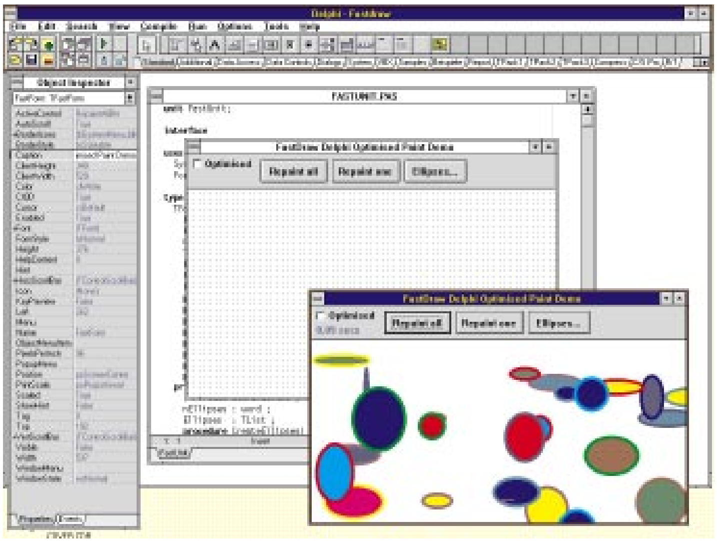
THE Delphi MAGAZINE

SPECIAL ELECTRONIC SAMPLE EDITION

1996

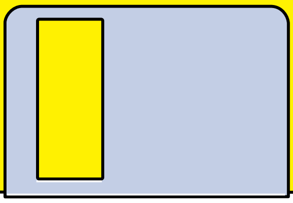


Included in this sample edition are all these articles from previous issues...

Optimising Display Updating

- ➤ **Subclassing Windows**
- ➤ **Writing Your Own Experts**
- ➤ **Inside TApplication**
- ➤ **Using Resource Files**

A FREE DISK IS INCLUDED WITH EACH ISSUE, WHICH CONTAINS ALL THE SOURCE CODE AND EXAMPLE FILES PLUS LOADS OF SHAREWARE AND FREEWARE COMPONENTS AND TOOLS AS WELL!



Delphi Internals

Dave Jewell shows how to detect CPU type

Under Construction

Bob Swart on creating custom property editors

Surviving Client/Server

An introduction to SQL by Steve Troxell

Plus...

The Delphi Clinic and Tips & Tricks

Contents

We have tried to include a varied selection of articles from past issues of The Delphi Magazine in this electronic sample edition and hope you enjoy them all. Current issues are 68 pages and contain more than we have been able to include in this electronic sampler. We currently have copies of all back issues, which come complete with their code disks containing the source and example files for these articles and a whole lot more. To receive back issues, just tick the box *'Please send me all available back issues'* on the subscription form. To go to the subscription form page please

CLICK HERE FOR ANY COUNTRY or **CLICK HERE FOR USA AND CANADA .**

Get Expert Help With All Your Delphi Projects...

DELIVERED TO YOUR DOOR, EVERY MONTH!

THE Delphi MAGAZINE

Issue 5, January 1996

UK £6.50

NOW MONTHLY!

Delphi 2.0 And The New 32-Bit BDE

- How To Use Resource Files
- Screen Capture
- Callbacks Part 2
- Typecasting Part 3

Delphi Internals
Detect CPU type with Dave Jewell

Under Construction
Bob Swart on adding custom events

Surviving Client/Server
TTable or TQuery? Ask Steve Troxell

INCLUDED ON THIS MONTH'S DISK ARE A DATA-AWARE OUTLINE COMPONENT, AN EQUATION PARSER, EASY MULTI-LANGUAGE SUPPORT AND MUCH MORE!

FULL DISK CONTENTS PAGE 64

ABOVE: Delphi 2.0's new multi-object grid , just one of the good things coming our way soon...

RIGHT: Add spell checking to your apps with just two lines of code using Eminent Domain's shareware EDSSpell component on the disk!

The Delphi Magazine is produced by Delphi developers for Delphi developers. We aim to provide the best technical material to help you develop better applications more quickly.

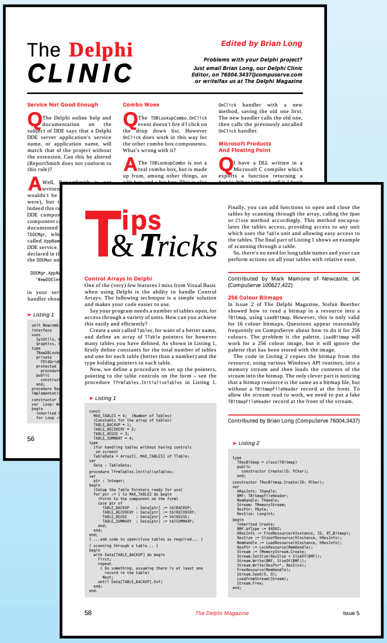
➤ Published monthly, each issue includes regular columns on building components and experts, client/server development, system-level issues, with ***The Delphi Clinic*** to solve your problems and ***Tips & Tricks*** too.

➤ Plus there are in-depth feature articles. Topics we've covered so far include: performance optimisation, resource files, callbacks, file handling, screen capture, custom database logins, typecasting, moving from other languages, custom clipboard formats, display updating and drag & drop.

➤ And in case that isn't enough there's product and book reviews and news. With each issue we include a ***free disk*** containing all the source and example files plus shareware/freeware components and tools as well.

There's more information and a subscription form overleaf. Subscribe now and we'll send you 13 issues for the price of 12!

For more information, call or fax us on +44 (0)181 460 0650
or email us on 70630.717@compuserve.com



The Delphi Clinic

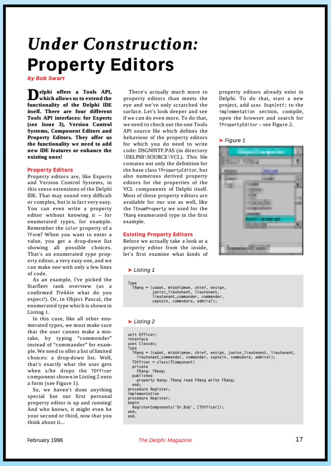
Edited by author, consultant and trainer Brian Long, the Clinic answers readers' "How do I..." and "Why won't it" queries every month.

Tips & Tricks

Readers' hints on getting the best out of Delphi, from Windows API insights to customising the VCL.

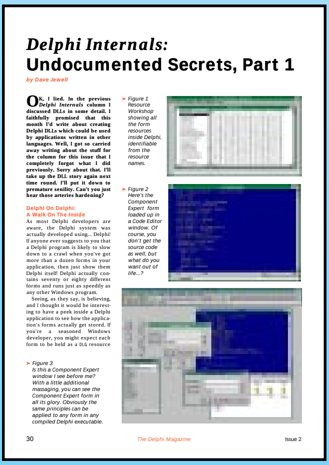
Surviving Client/Server

Written by Steve Troxell, this column provides insights and practical help on your Client/Server projects. Topics have included an introduction to SQL, choosing TTable or TQuery, plus stored procedures and triggers.



Under Construction

Bob Swart, known to many as "Dr.Bob", writes this monthly column on how to build your own Delphi components and experts, providing practical insights from his own development experience into what many have found to be an extraordinarily productive area of Delphi development. Topics covered so far include component building basics, encapsulating DLLs, extending components, adding built-in help, custom events, property editors and building data-aware components.



Delphi Internals

Dave Jewell helps make sense of the system-level issues which are so rarely discussed in programming books, with practical advice on topics such as converting projects to 32-bit, how to identify CPUs and drives, disk formatting and creating/using DLLS.

Plus there's much more...

Of course there are also in-depth feature articles, reviews of Delphi add-ons and books, plus an update on news in the Delphi world, in every issue of The Delphi Magazine. Our team of authors are some of the most knowledgeable and experienced Delphi developers around. Readers from over 40 countries are already benefiting from their insights – why not join them? Fill in the subscription form below and send it off today, or just telephone/email us.

FAX NOW!

Yes! Please enter my subscription to The Delphi Magazine

Please complete and fax or post to: The Delphi Magazine, 41 Recreation Road, Shortlands, BROMLEY, Kent BR2 0DY, United kingdom. Tel/Fax: +44 (0)181 460 0650. Email: CompuServe 70630,717

Name (Mr/Ms) ____

Position ______________________ Company ______________________

Address ____

______________________________ Town ______________________

County/State ______________ Post/Zip code ______________ Country ______________

Telephone ______________ Fax ______________ Email ______________

FL1

Subscription: 13 ISSUES FOR THE PRICE OF 12, please tick one:

☐ United Kingdom £75 ☐ Europe £80 ☐ USA/Canada £90 ☐ Rest of the World £95

☐ Please debit my VISA / MasterCard account by £ _______ Card Number: ______________________
Expiry Date: _____ / _____ Cardholder name: ______________________ Signature: ______________________

☐ I enclose a Sterling cheque drawn on a United Kingdom bank, or a Sterling Eurocheque, for £ _______
made payable to iTec *(We'll send you a receipt. Sorry, NO purchase orders! Please do not send payment in other currencies)*

Please tick one: ☐ Please start my subscription with the current issue *OR,* ☐ Please send me all available back issues

☐ Please tick here if you do NOT wish to receive information on relevant products and services from other companies

CHECK OUR WEB PAGE AT http://ourworld.compuserve.com/homepages/DelphiMagazine

Subscribing To The Delphi Magazine From The United States & Canada

If you live in the USA or Canada you can subscribe directly with our North American Agent in US dollars. Here's how.

We believe the best way to stay at the cutting edge of Delphi development is to read The Delphi Magazine! You will receive 12 issues a year, plus an ***extra free issue***, with in-depth technical articles, regular columns, Tips & Tricks, the Delphi Clinic, news, book and product reviews. We aim to make you more productive and help you enjoy your Delphi development even more.

As well as all this we include a free disk with each issue, which contains all the code and example files from that issue, plus extra shareware and freeware components and tools.

We think you'll agree that this represents excellent value for money and we are sure you won't want to miss a single issue. Your copies will be airmailed to you direct from the publishers in England, so you'll be sure to get them *FAST*.

Simply fill out the subscription form below and mail it as indicated. If you prefer, you can subscribe with a credit card by CompuServe™ email by sending the information on the form in an email message to 70602,1215. Or, just call us with your credit card details on (802) 244 7820.

The Delphi Magazine Subscription Request

Please complete this form and mail to:

ABAT

The Delphi Magazine (USA), RR1 Box 6020, WATERBURY CENTER, Vermont 05677

Payment must be included, we'll send a receipt to you. Sorry, **NO** purchase orders!

Name (Mr/Ms) ____________________

Position ____________________ Company ____________________

Address ____________________

____________________ City/Town ____________________

State ____________________ Zip ____________________

Tel ____________________ Fax ____________________

Email ____________________

USA & Canada Subscription US$140: 13 ISSUES FOR THE PRICE OF 12, including FREE disks*

Payment Method (please tick):

☐ Please debit my ☐ Visa ☐ Mastercard credit card (tick as appropriate) by US$140.00

Card Number: ____________________ Expiration date: Month:_____ Year:_____

Name on Card:____________________ Signature:____________________

☐ I enclose a check for US$140 payable to 'The Delphi Magazine (USA)'

Sorry, we can't accept payment by bank transfer

Note: please send payment only in **US dollars**

Would you like back issues? Please tick one:

☐ Please send me all available back issues ☐ Please start my subscription with the next issue

☐ Please tick here if you do **not** wish to receive information on relevant products and services from other companies.

**Note:* for subscriptions to other countries please contact the publishers direct:
iTec, 41 Recreation Road, Shortlands, BROMLEY, Kent BR2 0DY, United Kingdom
Telephone/Fax: +44 (0)181 460 0650 Email: 70630.717@compuserve.com

Under Construction:
Property Editors

by Bob Swart

Delphi offers a Tools API, which allows us to extend the functionality of the Delphi IDE itself. There are four different Tools API interfaces: for Experts (see Issue 3), Version Control Systems, Component Editors and Property Editors. They offer us the functionality we need to add new IDE features or enhance the existing ones!

Property Editors

Property editors are, like Experts and Version Control Systems, in this sense extensions of the Delphi IDE. That may sound very difficult or complex, but is in fact very easy. You can even write a property editor without knowing it – for enumerated types, for example. Remember the `color` property of a `TForm`? When you want to enter a value, you get a drop-down list showing all possible choices. That's an enumerated type property editor, a very easy one, and we can make one with only a few lines of code.

As an example, I've picked the Starfleet rank overview (as a confirmed *Trekkie* what do you expect!). Or, in Object Pascal, the enumerated type which is shown in Listing 1.

In this case, like all other enumerated types, we must make sure that the user cannot make a mistake, by typing "commonder" instead of "commander" for example. We need to offer a list of limited choices: a drop-down list. Well, that's exactly what the user gets when s/he drops the `TOfficer` component shown in Listing 2 onto a form (see Figure 1).

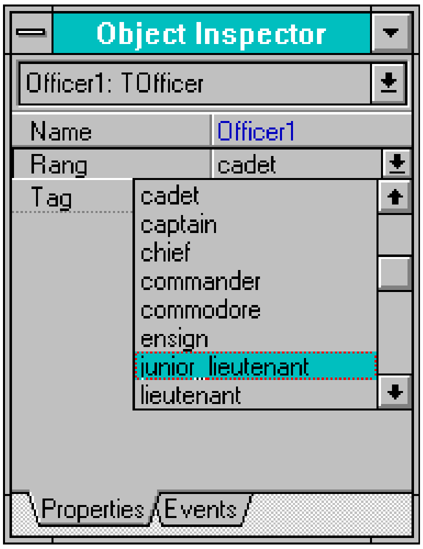
So, we haven't done anything special but our first personal property editor is up and running! And who knows, it might even be your second or third, now that you think about it...

There's actually much more to property editors than meets the eye and we've only scratched the surface. Let's look deeper and see if we can do even more. To do that, we need to check out the one Tools API source file which defines the behaviour of the property editors for which you do need to write code: DSGNINTF.PAS (in directory \DELPHI\SOURCE\VCL). This file contains not only the definition for the base class `TPropertyEditor`, but also numerous derived property editors for the properties of the VCL components of Delphi itself. Most of these property editors are available for our use as well, like the `TEnumProperty` we used for the `TRang` enumerated type in the first example.

Existing Property Editors

Before we actually take a look at a property editor from the inside, let's first examine what kinds of property editors already exist in Delphi. To do that, start a new project, add `uses DsgnIntf;` to the `implementation` section, compile, open the browser and search for `TPropertyEditor` – see Figure 2.

➤ *Figure 1*



Name: Officer1
Rang: cadet

➤ *Listing 1*

```
Type
  TRang = (cadet, midshipman, chief, ensign,
           junior_lieutenant, lieutenant,
           lieutenant_commander, commander,
           captain, commodore, admiral);
```

➤ *Listing 2*

```
unit Officer;
interface
uses Classes;
Type
  TRang = (cadet, midshipman, chief, ensign, junior_lieutenant, lieutenant,
    lieutenant_commander, commander, captain, commodore, admiral);
  TOfficer = class(TComponent)
  private
    FRang: TRang;
  published
    property Rang: TRang read FRang write FRang;
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Dr.Bob', [TOfficer]);
end;
end.
```

If I count correctly, there are at least 21 property editors registered by the DSGNINTF unit. Note, however, that there are actually even more property editors available, like the `TPictureEditor` in \DELPHI\LIB\PICEDIT.DCU (I'll return to this later on...).

TPropertyEditor

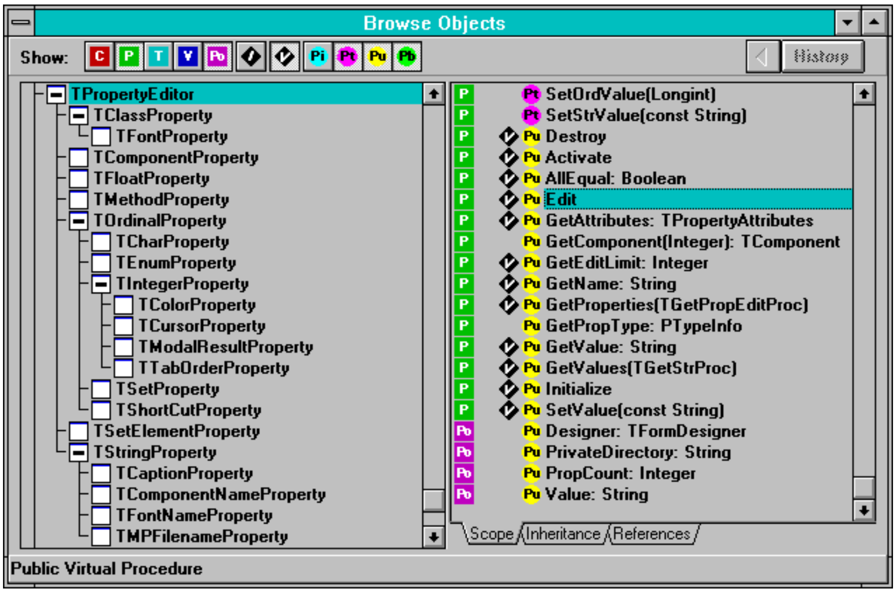
What does a property editor look like? Well, like an expert, it is derived from a base class, from which we need to override some methods in order to make things work our way. The `TPropertyEditor` base class is defined as shown in Listing 3 (I've left out the `private` parts, as we can't touch them anyway).

A `TPropertyEditor` edits a property of a component, or a list of components, selected into the Object Inspector. The property editor is created based on the type of the property being edited, as determined by the types registered by `RegisterPropertyEditor`. A `TPropertyEditor` is used by the Object Inspector whenever a property is modified.

For now, we will just focus on a subset of the methods which can be overridden to change the behaviour of the property editor (we'll get back to the others in a future instalment of this column).

`GetAttributes` is the most important method, as it determines the kind of property editor and its behaviour. There are three kinds of property editors (other than the default editbox-type): a dropdown value list (we've seen that one before), a sub-property list and a dialog. `GetAttributes` returns a set of type `TPropertyAttributes`:

- `paValueList`: The property editor can return an enumerated list of values for the property. If `GetValues` calls `Proc` with values then this attribute should be set. This will cause the dropdown button to appear to the right of the property in the Object Inspector.
- `paSubProperties`: The property editor has sub-properties which will be displayed indented and below the current property in standard outline format. If `GetProperties` will generate property objects then this attribute should be set.



```
TPropertyEditor
  TClassProperty
    TFontProperty
  TComponentProperty
  TFloatProperty
  TMethodProperty
  TOrdinalProperty
    TCharProperty
    TEnumProperty
    TIntegerProperty
      TColorProperty
      TCursorProperty
      TModalResultProperty
      TTabOrderProperty
    TSetProperty
    TShortCutProperty
  TSetElementProperty
  TStringProperty
    TCaptionProperty
    TComponentNameProperty
    TFontNameProperty
    TMPFilenameProperty
```

```
P   Pt SetOrdValue(Longint)
P   Pt SetStrValue(const String)
P   Pu Destroy
P   Pu Activate
P   Pu AllEqual: Boolean
P   Pu Edit
P   Pu GetAttributes: TPropertyAttributes
P   Pu GetComponent(Integer): TComponent
P   Pu GetEditLimit: Integer
P   Pu GetName: String
P   Pu GetProperties(TGetPropEditProc)
P   Pu GetPropType: PTypeInfo
P   Pu GetValue: String
P   Pu GetValues(TGetStrProc)
P   Pu Initialize
P   Pu SetValue(const String)
Pb  Pu Designer: TFormDesigner
Pb  Pu PrivateDirectory: String
Pb  Pu PropCount: Integer
Pb  Pu Value: String
```

Public Virtual Procedure

➤ *Figure 2*

```
Type
  TPropertyEditor = class
  protected
    function GetPropInfo: PPropInfo;
    function GetFloatValue: Extended;
    function GetFloatValueAt(Index: Integer): Extended;
    function GetMethodValue: TMethod;
    function GetMethodValueAt(Index: Integer): TMethod;
    function GetOrdValue: Longint;
    function GetOrdValueAt(Index: Integer): Longint;
    function GetStrValue: string;
    function GetStrValueAt(Index: Integer): string;
    procedure Modified;
    procedure SetFloatValue(Value: Extended);
    procedure SetMethodValue(const Value: TMethod);
    procedure SetOrdValue(Value: Longint);
    procedure SetStrValue(const Value: string);
  public
    destructor Destroy; override;
    procedure Activate; virtual;
    function AllEqual: Boolean; virtual;
    procedure Edit; virtual;
    function GetAttributes: TPropertyAttributes; virtual;
    function GetComponent(Index: Integer): TComponent;
    function GetEditLimit: Integer; virtual;
    function GetName: string; virtual;
    procedure GetProperties(Proc: TGetPropEditProc); virtual;
    function GetPropType: PTypeInfo;
    function GetValue: string; virtual;
    procedure GetValues(Proc: TGetStrProc); virtual;
    procedure Initialize; virtual;
    procedure SetValue(const Value: string); virtual;
    property Designer: TFormDesigner read FDesigner;
    property PrivateDirectory: string read GetPrivateDirectory;
    property PropCount: Integer read FPropCount;
    property Value: string read GetValue write SetValue;
  end;
```

➤ *Listing 3*

- `paDialog`: Indicates that the `Edit` method will bring up a dialog. This will cause the '...' button to be displayed to the right of the property in the Object Inspector.
- `paSortList`: The Object Inspector will sort the list returned by `GetValues` (by name).
- `paAutoUpdate`: Causes the `SetValue` method to be called on each change made to the editor instead of after the change has been approved (eg the `Caption` property).

- `paMultiSelect`: Allows the property to be displayed when more than one component is selected. Some properties are not appropriate for multi-selection (eg the `Name` property).
- `paReadOnly`: The value is not allowed to change.

`GetValue` returns the string value of the property. By default this returns `(unknown)`. This should be overridden to return the appropriate value. `GetValues` is called when `paValueList` is returned in `GetAttributes`. It should call the argument `Proc` for every value which is acceptable for this property. `TEnumProperty` will pass every element in the enumeration, as we can see in Figure 1.

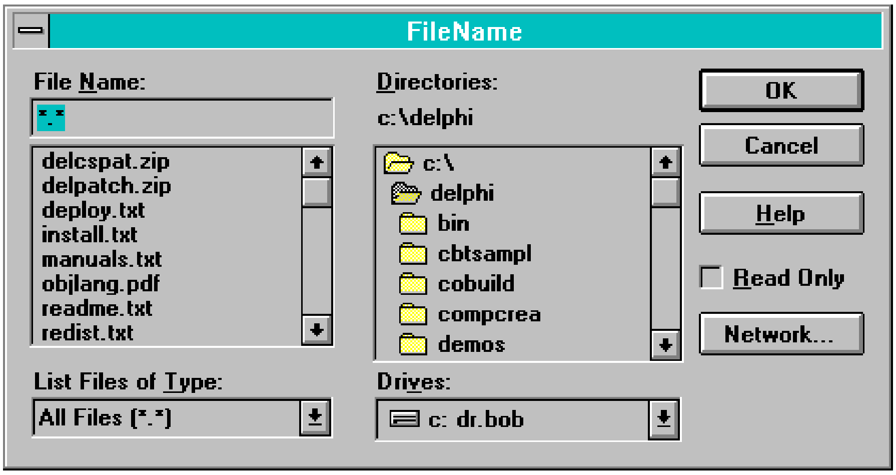
`SetValue(Value)` is called to set the value of the property. The property editor should be able to translate the string and call one of the `SetXxxValue` methods. If the string is not in the correct format or not an allowed value, the property editor should generate an exception describing the problem. `SetValue` can ignore all changes and allow all editing of the property to be accomplished through the `Edit` method (eg the `Picture` property).

`Edit` is called when the '...' button is pressed or the property is double-clicked. This can, for example, bring up a dialog to allow the editing the property in some more meaningful fashion than by text (eg the `Font` property).

TFileNameProperty

With these few basic methods we now have enough power at our disposal to write our first non-trivial property editor: an open filename dialog property editor for filename properties.

We must remember that writing components is essentially a non-visual task and writing property editors is no different. We have to write a new unit by hand in the editor (see Listing 4). We need to specify that we want a dialog type of property editor, so we return `[paDialog]` in the `GetAttributes` function. Then, we can do as we like in the `Edit` procedure, which in this case involves a `TOpenDialog` to let us find any existing file.



➤ *Figure 3*

```
unit FileName;
interface
uses DsgnIntf;
Type
  TFileNameProperty = class(TStringProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
  end;
implementation
uses
  Dialogs, Forms;
function TFileNameProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog]
end {GetAttributes};
procedure TFileNameProperty.Edit;
begin
  with TOpenDialog.Create(Application) do
  try
    Title := GetName; { name of property as OpenDialog caption }
    Filename := GetValue;
    Filter := 'All Files (*.*)|*.*';
    HelpContext := 0;
    Options := Options + [ofShowHelp, ofPathMustExist, ofFileMustExist];
    if Execute then SetValue(Filename);
  finally
    Free
  end
end {Edit};
end.
```

➤ *Listing 4*

Note that we call the `GetName` function of the property editor to get the name of the actual property for which we want to fire up the `TOpenDialog`. For a property called `FileName`, this would result in the example shown in Figure 3.

In just a few lines of code we've written a `TFileName` property editor which will give great support at design time for all our components which use a property of type `TFileName`. This illustrates that property editors have an enormous potential for designers of Delphi components.

TFileModeProperty

We've seen how we can create and execute a simple `TOpenDialog` as a property editor. But instead of just a `TOpenDialog` component, we can of course show a complete newly designed form in our property editor's `Edit` method.

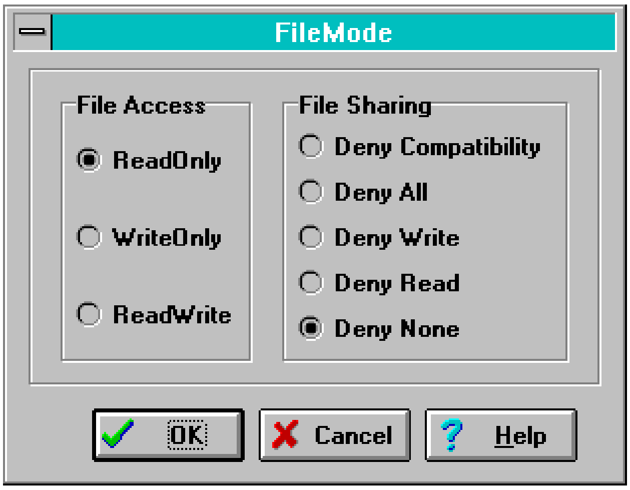
Let's design a `FileMode` property editor in which we can specify both the file access (read-only, write-only, read-write) and file sharing (deny-none, deny-read, deny-write, deny-all) values for a file. The code for a simple form to enable us to pick these options using two `TRadioGroup` controls is shown in Listing 5.

I've used two `TRadioGroups` to hold the items we can choose from in a list, just like a listbox or combobox. I think the `TRadioGroup` is probably one of the most underestimated components on the palette: one `RadioGroup` is capable of showing several radio buttons which are all easily accessible.

The form is shown in action in Figure 4, specifying a read-only deny-none filemode.

To turn this form into a property editor, we only need to activate it at the right time (ie in the `Edit` method of the `TFileModeProperty` editor), and set and get the actual value of the filemode property we need. See Listing 6.

➤ *Figure 4*



FileMode

File Access
● ReadOnly
○ WriteOnly
○ ReadWrite

File Sharing
○ Deny Compatibility
○ Deny All
○ Deny Write
○ Deny Read
● Deny None

```
unit FileMode;
interface
uses
  Forms, Buttons, StdCtrls, ExtCtrls;
Type
  TFileModeDlg = class(TForm)
    OKBtn: TBitBtn;
    CancelBtn: TBitBtn;
    HelpBtn: TBitBtn;
    Bevel1: TBevel;
    FileAccess: TRadioGroup;
    FileSharing: TRadioGroup;
    procedure OKBtnClick(Sender: TObject);
    procedure FormActivate(Sender: TObject);
  public
    FileShareMode: Word;
  end;
implementation
{$R *.DFM}
procedure TFileModeDlg.FormActivate(Sender: TObject);
begin
  FileAccess.ItemIndex := (FileShareMode MOD $10);
  FileSharing.ItemIndex := (FileShareMode SHR 4)
end;
procedure TFileModeDlg.OKBtnClick(Sender: TObject);
begin
  FileShareMode := FileAccess.ItemIndex + (FileSharing.ItemIndex SHL 4)
end;
end.
```

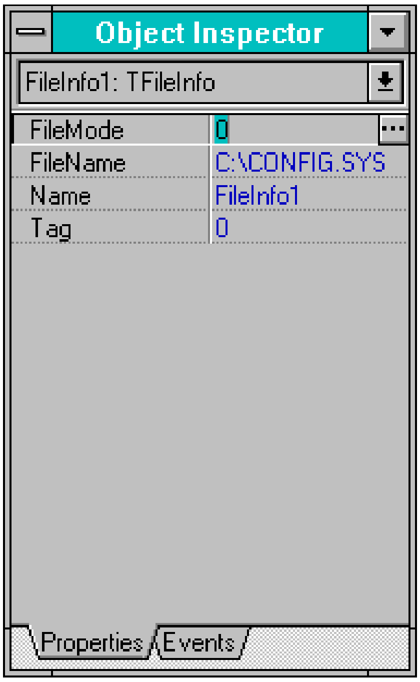
➤ *Listing 5*

TFileInfo

To illustrate the use of this `TFileModeProperty` editor, I've designed a new component called `TFileInfo`, derived from a standard non-visual `TComponent`, with two new properties: `FileName` and `FileMode`. The first one is connected to the `TFileName` property editor we created earlier, while the second one uses the `TFileMode` property editor to interactively set the desired `FileMode`. See Listing 7.

Note that this component registers its own property editors in the `Register` procedure where the component itself is registered. Figure 5 shows the `TFileInfo` component in action with the two new property editors.

➤ *Figure 5*



Object Inspector

FileInfo1: TFileInfo

FileMode	0
FileName	C:\CONFIG.SYS
Name	FileInfo1
Tag	0

TBUUCode

While `TFileInfo` is but an example component, we can easily extend the `TBUUCode` components for file UUEncoding and UUDecoding from the last few issues, using these property editors for the `InputFile` property.

In fact, I've already done so and the new source code for the `TBUUEnCode` and `TBUUDeCode` components is (again) included on the subscribers' disk with this issue.

TPictureEditor

We've already seen how to make picture editors that behave like dialogs. And this reminds me of the most irritating property editor in Delphi: the picture editor for glyphs, icon, bitmaps etc.

It's not the fact that it doesn't work, it's just the fact that it isn't very user friendly. If I click on the `Load` button I get a `TOpenDialog` that gives me the option to select a .BMP, .ICO or whatever file I wish. However, I don't get to see what's actually inside this file until I close the `TOpenDialog`. Then I'm back in the `Picture Editor` and if I decide it's not the file I really want I have to click on the `Load` button again

```
unit FileProp;
interface
uses DsgnIntf;
Type
  TFileModeProperty = class(TIntegerProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
  end;
implementation
uses
  SysUtils, Controls, FileMode;
function TFileModeProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog]
end {GetAttributes};
procedure TFileModeProperty.Edit;
begin
  with TFileModeDlg.Create(nil) do
  try
    FileShareMode := GetOrdValue;
    if ShowModal = mrOk then
      SetOrdValue(FileShareMode)
  finally
    Free
  end
end {Edit};
end.
```

➤ *Listing 6*

```
unit FileInfo;
interface
uses
  Classes, SysUtils;
Type
  TFileInfo = class(TComponent)
  private
    FFileName: TFileName;
    FFileMode: Word;
  published
    property FileName: TFileName read FFileName write FFileName;
    property FileMode: Word read FFileMode write FFileMode;
  end;
procedure Register;
implementation
uses
  DsgnIntf, FileProp, FileName;
procedure Register;
begin
  RegisterComponents(’Dr.Bob’, [TFileInfo]);
  RegisterPropertyEditor(TypeInfo(Word), TFileInfo, ’FileMode’,
     TFileModeProperty);
  RegisterPropertyEditor(TypeInfo(TFilename), nil, ’’, TFilenameProperty)
end;
end.
```

➤ *Listing 7*

and start all over again. It’s especially annoying if you have to browse through a directory with many `TBitBtn` bitmap files.

I would like a *preview* option, so I can see what the image in a file looks like *while* I’m browsing through a directory! It sounds exactly like a new property editor to me (Note: since Borland didn’t provide us with the source of PICEDIT.DCU, we don’t have the PICEDIT.DFM form either and have to write our own picture editor instead of enhancing the already existing one).

TImageForm

First of all, we have to design the actual dialog or form which will be used by our new property editor. Figure 6 shows my form: the area where the image of Dr.Bob is shown, in the lower-right corner, is used to display the image of any file which is currently selected in the file listbox. Depending on your needs, you can even stretch this image (not recommended for little `TBitBtn` bitmap images, but useful if you have large bitmaps and only want a preview).

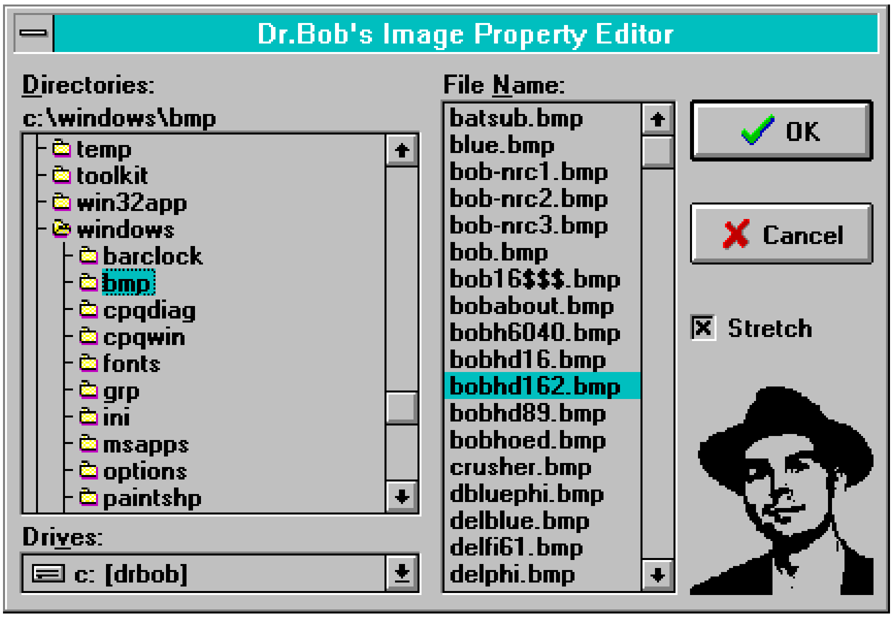
Note that I’ve used the component `TDirectoryOutliner` (from my *Performance Optimisation* articles in Issues 4 and 5) in this form. The complete source code is on the disk with this issue, but I’ve also uploaded the .DCU version of this property editor to the DELPHI forum on CompuServe, so if you don’t have a subscription yet (shame on you!) then you can at least get it to work.

Now that we have a form to ask for which image to use, let’s see how we can get this to work as a property editor. We need to take a look at GRAPHIC.PAS to see what kinds of pictures, glyphs and images exist in the first place. It seems we are limited to two descendants of `TPersistent`: `TPicture` and `TGraphic`, with descendent `TBitmap` of `TGraphic`. For this column, let’s just focus on .BMP files, and hence on the `TPicture` and `TBitmap` classes only. This means we want to offer the new Image Property Editor for properties of type `TPicture` and `TBitmap`. See Listing 8.

Note that since we don’t explicitly want the `TPictureEditor` to belong to one specific component, we have to register it ourselves here, and install it just like any other custom component or expert from the Delphi IDE’s `Options | Install Components` dialog. After rebuilding your COMPLIB.DCL (remember to make that backup first!), you will get the new picture editor for each `TPicture` (in a `TImage`) or `TBitmap` (for example in a `TSpeedButton` or `TBitBtn`).

One last important thing: Delphi already had a property editor installed for `TPictures` and `TBitmaps`, namely the picture editor Borland provided. Won’t we get into trouble if we want to use our own?

No, we won’t, because it seems that the last property editor which has been registered for a particular component or property type will override the previous one. So, if you ever install another property



Directories: c:\windows\bmp

File Name: bobhd162.bmp

Drives: c: [drbob]

☒ Stretch

➤ *Figure 6*

```
unit PictEdit;
interface
uses DsgnIntf;
Type
  TPictureEditor = class(TClassProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
  end;
  procedure Register;

implementation
uses
  SysUtils, Controls, Graphics, TypInfo, ImageFrm;

function TPictureEditor.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog]
end {GetAttributes};

procedure TPictureEditor.Edit;
begin
  with TImageForm.Create(nil) do
  try
    ImageDrBob.Picture := TPicture(GetOrdValue);
    if ShowModal = mrOk then begin
      if (GetPropType^.Name = ’TPicture’) then
        SetOrdValue(LongInt(ImageDrBob.Picture))
      else { Bitmap }
        SetOrdValue(LongInt(ImageDrBob.Picture.Bitmap))
    end
  finally
    Free
  end
end {Edit};

procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TPicture), nil, ’’, TPictureEditor);
  RegisterPropertyEditor(TypeInfo(TBitmap), nil, ’’, TPictureEditor)
  end;
end.
```

➤ *Listing 8*

editor for `TBitmaps`, for example, you will override the one we've just built.

More Property Editors

So far, we've only seen a few of the possible kinds of property editors we can write. We've focussed especially on `paDialog` property editors – in my view the easiest way to customise the entry of property values at design time. However, there are many more kinds of property editors and ways to write them, so we'll just have to come back later to explore these in detail in future columns. For now, I hope I've given you enough to chew on for a month!

Next Time

Next time we'll focus on a type of component we haven't touched at all so far: data-aware components. We'll explore how they work, learn what makes them tick and even develop one of our own: a data-aware multi-media player. After that, we'll slowly return to the subject of Tools APIs and Delphi IDE Experts when we start exploring the so-called Component Editors in the April issue.

Stay tuned and make sure you've always got a backup of your COMPLIB.DCL in a *save* place!

Bob Swart (you can email him at 100434.2072@compuserve.com) is a professional 16- and 32-bit software developer using Borland Delphi and sometimes a bit of Pascal or C++. In his spare time, he likes to watch video tapes of Star Trek Voyager with his almost two year old son Erik Mark Pascal.

Using Resource Files In Delphi

by Dave Bolt

For those of us who have previously used C and C++ to write our Windows programs, Delphi's use of resource files seems a little strange. The only real problem is that when a project is created in Delphi, a resource file is also created with the same name as the project. ***Delphi*** controls this file. Using the Image Editor to add bitmaps, cursors or icons to the Delphi-controlled resource file is doomed to failure under normal circumstances.

However, all is not lost. In this article I'll work through some examples of the use of bitmaps, icons and cursors from resources, then finish off by outlining a method by which the Delphi-controlled resource file can be manipulated without raising any objections from Delphi.

The Image Editor

A word of caution should be given about the Image Editor. Apart from my own programming errors, this is the only part of Delphi that has caused problems to date.

I would advise anyone using this tool to make sure that everything else which is open has been saved first. I would also advise that any work done in the Image Editor be saved regularly and that each open image and file be closed before closing the editor. This will not prevent problems totally, but will reduce the likelihood of work being lost.

Loading From .BMP Files

This first project loads a bitmap from a file which is not part of a resource file. In the project itself, the name of the bitmap is given explicitly. Unless a path is defined, Delphi assumes that the bitmap is in the same directory as the executable file. You can of course include the full path and file name for the bitmap in the code, but this would tie your application to a specific directory structure.

If we create a new project, rename UNIT1.PAS to BIT1.PAS and PROJECT1.DPR to BITPRJ1.DPR, then add a `TImage` component to the form, we have the basic framework to display an image. We can initialise the image component by adding code to the FormCreate handler as in Listing 1.

Compiling and running this project results in the bitmap being displayed. If the bitmap is not found when the program runs, an `EFOpenError` exception is generated. This can easily be detected and handled if required.

The object `MyBitmap` is our responsibility, and must be created before use then destroyed afterwards by our code. A separate copy of the bitmap is stored in the `TImage` component and is automatically allocated and de-allocated by the program.

Loading From Resource Files

To load a resource from a resource file, the file must have a different name to the project in which it is used. If we create a new project exactly as above, but call the files BIT2.PAS and BITPRJ2.DPR, we find that after saving the project there is a resource file called BITPRJ2.RES in the directory where the project was saved. In order to access our own resource file, it must be referenced explicitly in the project file (.DPR) using, eg:

```
{$R BITS.RES}
```

for a file called BITS.RES (which is on this issue's disk along with the other files from this article, and contains a bitmap resource called `FIRST`). You might think that the `{$R *.RES}` statement included by default in the .DPR file will pull in your own .RES files, but this is strangely not the case! The best place to insert your `{$R ...}` statement is *after* the existing one, after the uses statement.

If we add the same code to the `FormCreate` method as in Listing 1, but change the `LoadFromFile` command to:

```
MyBitmap.Handle :=
  LoadBitmap(hInstance,
  'FIRST');
```

then tidy up the display by adding the following two lines:

```
Image1.Width :=
  MyBitmap.Width;
Image1.Height :=
  MyBitmap.Height;
```

then run the program, we should get the same image displayed as before. The extra lines of code adjust the size of the `TImage` component to fit the bitmap at run time.

Caution

If the bitmap which is to be loaded is not in the resource file, the program will still run, it just won't display the bitmap. This situation can be avoided by testing to see if the handle is zero after a call to `LoadBitmap` and taking the appropriate action. Failure can also occur if there is insufficient memory to load the bitmap.

➤ *Listing 1*

```
procedure TForm1.FormCreate(Sender : TObject);
var
  MyBitmap : TBitmap;
begin
  MyBitmap := TBitmap.Create;
  MyBitmap.LoadFromFile('FIRST.BMP');
  Image1.Canvas.Draw(0, 0, MyBitmap);
  MyBitmap.Destroy;
end;
```

Custom Cursors

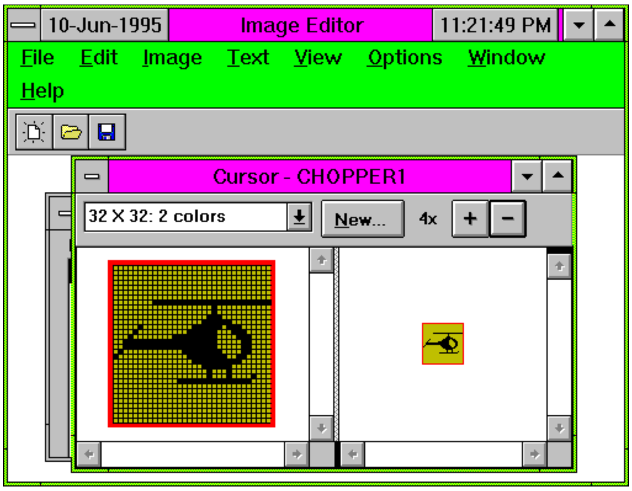
by Ken Otto

Creating custom cursors in Delphi can be a confusing venture. Not because it is hard, but because it is so poorly illustrated (and even wrong) in the Delphi documentation and help files.

A cursor actually contains two 32 x 32 monochrome bitmaps. One of these bitmaps is referred to as the 'XOR' bitmap, and the other is known as the 'AND' bitmap. When you create a cursor with Delphi's Image Editor, you won't need to worry about this. The cursor also has two fields defining the 'hot-spot': the point on the cursor representing its exact location.

There are 17 pre-defined cursor constants for your Delphi application. Some of the constant values in the Delphi Help file are incorrect. A corrected listing of cursors and their constants is shown opposite.

To create your own cursor, first create a resource (.RES) file. From the Delphi IDE select `Tools|Image Editor` then select `File|New`. By default, the 'Resource File (RES)' radio button will be selected. Choose `OK`. A window appears, captioned 'Untitled1.RES'. Click `New`, and a window will pop up asking for the type of resource you want to create. Choose `Cursor` and click `OK`. You may need to maximize the window at this point. Select a tool and begin drawing your cursor. If you make a mistake, select `Edit|Undo` to erase the last image written (this will continue to remove several layers each time it is selected). You can enlarge the drawing area by clicking the `Zoom` button. If you prefer a grid on the drawing area, select `Options|Show Grid On Zoom`. If you want the hot-spot to be the center of the cursor, select `Image|Hotspot...` and place 16 in the X and Y fields.



When you have finished drawing your cursor, you will probably want to rename it from the default name `CURSOR_1`: close the cursor editor window, ensure the Cursors tab on the untitled project window is selected, select `CURSOR_1` and click the `Rename` button. Make sure the new name is all capital letters. Now save the .RES file by selecting `File|Save As`, taking care not to give the file the same name as your project. A sample is included in the CHOPDEMO.LZH archive on the disk.

Ken Otto writes Pascal applications on the HP3000 in Sacramento, CA, USA; programming in Delphi is a hobby he enjoys. He can be reached on CompuServe at 73041,1336

Resources By Number

by Brian Long

When you name a resource in a resource file, Windows lets you choose numbers instead of names. In fact Microsoft recommends you use numbers instead of names for efficiency. However, the Image Editor will only store character strings as resource identifiers.

To generate a resource file for a cursor that marks resources by number, make a cursor resource file (.CUR file) with the Image Editor. One is supplied on the disk in the NUMCURS.LZH archive as file TARGET.CUR. A text file then acts as a resource script (a file with a .RC extension), and can look like this file, CURSOR2.RC (see my Typecasting Part 3 article in this issue for details of how to share constants between the resource script and the Delphi project):

```
 2 CURSOR TARGET.CUR
```

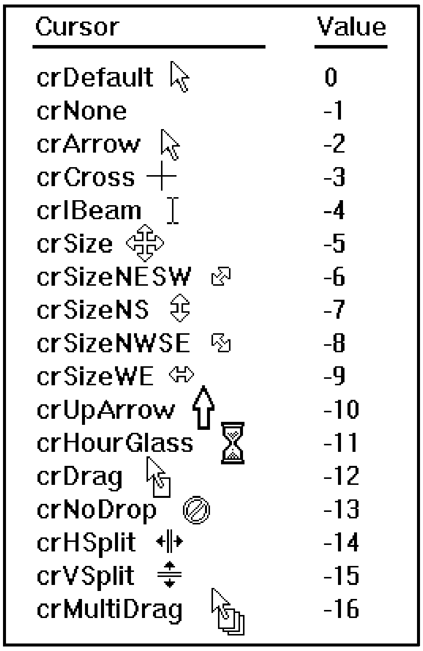
This can be compiled with the command-line resource compiler BRCC.EXE (found in the directory DELPHI\BIN) with the command: `BRCC CURSOR2.RC`

The `{$R CURSOR2.RES}` compiler directive will bind in the resulting CURSOR2.RES and `LoadCursor` can also be used to load up a numbered, as opposed to named, resource. The last parameter to `LoadCursor` needs to be a `PChar` type, but we wish to specify a number. The parameter can be either `PChar(2)`, `'#2'` or `MakeIntResource(2)`. So, the `OnCreate` handler will look like:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Screen.Cursors[crTarget] :=
    LoadCursor(HInstance, PChar(2)
  Form1.Cursor := crTarget;
end;
```

The complete example project in NUMCURS.LZH on the disk uses a 'named' cursor for the form and a different 'numbered' cursor for a button on the form.

Brian Long... well, by now surely he needs no introduction!



Cursor	Value
crDefault	0
crNone	-1
crArrow	-2
crCross	-3
crIBeam	-4
crSize	-5
crSizeNESW	-6
crSizeNS	-7
crSizeNWSE	-8
crSizeWE	-9
crUpArrow	-10
crHourGlass	-11
crDrag	-12
crNoDrop	-13
crHSplit	-14
crVSplit	-15
crMultiDrag	-16

➤ *Standard Cursors in Delphi*

A Simple Animation

This animation is so simple that it only has two frames. The program cycles through the images to hopefully give the impression of movement. This is the kind of thing which makes buttons appear to be pressed in or out in Windows programs and to animate logos.

If we take the project in the previous example and save it using the names BIT3.PAS and BITPRJ3.DPR, we have the basis for the next step. We need to add a timer to the form and the following variable declarations in the public part of the BIT3.PAS unit:

```
MyBitmap :
  array [0..1] of TBitmap;
BitMapNum : integer;
```

We also need `FormCreate`, `Timer1` and `FormDestroy` handlers as in Listing 2.

This project will now load two images from a resource file into an array of `TBitmap` objects and alternately display them in the image component. The image displayed is updated every fifth of a second in response to the timer. Note that the `TBitmap` array is declared as part of the `TForm1` object, instead of being local to the `FormCreate` handler, and must be initialised in `FormCreate` and destroyed in `FormDestroy`.

```
procedure TForm1.FormCreate(Sender : TObject);
begin
  MyBitmap[0] := TBitmap.Create;
  MyBitmap[1] := TBitmap.Create;
  MyBitmap[0].Handle := LoadBitmap(hInstance, ’FIRST’);
  MyBitmap[1].Handle := LoadBitmap(hInstance, ’SECOND’);
  Image1.Width := MyBitmap[0].Width;
  Image1.Height := MyBitmap[0].Height;
  { Show the image component copy of the bitmap }
  Image1.Canvas.Draw(0, 0, MyBitmap[0]);
  BitMapNum := 0;
  Timer1.Interval := 200;
end;
procedure TForm1.Timer1Timer(Sender : TObject);
begin
  BitMapNum := (BitMapNum+1) MOD 2;
  Image1.Canvas.Draw(0, 0, MyBitmap[BitMapNum]);
end;
procedure TForm1.FormDestroy(Sender : TObject);
begin
  MyBitmap[0].Destroy;
  MyBitmap[1].Destroy;
end;
```

➤ *Listing 2*

```
procedure TForm1.FormCreate(Sender : TObject);
begin
  Screen.Cursors[1]:=LoadCursor(HInstance,’ONE’);
  Cursor:=1; { You could also reference your cursor as a constant:        }
end;         { “Cursor := crMyCursor;” just by including the statement     }
             { “const crMyCursor = 1;” before “implementation” in the unit }
```

➤ *Listing 3*

We have now loaded bitmaps from a bitmap file and from resource files. The loaded data has been copied to a `TImage` component either once only (at form creation), or repeatedly in response to an event. Similar things can be done with cursors.

Loading A Cursor

As they say in all the best recipes, first create your cursor (see opposite). Then, as for the bitmaps, create a new project, name the files CUR1.PAS and CURPRJ1.DPR, and add the `FormCreate` handler which is shown in Listing 3. Also, in the CUR1.PAS unit file, add a `{$R}` statement after the `interface` keyword: `{$R CURSORS.RES}`.

Although Delphi will permit you to insert this statement in a number of places in the unit, this seems to be the place to put it to get it to work. The `{$R}` statement can also go after the default `{$R}` statement in the .DPR file, as for the bitmap projects. If the unit is to be used in another project, this could lead to problems remembering to include the correct file so I prefer to add it to the relevant unit.

`Screen.Cursors[]` is an array of cursors supplied by Delphi. The default cursors use index numbers from 0 for the default cursor to -17 for `crSQLWait`. Unless we wish to replace any of the defaults, the best strategy is to use cursor numbers starting from 1 and working up.

Running the program should give a strange cursor consisting of an angle and a ring containing a black quadrant. If an attempt is made to load a cursor resource, but there is nothing matching that name in the resource files, the handle will be zero. If a resource other than a cursor is found, the handle will not be zero, so take care to only reference *cursors* in `LoadCursor`.

An Animated Cursor

We can create a new project by saving the previous one as CUR2.PAS and CURPRJ2.DPR. For the animation we need a `TTimer` component on the form and this time also a `TButton`. Caption the button ‘Finish’ so that there is some point to including it, and it helps if the size is a little larger than normal. Also set the `Cursor` property of the button to `crCross`.

In the `Public` part of the type `TForm1` declaration include:

```
 CursorCount : Integer;
```

The handlers in Listing 4 are also required.

`CursorCount` is used to keep track of the next cursor to load. Having it zero-based simplifies the arithmetic. The first user-defined cursor is 1 and the last is 4, so we add 1 to `CursorCount` to give the cursor number to use. When this program is run the cursor will have a rotating quadrant as part of it. The effect can be improved considerably by drawing eight versions with the quadrant moved 45 degrees from the last position.

Compare the behaviour of the cursor with the previous version. As the cursor moves across the non-client area of the form, it starts flashing between the custom cursor and the relevant default cursor. This undesirable behaviour is brought on by the slightly simplistic method of changing the cursor in the timer handler. Note also that when the cursor is over the ‘Finish’ button it changes to the shape set at design time. Try this with the `Button1.Enabled` property set to `False`.

One method of modifying the cursor behaviour further would be to use `GetCursorPos` to find the position in terms of global co-ordinates, then convert to client area co-ordinates and check if the cursor is actually in a valid region or not.

Another method, which is demonstrated in CURPRJ2A.PRJ, uses one of the less known windows messages `WM_NCMOUSEMOVE`, which is generated in response to mouse movement in the non-client area of a window. This just happens to be perfect for controlling the flashing effect in CURPRJ2.PRJ.

We need to amend the `private` and `public` declarations of the type section in the unit file (now CUR2A.PAS) as in Listing 5.

```
procedure TForm1.FormCreate(Sender : TObject);
begin
  Screen.Cursors[1] := LoadCursor(HInstance,’ONE’);
  Screen.Cursors[2] := LoadCursor(HInstance,’TWO’);
  Screen.Cursors[3] := LoadCursor(HInstance,’THREE’);
  Screen.Cursors[4] := LoadCursor(HInstance,’FOUR’);
  Cursor := 1;
  CursorCount := 0;
  Timer1.Interval := 100;
end;
procedure TForm1.Timer1Timer(Sender : TObject);
begin
  CursorCount := (CursorCount+1) MOD 4;
  Cursor := CursorCount+1;
end;
procedure TForm1.Button1Click(Sender : TObject);
begin
  Close;
end;
```

➤ *Listing 4*

```
private
  procedure WMNCMouseMove(var AMessage : TMessage); message WM_NCMouseMove;
public
  CursorCount : integer;
  DeadArea : BOOL;   { True if NonClient Area }
end;
```

➤ *Listing 5*

```
procedure TForm1.WMNCMouseMove(var AMessage : TMessage);
begin
  DeadArea := TRUE; {Cursor is on the form but in an invalid area}
  AMessage.Result := 0; { Win API Help says return 0 if message was handled}
  inherited;
end;
procedure TForm1.Timer1Timer(Sender : TObject);
begin
  { Calculate the next cursor }
  CursorCount:= (CursorCount+1) MOD 4;
  { If in a valid region, Update cursor }
  if not DeadArea then Cursor := CursorCount+1;
end;
procedure TForm1.FormMouseMove(Sender : TObject; Shift : TShiftState;
  X,Y : Integer);
begin
  DeadArea := FALSE; { If this handler is called, DeadArea must be false }
  Cursor := CursorCount+1; { Update Cursor }
end;
```

➤ *Listing 6*

The `WMNCMouseMove` procedure is a message handler for the required message. It overrides the default message handler, but instead of the `override` keyword it uses the `message` keyword. (For more information use Search All in the Help system, and look up ‘message’). This method of overriding windows message handlers can be used for any message and also to handle additional messages generated by our own programs.

The message handling function is given in Listing 6, and goes in the implementation section of the CUR2A.PAS unit. The timer and mouse movement handlers also need to be amended.

Note that the `WMNCMouseMove` handler will only be called if Windows updates the mouse position while the mouse cursor happens to be on a non-client area of the form.

Icons

Looking at Delphi’s `Options|Project|Application` settings we find that Delphi supplies a default icon. This can be replaced quite simply by clicking the ‘Load Icon’ button and browsing around until we find a suitable icon. After compiling the project and running it, it will quite nicely change itself into the new icon as requested. Further, if we use Image Editor to open the resource file that Delphi maintains for the project and edit `MAINICON`, we find that the resource file actually contains the requested icon. This is the only situation where I have found Delphi giving the user any control over the default resource file for a project.

If we create a new project, we can note that the form properties in the Object Inspector list an Icon property, which is blank by default. Double-clicking on the '...' for that field brings up Picture Editor, which can be used to load another icon, in addition to the default one already discussed. This means that in a multi-form project we can have a different icon for each form.

As with the bitmaps, we can load icons from files using the method `LoadFromFile` and from resource files using the `LoadIcon` function from the Windows API. Listing 7 shows a simple example of loading from a resource file, the project is ICOPRJ1.DPR. Once again, I have included an animation. After the previous examples, this project should be clear enough. The animation is of course only visible when the project is minimised. The 'Shrink Me' button in the middle of the form minimizes the program.

If an icon name is supplied in `LoadIcon` but the icon is not found, the handle returned will be `NUL`.

```
unit Ico1;
interface
{$R ICONS.RES}
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls;
type
  TForm1 = class(TForm)
  Timer1 : TTimer;
  Button1 : TButton;
  procedure FormCreate(Sender : TObject);
  procedure Timer1Timer(Sender : TObject);
  procedure FormDestroy(Sender : TObject);
  procedure Button1Click(Sender : TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    IconNumber:integer;
    Ico : array [0..1] of TIcon;
  end;
var  Form1 : TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender : TObject);
begin
  Ico[0] := TIcon.Create;
  Ico[1] := TIcon.Create;
  Ico[0].Handle := LoadIcon(HInstance, 'Icon_1');
  Ico[1].Handle := LoadIcon(HInstance, 'Icon_2');
  Icon := Ico[0];
  IconNumber := 0;
  Timer1.Interval := 200;
end;

procedure TForm1.FormDestroy(Sender : TObject);
begin
  Ico[0].Destroy;
  Ico[1].Destroy;
end;

procedure TForm1.Timer1Timer(Sender : TObject);
begin
  IconNumber := (IconNumber+1) MOD 2;
  Icon := Ico[IconNumber];
end;

procedure TForm1.Button1Click(Sender : TObject);
begin
  Application.Minimize;
end;
end.
```

➤ *Listing 7*

Side-Stepping Delphi

At the start of this article I mentioned that Delphi maintains a resource file with the same name as each project and that attempting to manipulate the contents is generally a waste of time.

However, if you wish to alter the resources available in the default file, this can be done as long as the associated project is not active within Delphi, ie if you want to alter CURPRJ1.RES, either close the CURPRJ1 project, or open another project. Delphi will then be quite happy for you to alter the contents of the default .RES file.

A direct result of this behaviour is that it is possible to alter a project resource file without intending to, simply by forgetting that a particular resource file is the default for a project that has not been worked on for some time and adding or removing resources. If a project suddenly starts failing to compile because of duplicate resource identifiers, check all the resources included in case this has happened. The project must be re-compiled in order to make use of revised resources.

Personally, I feel that there is very little advantage in altering the default resource file. Firstly, there is too much to do, compared with just adding another file reference into either the project (.DPR) or a unit (.PAS) file. Secondly, it makes life difficult if you wish to copy a unit into another project, since you then have to manipulate the default resource file for that project as well. Thirdly, the work-around is not something that Borland have recommended. If you contact them, they specifically tell you not to try to modify the default resource file.

My real reason for including the technique here is that if like me you have accidentally replaced a default resource file in a project, you are now in a position to sort it out without deleting the whole project and starting again.

One final comment. The names of resources in the .RES file can in theory be in either upper or lower case. The feedback I have had from various people is that upper case throughout is needed to ensure that everything works correctly.

Dave Bolt hails from Barnsley in Yorkshire and can be contacted on CompuServe as 100112,522

Delphi Internals: CPU Type

Component-based CPU Type determination

by Dave Jewell

A few weeks ago, I was given the task of writing a readership survey program for a popular UK magazine. Not unnaturally, I decided to write the program in Delphi! Much of the required information was provided by getting the user to answer a series of questions, but it was also important to automatically gather some information on the hardware which the user was running. One item that was of interest to the publishers was the CPU type.

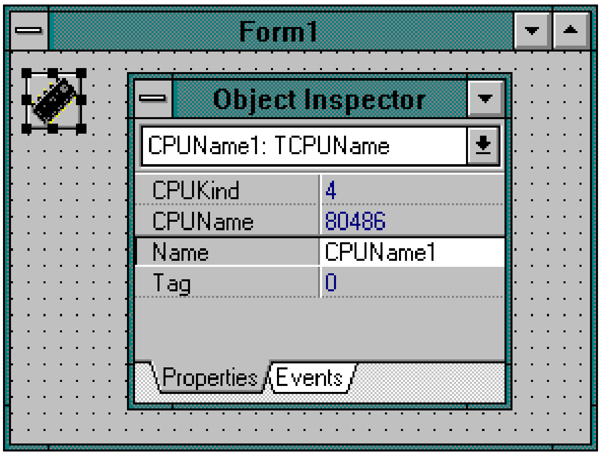
Initially, I planned to use the `GetWinFlags` API routine to determine what sort of processor was in use. However, I discovered that this routine doesn't even recognise Pentium processors let alone the Pentium Pro (P6) chip. It seemed to me that this wasn't adequate for the job so I decided to look further afield. Somewhat later, I discovered a call you could make on the built-in Windows DPMI server which will return the CPU type. This call is accessed by a `INT $31` call. Again, this looked promising, but as before there was no information on how to recognise Pentium or Pentium Pro chips.

In desperation I started trawling the bulletin boards and eventually found what I was looking for in the Delphi forum on CompuServe. I discovered some code (kindly donated by Intel) which reliably checks for all existing processor types – oh joy!

Introducing The TCPUName Component

As it stood, the code in question was rather untidy and mainly comprised a single large in-line assembler routine. I spent some time tidying this up and decided to re-package it as a component. One might as well do things properly! The screenshot above shows how the component looks from the viewpoint of the Object Inspector.

➤ Our TCPUName component in action! The corresponding Object Inspector window is also shown



CPUName1: TCPUName

CPUKind	4
CPUName	80486
Name	CPUName1
Tag	0

The component has two special properties in addition to the `Name` and `Tag` fields. Both of these properties are read-only. If you try editing them in the Object Inspector, they'll immediately snap back to whatever values they had beforehand. More on that shortly!

The first property, `CPUKind`, is a simple integer value which returns the type of CPU we're dealing with. This number will generally take one of the following values:

```
const
  i8086       = 1; {also 8088}
  i80286      = 2;
  i80386      = 3;
  i80486      = 4;
  iPentium    = 5;
  iPentiumPro = 6;
```

Although the constants defined here only go up as far as the Pentium Pro, you can reliably expect that what comes after will return a value of 7, provided that Intel continue to consistently implement the `CPUID` instruction which is used by the `TCPUName` code. `CPUID`, in case you haven't encountered it before, is a special instruction which returns the type of processor being used along with other assorted information. Unfortunately Intel didn't think of implementing the `CPUID` instruction until they got as far as the 80486 processor (amazing how obvious things seem given the benefit of hindsight!) which is why the code in Listing 1 is a good deal more complex than it would otherwise have to be.

The second property, `CPUName`, returns a plain-English description of the processor. This is for use by those applications (like my reader survey program) which simply wish to report the CPU type without necessarily doing anything else with it. The possible values which this string can take are:

```
'8086'
'80286'
'80386'
'80486'
'Pentium'
'Pentium Pro'
'Px'
```

The last item here is intended for upward compatibility with future processors. Although we can't predict Intel's marketing names for each processor, the 'Px' designation should be consistent. I've already got my order in for a quad, 2 GHz P10 motherboard...

How It Works

Listing 1 shows the component in its entirety. Two private member functions are implemented within the class definition, `GetCPUKind` and

`GetCPUName`. These methods are used to return the processor type and processor name respectively for the implementation of the two properties described above. It's likely that you'll only ever have one instance of the `TCPUName` component in any one application, but for the sake of simplicity, and to avoid re-interrogating the CPU where multiple instances might be present, I decided to store the CPU identifier using a single integer global variable. This is set up in the unit initialisation code.

You'll also notice that there are two dummy routines, `NOPInteger` and `NOPString`. These are used to provide dummy write methods for the two component properties. The recommended way of creating read-only properties (according to the Delphi on-line help) is to declare properties without an associated write clause, like this:

```
published
  { Published declarations }
  property CPUKind: Integer
   read GetCPUKind; {read-only!}
  property CPUName: String
   read GetCPUName; {read-only!}
end;
```

➤ *Listing 1*

```
unit CPUKind;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs;
type
  TCPUName = class(TComponent)
    private
      { Private declarations }
      function GetCPUKind: Integer;
      function GetCPUName: String;
      procedure NOPInteger (val: Integer);
      procedure NOPString (val: String);
    protected   { Protected declarations }
    public      { Public declarations }
    published
      { Published declarations }
      property CPUKind: Integer read GetCPUKind
        write NOPInteger; { read-only! }
      property CPUName: String read GetCPUName
        write NOPString;   { read-only! }
  end;

procedure Register;

implementation
const
  i8086       = 1;             { includes 8088 CPU as well }
  i80286      = 2;
  i80386      = 3;
  i80486      = 4;
  iPentium    = 5;             { P5 - Pentium }
  iPentiumPro = 6;             { P6 - Pentium Pro }
var
  id: Integer;

function CpuID: Integer; assembler;
{ Assembly function to get CPU type incl Pentium and later }
asm
  push        ds             { first, check for 8086 -
                               Flag bits 12-15 always set }
  call        GetWinFlags    { call Windows API }
  or          ax,wf_CPU286   { or with 80286 processor bit }
  mov         ax,i80286      { assume 286 }
  jz          @@1            { branch if it was }
  { Not a 80286 - let’s check for a 8088/8086 next }
  pushf                      { save EFLAGS }
  pop         bx             { store EFLAGS in BX }
  mov         ax,0fffh       { clear bits 12-15 }
  and         ax,bx          { in EFLAGS }
  push        ax          { store new EFLAGS value on stack }
  popf                       { replace current EFLAGS value}
  pushf                      { set new EFLAGS }
  pop         ax             { store new EFLAGS in AX }
  and         ax,0f000h  { if bits 12-15 are set, then 8086 }
  cmp         ax,0f000h      { is an 8086/8088 ? }
  mov         ax,i8086       { turn on 8086/8088 flag }
  je          @@1            { yes - all done }
  { To test for 386 or better, we need to use 32 bit
    instructions, but the 16-bit Delphi assembler does not
    recognize the 32 bit opcodes or operands.  Instead, use
    the 66H operand size prefix to change each instruction to
    its 32-bit equivalent. For 32-bit immediate operands, we
    also need to store the high word of the operand
    immediately following the instruction. The 32-bit
    instruction is shown in a comment after the 66H
    instruction. }
  db         66h             { pushfd }
  pushf
  db         66h             { pop eax }
  pop        ax              { get original EFLAGS }
  db         66h             { mov ecx, eax }
  mov        cx,ax           { save original EFLAGS }
  db         66h             { xor eax,40000h }
  xor        ax,0h           { flip AC bit in EFLAGS }
  dw         0004h
  db         66h             { push eax }
  push       ax              { save for EFLAGS }
  db         66h             { popfd }
  popf                       { copy to EFLAGS }
  db         66h             { pushfd }
  pushf                      { push EFLAGS }
  db         66h             { pop eax }
  pop        ax              { get new EFLAGS value }
  db         66h             { xor eax,ecx }
  xor        ax,cx      { can’t toggle AC bit, CPU=Intel386 }
  mov        ax,i80386       { turn on 386 flag }
  je         @@1
  { i486 DX CPU / i487 SX MCP and i486 SX CPU checking
    Checking for ability to set/clear ID flag (Bit 21) in
    EFLAGS which indicates the presence of a processor with
    the ability to use the CPUID instruction }
  db         66h             { pushfd }
  pushf                      { push original EFLAGS }
  db         66h             { pop eax }
  pop        ax              { get original EFLAGS in eax }
  db         66h             { mov ecx, eax }

  mov        cx,ax           { save original EFLAGS in ecx }
  db         66h             { xor eax,200000h }
  xor        ax,0h           { flip ID bit in EFLAGS }
  dw         0020h
  db         66h             { push eax }
  push       ax              { save for EFLAGS }
  db         66h             { popfd }
  popf                       { copy to EFLAGS }
  db         66h             { pushfd }
  pushf                      { push EFLAGS }
  db         66h             { pop eax }
  pop        ax              { get new EFLAGS value }
  db         66h             { xor eax, ecx }
  xor        ax, cx
  mov        ax,i80486       { turn on i486 flag }
  je         @@1
  { if ID bit cannot be changed, CPU=486 without CPUID
    instruction functionality }
  { Execute CPUID instruction to determine vendor, family,
    model and stepping.  The CPUID instruction used in this
    program can be used for B0 and later steppings of P5 }
  db         66h             { mov eax, 1 }
  mov        ax, 1           { set up for CPUID instruction }
  dw         0
  db         66h             { cpuid }
  db         0Fh   { Hardcoded opcode for CPUID instruction }
  db         0a2h
  db         66h             { and eax, 0F00H }
  and        ax, 0F00H       { mask everything but family }
  dw         0
  db         66h             { shr eax, 8 }
  shr        ax, 8    { shift the cpu type down to low byte }
@1:
  pop        ds
end;

procedure TCPUName.NOPInteger(val: Integer); begin end;

procedure TCPUName.NOPString(val: String); begin end;

function TCPUName.GetCPUKind: Integer;
begin
  Result := id;
end;

function TCPUName.GetCPUName: String;
begin
  case id of
    i8086:       Result := ’8086’;
    i80286:      Result := ’80286’;
    i80386:      Result := ’80386’;
    i80486:      Result := ’80486’;
    iPentium:    Result := ’Pentium’;
    iPentiumPro: Result := ’Pentium Pro’;
  else
    Result := Format (’P%d’, [id]);
  end;
end;

procedure Register;
begin
  RegisterComponents (’Pilgrim’’s Progress’, [TCPUName]);
end;

begin
  id := CpuID;          { unit initialisation }
end.
```

Unfortunately, if you try this, you'll find that not only are the properties read-only, but they are also invisible to the Object Inspector. I found this rather irritating as I wanted to see the properties in the Object Inspector window. In order to get the read-only properties to appear, it was necessary to fool Delphi into thinking that the properties were writeable, hence the need for the dummy write methods. I think this is a shortcoming. After all, it's called an Object *Inspector*, so you would imagine that it ought to be able to *inspect* read-only properties without any implication that the properties in question are writeable! Ho-hum...

The most important routine in Listing 1 is the `CpuID` function. This first calls the `GetWinFlags` API function in order to determine if a 286 is in use. Although the original Intel source included 286 detection code, this was commented out and I got the impression it caused GPF problems under Windows.

The program then tries to see if it's dealing with an 8088/8086 by checking to see if certain bits are 'stuck' in the `EFLAGS` register. If you're paying attention, you'll know that there's a zero percent chance of detecting a 8088/8086 processor while running a Delphi program since these processors aren't even capable of entering protected mode, let alone running Windows! Nevertheless, I've left the code in so that you can adapt it for use in a DOS application if you wish. The same argument applies to 286 detection. Most modern software, including Windows 95 itself, requires a minimum of a 386 processor, but again, I've left the code in for the sake of completeness.

From then onwards, the code uses 32-bit instructions to test for 386 (and higher) processors. Unfortunately, the in-line assembler built into 16-bit Delphi won't recognise anything higher than 286 instructions, so we have to manually prefix each 32-bit instruction with the value `$66`. This looks rather untidy, but the result is the same. Each `$66` op-code tells the processor to treat the next instruction as being 32-bit rather than 16. Thus, the op-code `$50`, which is interpreted as `PUSH AX`, will normally only push the contents of the 16-bit `AX` register onto the stack. However, if we precede the `$50` op-code with `$66`, then it becomes a `PUSH EAX` instruction, pushing the entire contents of the 32-bit `EAX` register.

Incidentally, 32-bit Delphi will be able to assemble 32-bit instructions directly, so if you want to port this code over you'll be able to significantly tidy it up and get rid of the 8088/8086 and 286 checks at the same time!

By the time the code has got to this point, we know that it's a 486 processor (or better). However, not all 486 processors implement the `CPUID` instruction which returns the processor family and other information. Accordingly, the code has to perform another check, testing bit 21 in the `EFLAGS` register to see if `CPUID` functionality is present. If it isn't, we know it's just a low-end 486 chip. Otherwise the code executes a `CPUID` instruction and returns the family identifier directly as the function result.

The CPUID Instruction

If you wanted to, you could easily obtain more detailed information from the call to `CPUID`. If you look at Table 3 on page 4 of the Intel PDF file on the disk (see '*What's On The Disk*'), you'll see a description of the format of the `EAX` register immediately after `CPUID` has been executed. This gives you the minor stepping number, major stepping number, family information (which is what is picked up by the `TCPUName` component) and model number. Using this information, you could easily add a model number and/or string to the `TCPUName` property list, and the same for the stepping information. This is left as an exercise for the reader, but it's clearly very straightforward.

Getting the model and stepping information might be advantageous in a hardware diagnostic program, but bear in mind that if you're seeking to detect the notorious Pentium division bug you won't get very far because Intel didn't change the processor stepping numbers when they fixed the divide instruction. Instead, you actually have to perform some division and see if it checks out OK.

After the `CPUID` instruction has executed, the `EDX` register contains a list of 'feature flags' for the processor. The format of these flags is given in the Intel PDF file, although I think the information here is a bit too esoteric to be of much general use. I can't help wondering, though, if the undocumented feature flags perhaps provide an easy way to detect the fixed division problem, amongst other things.

What's On The Disk

The disk with this issue contains the component files as well as the Intel documentation on the `CPUID` instruction, in Adobe Acrobat format as CPUAP.PDF (the Acrobat reader is on your Delphi CD).

Dave Jewell is a freelance technical journalist, computer consultant and author of *Instant Delphi* from Wrox Press. This article is based on part of his forthcoming book on Delphi component writing, which will be published in the first half of 1996. You can reach Dave by email on the internet as djewell@cix.compulink.co.uk or on CompuServe as 102354,1572

The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

Just email Brian Long, our Delphi Clinic Editor, on 76004.3437@compuserve.com or write/fax us at The Delphi Magazine

---

Radio Group Focus

Q When I try and set focus to a radio group, with the intention of focusing on the current radio button within it, the focus disappears. What seems to happen is that the radio group itself takes focus. I would explicitly set focus to one of its radio buttons, but they are represented by the `Items` property, a `TStrings` object, and so I can't find a window handle. What do I do?

A Well, when you get the group box focused, a press of the `Tab` key sets focus on the current radio button. You need a program statement that achieves the same thing. Something like this:

```
RadioGroup1.SetFocus;
SelectNext(ActiveControl,
  True, True);
```

`SelectNext` focuses on the next component on the form in the tab order. It appears the radio buttons come after the group box.

Drag And Drop On Grid Cells

Q I have a `DBGrid` that I would like to drag and drop a value onto. It works fine when I drop the value into the currently selected row; however, I would like to be able to drop it into the record the drag cursor is over when the mouse button is released. Is there any way to change the selected record in the grid based on the location of the drag cursor?

A To change the record normally, you click with the mouse. Here is one possible solution, where the `OnDragDrop` event handler simulates a mouse click at the cursor position. Rather than limit this answer to a `TDBGrid`, I have also implemented a handler for a `TStringGrid`. In my example project (DRAGDROP.DPR on the disk) there is a string grid and a database grid, which will both receive a value dragged over from a file list box (`SourceControl`). The two event handlers are shown in Listing 1.

The string grid writes a value to a cell using the two-dimensional string array property, `Cells`, using `Selection` to identify the currently selected cell. The database grid uses the `SelectedField` property to write a value to the target field: it also ensures the dataset the field belongs to is in `Edit` mode first. The `OnDragOver` method is simply:

```
Accept :=
  Source = SourceControl;
```

256 Colour TImages

Q When I set the `Stretch` property of a `TImage` which contains a 256 colour bitmap to True, its colours get mashed. Is this a VCL bug, and is there a fix?

A I have heard that this does not occur on some video drivers, and so the problem could be a `TImage` problem or a Windows video driver feature. Rather than decide who is to blame, let's see if we can fix it.

There are two approaches here, and both involve using an intermediate bitmap to copy the `TImage`'s bitmap onto. It seems that when the bitmap is copied onto a canvas directly, the palette is not fixed up, or 'realized' correctly, however if it is copied onto another bitmap, it magically is. If you are brave enough to modify the VCL source (which you may not even have, depending on what products you have purchased), you can try the first approach, otherwise you can use the substitute component presented below.

First Method. Find the EXTCTRLS.PAS file and locate the `TImage.Paint` method. Add to it a second local variable:

```
Bmp: TBitmap;
```

Now go down to the end of the method and find:

➤ *Listing 1*

```
procedure TForm1.StringGridDragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  if Source <> SourceControl then Exit;
  with Sender as TStringGrid do begin
    Perform(wm_LButtonDown, 0, MakeLong(X, Y));
    Perform(wm_LButtonUp,   0, MakeLong(X, Y));
    Cells[Selection.Left, Selection.Top] := SourceControl.FileName;
  end;
end;
procedure TForm1.DBGridDragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  if Source <> SourceControl then Exit;
  with Sender as TDBGrid do begin
    Perform(wm_LButtonDown, 0, MakeLong(X, Y));
    Perform(wm_LButtonUp,   0, MakeLong(X, Y));
    SelectedField.DataSet.Edit;
    SelectedField.AsString := SourceControl.FileName;
  end;
end;
```

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

Before these lines insert the block of code in Listing 2. If this compiles okay, copy the resultant EXTCTRLS.DCU file into your DELPHI\LIB directory (backing up the old one first) and then rebuild the component library (`Options | Rebuild Library`).

You can see that if the `Stretch` property is set, the code copies the picture to another bitmap before drawing it on the image's canvas. The use of `inherited` against a property in the original and modified code is worth exploring here, as it causes a headache when attempting to put similar code in a new component. The `TImage` is derived from a `TGraphicControl` which has a `Canvas` property, referring to the screen space where it will draw. `TImage` redefines `Canvas` to refer to the bitmap's canvas instead. When it comes to draw itself in the `Paint` method, it must use the word `inherited` to access the real canvas, declared in the `TGraphicControl` class.

Listing 2

```
if Stretch then begin
  Bmp := TBitmap.Create;
  try
    Bmp.Height := Picture.Height;
    Bmp.Width  := Picture.Width;
    Bmp.Canvas.Draw(
      0, 0, Picture.Graphic);
    inherited Canvas.StretchDraw(
      Dest, Bmp);
  finally
    Bmp.Free;
  end;
end
  else
```

Unfortunately there is no way for a class inherited from `TImage` to get access to this proper canvas, two levels up the inheritance tree, so we have to use other sneaky devices to achieve the goal.

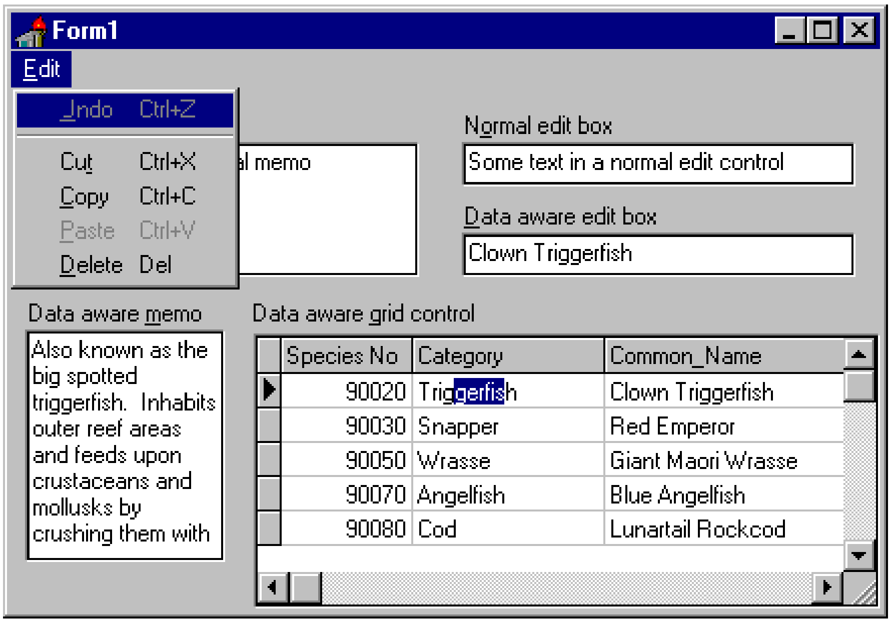
Second Method. The `TNewImage` component in Listing 3 (the file IMAGE2.PAS) traps the `wm_Paint` message (well it's not a real message: a `TImage` does not have a window handle, but let's not get too involved here) and obtains a Windows device context handle from the message parameters. It uses this to set up a temporary canvas that can be used by the new `Paint` method. Notice that to allow the entirety of the component to be seen at design time, the new code only executes if a stretched bitmap is present. In other cases, the usual surrounding dashed line will be seen.

Using this approach has a side benefit, as now things other than bitmaps can be stretched as well: try and load an icon or a metafile into a `TImage` and set the `Stretch` property to `True`. In case you have lots of `TImage` components in use that you want to replace with `TNewImage`s, but can't face the ordeal of deleting the originals, adding `TNewImage`s, setting the `Stretch` property and loading a `Picture`, here is an alternative. Load your form as a text file using `File | Open File`, and choosing `Form Files` from the `file` types combo box. Now do a search and replace of `TImage` with `TNewImage` and close the file. Finally open the form's unit file normally and do the same search and replace through the form class definition. Problem solved.

This approach comes in very handy if you start working on a `TTable` component, use the `Fields Editor` to set up all the field objects and start writing code, only to realise that you should have started with a `TQuery`. If you were to delete the `TTable`, all the field objects would also be deleted. It is much easier to change the definition of a `TTable` to be a `TQuery` and alter the differing property (ie change the `TableName` property to be an SQL property formatted appropriately – you can find what the right format is by examining an existing `TQuery` component in text mode).

Listing 3

```
unit Image2;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, ExtCtrls;
type
  TNewImage = class(TImage)
  private
    FCanvas: TCanvas;
    FBmp: TBitmap;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure WMPaint(var Msg: TWMPaint);
      message wm_Paint;
    procedure Paint; override;
  end;
procedure Register;
implementation

constructor TNewImage.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { Can't draw on the TImage canvas - that turns out to
    be the bitmap object's canvas }
  FCanvas := TCanvas.Create;
  { Temporary bitmap to cause palette realization }
  FBmp := TBitmap.Create;
end;

destructor TNewImage.Destroy;
begin
  FBmp.Free;
  FCanvas.Free;
  inherited Destroy;
end;

procedure TNewImage.WMPaint(var Msg: TWMPaint);
begin
  { Identify what the real canvas is }
  FCanvas.Handle := Msg.DC;
  { Do normal stuff, like call Paint }
  inherited;
  { Now forget about it }
  FCanvas.Handle := 0;
end;

procedure TNewImage.Paint;
begin
  { Only do new stuff if it is a stretched image }
  if (Picture.Graphic = nil) or not Stretch then
    inherited Paint
  else begin
    FBmp.Height := Picture.Height;
    FBmp.Width  := Picture.Width;
    FBmp.Canvas.Draw(0, 0, Picture.Graphic);
    FCanvas.StretchDraw(ClientRect, FBmp);
  end
end;

procedure Register;
begin
  RegisterComponents('Samples', [TNewImage]);
end;
end.
```



l memo

Normal edit box

Some text in a normal edit control

Data aware edit box

Clown Triggerfish

Data aware memo

Also known as the big spotted triggerfish. Inhabits outer reef areas and feeds upon crustaceans and mollusks by crushing them with

Data aware grid control

Species No	Category	Common_Name
90020	Triggerfish	Clown Triggerfish
90030	Snapper	Red Emperor
90050	Wrasse	Giant Maori Wrasse
90070	Angelfish	Blue Angelfish
90080	Cod	Lunartail Rockcod

Clipboard Stuff

Q How can I implement generic cut/copy/paste menubar functionality right across `TEdit`, `TDBEdit`, `TStringGrid`, `TDBGrid`, `TMemos` and `TDBMemo` controls?

A To make common code work across these components we need some functionality common to them all. This is ok for edits and memos, as they're all based at some point on `TCustomEdit`, but what about the grids?

Well, when it comes to editing on a grid the component makes use of a specialised in-place edit control called a `TInPlaceEdit`. This is also based on `TCustomEdit`, and so we need to know how to get a handle on this object, so we can call its clipboard-type functionality.

The code in Listing 4 contains two routines used in the program shown in the screenshot. When the edit menu is invoked, the `EditMenuClick` event handler is invoked to identify if there is a `TCustomEdit` derivative around to work with. If the active component is an edit or memo, then the target has been found. However if it is a grid, it is more involved. Even when a grid has an in-place editor active, the grid is still the active component as far as the form is concerned. If a `TCustomGrid` is active, the code cycles through its components until it finds a visible `TInPlaceEdit`.

When a `TCustomEdit` descendant is located it is assigned to a data field called `EditCtl` which I have added to the form's declaration. If an edit control is found, the various menu items need to be enabled or disabled, depending on the current state of both it and the clipboard; ie whether there is any selected text, if there is any text in the clipboard, etc.

All the menu items that hang off the `Edit` menu use the same event handler. To distinguish between the menu item that triggered the event, they have all had their `Tag` properties set to unique values. Providing `EditCtl` refers to a valid object, the code performs a standard edit control clipboard type of operation, such as `CutToClipBoard` or `ClearSelection`. The one exception is for the `Undo` menu item, which uses a windows message to achieve its goal instead.

Updates From Issue 4

Several readers contacted us with more elegant or efficient ways of doing a left zero fill. Jack Bakker and Niek de Ruitjer reminded us about Delphi's `Format` routine, which will do the job quite nicely:

```
Format('%.5d', [123]); {00123}
```

See the online help for 'Format Strings' for more details (the features are quite comprehensive so it should meet most needs).

Also, Alan Gregory send in a less dirty solution for updating a file listbox. Simply call its `Update` method.

Acknowledgements

Thanks to Roy Nelson of Borland for the `TDBImage` VCL fix and idea for the `TDBImage` replacement.

➤ *Listing 4*

```
procedure TForm1.EditMenuClick(Sender: TObject);
var
  Loop: Byte;
begin
  EditCtl := nil;
  if ActiveControl is TCustomEdit then
    EditCtl := ActiveControl as TCustomEdit
  else if (ActiveControl is TCustomGrid) then
    with TCustomGrid(ActiveControl) do
      { When editing in a grid, the grid is the active
        control not the in-place editor, so we need to find
        the editor in the grid. If grid owns any controls,
        cycle through them checking for editor}
      if ControlCount > 0 then
        for Loop := 0 to Pred(ControlCount) do
          if Controls[Loop] is TInPlaceEdit then
            { Editor is visible when being used }
            if Controls[Loop].Visible then begin
              EditCtl := TInPlaceEdit(Controls[Loop]);
              Break;
            end;
  if Assigned(EditCtl) then begin
    Undo1.Enabled :=
      Bool(EditCtl.Perform(em_CanUndo, 0, 0));
    Cut1.Enabled := EditCtl.SelLength > 0;
    Copy1.Enabled := Cut1.Enabled;
    Paste1.Enabled := ClipBoard.AsText <> '';
    Delete1.Enabled := EditCtl.SelLength > 0;
  end else begin
    Undo1.Enabled := False;
    Cut1.Enabled := False;
    Copy1.Enabled := False;
    Paste1.Enabled := False;
    Delete1.Enabled := False;
  end;
end;

procedure TForm1.MenuClick(Sender: TObject);
begin
  if Assigned(EditCtl) then with EditCtl do
    case (Sender as TComponent).Tag of
      1: Perform(em_Undo, 0, 0);
      2: CutToClipBoard;
      3: CopyToClipBoard;
      4: PasteFromClipBoard;
      5: ClearSelection;
    end;
end;
```

Inside TApplication

by Nick Hodges

Most of the components used in building Delphi applications can be clearly seen on the Component Palette and manipulated with the Object Inspector at design time. A click on the palette and a click on the form, and any component on the palette is ready for use. However, the most important component to any Delphi application is not on the Component Palette, nor will its properties be found in the Object Inspector. `TApplication` is the foundation for all Delphi VCL based projects. It contains the lowest level of code needed to run a Windows application, creating the ever-patient message loop and handling all the low level calls to the Windows API that create and run an application. Like a Secret Service agent, `TApplication` is there, not quite noticed, but very capable and ready to serve.

Strangely enough, `TApplication` is actually a component, descending directly from `TComponent`. `TApplication` itself is declared in the `Forms` unit of the runtime library. The instance of `TApplication` that is declared for all Delphi projects, `Application`, is actually a Window, created directly with a call to the API `CreateWindow`. It is intialized with zero height and zero width, so it never actually appears on the screen. `Application` knows how to create and manage the main form of a Delphi project at runtime. `TApplication` has properties and events just like any other component. The best part is that a number of these events and properties contain valuable information for the Delphi programmer. That information is not readily apparent, but easily surfaced.

Application.ProcessMessages

Frequently, an application will have to perform a task that takes a rather large chunk of processor time. Often, this involves some sort of loop. Because Windows 3.x multi-tasks cooperatively, a well-behaved Windows application has to allow other applications a shot at processing their messages. `Application` provides a simple way to allow messages to be processed while a project is busy doing some other menial task. A call to `Application.ProcessMessages` anywhere in your code will ensure that your application will give other Windows programs space to do their thing. Periodic calls inside a loop will allow all applications to process messages that would otherwise be bottled up.

The sample application (included on the disk and shown in action over the page) demonstrates how this works. When the *Waste Time* check box is selected, the demo continuously counts up and down from 0 to 100 and displays the status in a gauge (see Listing 1). The `repeat...until` loop would normally seize control of the Windows environment, not allowing any other applications access to the message queue. However, a simple call to `Application.ProcessMessages` in the middle of the loop causes the demo to peek into the message queue and process any messages waiting there. As a result, Windows can function normally despite a loop running continuously in the background.

However, note that `Application.ProcessMessages` will not close an application when the `wm_quit` message is encountered inside a loop. Therefore, the loop itself includes a check `Application.Terminated`. This ensures that `Application.ProcessMessages` actually processes all the waiting messages for the application before terminating the application. `Application.ProcessMessages` actually sets `Terminated` to true, but the programmer must explicitly check for it to allow it to be processed. To see this work, try commenting out the call in the `until` clause, run the program and notice that the program won't close until the *Waste Time* check box is deselected.

You can call `Application.ProcessMessages` anywhere at any time, but it is best used when any action a program takes might interfere with the free flow of Windows messages. Interestingly, the code is the same as that invoked by `TApplication` when it sets up the message loop and waits for user input in any Delphi program.

Starting Out Minimized

Employing a zero-sized window to run a Delphi application and to manage all of its associated forms causes the application to behave slightly differently to what might normally be expected.

Despite how it may appear to a developer within Delphi itself, the real main window of any Delphi application is the `TApplication` window itself. It is this window which is displayed when the application is minimized and it is this window which is queried by Windows

➤ *Listing 1*

```
procedure TForm1.CheckBox3Click(Sender: TObject);
var Increment: Integer;
begin
  Increment := 1;
  repeat {Waste time, but allow processing of Windows messages}
    Gauge1.Progress := Gauge1.Progress + Increment;
    if Gauge1.Progress = Gauge1.MaxValue then Increment := -1;
    if Gauge1.Progress = Gauge1.MinValue then Increment := 1;
    Application.ProcessMessages;
  until (not CheckBox3.Checked) or (Application.Terminated);
  Gauge1.Progress := 0;
end;
```

shells such as Program Manager when seeking an icon.

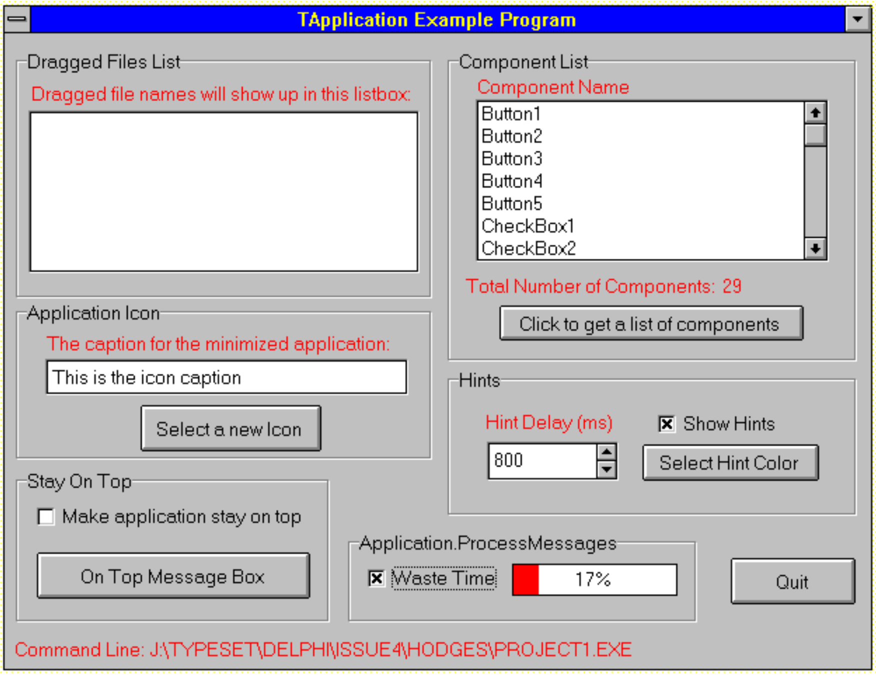
One of the easiest ways to show this slightly unusual trait is to create a simple Delphi application, install it in a group in Program Manager and then tell Program Manager to run the application minimized. The Delphi-built application will ignore the command when run from Program Manager. Since `Application` is really the main form of the application, and it creates and displays what the developer calls the main window of the application, the message never gets to the application to start in a minimized state. `TApplication` doesn't process the `CmdShow` parameter which defines how the program will be displayed on startup.

Fortunately, there is an easy fix to this seemingly anomalous behavior. The demo application, if started with the Run Minimized command set in Program Manager, will behave as expected. In the main form of the demo program, the `FormCreate` method checks the `CmdShow` value that was passed to `TApplication` and stored in the `CmdShow` variable in the `System` unit (see Listing 2). The `FormCreate` constructor checks the value and sets the `WindowState` accordingly. A call to the `ShowWindow` API would do the same thing, but wouldn't necessarily set the proper `WindowState` value for the main form.

Icon And Icon Caption

Some developers may notice that the once the application is properly minimized when called from Program Manager, the icon that is displayed in Program Manager is not the one attached to the main form's `Icon` property.

This is another symptom of the distinction between `TApplication` and the main form. The icon that is bound into the executable and found by Program Manager is the icon attached to the application itself. This icon can be set through Delphi's IDE on the `Options|Project|Application` page. You can also change the application's main icon at runtime with a simple assignment statement.



Dragged file names will show up in this listbox:

Total Number of Components: 29

The caption for the minimized application: This is the icon caption

Hint Delay (ms) 800

☒ Show Hints

☐ Make application stay on top

☒ Waste Time 17%

Command Line: J:\TYPESET\DELPHI\ISSUE4\HODGES\PROJECT1.EXE

➤ *The example program in action*

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  {Ensure Window opens itself in state set by CmdShow}
  case CmdShow of
    sw_ShowMinimized,
    sw_ShowMinNoActive     : WindowState := wsMinimized;
    sw_ShowMaximized       : WindowState := wsMaximized
  else
    WindowState := wsNormal
  end; { case }
  { ... more code here, see files on the disk ... }
end;
```

➤ *Listing 2*

The solution to this dilemma is to do one of two things: either ensure that the `TApplication` icon and the main form icon are the same, or leave the `Icon` property of the main form blank and let `TApplication` do all the icon management.

It is also easy to assume that the main form's caption will become the caption for the icon, but such is not the case. The `Title` property of `TApplication` holds a string that will be displayed as the minimized application's caption. The default can be set in the project's `Option` dialog, and can be easily changed at run-time.

The ever-present demo demonstrates the use of icons and their captions. Note that if placed in Program Manager, the demo will display the Delphi default icon. When run and minimized, that same icon will be displayed. However, `TApplication` has an `Icon` property that can be set at design time. The demo allows you to do that – see Listing 3. Assigning an `Icon` to the main form's `Icon` property at design time would cause that icon to be displayed on minimization, but not as the icon representing the application in Program Manager. Note, too, that the icon assigned at runtime is only temporary, and that the icon assigned to `TApplication` at design time is the one bound into the program at compile time as its main icon.

Finally, the caption can be easily changed by entering a string into the supplied edit box. That string is then assigned to `Application.Title`.

Dragged Files

The fact that the icon shown on minimization is not the icon representing the project's main form brings about more unusual, but fixable, behavior in Delphi applications. Because the icon displayed when the program is minimized is owned by the application and not by the main form, dragging files from File Manager to the iconized application does not function as expected. You can cause an application's main form to accept dragged files as usual by calling the `DragAcceptFiles` API and responding to the `wm_DropFiles` message to gather information about those files. Once this is done, Delphi applications that are in the restored state will accept these files gladly; however, when minimized they will not.

`TApplication` has to be set up to accept files as well. `TApplication` has an event called `OnMessage` that is invoked every time a message is received by the application. By calling `DragAcceptFiles` and passing `Application.Handle`, and by writing a special handler to catch the `wm_dropfiles` message inside the `OnMessage`, a minimized application can respond to files dragged to it in exactly the same way as does a restored program. Note that the `OnMessage` event could be used to trap any Windows message that might need special handling by the `Application` instance, such as `wm_paint` for painting on the icon.

The demo application illustrates how a Delphi application can be set up to accept files in any state. Both `TApplication` and the program's main form are able to accept dragged files, and both respond to the `wm_dropfiles` message by gathering the names of all the dragged files in a `TStringList` and then placing that list into a listbox on the form.

Hints

Windows applications these days aren't considered complete without fly-by help boxes for buttons, tool bars and other components. Delphi makes it incredibly easy to supply these little hint boxes. The `TApplication` object makes it very easy to customize them. `TApplication` supplies properties to change the time a user waits to see the hints, whether the hints are displayed at all, and even the background color of the hints themselves, in case a programmer wants to be different and not display hints with the standard yellow background.

These features can be easily seen in the demo application. The hints can be turned on and off using the so-named check box. The hint delay time, in milliseconds, can be set using the spin edit box. Note that the delay is set only for the first hint, once the first hint is shown all hints after that are immediately displayed without delay. This allows users to see all the hints without having to wait for the delay for each control. Moving the mouse off the main window resets the delay. The background color of the hints can be changed with a simple call to a `ColorDialog` box. The selected color is then set to the `TApplication` `HintColor` property.

```
procedure TForm1.Button1Click(Sender: TObject);
var Icon: TIcon;
begin
  {Get an icon and load it into the Application.  The new icon will now
   show up when the application is minimized}
  Icon := TIcon.Create;
  if OpenDialog1.Execute then begin
    Icon.LoadFromFile(OpenDialog1.Filename);
    Application.Icon := Icon;
  end;
  Icon.Free;
end;
```

➤ *Listing 3*

```
procedure TForm1.Button5Click(Sender: TObject);
begin
  Application.NormalizeTopMosts; {Allow dialogs on top}
  MessageBox(Form1.Handle, 'This should be on top.', 'Message Box', MB_OK);
  Application.RestoreTopMosts; {Return to normal}
end;
```

➤ *Listing 4*

Stay-On-Top

Some Delphi developers may want to create an application that always remains on top of all the other windows on the screen.

Interestingly, `fsStayOnTop` is the only `FormStyle` property setting that can be changed at runtime. However, when in this state, problems can arise when the application tries to call another dialog. Dialogs called by programs in stay-on-top mode can end up *behind* the calling window. If such a dialog is modal, it can lock up Windows entirely! `TApplication` allows you to place dialogs on top of forms that have `fsStayOnTop` set. The two methods `NormalizeTopMosts` and `RestoreTopMosts` allow a programmer to toggle in and out of a state that allows dialogs to be place on top of a stay-on-top application (see Listing 4).

The trusty demo application can be switched to stay-on-top mode and then can calls a message box which is displayed on top of the form. Without the calls altering the topmost state, the message box would be placed behind the app and out of reach, causing Windows to become modal with no escape. Even worse, users wouldn't even know what had happened! To demonstrate this, try setting the stay on top checkbox and then try to change the hint color!

Listing Components

Frequently, a Delphi developer may wish to gain access to a particular type of control or a certain set of controls on a form at run time. `TApplication` contains a list of all the components owned by the main form in its `Components` property. `ComponentCount` contains the total number of controls in the array. By using run-time typing information and a `for` loop, a programmer can cycle through all of the main form's components and find components that are of a certain type or that have certain properties. The demo shows this by gathering all of the names of the components on a form and putting them in a list box (see Listing 5). It also picks out all the `TLabel` controls and alternates their font color between red and black. You can use this technique to find any specific control or type of control.

The Command Line

`TApplication` also stores details at run time about the command line used to run itself. The `EXEName` property stores the full path of the executable, which can be broken down using functions from `SysUtils`, such as `ExtractFilePath` and `ExtractFileName`. The demo displays its command line in a label upon execution.

Conclusion

`TApplication` can perform a number of other tricks, including restoring and minimizing itself as well as making it easy for to invoke a program's help file. Despite being hidden away, `TApplication` has a wealth of capabilities. Knowing a few tricks we can take advantage of the strengths of `TApplication` and overcome its few quirks.

Nick Hodges, an experienced Delphi and Pascal developer, is known to many as the author of TSmiley and the inspiration behind a whole raft of SmileyWare! He can be contacted via CompuServe on 71563,2250

➤ *Listing 5*

```
procedure TForm1.Button2Click(Sender: TObject);
var I: Integer;
begin
  ListBox2.Clear;
  for I := 0 to ComponentCount - 1 do begin
    Listbox2.Items.Add(Components[I].Name);
    if Components[I] is TLabel then begin
      {Use Run-time typing to check type. Toggle the text of only TLabels
       between Red and Black. Note that each component must be typecast
       first.}
      if TLabel(Components[I]).Font.Color = clBlack then
        TLabel(Components[I]).Font.Color := clRed
      else
        TLabel(Components[I]).Font.Color := clBlack;
    end
  end;
  Label7.Caption := IntToStr(ComponentCount);
end;
```

Several readers responded to the query in Issue 3's Delphi Clinic about how to make an iconised Delphi application stay on top of all other windows – thanks! Here Hallvard Vassbotn provides a usefully generalised solution, as well as revealing other useful facets of `TApplication`.

Icon On Top

For a Delphi application, even if the main form's `FormStyle` property is set to `fsStayOnTop`, the icon which shows when minimized is not on top of the other windows. The reason for this is that the `Application` object maintains a hidden window that is the actual main window of the application. This hidden window will distribute commands to what Delphi considers the main form as it sees fit. When the Delphi main form is minimized, it will actually be *hidden* and the `Application` window is responsible for drawing the main form's icon.

With that in mind, and remembering that the `Application` window's handle can be accessed with it's `Handle` property, we can solve the problem using the code in Listing 6.

We simply hook the `OnMinimize` event of the `Application` object. Whenever the application is minimized, and thus the icon is showed, the code in `AppMinimize` will be run. Here we check if the `FormStyle` property of the main form indicates that the icon should be made on top. If so we use the `WinProcs` routine called `SetWindowPos` to change the display attributes of the icon.

Tile And Cascade

You might have noticed that when running an application created with Delphi it doesn't respond properly to the Tile and Cascade commands from the Task Manager. Delphi itself has this behaviour (it was, after all, written in Delphi!).

You can test this by running a Delphi app together with one or more non-Delphi apps. Bring up the Task Manager by double-clicking on the background or pressing Ctrl+Esc. Click the Tile and Cascade buttons. All non-Delphi applications are resized and positioned correctly. The Delphi app doesn't move, but instead an empty square is left where the window should have been placed.

This is another effect of the fact that the `Application` object in Delphi maintains its own hidden window which is the actual main window in Windows terms. The blank space you see when tiling is actually this hidden window.

To overcome this problem, we can use a little known feature of the `Application` object, the method `HookMainWindow`, which lets us hook into the message handler of the `Application` window. This way we can monitor and override any functionality of the main window.

By using the WinSight utility provided with Delphi, I found that monitoring `WM_WindowPosChanging` messages sent by Windows to the `Application` window would let me resize the main form correctly when tiling and cascading. The solution is shown in Listing 7.

First we hook the message handler of the application window with the `HookMainWindow` method. Note that `Application` keeps track of a list of hooks, so that there might be several hooks installed at once. When the main form is destroyed we act politely and clean up after ourselves by calling `UnHookMainWindow`.

The `HookProc` method will now be called for every message that arrives in the `Application` window's message queue. We are only interested in monitoring the messages, not overriding the default behaviour, so we always return false.

If it is a `WM_WindowPosChanging` message, we are interested in it and type-cast the message record to the `TWMWindowPosMsg` defined in `Messages`. The `WindowPos` field is a pointer to a record that contains all the useful information, so we de-reference this pointer as well. Now to be on the safe side we check that the message was indeed intended for the `Application` window, that we are not an icon and that the size of the window is not zero.

If all is well so far, we know that we should resize the main form. To keep things unobstructed, we fiddle with the flag bits to make sure that the Z-order is not affected and that the size of a fixed-size window isn't changed.

Now Tile and Cascade from the Task Manager should work the way they are supposed to do. This code example also shows how to monitor and/or override the application window's behaviour – again demonstrating Delphi's power and extensibility!

Hallvard Vassbotn lives and works in Norway and can be reached by email at hallvard@falcon.no

➤ Listing 6

```
unit Unit1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private   { Private declarations }
    procedure AppMinimize(Sender: TObject);
  public    { Public declarations }
  end;
var
  Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMinimize := AppMinimize;
end;

procedure TForm1.AppMinimize(Sender: TObject);
begin
  if FormStyle = fsStayOnTop then
    SetWindowPos(Application.Handle, HWnd_TopMost, 0, 0, 0, 0,
                 SWP_NoActivate or SWP_NoSize or SWP_NoMove);
end;
end.
```

➤ Listing 7

```
unit Unit2;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private   { Private declarations }
    function HookProc(var Message: TMessage): boolean;
  public    { Public declarations }
  end;
var
  Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.HookMainWindow(HookProc);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  Application.UnHookMainWindow(HookProc);
end;

function TForm1.HookProc(var Message: TMessage): boolean;
var
  LocalFlags: word;
begin
  Result := false;
  if Message.Msg = WM_WindowPosChanging then begin
    with TWMWindowPosMsg(Message).WindowPos^ do begin
      if (hWnd = Application.Handle)
      and not IsIconic(hWnd)
      and (cx > 0) and (cy > 0) then begin
        LocalFlags := flags or SWP_NoZOrder;
        if BorderStyle = bsSizeable then
          LocalFlags := LocalFlags and not SWP_NoSize
        else
          LocalFlags := LocalFlags or SWP_NoSize;
        SetWindowPos(Self.Handle, 0, x, y, cx, cy, LocalFlags);
      end;
    end;
  end;
end;
end.
```

Surviving Client/Server:
Getting Started With SQL Part 1

by Steve Troxell

Delphi incorporates some very useful database components that make developing database applications a breeze. And because Delphi ships with a local version of Borland's InterBase database server, a great number of application developers now have an easy, inexpensive means of exploring the world of Client/Server technology right at their fingertips. For these reasons, many fledgling Delphi programmers may be introduced to Structured Query Language (SQL) for the first time. This article is intended to introduce you to the principal SQL data access statements: `SELECT`, `INSERT`, `UPDATE` and `DELETE`. These are the workhorses of SQL and by the end of this article you'll be able to effectively use SQL to accomplish a wide variety of tasks. We'll continue to expand our knowledge of SQL in the next issue.

All of the examples in this article use the sample InterBase database EMPLOYEE.GDB that ships with Delphi. You may find it helpful to connect to this database through the Windows Interactive SQL (ISQL) program that ships with Delphi and try the examples as you read about them. It's also handy to be able to experiment as ideas come to you while reading the text. This database is located in the \IBLOCAL\EXAMPLES directory if you installed Delphi with the default directories. We will be adding information in this database so you may want to make a copy of the EMPLOYEE.GDB file and work with the copy only.

Using ISQL

To use Windows ISQL, start the ISQL program from the Delphi program group. From the `File` menu, select `Connect to Database`. In the `Database Connect` dialog box, make sure you've selected `Local Server`, enter the path and filename for the EMPLOYEE.GDB database, enter `SYSDBA` for the user name, and enter `masterkey` for the password (make sure you enter the password in lower case). This is the default system administrator login for InterBase databases.

Using the ISQL program is simple: type in the SQL statement you want to execute in the `SQL Statement` window and click the `Run` button to execute the statement. The results of your statement will appear in the `ISQL Output` window. Once you run an SQL statement, it disappears from the `SQL Statement` window. If you want to run it again (and perhaps make small changes to it), you can retrieve any previous SQL statement by clicking the `Previous` button.

SQL Preliminaries

Before we get started, let's look at a few of the ground rules for working with SQL. First, three new terms: the familiar structures of file, record, and field are called table, row, and column in relational databases. A database is a collection of tables; in Paradox each .DB file represents a table, in InterBase each .GDB file represents a database and the tables are managed internally.

Second, let's take a look at the syntax of a SQL statement. A typical SQL statement might be:

```
SELECT name, address
  FROM customers;
```

SQL itself is case-insensitive, but in this article SQL keywords are shown in all uppercase and table names, column names, etc. are shown in all lower case. SQL generally requires a semi-colon at the end of each statement, but in certain tools (such as ISQL) it's optional.

Third, you can break an SQL statement across multiple lines by pressing `RETURN` anywhere you can legally place a space in the statement.

Finally, you can enclose literal strings with single quotes or double quotes. We'll use single quotes here.

Introducing The Column...

Delphi users seem to have settled down into several groups: firstly occasional programmers or first-time programmers (attracted by Delphi's ease of use), secondly professional full-time developers doing general Windows application building (enthralled by Delphi's amazing productivity) and thirdly those putting together Client/Server systems (impressed by Delphi's robustness and power). This column is aimed at the third group, but especially those who may be dipping a toe into the waters of Client/Server for the first time.

Steve is involved in developing a variety of Client/Server systems in his work at TurboPower Software, using different server databases, and we are looking forward to learning from his experience over the months. As well as SQL – an essential part of the Client/Server developer's skill set – Steve plans to cover a variety of other topics and also provide lots of hints and helps along the way. If there are things which you need some help with, why not drop us a line and let us know!

Reading Rows

SELECT is SQL's data retrieval statement and is probably the most frequently used SQL statement. The basic form of the SELECT statement is:

```
SELECT <column(s)>
  FROM <table(s)>;
```

For example, try the following SELECT statement in ISQL (the results are shown in Figure 1):

```
SELECT last_name, first_name
  FROM employee;
```

We selected the `last_name` and `first_name` columns from the `employee` table. By default, SELECT returns all the rows from the table, but only shows data for the columns we requested (called the select list). If you wanted to see all of the columns in a table, it would be cumbersome to enter the name of every column in the SELECT statement, so SQL provides a convenient shortcut: an asterisk can be used to indicate *all columns*. Try the following in ISQL:

```
SELECT * FROM employee;
```

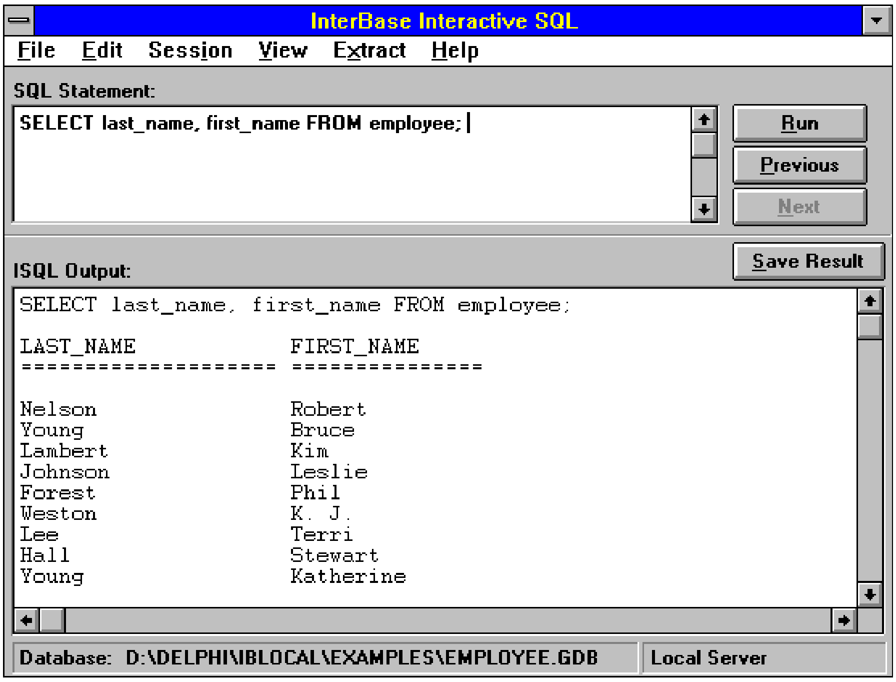
Computed Columns

You can also show calculated results with a SELECT statement. In this case, you simply provide the expression used to make the calculation in place of a column name in the select list. The example below estimates the monthly earnings of each employee:

```
SELECT last_name, salary / 12
  FROM employee;
```

Take a look at the results of this statement in Figure 2. Notice that there isn't a column name for the salary calculation. That's because the data shown doesn't exist as a named column in the table we selected from. A column without a name is just not a very useful thing to have, so we need to assign an alias to the column (see Figure 3):

```
SELECT last_name, salary / 12
  AS monthly_salary
  FROM employee;
```



SELECT last_name, first_name FROM employee;

```
SELECT last_name, first_name FROM employee;

LAST_NAME            FIRST_NAME
==================== ===============

Nelson               Robert
Young                Bruce
Lambert              Kim
Johnson              Leslie
Forest               Phil
Weston               K. J.
Lee                  Terri
Hall                 Stewart
Young                Katherine
```

Database: D:\DELPHI\IBLOCAL\EXAMPLES\EMPLOYEE.GDB

➤ *ISQL in action*

```
SELECT last_name, first_name FROM employee;

LAST_NAME            FIRST_NAME
==================== ===============

Nelson               Robert
Young                Bruce
Lambert              Kim
Johnson              Leslie
Forest               Phil
Weston               K. J.
```

➤ *Figure 1 (partial listing of rows)*

```
SELECT last_name, salary / 12 FROM employee;

LAST_NAME
==================== ======================

Nelson                                 8825
Young                                  8125
Lambert                              8562.5
Johnson                             5386.25
Forest                                 6255
Weston                          7191.078125
```

➤ *Figure 2 (partial listing of rows)*

There's no reason why we couldn't do the same thing to a regular named column as well, if we wanted to reference the data by something other than its defined column name. You might need to do this if you were selecting data from two or more tables that use the same column name.

Selecting Subsets Of Rows

The selects we've looked at so far return all the rows in the table. Many times you're not interested in wading through all of the rows to get the information you want. Usually all you want is a specific row, or a set of rows with something in common. You use the WHERE clause to restrict the rows returned by SELECT.

For example, the following SQL statement selects all the employees in the Software Development department (see Figure 4):

```
SELECT * FROM employee
  WHERE dept_no = 621;
```

The criteria defined by a `WHERE` clause is usually referred to as the *search condition*. The `SELECT` statement returns only those rows that meet the search condition. `WHERE` supports a variety of conditional operators and allows multiple criteria to be logically combined using `AND` and `OR`. Some of the common operations that can be performed are listed in Figure 5. Many SQL databases offer additional operations.

The following examples illustrate some of the expressions you can use to formulate search conditions. They are all valid search conditions for the `employee` table, so feel free to try them out in ISQL before continuing (try `SELECT * FROM employee` using each of the following `WHERE` clauses).

```
WHERE job_grade IN (1,3,4) AND
  job_country = 'USA'

WHERE salary / 12 > 10000

WHERE phone_ext IS NULL

WHERE hire_date BETWEEN
  '1-1-90' AND '12-31-90'

WHERE UPPER(first_name)
  LIKE 'ROBERT%'
```

It should be noted that the columns used in the `WHERE` clause do not have to be part of the select list; they only have to be available in the table defined in the `FROM` clause of the `SELECT` statement.

Wildcarding

You can use wildcards to select on a character column matching a given pattern (amazingly enough, this is sometimes referred to as *pattern matching*).

SQL supports two wildcard characters: `%` is used to match any number of characters (including zero), and `_` is used to match exactly one character. You must use the `LIKE` operator to use wildcards in a character search, otherwise the wildcards are taken literally.

```
SELECT last_name, salary / 12 AS monthly_salary
  FROM employee;

LAST_NAME                    MONTHLY_SALARY
==================== ======================

Nelson                                 8825
Young                                  8125
Lambert                              8562.5
Johnson                             5386.25
Forest                                 6255
Weston                          7191.078125
```

➤ *Figure 3 (partial listing of rows)*

```
SELECT * FROM employee WHERE dept_no = 621;

EMP_NO FIRST_NAME      LAST_NAME            PHONE_EXT   HIRE_DATE
====== =============== ==================== ========= ===========

     4 Bruce           Young                233       28-DEC-1988
    45 Ashok           Ramanathan           209        1-AUG-1991
    83 Dana            Bishop               290        1-JUN-1992
   138 T.J.            Green                218        1-NOV-1993
```

➤ *Figure 4 (partial listing of columns)*

Operator	**Meaning**
=	Equal to
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
BETWEEN x AND y	Range: greater than or equal to <x> and less than or equal to <y>
IS NULL	Contains null value
IS NOT NULL	Contains non-null value
IN (x,y,...)	Value found in a list
NOT IN (x,y,...)	Value not found in a list
LIKE	Matches a wildcard pattern

➤ *Figure 5 Common SQL operators*

Suppose you needed to look up a customer and all you could remember was that they were located on Newbury Street or Avenue or Newbury something. You could use the following SQL statement:

```
SELECT customer, address_line1
  FROM customer
  WHERE address_line1
  LIKE '%Newbury%';
```

As Figure 6 shows, this select finds all rows containing the word `Newbury` anywhere in the column `address_line1`, regardless of what characters (if any) preceded or followed the word.

Sorting Rows

You can sort the rows returned by a `SELECT` statement by including an `ORDER BY` clause. `ORDER BY` simply names the columns you want to sort on. Suppose you wanted a list of sales orders sorted by the amount of the order (see Figure 7 for the output):

```
SELECT cust_no, order_status,
  total_value
  FROM sales
  ORDER BY total_value;
```

Like WHERE, the columns defined in ORDER BY do not have to appear in the select list. This is true for InterBase but may not apply to other SQL databases.

You can define the sort sequence for each sort column by adding the ASC (ascending) or DESC (descending) keyword after the column name in the ORDER BY clause. Here's how you would make the query above return a list of sales orders in order of decreasing amount:

```
SELECT cust_no, order_status,
  total_value
  FROM sales
  ORDER BY total_value DESC;
```

As an alternative to specifying the name of the column to sort on, you can specify the number of the column from the select list. This is useful if you've included an unaliased computed column in the select list. For example, suppose you wanted a list of employees in order by monthly salary (see Figure 8):

```
SELECT full_name, salary / 12
  FROM employee
  ORDER BY 2;
```

Here the 2 in the ORDER BY means "order by the second column in the select list"; that is, by the computed column salary / 12.

If you use both a WHERE clause and an ORDER BY clause in the same select statement, the WHERE clause must appear before the ORDER BY clause, as in the following example:

```
SELECT * FROM employee
  WHERE dept_no = 621
  ORDER BY phone_ext;
```

```
SELECT customer, address_line1 FROM customer
  WHERE address_line1 LIKE '%Newbury%';

CUSTOMER                  ADDRESS_LINE1
========================= ==============================

Buttle, Griffith and Co.  2300 Newbury Street
```

➤ *Figure 6*

```
SELECT cust_no, order_status, total_value
  FROM sales
  ORDER BY total_value;

    CUST_NO ORDER_STATUS TOTAL_VALUE
=========== ============ ===========

       1003 waiting             0.00
       1001 shipped             0.00
       1006 shipped            47.50
       1014 shipped           100.02
       1010 shipped           210.00
```

➤ *Figure 7 (partial listing of rows)*

```
SELECT full_name, salary / 12 FROM employee ORDER BY 2;

FULL_NAME
===================================== ======================

Bennet, Ann                                          1911.25
Brown, Kelly                                            2250
O'Brien, Sue Anne                                    2606.25
Guckenheimer, Mark                          2666.66666666667
Reeves, Roger                                      2801.71875
```

➤ *Figure 8 (partial listing of rows)*

Selecting From Multiple Tables (Joins)

One of the most powerful features of the SELECT statement (and of SQL itself) is the ease with which you can combine data from multiple tables into one informative view.

For example, suppose you want a roster of all employees by department. Employee names are stored in the employee table and department names are stored in the department table, so you'll have to combine this information somehow into a single report. This is where the concept of a "join" comes into the picture.

A join can occur between two or more tables where each pair of tables can be linked by a common field.

For example, departments are identified by the dept_no column in both the employee table and department table, so this column can be used to link the two tables in a join. In SQL you can use the WHERE clause to specify the link field for a join between two tables. The select statement below produces the employee roster we want:

```
SELECT full_name, department
  FROM employee, department
  WHERE employee.dept_no =
    department.dept_no
  ORDER BY department;
```

This means "show the selected columns from the given tables and combine the rows such that dept_no in the employee table matches dept_no in the department table" (in this case the department table also happens to contain a column called department). Take a look at the results shown in Figure 9. The result of a join select is indistinguishable from a single table select.

The WHERE clause defines the association between the two tables. The linking columns are not required to have the same name, but they must be compatible data types. As an alternative to the WHERE clause, you can also define a join in the FROM clause as follows:

```
SELECT full_name, department
  FROM employee JOIN department
    ON employee.dept_no =
      department.dept_no
  ORDER BY department;
```

You can join more than two tables by simply ANDing the join expressions together in the `WHERE` clause (or concatenating `JOIN`s in the `FROM` clause). The linking columns do not have to be the same for all tables in the join. For example, to get a list of all employees assigned to a project and the name of the projects they are assigned to, you must join the `employee`, `employee_project`, and `project` tables:

```
SELECT full_name, proj_id,
  proj_name
  FROM employee,
    employee_project, project
  WHERE employee.emp_no =
    employee_project.emp_no
  AND
   employee_project.proj_id =
   project.proj_id
  ORDER BY full_name;
```

You can see the results in Figure 10.

One improvement we could make is to reduce the bulk of this statement a little by assigning aliases to the tables just as we assigned aliases to columns previously. We do this in the same fashion by following the actual table name with its alias, however we do not use the `AS` keyword in between. We're going to redefine the `employee`, `employee_project` and `project` tables to have the aliases `a`, `b` and `c` respectively. So our final select statement now looks like:

```
SELECT full_name, proj_id,
  proj_name
  FROM employee a,
   employee_project b,
   project c
  WHERE a.emp_no = b.emp_no
  AND b.proj_id = c.proj_id
  ORDER BY full_name;
```

Summing Up SELECT

`SELECT` is where most of the power of SQL lies. There are even more clauses and functionality to `SELECT` than were covered here, so check your manual. This was meant just to show you the basic nuts-and-bolts needed to do anything really useful with `SELECT`.

In the next issue we'll cover a lot more on `SELECT`, but now that we've got a handle on looking at the data, we'll turn to altering the data.

```
SELECT full_name, department FROM employee, department
  WHERE employee.dept_no = department.dept_no
  ORDER BY department;

FULL_NAME                             DEPARTMENT
===================================== =========================

O'Brien, Sue Anne                     Consumer Electronics Div.
Cook, Kevin                           Consumer Electronics Div.
Lee, Terri                            Corporate Headquarters
Bender, Oliver H.                     Corporate Headquarters
Williams, Randy                       Customer Services
Montgomery, John                      Customer Services
```

➤ *Figure 9 (partial listing of rows)*

```
SELECT full_name, proj_id, proj_name
  FROM employee, employee_project, project
  WHERE employee.emp_no = employee_project.emp_no AND
    employee_project.proj_id = project.proj_id
  ORDER BY full_name;

FULL_NAME                             PROJ_ID PROJ_NAME
===================================== ======= ====================

Baldwin, Janet                        MKTPR   Marketing project 3
Bender, Oliver H.                     MKTPR   Marketing project 3
Bishop, Dana                          VBASE   Video Database
Burbank, Jennifer M.                  VBASE   Video Database
Burbank, Jennifer M.                  MAPDB   MapBrowser port
Fisher, Pete                          GUIDE   AutoMap
Fisher, Pete                          DGPII   DigiPizza
```

➤ *Figure 10 (partial listing of rows)*

Committing Your Work

One of the key characteristics of SQL is that when you add or modify data in the SQL table, the changes are not permanently recorded in the table and other users of the database will not see them, until you commit them. In this way you can work with the data as much as you like until you get it just the way you want it and then commit it permanently to the database.

In ISQL you commit your changes by selecting `Commit Work` from the `File` menu. If you want to undo your modifications, you can select `Rollback Work` from the `File` menu. This will rollback all the changes you've made since the last time you committed your work. Think of this as a refresh of the data you're working with.

Be careful if you decide to experiment with these data modification statements outside the examples given. The tutorial database provided with InterBase defines some data validation and referential integrity constraints that may give you errors if you don't modify the data just right (this is yet another boon of SQL by helping preserve data integrity through automatic means).

Adding New Rows

We use the `INSERT` statement to add a new row to a table. `INSERT` expects us to enumerate the values of each column in the table for the new row.

For example, the `country` table identifies the currency used in a particular country and contains two columns: `country` and `currency`. To add a new country row in this table we would use:

```
INSERT INTO country
  VALUES ('SteveLand',
    'Twinkies');
```

In this case Twinkies are the form of currency in SteveLand. Try entering this statement into ISQL and then select all the rows from country to see the result.

The data values must appear in the same order as the columns are defined; the first value given will be inserted into the first column, the second value into the second column, and so on. Alternatively, you can enter the values in any order you like as long as you specify the column names after the table name. The data values must then be in the order of the columns as given. For example:

```
INSERT INTO country (currency,
  country)
  VALUES ('Clams',
   'Troxellvania');
```

Using this same syntax you can insert data into only certain columns instead of all of them. Any columns not specifically included in the `INSERT` receive a null value. Try this in ISQL and then select all rows to see what happens:

```
INSERT INTO customer (customer)
  VALUES
    ('Bigwig International');
```

The results of this statement are shown in Figure 11. Notice that the `customer` column was set as we specified, and all the other columns except `cust_no` were set to null. `Cust_no` was set to a new value because of two advanced concepts called *triggers* and *generators*; concepts that are outside the scope of this article, but we'll cover them in a future issue.

Copying Rows

It is possible to combine the `INSERT` and `SELECT` statements to allow you to copy specific columns from existing rows in one table to another table. In this case, you simply omit the `VALUES` clause containing the explicit values and replace it with any legal `SELECT` statement with the same number and type of columns that you are inserting. It's difficult to illustrate this concept with the sample `employee` database we've been using, so here is a contrived example:

```
INSERT INTO shipped_orders
  (order_num, amount)
  SELECT order_num,
    order_total FROM orders
    WHERE order_status =
      'shipped';
```

In this example, we are reading all the rows from the `orders` table that have a status of `shipped`. For each row we find, we are inserting a row into the `shipped_orders` table that consists of the `order_num` and `order_total` columns from `orders` (realistically, we would probably delete these rows from orders after we've copied them). If there are any additional columns in `shipped_orders`, they default to null since we did not provide a value for them. Notice that the column names in the read table do not have to match the column names in the write table. We only need to have the same number of columns and of the same data type.

Changing Rows

Now that we have a new customer, let's add some useful information. To change the value of a column within an existing row we use the `UPDATE` statement. Let's add Joe Johnson as the contact for our new customer. Try the following in ISQL and examine the results:

```
UPDATE customer
  SET contact_first = 'Joe',
    contact_last = 'Johnson'
  WHERE customer =
    'Bigwig International';
```

(Note: `Bigwig International` must be spelled and cased exactly as you originally entered it in the `INSERT` statement previously).

The `SET` clause specifies a list of columns to modify and their new values. The `WHERE` clause operates just like the `WHERE` clause in the `SELECT` statement and defines the rows to apply the change to. If you omit the `WHERE` clause, the change will be applied to all rows in the table.

`UPDATE` can be used to set columns with calculated values as well. If you were in a generous mood and wanted to double everybody's salary, you could try:

```
UPDATE employee
  SET salary = salary * 2;
```

Removing Rows

Sooner or later, you'll need to remove some of the data from a table. To delete rows we use the `DELETE` statement. `DELETE` simply uses a `WHERE` clause to identify the rows to delete from a given table.

Let's say our company is downsizing and we now need to remove the Software Development department from the `department` table. Try the following in ISQL and then select all rows from the table to see the results:

```
DELETE FROM department
  WHERE dept_no = 621
```

If the `WHERE` clause is omitted, then all of the rows in the table are deleted.

Conclusion

There you have it: the real meat of SQL. We've covered the principal statements of SQL and with these you can perform a great deal of the common tasks of relational databases. In the next issue we'll cover some more features of `SELECT` that simplify creating reports from SQL data.

Steve Troxell is a Software Engineer with TurboPower Software where he is developing Delphi Client/Server applications using InterBase and Microsoft SQL Server for parent company Casino Data Systems. Steve can be reached on CompuServe at 74071,2207

➤ *Figure 11 (partial listing)*

```
SELECT * from customer

CUST_NO CUSTOMER                  CONTACT_FIRST   CONTACT_LAST
======= ========================= =============== ===================

   1008 Anini Vacation Rentals    Leilani         Briggs
   1009 Max                       <null>          <null>
   ...more rows...
   1015 GeoTech Inc.              K.M.            Neppelenbroek
   1016 Bigwig International      <null>          <null>
```

Writing Your Own Experts

by Bob Swart

Delphi is an open development environment, in that it has interfaces to enable you to integrate your own tools and experts with it. This article will focus on writing and integrating new Experts with Delphi.

There are three kinds of experts: project, form and standard. The first two can be found in the `Options | Gallery` dialog, while standard experts are on the Help menu (like the Database Form Expert).

Project and form experts can be activated whenever you create a new project or form (just like project and form templates). Standard experts generally do not create a new project or form, but just a new file, or unit. A project expert develops an entire project for you based on your specific preferences. A form expert develops custom forms that are added to your current project.

These example experts are not external tools that can be started from Delphi, they actually communicate with Delphi and form an integral part of the development environment. While this is not so strange for the existing Delphi experts (after all, they were developed and added by the same team that developed Delphi in the first place), it sounds intriguing at least to know that we, too, can write a Delphi expert that is able to communicate with Delphi in the same way. Could we write an expert that also opens files in the IDE, and can start a new project from scratch? Yes, all this is possible, and more, as we will see shortly!

TIExpert

The major reason why everybody thinks experts are difficult is because they are not documented. Not in the manuals or on-line Help, that is. If you take a look at the documentation and source code on your hard disk, though, you'll find some important files and even two example experts. The key example files can be found in the DELPHI\DOC subdirectory and are EXPTINTF.PAS and TOOLINTF.PAS. The first one shows how to derive and register our own Expert, while the second one shows how to use the tool services of Delphi to make the integration complete.

If we want to derive our own expert, say `TMyFirstExpert`, we have to derive it from the abstract base class `TIExpert`, which has seven abstract member functions (`GetStyle`, `GetName`, `GetComment`, `GetGlyph`, `GetState`, `GetIDString` and `GetMenuText`) and one member procedure (`Execute`).

My First Expert: TMy1stExp

Let's have a closer look at our first expert from Listing 1. Since `TIExpert` is an abstract base class, we need to override every function. First of all, we need to specify the style of the expert with the `GetStyle` method that can return one of three possible values: `esStandard` to tell the IDE to treat the interface to this expert as a menu item on the `Help` menu, `esForm` to tell the IDE to treat this expert interface in a fashion similar to form templates, or `esProject` to tell the IDE to treat this interface in a fashion similar to project

➤ Listing 1 Source Code for My First Expert (MY1STEXP.PAS)

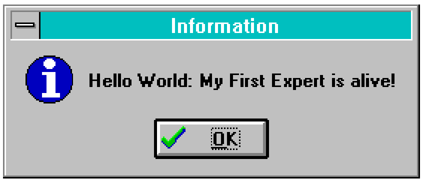
```
unit My1stexp;
interface
uses
  WinTypes, Dialogs, ExptIntf;
Type
  TMy1stExp = class(TIExpert)
  public
    function GetStyle: TExpertStyle; override;  { Style }
    { Expert Strings }
    function GetName: string; override;
    function GetComment: string; override;
    function GetGlyph: HBITMAP; override;
    function GetState: TExpertState; override;
    function GetIDString: string; override;
    function GetMenuText: string; override;
    procedure Execute; override;  { Launch the Expert }
  end;
  procedure Register;
implementation

function TMy1stExp.GetStyle: TExpertStyle;
begin
  Result := esStandard
end;

function TMy1stExp.GetName: String;
begin
  Result := ’My First Expert’
end;

function TMy1stExp.GetComment: String;
begin
  Result := ’’ { not needed for esStandard }
end;

function TMy1stExp.GetGlyph: HBITMAP;
begin
  Result := 0 { not needed for esStandard }
end;

function TMy1stExp.GetState: TExpertState;
begin
  Result := [esEnabled]
end;

function TMy1stExp.GetIDString: String;
begin
  Result := ’DrBob.MyFirstExpert’
end;

function TMy1stExp.GetMenuText: String;
begin
  Result := ’&My First Delphi Expert...’
end;

procedure TMy1stExp.Execute;
begin
  MessageDlg(’Hello World: My First Expert is alive!’,
             mtInformation, [mbOk], 0)
end;

procedure Register;
begin
  RegisterLibraryExpert(TMy1stExp.Create)
end;
end.
```

templates. For our `TMy1stExp`, a *standard* type expert that shows a `MessageDlg` to indicate it is alive, we can use the `esStandard` style.

After we've set the style of the expert, all we need to do is fill in the other options accordingly. `GetName` must return a unique descriptive name identifying this expert, like 'My First Expert'. If style is `esForm` or `esProject` then `GetComment` should return a short sentence describing the function of this expert. Since the style is `esStandard`, we can return an empty string. If style is `esForm` or `esProject` then `GetGlyph` should return a handle to a bitmap to be displayed in the form or project list boxes or dialogs. This bitmap should have a size of 60x40 pixels in 16 colours. Again, since the style is `esStandard`, we can return 0 here. If the style is `esStandard` then `GetState` returning `esChecked` will cause the menu to display a checkmark. This function is called each time the expert is shown in a menu or listbox in order to determine how it should be displayed. We just leave it `esEnabled` for now. The `GetIDString` should be unique to each expert. By convention, the format of the string is: `CompanyName.ExpertFunction`. If the style is `esStandard` then `GetMenuText` should return the actual text to display for the menu item, like 'My First Delphi Expert'. Since this function is called each time the parent menu is pulled down, it is even possible to provide context sensitive text.

Finally, the `Execute` method is called whenever this expert is invoked via the menu, form gallery dialog, or project gallery dialog. The style will determine how the expert was invoked. In this case, we just call a `MessageDlg` in the `Execute` method to indicate that the expert is actually alive.

To install our first expert, all we need to do is act like it's a new component: pick `Options | Install` and add it to the list of installed components. When Delphi is done with compiling and linking COMPLIB.DCL, you can find our first new expert in the `Help` menu. Just click on it and it will show that it's alive (see Figure 1).



Hello World: My First Expert is alive!

➤ *Figure 1*
My First Delphi Expert is alive!

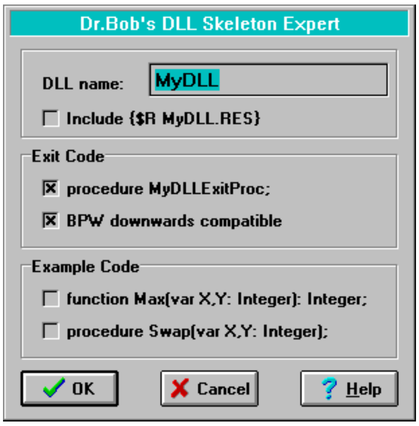
And Now For Something Completely Different...

Now that we've seen our first nice, but useless, Delphi expert, it's time to move on to more serious matters. I want to make a little sidestep to a subject that will make a good example of a more serious Delphi expert.

On the CompuServe DELPHI forum, one of the queries that comes up rather frequently is "*How do I write a DLL with Delphi?*". The answer is not just that you need to write the code starting with `library` and so on, the answer also needs to explain how to compile the source for a DLL with Delphi. In their wisdom, Borland made the Delphi IDE only capable of compiling the current project. If you just open a single file with the source for the DLL and press Ctrl-F9 to compile it, you won't get what you want. You must actually open your DLL source file as a *project* and then you can compile your DLL. During this process, Delphi will generate .OPT and .RES files if these don't already exist. All things considered, I would like something that enables me to open a new or existing DLL source file at once so I can compile it.

Speaking of DLLs, whenever I sit down to write a DLL in Delphi (or Borland Pascal, for that matter), I pick up an old one to use as skeleton. Mostly, I re-use the setup for the `ExitProc` routine and the `exports` settings. For this purpose, I've written a DLL skeleton that can be loaded every time I need it. Considering the fact that some of my friends also use this skeleton for their new DLLs, I decided to make it something truly re-usable: a Delphi DLL Skeleton Generator (see Figure 2).

As you can see, I've included the key functionality all in one Form: *How do I write a Resource-only DLL? How do I write my own WEP* (the same as `ExitProc`)? *How do I export routines from a DLL?* All these questions can be answered if you just select the appropriate options and click OK to generate the DLL skeleton source code. A sample skeleton DLL with all options enabled, except BPW compatibility (which does not include the `SysUtils` unit and `AddExitProc` routine but requires you to setup the `ExitProc` chain by hand), can be found in Listing 2.



DLL name: MyDLL
☐ Include {$R MyDLL.RES}
Exit Code
☒ procedure MyDLLExitProc;
☒ BPW downwards compatible
Example Code
☐ function Max(var X,Y: Integer): Integer;
☐ procedure Swap(var X,Y: Integer);

➤ *Figure 2*
Delphi DLL Skeleton Generator

Behind the `OkButtonClick` is the source code generator that writes the selected source code to file. Now I want something like this integrated into Delphi itself, so I can generate a new Delphi DLL Skeleton and open it as my new project at the same time. In order to make the DLL Skeleton Generator a Delphi expert, all we have to do is connect our expert `Execute` method with our DLL Skeleton Generator Form, as in Listing 3.

So, whenever the expert is executed, it will see if our DLL Skeleton Generator Form already exists (ie if the expert is already being executed) and create it if it doesn't exists. It will then show the form and give it the input focus. The DLL Skeleton Generator Form is then in control.

The Final Frontier...

Only one thing remains: the final integration with the Delphi IDE. I would like to be able open a new project with the source of the generated DLL Skeleton inside. For this, we need to communicate with the Delphi IDE itself. This is possible with the special `ToolServices` that are provided from Delphi to its experts. Like the expert interface, the `ToolServices` are not documented in the manual or on-line help. The only place you can find more information on this is in the TOOLINTF.PAS file, again in the DELPHI\DOC directory.

First of all, we need to check if the `ToolServices` are available to us. This is just a check to see if `ToolServices` (a global variable from the `TOOLINTF` unit) is not `nil`. If `ToolServices` are available, we can do several things. I would like to close the current project, which can be done with the function `ToolServices.CloseProject`. Then, I would like to open a new project, with the generated DLL Skeleton source file as the filename, which can be done with the function `ToolServices.OpenProject`.

The last part of the `OkButtonClick` method of the DLL Skeleton Generator Form is therefore as shown in Listing 4.

Simple, eh? That's all we need to communicate with Delphi and write a truly integrated Delphi standard expert.

Project Expert

The DLL Skeleton Generator Expert is still a standard expert, only accessible from the `Help` menu. I would like to make it a project expert, so we can select it when we start a new project. To do this, we have to derive the project expert from the standard expert and override four methods. First of all, we have to override `GetStyle` and return `esProject`. Also, we need to return a comment (this is not really needed) and a bitmap to display the expert in the Gallery.

```
library MyDLL;
{ Generated by DLL Skeleton Expert (c) 1995 by Dr.Bob for The Delphi Magazine }
uses WinTypes, WinProcs, SysUtils;
{$R MyDLL.RES}
function Max(X,Y: Integer): Integer; export;
begin
  if X > Y then Max := X
    else Max := Y
end {Max};
procedure Swap(var X,Y: Integer); export;
var Z: Integer;
begin
  Z := X;
  X := Y;
  Y := Z
end {Swap};
exports max index 1,
        swap index 2;
procedure MyDLLExitProc; far;
begin
  { WEP & cleanup }
end;
begin
  AddExitProc(MyDLLExitProc);
end.
```

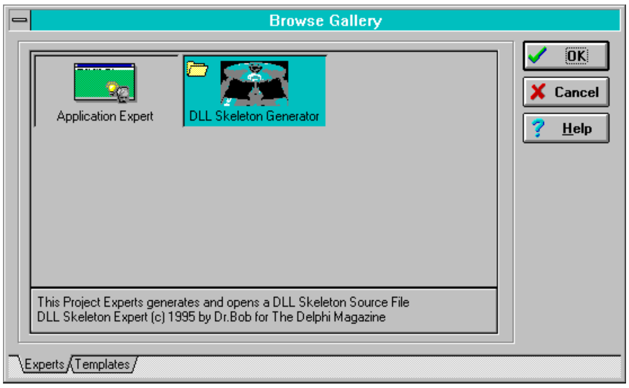
➤ *Listing 2 Generated DLL skeleton source code*

```
procedure TDLLSkExp.Execute;
begin
  if not Assigned(DLLSkeletonGenerator) then
    DLLSkeletonGenerator := TDLLSkeletonGenerator.Create(Application);
  DLLSkeletonGenerator.Show;
  DLLSkeletonGenerator.SetFocus
end;
```

➤ *Listing 3*

```
if ToolServices <> nil then begin
  { I’m an expert!! }
  if ToolServices.CloseProject then
    ToolServices.OpenProject(ExtractFileName(DLLName.Text)+’.PAS’)
end
```

➤ *Listing 4*



This Project Experts generates and opens a DLL Skeleton Source File
DLL Skeleton Expert (c) 1995 by Dr.Bob for The Delphi Magazine

➤ *Figure 3 DLL Skeleton Generator installed ready for use*

Standard And Project?

Remember the Database Form Expert? This can be found in the Gallery as a form expert *and* in the `Help` menu as a standard expert. It seems to be both.

I would like to be able to use my DLL Skeleton Generator Expert not only as a standard expert but also as a project expert. In that case I have to modify the expert functions from Listing 1 to include both the `esStandard` and `esProject` styles (the result is in Listing 5). Also, `GetIDString` needs to return unique ID strings for *both* the standard and the project expert. Even though the two are essentially the same, I need to return two special IDs. If you don't, Delphi will just GPF when you try to install the experts. Which leads back to Rule #1 from the *Under Construction* column: always have a backup of COMPLIB.DCL at hand when you start to play with components and experts.

Now, if we install the expert, as before, we get both a standard expert in the Help menu and the project expert in the Gallery (see Figure 3). If we enable the gallery from the environment options, we can generate and open a DLL Skeleton source file every time we start a new project.

```
unit Dllskexp;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, Buttons, StdCtrls,
  ExptIntf, ToolIntf;
{ definition of class TDLLSKeletonGenerator is on disk}
Type
  TDLLSkeletonStandardExpert = class(TIExpert)
  public
    { Expert Style }
    function GetStyle: TExpertStyle; override;
    { Expert Strings }
    function GetIDString: string; override;
    function GetName: string; override;
    function GetComment: string; override;
    function GetGlyph: HBITMAP; override;
    function GetState: TExpertState; override;
    function GetMenuText: string; override;
    procedure Execute; override;    { Launch the Expert }
  end;
  TDLLSkeletonProjectExpert =
    class(TDLLSkeletonStandardExpert)
  public
    { Expert Style }
    function GetStyle: TExpertStyle; override;
    { Expert Strings }
    function GetIDString: string; override;
    function GetComment: string; override;
    function GetGlyph: HBITMAP; override;
  end;
  procedure Register;

implementation
{$R *.DFM}
{ class TDLLSKeletonGeneratorm implementation is on disk}

function TDLLSkeletonStandardExpert.GetStyle:
  TExpertStyle;
begin
  Result := esStandard
end;

function TDLLSkeletonStandardExpert.GetIDString: String;
begin
  Result := ’DrBob.StandardDLLSkExp’
end;

function TDLLSkeletonStandardExpert.GetComment: String;
begin
  Result := ’’ { not needed for esStandard }
end;

function TDLLSkeletonStandardExpert.GetGlyph: HBITMAP;
begin
  Result := 0 { not needed for esStandard }
end;

function TDLLSkeletonStandardExpert.GetName: String;
begin
  Result := ’DLL Skeleton Generator’
end;

function TDLLSkeletonStandardExpert.GetState:
  TExpertState;
begin
  Result := [esEnabled]
end;

function TDLLSkeletonStandardExpert.GetMenuText: String;
begin
  Result := ’Dr.&Bob’’s DLL Skeleton Expert...’
end;

procedure TDLLSkeletonStandardExpert.Execute;
begin
  if not Assigned(DLLSkeletonGenerator) then
    DLLSkeletonGenerator :=
      TDLLSkeletonGenerator.Create(Application);
  DLLSkeletonGenerator.Show;
  DLLSkeletonGenerator.SetFocus
end;
{$R DLLSKEXP.RES}
Const DLLSKEXPBITMAP = 666; { Bitmap ID }

function TDLLSkeletonProjectExpert.GetStyle:
  TExpertStyle;
begin
  Result := esProject
end;

function TDLLSkeletonProjectExpert.GetIDString: String;
begin
  Result := ’DrBob.ProjectDLLSkExp’
end;

function TDLLSkeletonProjectExpert.GetComment: String;
begin
  Result := ’This Project Experts generates and opens ’+
    ’a DLL Skeleton Source File’#13+ ’DLL Skeleton ’+
    ’Expert (c) 1995 by Dr.Bob for The Delphi Magazine’;
end;

function TDLLSkeletonProjectExpert.GetGlyph: HBITMAP;
begin
  Result := LoadBitMap(HInstance,
              MakeIntResource(DLLSKEXPBITMAP))
end;

procedure Register;
begin
  RegisterLibraryExpert(
    TDLLSkeletonStandardExpert.Create);
  RegisterLibraryExpert(
    TDLLSkeletonProjectExpert.Create);
end;
end.
```

➤ *Listing 5 DLL Skeleton Generator Standard and Project Expert*

If we select the DLL Skeleton Expert, we can then select the required options (as in Figure 2). If we click on OK, the expert closes and we're in our main project: the generated source of the DLL.

Since the generated DLL source code is opened as a new project, we can instantly compile it by pressing Ctrl-F9. And once you have a skeleton DLL, it's easy to build on it and add your own functions.

Serious Business...

The example DLL Skeleton Generator Expert is included on the subscribers' disk with this issue. You'll also find another expert, the one I wrote about in the last issue: *HeadConv*. This solves a more serious problem that many people have: *"How do I use this foreign DLL written in C, as I only have the C header file with it and no Delphi import unit?"* The answer is to convert the C DLL header file to a Delphi import unit. This is no simple task, especially for large header files, and my *HeadConv C DLL Header Converter Expert* tries to assist in this task by creating an initial conversion from which to start. For some headers, the initial conversion is good enough, for others extra work might be needed. Specifically, the declaration of nested structs and actual code (as opposed to function declarations) will be a source of problems (pun intended).

I've decided to sell *HeadConv* as a shareware tool. The version on the disk is fully functional, but for a registration fee of $25 you get a more advanced version with explicit import unit capabilities and the source code of the expert (but not of the parser). The CompuServe SWREG forum registration ID is 6533). See the advert in this issue for more details.

Bob Swart is a professional software developer using Borland Pascal, C++ and Delphi; email: 100434.2072@compuserve.com

Tips & Tricks

This is ***your*** column! Here is your opportunity to share with your fellow Delphi enthusiasts those hard-won hints and helps that make your life easier day by day. Please do send in your Tips & Tricks to us (preferably by email, to the Editor, at 70630.717@compuserve.com, or alternatively on disk), whether large or small, on any aspect of Delphi or related issues. We're looking forward to hearing from you!

Popup Menus

A question arose at a Delphi Developers' Group meeting in London recently. How do you find out which component caused a popup menu to pop up? Remember that a popup menu may be associated with many components. No one knew an immediate answer, and so I set to work looking for a solution. I came up with the techniques programmed in Listing 1 before finding the documented `PopupComponent` run-time property. If a right-click on a component caused a menu to pop up, this property points to that component. I just wasted all that time working out my own method! Or so I thought.

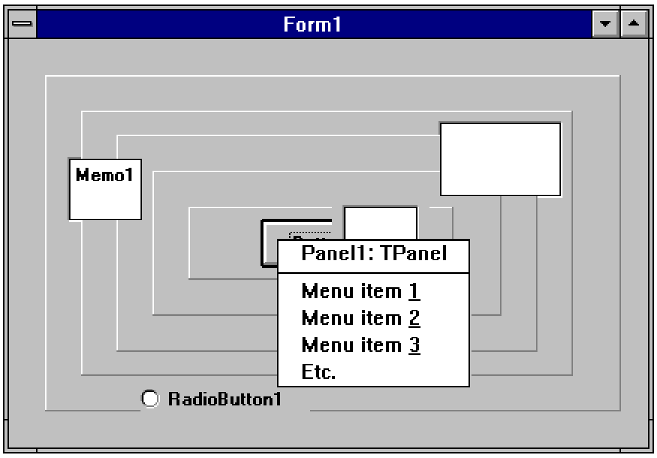
It seems `PopupComponent` doesn't work consistently. It often gives mis-information, as shown in Figure 1: when I right-click over a button that's on some panels, the popup thinks that `Panel1` is under the cursor. So, let's dig that old code out of the bin and persevere. But there's another thing we can do...

Delphi uses popup menus in the form designer. However, when you right click, a menu pops up which is specific to the currently selected component, not the component under the mouse cursor. Delphi can generate its popup using a keystroke, `Alt-F10` and again, the selected component is used.

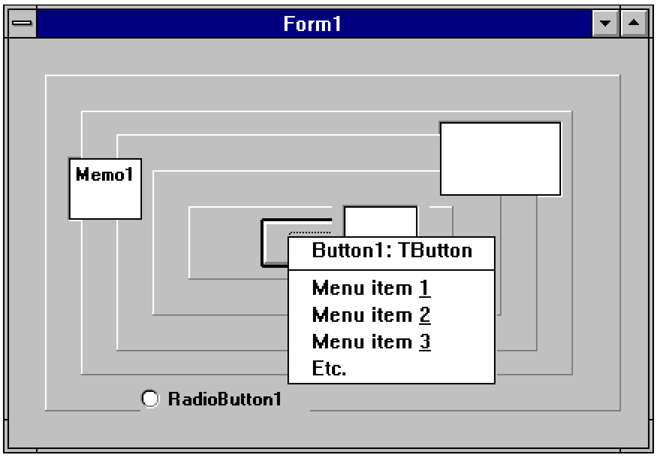
Given that VCL popups work on the current cursor position, it may also be desirable to allow a keystroke (`Alt-F10`) to popup a menu, but again, make it specific to the component under the cursor, if there is one.

We can use the technique implemented in Listing 1 to do this. The menu that's generated is a bit like a Paradox for Windows popup. The first item is descriptive (in this case it tells you the component name and class) and non-selectable. It is disabled, but not greyed out. If there is no component under the cursor, the top menu item and its separator are hidden.

The `Alt-F10` keystroke is trapped by setting the form's `KeyPreview` property to `True` and using an `OnKeyDown` event handler. It then pops up the menu at the current mouse position.



➤ *Figure 1 PopupComponent gets it wrong...*



➤ *Figure 2 FindComponentAtCursor gets it right!*

To find which component is under the mouse cursor when a popup menu pops up we need to implement an `OnPopup` event handler for the menu itself and then do some exploratory work. The `PopupMenu1Popup` event handler in Listing 1 calls `FindComponentAtCursor` to do the job. `FindComponentAtCursor` returns the appropriate component, and the event handler sets the first menu item's caption to the name and class of the component. It then ensures the menu item is disabled, but not greyed, by using a Windows API call.

The `FindComponentAtCursor` routine works by identifying the current position of the cursor and then using another API call to identify the handle of the window that contains the cursor position. This handle is passed to `FindControl` to identify which `TWinControl`-based component has that handle as its `Handle` property.

Contributed by Brian Long (whose email address is 76004.3437@compuserve.com)

Form Design Quick Keys

Delphi makes designing the visual interface of your applications so much easier, but there are tricks that can make design even slicker! If you find it a pain

```
unit Popupsu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, Menus, StdCtrls,
  ExtCtrls, Grids, Outline;
type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    Panel3: TPanel;
    Panel4: TPanel;
    Panel5: TPanel;
    Button1: TButton;
    PopupMenu1: TPopupMenu;
    DummyItem: TMenuItem;
    Memo1: TMemo;
    ListBox1: TListBox;
    RadioButton1: TRadioButton;
    Notebook1: TNotebook;
    Outline1: TOutline;
    N1: TMenuItem;
    Menuitem1: TMenuItem;
    Menuitem2: TMenuItem;
    Menuitem3: TMenuItem;
    EtcItem: TMenuItem;
    procedure PopupMenu1Popup(Sender: TObject);
    procedure FormKeyDown(Sender: TObject;
      var Key: Word; Shift: TShiftState);
  private
    { Private declarations }
  public
    { Public declarations }
    function FindComponentAtCursor: TWinControl;
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
```

```
function TForm1.FindComponentAtCursor: TWinControl;
var  Pt: TPoint;
begin
  GetCursorPos(Pt);
  Result := FindControl(WindowFromPoint(Pt));
end;
procedure TForm1.PopupMenu1Popup(Sender: TObject);
begin
  with PopupMenu1 do begin
    PopupComponent := FindComponentAtCursor;
    { Write to the menu first, to make sure it is
      brought into life. If you do this last, the
      EnableMenuItem call will have had no effect,
      since the menu won't actually exist }
    if PopupComponent <> nil then
      DummyItem.Caption := PopupComponent.Name + ': ' +
        PopupComponent.ClassName;
    { No component of ours under cursor  so get rid of
      menu item ... }
    DummyItem.Visible := PopupComponent <> nil;
    { ... and seperator }
    N1.Visible := PopupComponent <> nil;
    { Disable the dummy menu item, but _don't_ grey it
      out. The Enabled property does grey the menu item
      when set to False }
    EnableMenuItem(Handle, DummyItem.Command,
      mf_ByCommand or mf_Disabled);
  end;
end;
procedure TForm1.FormKeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
var  Pt: TPoint;
begin
  GetCursorPos(Pt);
  if (ssAlt in Shift) and (Key = vk_F10) then
    PopupMenu1.Popup(Pt.X, Pt.Y);
end;
end.
```

➢ *Listing 1*

moving controls on your forms into exactly the right position using the mouse, try these:

- Use `Ctrl` plus the cursor keys to move the current control on the form in one pixel increments;
- Use `Shift` plus the cursor keys to re-size the current control on the form in one pixel increments;

Also, pressing the `Esc` key while a control is selected on a form passes the focus to the underlying control or form.

Contributed by Tony McKiernan

More Editor Shortcuts

Ever wanted to remove a column of text at the end of your lines? For example:

```
Statement1; { Comment 1 }
Statement2; { Comment 2 }
Statement3; { Comment 3 }
Statement4; { Comment 4 }
Statement5; { Comment 5 }
Statement6; { Comment 6 }
Statement7; { Comment 7 }
```

Suppose you wanted to remove the Comments. Normally you would have to remove them one by one by going to the beginning of the comment and pressing `Ctrl-Q-Y` (to remove all the text up to the end of the line). The new Borland IDEs (Delphi 1.0 and BC++ 4.x) however support a new feature: you can now *mark a column of text*. To do this using the Default or Classic keyboard mapping:

Using the keyboard:

- Type `Ctrl-O-C` (to enter column selection mode),
- Now select the part you want to remove, by using the Shift and cursor keys,
- After this you can remove the text by pressing `Ctrl-Del` (or `Shift-Del` to cut it to the clipboard).

This method is also ideal if you want to *swap* two columns of text: just mark the column, cut it to the clipboard and paste it where you want it. To return to the normal selection mode, press `Ctrl-O-K`.

Using the mouse:

- Use `Alt Left Mouse Button` to select a block of text.

You can also try out the other marking methods like inclusive block marking or line marking (`Ctrl-O-I` and `Ctrl-O-L` respectively). Here's some more editor tips:

- Ever wanted to put a complete word into uppercase or lowercase in the IDE? While the cursor is in the word, press `Ctrl-K-E` to change it to lowercase, or press `Ctrl-K-F` for lowercase.
- Ever wanted to go directly to a specific line number in the IDE? Press `Ctrl-O-G` and then enter the line number you want to go to.

Contributed by Arjan Jansen

Replacing if..then..else

Rather than using clumsy `if..then..else` statements such as:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  if Button1.Top + Button1.Height div 2 <
    ClientHeight div 2 then
    Button1.Caption := 'Top Half'
  else
    Button1.Caption := 'Bottom Half';
end;
```

you can take advantage of the anomalous typed constant construct in Delphi, and also the fact that arrays can have elements indexed by any ordinal type. So, the condition above becomes:

```
procedure TForm1.FormResize(Sender: TObject);
const
  Captions: array[False..True] of String[11] =
    ('Bottom Half', 'Top Half');
begin
  Button1.Caption :=
    Captions[Button1.Top + Button1.Height div 2 <
    ClientHeight div 2];
end;
```

Notice that we are using the result of a Boolean expression to index the array, which was set up with Boolean element indices. Another example uses a compound statement in the `if` statement:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  if Button1.Top + Button1.Height div 2 <
    ClientHeight div 2 then begin
    Button1.Caption := 'Top Half';
    Button1.Enabled := True;
  end else begin
    Button1.Caption := 'Bottom Half';
    Button1.Enabled := False;
  end;
end;
```

A simplification of this becomes:

```
procedure TForm1.FormResize(Sender: TObject);
const
  Captions: array[False..True] of
    String[11] = ('Bottom Half', 'Top Half');
begin
  Button1.Enabled :=
    Button1.Top + Button1.Height div 2 <
      ClientHeight div 2;
  Button1.Caption := Captions[Button1.Enabled];
end;
```

Contributed by Brian Long

Are We In Range?

If we want to check if an ordinal value is in a particular range, or is one of a number of values, many third and fourth generation languages instill the following techniques in us:

```
if (TestValue >= 3) and (TestValue <= 7) then
  ShowMessage('It''s between 3 and 7');
if (TestValue <> 3) and (TestValue <> 5) and
  (TestValue <> 7) then
  ShowMessage('Found it');
```

However Object Pascal offers us alternative constructs to express this with:

```
if TestValue in [3..7] then
  ShowMessage('It''s between 3 and 7');
if not (TestValue in [3, 5, 7]) then
  ShowMessage('Found it');
```

These expressions make use of *sets* which are a convenient aid to reducing typing and making code (in my opinion) more readable. Sets have limitations here though: they only take values which take one byte, in other words no `Integer`, `Longint` or `Word` variables, amongst others. So the following *won't* work:

```
if not (Message.Msg in [wm_LButtonDown,
  wm_LButtonDblClk]) then
  inherited WndProc(Message);
```

To cater for other types, we can avoid saying:

```
if (Message.Msg <> wm_LButtonDown) and
  (Message.Msg <> wm_LButtonDblClk) then
  inherited WndProc(Message);
```

by using a `case` statement. This allows you to specify ranges and lists of values, as you can in a set, but without the one byte restriction. So, for example, the following may be a preferable scheme (bear in mind that in many cases the tests we need to perform in our programs may be rather larger than these simple ones):

```
case Message.Msg of
  wm_LButtonDown, wm_LButtonDblClk: ;
else
  inherited WndProc(Message);
end;
```

Contributed by Brian Long

Thanks to all our contributors to this issue's Tips & Tricks column. Sorry to those who submitted Tips that we didn't have space for in this issue, they're in the file for Issue 4!

Subclassing Windows

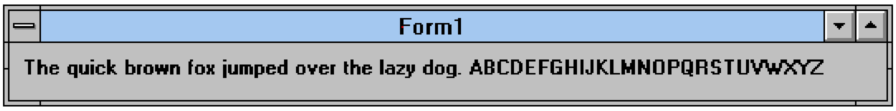
by Brian Long

So what is subclassing? Well, the answer depends on the context of the question. It means different things to different people. Let's look at definitions from some industry gurus.

Firstly Bjarne Stroustrup, the guy who designed the C++ programming language, in his book *The C++ Programming Language, Second Edition*, says: "A base class is sometimes called a superclass and a derived class a subclass." He goes on to mention that this is a confusing definition given that "an object of a derived class has its base class as a subobject and also that a derived class is larger than its base class in the sense that it holds more data and provides more functions." From this we learn that in the OOP world, subclassing can be taken to mean deriving new objects.

As for Charles Petzold, in *Programming Windows 3.1* in a paragraph discussing scroll bars says "the window procedure for the scroll bar controls is somewhere inside Windows. However, you can obtain the address of this window procedure by a call to `GetWindowLong` using the `GWL_WNDPROC` identifier as a parameter. Moreover, you can set a new window procedure for the scroll bars by calling `SetWindowLong`. This technique, called 'window subclassing' is very powerful."

So a subclass is a derived object, but window subclassing (or more correctly, window instance subclassing, as there is also a concept of global window subclassing) involves changing the functionality of a window/control. The reason we get (at least) two definitions is that the base term *class* is an OOP term, but it is also the term Microsoft chose to apply to a set of information that is required when creating a window. Most important of this information is the window procedure, the subroutine that responds to messages sent to the window, dictating how the window will appear and function.


Form1

The quick brown fox jumped over the lazy dog. ABCDEFGHIJKLMNOPQRSTUVWXYZ

➤ *Our all-singing, all-dancing example!*

With tools such as Delphi, the window procedure is tucked away under the plush carpet of the class library, but Delphi does not prevent us accessing it directly.

Typically when people use the term subclassing loosely, it ends up meaning a combination of the two ideas mentioned above: changing the behaviour of a particular window instance by deriving a new object class. This ends up being the most convenient way of changing the functionality accessed by the window procedure associated with the window class of the window.

So with all that borne in mind and with Delphi both giving us high level encapsulations of all things Windows-based and also allowing us to get to the low-level nuts and bolts of Windows, what options do we have for subclassing a Window? The answer is several.

To explore them all, let's take a trivial task and implement the relevant subclassing in as many different ways as possible. The task will be writing to the caption of a label as characters are typed on the keyboard when the form in a simple application has the focus, in other words changing the default functionality that occurs when the form window receives a `wm_Char` message. However, before we start it will be useful to view the course a message takes as it is digested through a Delphi application.

Message Execution Flow

When Windows has a message that needs to be delivered to an application, it normally places it in the application's message queue (which defaults to a capacity of eight messages). At least this is the case if the message was sent using `PostMessage`. If `SendMessage`, or the Delphi method `Perform`, were used instead, the message is passed directly to the appropriate window procedure by Windows/Delphi.

Messages which are sent using `PostMessage`, or which manage to get into the application message queue in more elaborate ways, are called *queued* messages. Those that go directly to the window procedure are *unqueued* messages.

The mechanism by which queued messages get delivered to the appropriate window in the program is wrapped up in an `Application` object method called `ProcessMessages`, normally called from `Application.HandleMessage` (itself called repetitively from `Application.Run`), but is also called by the yielding method `Application.ProcessMessages` .

Inside `ProcessMessages`, when a message is plucked from the queue, it is given to the `Application` object's user-supplied `OnMessage` handler if one exists, which has the option of terminating the message's existence, if it deems it appropriate. If the message survives this first hurdle, it is sent to the appropriate window procedure using the Windows API function `DispatchMessage`.

The window procedure of the form or control (or any object descended from `TWinControl`) that receives the message is a small stub of code which calls the non-virtual method `MainWndProc`, which in turn passes the message straight onto the virtual method `WndProc`, originally defined in the `TControl` object. Each object that overrides `WndProc` in the Visual Component Library adds additional default handling. It is in this default handling that the message may again get swallowed. If it makes it through this point, the message gets passed to the `TObject` method `Dispatch`, which invokes the dynamic method dispatching system to allow any specific message handlers (defined using the `message` keyword) that have been implemented to be called. Pre-written message handlers in the VCL have the task of calling any event handlers that have been written at relevant points.

With that meandering tale finished, it's on with the show.

Method 1:
The OnMessage Event

If we do this chronologically, the earliest place we can encroach upon the default message handling scheme (for queued messages) is by defining an event handler for the `Application` object's `OnMessage` event. Unfortunately we can't ask the Delphi environment to manufacture an `OnMessage` handler as the `Application` object does not have a visual representation; we have to do it manually, as shown in Listing 1 in the form's `OnCreate` handler, `FormCreate`.

The event handler, `MsgHandler`, will be triggered as soon as a message is picked from the application queue inside `Application.ProcessMessage`, regardless of the target window. So in the handler, we must check that the target window handle of the message matches our form's `Handle` property and that the message is a `wm_Char` message and then do what we want to do: determine what key was pressed and add it to the label's caption.

Look in the Delphi help file for more on the `OnMessage` event.

Method 2:
The Windows SDK Approach

If you say "subclass" to an experienced Windows SDK programmer, there is a strong possibility that, in a fashion usually associated with Pavlov's dogs, they will involuntarily say "`SetWindowLong`". In the world of procedural API programming, window instance subclassing is done using `SetWindowLong` to replace the current window procedure with another one of our choosing. Because of implementation issues of object methods, the new window procedure needs to be a global function, not a method, although we'll see how to overcome this in the next section.

Changing the window procedure is better than using an `OnMessage` handler since a window procedure deals with all messages, no matter how they are sent. The `OnMessage` event only reacts to queued messages and so the window procedure is the earliest place in the scheme of things that is guaranteed to see every message.

The example in Listing 2 replaces the form's window procedure with a global routine called `NewWndProc`. The parameters passed into the window procedure allow us to identify the target window (fairly redundant in this application since this window procedure is associated with only one window; however you may see fit at some point to write one window procedure to be associated with several windows), the message number and the two additional pieces of information associated with the message.

In the form's creation event, `SetWindowLong` is called with the parameter `gwl_WndProc` to specify that we wish to change a window procedure and the form's window handle to indicate which window is to be changed. In addition, the address of our replacement window procedure, typecast into a `Longint`, is also passed along. The return value from `SetWindowLong` is the address of the old window procedure: we need to call this from our replacement window procedure so we save this address.

In the new window procedure, we check if the message is `wm_Char` – if it is we update the label, if it isn't we invoke default functionality by using `CallWindowProc` to call the original window procedure, passing the same parameters as we were given. Notice that in the form's `OnClose` event handler, `FormClose`, we tidy up by setting the window procedure back to the original routine that we saved.

Look in the Windows API help file for more on `SetWindowLong`.

Method 3: Windows SDK ++

In the section above on the course taken by a message through an application, I mentioned that the window procedure for all forms was a small stub of code which calls the `MainWndProc` method. The

➤ *Listing 1*

```
unit Subu1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TWndProc = function(HWindow: HWND; Message: Word;
    WParam: Word; LParam: Longint): Longint;
  TForm1 = class(TForm)
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
  private { Private declarations }
  public  { Public declarations }
    procedure MsgHandler(var Msg: TMsg; var Handled: Boolean);
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.MsgHandler(var Msg: TMsg; var Handled: Boolean);
begin
  if (Msg.HWnd = Handle) and (Msg.Message = wm_Char) then
    Label1.Caption := Label1.Caption + Chr(Msg.WParam);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMessage := MsgHandler;
end;
end.
```

reason `MainWndProc` can't be set as the window procedure directly is that it is a method. Windows expects to be given a function which takes a particular set of four parameters.

The implementation of methods in the Object Pascal language causes them to have an additional hidden parameter, `Self`, used to identify the currently executing object instance, which would cause any attempt at parameter passing by Windows to become unsynchronised. That is why we have to supply a global routine. However, we can work around this limitation using the function `MakeObjectInstance` and its partner `FreeObjectInstance`.

`MakeObjectInstance` takes one parameter, the name of a method that you wish to use as a window procedure, which needs to be defined as taking a `TMessage` parameter, a record holding message information. It returns a pointer to a small block of code which can be given to `SetWindowLong` and which will invoke your window procedure method. When you are done with this window procedure, be sure to call `FreeObjectInstance` to deallocate this code block.

Both `MakeObjectInstance` and `FreeObjectInstance` were present in Borland Pascal 7, but weren't documented. In Delphi the *Component Writer's Guide* tells us what we can use them for. They can be seen as parallels to the Windows API functions `MakeProcInstance` and `FreeProcInstance` which deal with procedure instances: small chunks of code that Windows can safely call, which in turn safely call some exported routine in your code. These functions generate code snippets to safely allow Windows to indirectly call object methods.

In Listing 3, `MakeObjectInstance` is used to turn `NewWndProc`, a form method, into the form's window procedure. In the form's `OnClose` event handler, the window procedure is set back to its original value and our window procedure calling stub, which is returned back from `SetWindowLong`, is freed with `FreeObjectInstance`.

```
unit Subu2;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private { Private declarations }
  public  { Public declarations }
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
var OldWndProc: TFarProc;
function NewWndProc(HWindow: HWND;   Message: Word;
  WParam: Word; LParam: Longint): Longint; export;
begin
  Result := 0;
  if Message = wm_Char then
    Form1.Label1.Caption := Form1.Label1.Caption + Chr(WParam)
  else
    Result := CallWindowProc(OldWndProc, HWindow, Message, WParam, LParam);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  OldWndProc := TFarProc(SetWindowLong(Handle, gwl_WndProc,
    LongInt(@NewWndProc)));
end;
procedure TForm1.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  SetWindowLong(Handle, gwl_WndProc, LongInt(@OldWndProc))
end;
end.
```

➤ *Listing 2*

```
unit Subu3;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private { Private declarations }
  public  { Public declarations }
    FOldWndProc: TFarProc;
    procedure NewWndProc(var Message: TMessage);
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.NewWndProc(var Message: TMessage);
begin
  with Message do begin
    Result := 0;
    if Msg = wm_Char then
      Label1.Caption := Label1.Caption + Chr(WParam)
    else
      Result := CallWindowProc(FOldWndProc, Handle, Msg, WParam, LParam);
  end;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  FOldWndProc := TFarProc(SetWindowLong(Handle, gwl_WndProc,
    Longint(MakeObjectInstance(NewWndProc))));
end;
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  FreeObjectInstance(
    Pointer(SetWindowLong(Handle, gwl_WndProc, LongInt(FOldWndProc))));
end;
end.
```

➤ *Listing 3*

Method 4: Virtual Window Procedure Method

Having gone to all the trouble of finding how we can do what is essentially passing a method into `SetWindowLong`, we will now see that there was no real need to worry about it. As mentioned previously, there is a virtual method that acts as a window procedure method already waiting for us to override it. So we can now dispense with `SetWindowLong`, `MakeObjectInstance` and `FreeObjectInstance`.

The code in Listing 4 is pretty straightforward. We override the virtual `WndProc` method and inside it we update the label if a `wm_Char` message is received, otherwise we call upon the default `WndProc` functionality inherited from the `TForm` object.

Method 5: Message Handlers

In all the cases so far, we have needed to check the message that comes through to ensure it matches the one we are interested in trapping. There is an elegant construct which allows us to forego these comparisons. The message keyword, when used in conjunction with a method definition and an appropriate message identifier, allows us to write a specific message handler. The handler needs to take either a generic `TMessage` record as a parameter, or one of the specific message records defined in the `Messages` unit, or alternatively a user-defined message record if this is not a standard message.

In our case we are changing the behaviour of a `wm_Char` message so the `WMChar` method uses a specific `TWMChar` record type. To complete the declaration of the method in the class definition, the message keyword is followed by the `wm_Char` identifier.

Although this construct allows us to conveniently write a specific message handling method inside the target window class definition, there is a down side to it. Despite the implementation being mostly in hand-optimised assembler, the dynamic method dispatching scheme used in `TObject.Dispatch` is slower than the virtual method

```
unit Subu4;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
type
  TWndProc = function(HWindow: HWND; Message: Word; WParam: Word;
    LParam: Longint): Longint;
  TForm1 = class(TForm)
    Label1: TLabel;
  private { Private declarations }
  public  { Public declarations }
    procedure WndProc(var Message: TMessage); override;
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.WndProc(var Message: TMessage);
begin
  if Message.Msg = wm_Char then
    Label1.Caption := Label1.Caption + Chr(Message.WParam)
  else
    inherited WndProc(Message);
end;
end.
```

➤ *Listing 4*

```
unit Subu5;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
type
  TWndProc = function(HWindow: HWND; Message: Word; WParam: Word;
    LParam: Longint): Longint;
  TForm1 = class(TForm)
    Label1: TLabel;
  private { Private declarations }
  public  { Public declarations }
    procedure WMChar(var Msg: TWMChar); message wm_Char;
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.WMChar(var Msg: TWMChar);
begin
  Label1.Caption := Label1.Caption + Chr(Msg.CharCode);
end;
end.
```

➤ *Listing 5*

```
unit Subu6;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
type
  TWndProc = function(HWindow: HWND; Message: Word; WParam: Word;
    LParam: Longint): Longint;
  TForm1 = class(TForm)
    Label1: TLabel;
    procedure FormKeyPress(Sender: TObject; var Key: Char);
  private { Private declarations }
  public  { Public declarations }
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  Label1.Caption := Label1.Caption + Key;
end;
end.
```

➤ *Listing 6*

dispatching scheme, as discussed in Chapter 2 of the *Component Writer's Guide*.

An object's virtual method table has pointers to all virtual methods, be they inherited from ancestor objects or new ones. The dynamic method list on the other hand is much more space conservative (necessary since an object can amass message handlers for many Windows messages) and only holds pointers to new methods introduced in the current object. This means that to find the relevant address to jump to, `Dispatch` may need to search through the dispatch lists of all the object's ancestors. Because of this, if speed is key to your application, you would do better to use one of the previous window procedure or message event approaches.

More information on creating message handlers can be found in Chapter 7 of the *Component Writer's Guide*, or alternatively in the Component Writer's help file by searching for the messages section and the topic "Handling messages."

Method 6: Individual Events

After all that, for this particular scenario there is a much simpler and more familiar approach. A Delphi event handler can be manufactured using the Object Inspector's Events page for the `OnKeyPress` event for the form in the normal way, as shown in Listing 6. Inside the VCL, the `OnKeyPress` event's associated routine (or perhaps, more accurately, I should say the method pointed to by the `OnKeyPress` pointer property since the events listed in the Object Inspector are nothing more than properties for storing method addresses in) is called indirectly from a `wm_Char` message handler method in the `TWinControl` ancestor object of the `TForm`.

Conclusions

So, all this goes to show that with an advanced product like Delphi, which a lot of people treat like a clever 4GL, but which of course is really a very capable 3GL in a most sumptuous wrapping, there are more ways than one to skin the proverbial cat. In this particular example of window instance subclassing, we have six different approaches.

We can supply a routine that gets called for all queued messages targeted to any window in the application, though this will miss any nonqueued messages. There are two ways of writing traditional window procedure replacements, using either a global routine or an object method in conjunction with a Windows API function. Then there is the already present virtual window procedure method and also the elegant message dispatch methods. Last but by no means least is the standard Delphi event model. And that, I think, is plenty for one day.

Note: Brian's examples are on the free disk with this issue, as Delphi projects called SUB1 to SUB6.

Brian Long is an independent consultant and trainer specialising in Delphi. His email address is 76004.3437@compuserve.com

Optimising Display Updating

by Mike Scott

With its optimising compiler and efficient VCL class library, Delphi produces applications that run much faster than those from interpreted products such as Visual Basic. However, the methodology employed in VCL for screen updates has been simplified to ease programming. The downside is that it can be inefficient under certain circumstances.

In this article I will introduce you to two techniques to optimise screen updates which require only a few simple additions to your painting code. To illustrate this I have a written a sample application, FASTDRAW, which is included on the free disk you'll receive with Issue 2. As a bonus, I have also used code that illustrates the use of Windows complex regions for painting, exception handling to protect allocations of Windows resources and how to copy areas directly from and to the screen.

So to begin with, let's go back to first principles. When a window is created, or revealed by moving another window, for example, the parts which were previously obscured and have now been made visible need to be painted. We call these areas invalid regions.

Windows maintains a list of invalid regions and responds to the need for updates by sending messages to the appropriate windows to tell them to repaint themselves. In Delphi applications, the VCL receives this message for you and calls the `Paint` method of your component or form. If you have installed an `OnPaint` handler this is called too.

However, Windows supplies extra information to help you optimise the update but VCL does not pass this on to the `Paint` method or handler. It is non-essential information, your windows will look fine without it. The problem is that they may repaint much slower than necessary because in many cases only part of the window needs repainting. But because VCL doesn't tell you which part, you just have to code your paint method to paint everything. What you need is that extra information.

Fortunately, the Windows API is fairly helpful in this respect. There is a simple function which you can call to get a `TRect` that completely surrounds the invalid region. Then, all you need to do is check which parts of your form or component intersect that area and paint those. It may sound complicated but it's not. It's time to look at the sample.

Ellipses, Ellipses, Everywhere

The sample program simply creates a random pattern of ellipses of different colours spread out to cover an 800 by 600 pixel form. However, when the form first appears it is considerably smaller than this. The form has a toolbar with a checkbox that switches optimised painting on and off. When it's off, the paint method blindly paints every ellipse whether it needs to or not. When the form is in its initial size, for example, only about a quarter of the total number ellipses are visible, but it tries to draw all of them anyway. Of course, Windows makes sure that the ellipses outside the window don't appear, but it still does all the calculations, which wastes a lot of time. For a comparison see Figures 1 and 2.

I have defined a `TEllipse` class, shown in Listing 1. Notice the `Rect` field. It's a good idea to add a rect to classes that you are going to display so that you can quickly determine if they need to be drawn. `TEllipse`'s `Paint` method simply sets the line and fill colours and calls `TCanvas.Ellipse`.

Now let's look at the `Paint` -handlers for the form. I 've written two of these, one "dumb" and the other "smart". When you change the optimised checkbox, the appropriate handler is assigned to the form's `OnPaint` property. The "dumb" paint method simply iterates through the list of ellipses and calls their `Paint` methods as declared above. The loop looks like this:

```
for i := 0 to
  Ellipses.Count - 1 do
    TEllipse(
    Ellipses[i]).Paint(Canvas);
```

When you check optimised, the other, "smart", `OnPaint` handler is assigned. This has additional code to optimise painting. It does this by calling the Windows procedure `GetClipBox` which gets the `TRect` that encloses the invalid region. Then it iterates through the ellipses as before but uses the `IntersectRect` function to check if any part of each ellipse intersects the invalid rect to see if it needs to call `TEllipse.Paint`:

```
GetClipBox(Canvas.Handle,
  ClipRect);
for i := 0 to
  Ellipses.Count - 1 do
  with TEllipse(Ellipses[i]) do
    if IntersectRect(ARect,
    Rect, ClipRect) <> 0 then
      Paint(Canvas);
```

When you get your disk, try running the sample. Press the 'Repaint all' button with and without optimised drawing and note the difference in the times displayed on the toolbar. On my system I get a twelve-fold increase in speed! Interestingly, the time to

Listing 1

```
type
  TEllipse = class( TObject )
  protected
    LineColor : TColor;
    FillColor : TColor;
  public
    Rect      : TRect;
    constructor Create(
      const ARect : TRect;
      ALineColor  : TColor;
      AFillColor  : TColor);
    procedure Paint(ACanvas :
      TCanvas); virtual;
  end;
```

Figure 1

Repainting all the ellipses ***without*** *optimisation*

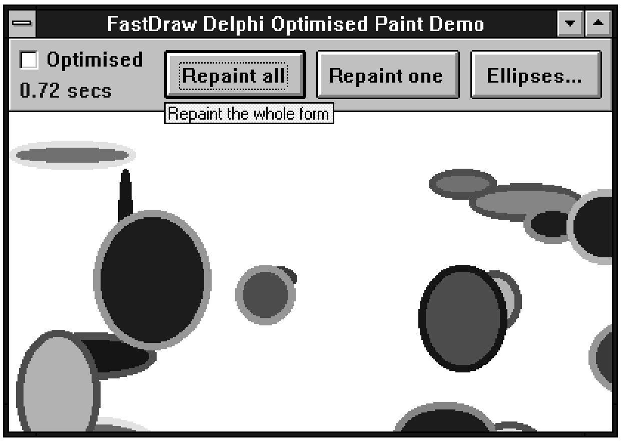
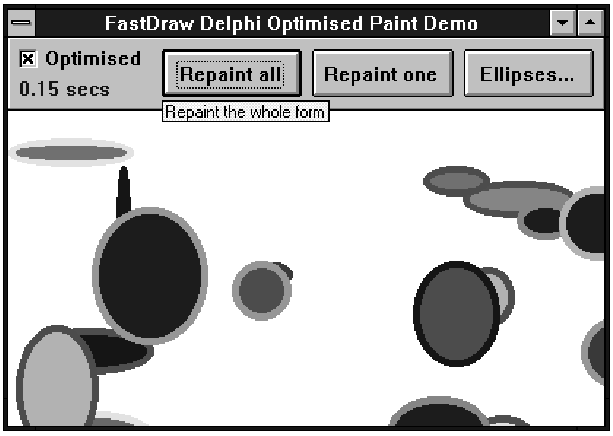


Figure 2

Now notice the difference in repaint time ***with*** *optimisation!*

The comparison when only one ellipse is repainted (Repaint one) is even more staggering.



repaint the form using the dumb technique is substantially longer when it is not maximised because of the extra calculations which Windows performs to clip each ellipse to the visible region. In contrast, the optimised method takes less time as the form is reduced in size.

If you maximise the running form, there is virtually no speed difference between dumb and smart redraws because all the ellipses have to be drawn in both cases anyway. The good news is that the additional code that is executed in the optimised case is so fast that you shouldn't notice any increase in the time recorded.

If you press the 'Ellipses...' button you can change the number of ellipses on the form. You will really notice the difference with a large number, say 500 or more. Also, you should try moving the 'Ellipses...' dialog around and noting how long it takes to repaint the revealed area.

You may notice that there is no repaint if you simply close the dialog without moving it. In this case, Windows has decided to take a copy of the background and replace it when the dialog is closed. Moving it forces a repaint and Windows discards the noted area.

Reducing Flicker

VCL has another simplification that can cause another type of annoyance – flicker.

Windows provides a function to invalidate an area of a window to force a paint message to be sent. VCL, however, supplies an invalidate method but this invalidates all of the component or form and tells Windows to erase the background before painting. This erase and then paint causes excessive flicker. A much better way is to erase only the rect that you want to redraw.

The ellipse sample program has an example of this. When you press the 'Invalidate one' button, a single ellipse is chosen at random and invalidated. This causes Windows to send a paint message and the optimised handler only updates the rect for that ellipse. So if you only need to update a part of your component or form, instead of calling its `Invalidate` method, call the Windows API `InvalidateRect` function instead. Here is the sample code that does this:

```
var InvalidRect : TRect;
begin
 ...
 InvalidRect :=
  TEllipse(
   TempList[Random(
   TempList.Count )]).Rect;
 InvalidateRect(Handle,
  @InvalidRect, true);
```

It's not necessary to copy `Rect` into `InvalidRect` but I did so for clarity. `InvalidateRect` takes three parameters: the first is a window handle which you can get from the form or component's `Handle` property. The second is a pointer to a `TRect`, so remember to prefix it the '@' or you'll get 'Error 26: type mismatch' when you try to compile. The last parameter is a boolean that tells Windows whether to erase the background or not. You should generally set this to true. Setting it to false can produce some unusual effects and is not recommended until you know what you're doing!

Regions

As I said at the start, I'll give you a bonus by including some region handling code. You might notice that I draw a thick frame around the ellipse when you click the 'Invalidate one' button to attract your attention to the area being drawn. The code inverts the frame and then inverts it again when the drawing is finished which restores the screen to its original state, preventing the need for invalidation and repainting.

I achieve this by using a region. This is an area which can be any shape and can include holes and gaps. There are a number of Windows API functions which you use to create a complex region by different combinations of simpler regions. To create the frame effect, first I create a `TRect` that is the size of the outside edge of the frame and use `CreateRectRgn` to create a region from this. I do the same for the inner edge and I have two rectangular regions, one defining the outer edge of the frame and the other defining the ‘hole’ in the middle. I then use the `CombineRgn` function with the `RGN_DIFF` operator which gives me a region which is the difference of the two. This effectively removes the ‘hole.’ I can then invert the region using the `InvertRgn` function. The code is in Listing 2.

Note the use of `try` blocks to protect the allocation of the region resources which Windows will not free automatically, even after the program quits. If you’re not careful when writing Windows code you can end up with severe resource leakage. A good habit to get into is to automatically type a `try` statement on the line following any allocation or operation which you have to undo or tidy up later. I then often type in the `finally...end` with the cleanup code before I even put in the rest of the lines. That way I am sure it won’t be forgotten and it’s easier to follow the indentation!

I use `try...finally` to deallocate the two source regions because they must always be freed whether there is an exception or not. I use the `try...except` block to free up `Result` only when there is an exception. In most cases you should call `Raise` at the end of your `try...except` block to pass the exception back up the stack. On the other hand, you don’t call `Raise` in a `finally` block because the appropriate processing continues anyway. Region functions generally make a copy of any region passed as a parameter so remember to free up the source regions as I have done in the example.

Listing 2

```
function
CreateFrameRegion(const ARect :
  TRect) : HRgn;
var Region1, Region2 : HRgn;
begin
  { creates a “frame” area using
    regions as an illustration -
    also illustrates protecting
    code with try blocks }
  with ARect do begin
    Region1 :=
      CreateRectRgn(Left - 6,
        Top - 6, Right + 6,
        Bottom + 6);
    try
      Region2 :=
        CreateRectRgn(Left, Top,
           Right, Bottom);
      try
        Result := CreateRectRgn(
          0, 0, 0, 0);
        try
          { remove region 2 from
            region 1 and delete
            the source regions }
          CombineRgn(Result,
            Region1, Region2,
            RGN_DIFF);
        except
          DeleteObject(Result);
          Raise;
        end;
      finally
        DeleteObject(Region2);
      end;
    finally
      DeleteObject(Region1);
    end;
  end;
end;
```

One very powerful use of regions is to control the clipping area when painting. Windows allows you to specify your own region where painting will be allowed. In the above example, I could have selected the region as the clipping region and then inverted the whole form with:

```
 InvertRect(Canvas.Handle,
   Rect(0, 0, Width, Height))
```

instead of using `InvertRgn`. The result would have been the same because Windows would limit the invert, or any other drawing operation, to the area defined by the frame region. This is a very powerful technique which you can use to fill or paint complex shapes.

Direct From The Screen And Back...

You may have noticed that I overlay a rectangular red box containing some text along with the inverted frame when the ‘Invert one’ button is pressed. When the frame is removed, so also is the text box but there is no time-consuming paint. I achieve this by creating a temporary memory bitmap the size of the box and using its canvas to copy the area straight off the screen. To remove the box, all I need to do is copy the area back when I’m finished.

Listing 3

```
var ABitmap : TBitmap;
begin
  ...
  ABitmap := TBitmap.Create;
  try
    ABitmap.Width :=
      DestRect.Right;
    ABitmap.Height :=
      DestRect.Bottom;
    { grab the pixels from
      the form’s canvas }
    ABitmap.Canvas.CopyRect(
      DestRect, Canvas,
      SourceRect);
    { ... do whatever you need
      to do ... }
    { & copy pixels back again}
    Canvas.CopyRect(SourceRect,
      ABitmap.Canvas, DestRect);
  finally
    ABitmap.Free;
  end;
```

Performing this update trick is actually very simple. You use the canvas `CopyRect` method which does all the work. All you need to do is create a bitmap, set its width and height and use this method to grab the pixels from the screen. When you’re done you use `CopyRect` in the reverse direction to put the pixels back again and then just free the bitmap. Simple! The code is in Listing 3.

`DestRect` is a `TRect` that defines the area on the form on the screen. In this case the other canvas used in `CopyRect` is that of the form, but it could be any other canvas. Again, I use `try...finally` to make sure `ABitmap` gets freed at the end.

You can use a similar technique to completely banish flicker altogether. Instead of painting straight on to the form’s canvas, you create a bitmap just like the above code fragment, set the width and height to the width and height of the area you’re updating on the form, paint to the bitmap’s canvas and then use `CopyRect` to blast the result straight on to the screen with no trace of flicker at all. Because you are not going across a bus to the screen card with every drawing operation, this technique is often faster than the usual method of writing direct to the form’s canvas. Space does not allow a proper example or full details. That is the topic for another article...!

Mike Scott is a Director of Mobius Software which specialises in Delphi VCL component tool kits and applications and is based in Edinburgh, Scotland. He can be contacted via CompuServe at 100140,2420 (on the internet it’s 100140.2420@compuserve.com), or telephone +44 (0)131-467 3267