THE Delphi MAGAZINE

Issue 1 **April 1995**

Here is your sample on-line copy of the launch issue of The Delphi Magazine! We hope you enjoy it and find it really useful in your Delphi development. The Delphi Magazine is published 6 times a year; each copy comes with a free disk containing all the code from that issue, plus shareware tools and libraries too. We airmail direct to readers in any country in the world; local distributors are also in place in certain countries (call for details).

This on-line document is largely identical to the printed paper copy, except that the ads have been taken out, there's no snazzy cover and we've amended this contents page. Because there are no adverts some pages are missing – we have taken out the blank pages but kept the original page numbering from the paper version, to maintain continuity.

We hope that, having read this free issue, you will decide to subscribe to The Delphi Magazine; the subscription form is on page 41. You can contact us as follows:

The Delphi Magazine, iTec, 41 Recreation Road
Shortlands, BROMLEY, Kent BR2 0DY, United Kingdom

Tel/Fax: 0181 460 0650 (International: +44 181 460 0650)
Email: 70630.717@compuserve.com

Contents

Don't forget the Subscription Form on Page 41!

From The Editor...

Welcome to the first issue of The Delphi Magazine. We've sent you this first issue completely **free of charge** so you can see for yourself how good it is and reach straight for your credit card or cheque book to subscribe! Our next issue will be out in late May, then we move into our regular bi-monthly cycle: July, September, etc, 6 issues a year in total (as well as this free sample).

From Issue 2 onwards, each magazine will come with a **free disk** containing all the source and other files from that issue, plus shareware and public domain libraries and tools. The files from this issue will be on the disk with Issue 2.

Welcome, too, to what is surely a revolution in Windows software development – Delphi itself. I've been using Delphi since summer of last year and for me it is the most exciting new software product for 5 years or more.

OK, there are other 'visual' development environments out there, but not linked to a language with low-level grunt, elegance and ease of learning, with a world-beating true compiler at its back. Personally, I never thought I would enjoy Windows programming in the way I have enjoyed DOS programming. With Delphi, it truly is enjoyable.

Back to The Delphi Magazine. What kind of publication is it? Our aim is to provide solid, practical, technical help to Delphi developers – the people at the sharp end! Quite simply, we want to help you write better Delphi applications more quickly and more enjoyably.

So, in each issue you will find in-depth articles on specific topics – for example in this issue Mike Scott shows how to do faster screen updates and Xavier Pacheco tells us how easy sprite animation is in Delphi.

There will be articles which discuss technologies and new developments, for example Sundar Rajan's introduction to Client/Server, but always with a practical rather than a 'philosophical' leaning!

We have regular columns: Bob Swart's *Under Construction* leads us gently into the fascinating area of Delphi components, and Dave Jewell's *Delphi Internals* looks 'under the hood' of Delphi.

We've got *The Delphi Clinic*, which provides the opportunity for you to have your queries and problems answered by our team of contributors. In the *Tips & Tricks* column you can share your successes with us all! And of course there's product and book reviews and news too.

We are very much aware that readers will have come from different programming backgrounds: some will have been long-time Borland Pascal aficionados and have come through the Beta tests, others will have switched from another language such as Visual Basic.

So, initially, we aim to provide a mix of material for the beginning and more experienced developer. Over time, you will find there may be fewer beginner articles and more at an intermediate or advanced level.

But, we'd like very much to hear from you about what you want to see in the magazine. As we get your feedback and your support (by subscribing!), we can make the magazine better and bigger. If we are not getting the mix right, tell us!

Enough of me, time for you to settle down in a comfortable chair and enjoy a good read...

Chris Frizelle, Editor

News

Mobius Components

Mobius is a British company (well, Scottish really!) which has several sets of native Delphi Components in development. We've seen demos of them all and they are certainly impressive. We will be reviewing each product in detail just as soon as we can (hopefully in the next issue), but to give you a taster here's a rundown:

➢ **Mobius Business Builder**

A suite of standard tables, forms, reports and experts to use as building blocks in business applications. The table classes have standard business rules built into them, so they all 'know' how to work with each other and link automatically. Included classes are customer, sales & purchase orders, despatch notes, sales and purchasing invoices, statements, job costings, inventory, payments and departments/staff.

➢ **Mobius WinG Sprite Kit**

Just about the easiest way to get the incredible speed and power of WinG graphics in your Windows applications. Included are classes for a WingG surface, WingG sprite surface and sprites.

➢ **Mobius Draw Kit**

Provides a set of building blocks for developing drawing and image editing type applications with minimal coding. Features include a draw surface with undo/redo, file handling, toolboxes, printing, etc. A number of standard tools are included such as line, freehand pen, Bezier curve and so on.

By the time you read this, at least one of these products should be available in final release. Mobius is also running an early experience programme for those developers who can't wait to get their sticky paws on them!

Contact Mobius Limited in the UK at 10 Coates Gardens, Edinburgh EH12 5LB, Tel: +44 (0)131 467 3267. In the USA contact Mobius Limited at PO Box 259, Hershey, PA 17033.

Don't forget the Subscription Form on Page 41!

Borland Wins Lotus Suit

It looks like Borland has finally won the 'look and feel' lawsuit brought by Lotus over the Quattro and Quattro Pro spreadsheets (now of course owned by Novell). The US Court of Appeals has reversed a District Court ruling against Borland.

All three judges held in favour of Borland, concluding "Because we hold that the Lotus menu command hierarchy is uncopyrightable subject matter, we further hold that Borland did not infringe the copyright by copying it."

Let's hope that this sees the end of these worthless lawsuits, which make lawyers very rich and bring no benefits to the software houses concerned or their users.

InfoPower

Woll2Woll Software has announced version 1.0 of InfoPower: a set of data-aware native Delphi Components designed for those developing database front-ends. Included are a Super Database Grid, Table Sort, Lookup Combobox, Advanced Filtering, Incremental Search, Auto-Expanding Memo, and a Table Lookup/Locate dialog.

The Super Database Grid can include check boxes, comboboxes etc, and allows multiple tables to be displayed in the same grid. The Lookup Combobox displays one or more columns of data in a drop-down listbox, with incremental search to allow easy locating of specific items. By using the Advanced Filtering component, you can limit the data displayed by any Delphi data-aware component to just what the user needs to see.

We look forward to bringing you more information on InfoPower in a future issue. The bad news is that InfoPower is not due to ship until 1st May. The good news, however, is that Woll2Woll are doing an introductory price of $149 ($50 off the normal price). Contact Woll2Woll by phone from inside the USA on 1 800 WOL2WOL, or from elsewhere on +1 408 293 9369, or alternatively by email at 76207.2541@compuserve.com

Delphi Developers' Group

A number of Delphi 'user groups' have sprung up very quickly and one which many of our readers will be interested in is the British *Delphi Developers' Group*. It provides a newsletter, regular meetings in various locations, a source for code libraries and tools, technical assistance by email or fax, a route for feedback to Borland, seminars and conferences, training and a developers list for those seeking contract help etc.

Meetings are held every other month and include an extended afternoon workshop, during which 'teasers' or practical Delphi problems are set for the members to resolve collectively. This is an interesting way of learning more about Delphi and stretching your programming muscles! After a refreshment break, in the evening regular or guest speakers discuss new developments, applications, Delphi add-ons and utilities. The inaugural meetings held during March were very well attended and seemed to be appreciated by all.

Membership costs £100 plus VAT for one year. More details from Joanna Pooley, Delphi Developers' Group, 8 High Street, Upavon, Wiltshire SN9 6EA, United Kingdom, Tel: +44 (0)1980 630032, Fax: +44 (0)1980 630602, Email: 100016.355@compuserve.com

Do let us know details of any Delphi user groups you are aware of, anywhere in the world, so we can include the information in regular listings.

HighEdit Word Processing

From Heiler Software in Germany comes the High Edit fully programmable word processor, available either as a VBX or a DLL. It features a true WYSIWYG display which makes use of all available printer and TrueType fonts, font attribute

Looking For Information?

One of the great things about Delphi is its ability to use VBX custom controls (Version 1.0), as well as native Delphi components and DLL-based controls. There are, however, a large number of companies who market VBX and other controls and it can sometimes be difficult to get hold of the right information from the right source – for example on Delphi compatibility. There are two useful CompuServe fora which have been set up for vendors of Windows components/controls: COMPA and COMPB. Some of the companies involved are:

- A&G Graphics (GO COMPA or GO SPEECH)
- Aardvark Software, Inc. (GO COMPA or GO AARDVARK)
- APEX Software Corporation (GO COMPA or GO APEX)
- Avanti Software (GO COMPA or GO AVANTI)
- Bennet Info Systems (GO COMPA or GO BTIS)
- Crescent Software (GO COMPA or GO CRESCENT)
- Desaware (GO COMPA or GO DESAWARE)
- Foxhall Publishing (GO COMPA or GO VBZFORUM or GO FOXHALL)
- ImageFX (GO COMPA or GO IMAGEFX)
- London Software (GO COMPB)
- Mabry Software (GO COMPA or GO MABRY)
- Media Architects (GO COMPA or GO MEDARCH)
- MicroHelp (GO COMPA or GO MICROHELP)
- Sax Software (GO COMPA or GO SAXSOFT)
- Sheridan Software Systems (GO COMPA or GO SHERIDAN)
- Stylus Innovation (GO COMPA or GO STYLUS)
- SuccessWare International (GO COMPB)
- The Young Software Works (GO COMPB)
- VideoSoft (GO COMPA or GO VIDEOSOFT)
- Viewpoint Technologies (GO COMPB)
- Teknowledge (GO COMPB)
- VisualTools (GO COMPA or GO VISTOOLS)

Thanks to Robert Scoble of Fawcette Technical Publications for this information.

control, full printing control, margins, tabs, paragraph formatting, search and replace, clipboard support, optional display of control characters, multiple document support, auto word-wrap, import/export in native, RTF or ASCII formats, graphics import in EMF, PCX, TARGA, TIFF, GIF or WMF formats, headers and footers and more!

One other feature which looks really useful is input fields, which promise to make things like form letters and mail-merge easier. Dear Heiler, can we have a native Delphi Component version please...?!

HighEdit is available in the United Kingdom from Bits Per Second at 14 Regent Hill, Brighton, East Sussex BN1 3ED, Tel: +44 (0)1273 727119, Fax: +44 (0)1273 731925.

Delphi Communications From TurboPower

As well as their Orpheus product, containing lots of native Delphi components (see the ad in this issue – to be reviewed soon!), TurboPower from Colorado, USA, are working on Delphi components for their acclaimed Async Professional for Windows communications library (APRO). When they are ready, we understand the Delphi components will be downloadable from CompuServe or the TurboPower BBS for existing APRO users (you'll need APROto use them). Fax support is also due to be added soon.

We understand TurboPower are also working on other exciting Delphi projects and we'll let you know more just as soon as we can. For details Tel: +1 719 260 9136, Fax: +1 719 260 7151, or email: 76004.2611@compuserve.com

Client/Server Load Testing

An important aspect of any development cycle has to be software testing, but how much does your new baby get really *stressed* before you ship it to the client?

Empower/CS from Performix is designed to capture and replay the activities of Client/Server applications running under Windows, stressing a database server by running multiple scripts from one driver to simulate the load of multiple PCs. In this way you can test your application with an effective load of hundreds or thousands of users with just one machine. Response times are measured and logged for analysis. Sounds ideal for thoroughly testing your new Delphi C/S apps!

Contact Performix in the United Kingdom at The Atrium Court, Apex Plaza, Reading, Berkshire RG1 1AX, Tel: +44 (0)1734 795016, Fax: +44 (0)1734 795017, email: sacha@performix.com. In the USA Performic Inc are at: 8200 Greensboro Drive, Suite 1475, McLean, Virginia 22102, Tel: +1 703 448 6606, Fax: +1 703 893 1939.

Information Please!

If you have products or services relevant to Delphi developers we want to hear about them! Send your information to us at the address on page 42 (or by fax or email), marked for the attention of the Editor.

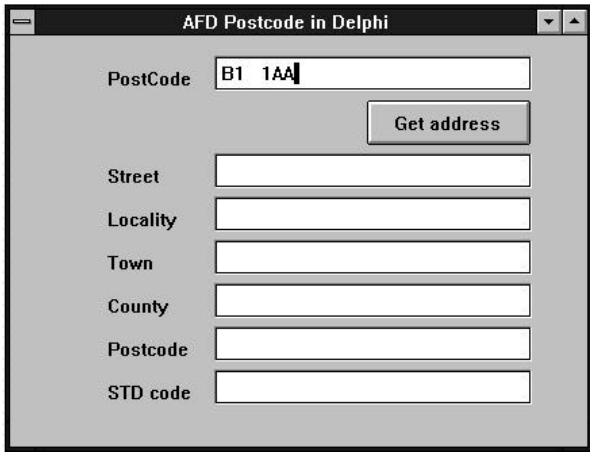
AFD postcode

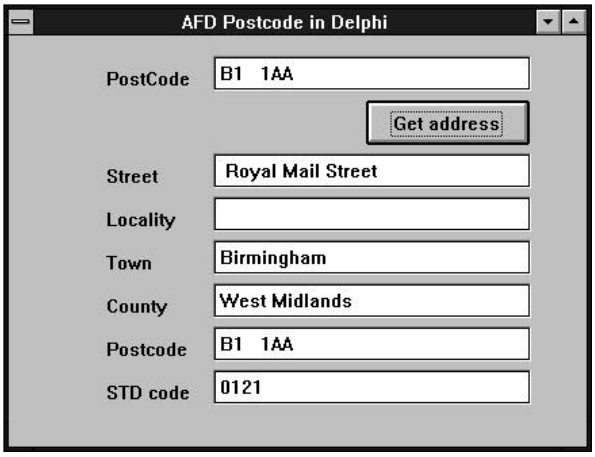
From AFD Computers comes a neat DLL-based package which all developers creating applications which capture addresses in the United Kingdom will find useful.

These screenshots demonstrate the postcode DLL in use in a simple Delphi example application. You just grab a Postcode from an edit box, as shown in the top screenshot, then make one call to the DLL, which returns the remainder of the address as shown in the bottom screenshot, plus the STD telephone area code as a bonus. All you then need to do is fill in the house number in the Street field and you have a complete address which the Royal Mail are guaranteed to like! In the example, which took me a few minutes to knock up, the call to the DLL is triggered by the user clicking the *Get address* button.

This tool is designed for developers' use and what makes it unique is the sensible pricing: between £75 and £15 per end-user (ie your client's users) for the software licence and an annual licence fee for the Postcode data which depends on the number of end users: for 1 user with no updates it's £55, but very good deals are available for site or company licences. Competing products are usually very much more expensive and out of the range of the small developer. The advantage for your users is that they save time and hence money when punching in address details.

Contact AFD Computers at 51 Meadowfoot Road, West Kilbride KA23 9BU, United Kingdom, Tel: +44 (0)1294 823221, Fax: +44 (0)1294 822905. An evaluation version (PCEVAL.EXE) can be downloaded from the CompuServe MS BASIC forum or from AFD's bulletin board on +44 (0)1294 829327 (to 19200 bps).





Don't forget the Subscription Form on Page 41!

The Delphi Idiom

by Steve Teixeira

Now you have Delphi in your hot little hands you can understand what the big deal has been about for the past several months! Delphi can handle practically any application development task you throw at it – and handle it faster and more elegantly than most any other tool. But you probably already know that or you wouldn't be here, right? I'm here to tell you what you need to know to be a Delphi programmer.

It's Not Just For Prototyping

Like most products of its genre, Delphi gives you a simple, productive point-and-click interface to program creation. Unlike with some products, though, your application doesn't have the silhouette of a cow nor the speed of a sloth. Delphi combines the advantages of visual development with the performance of a true compiler.

Borland didn't cook up the Delphi compiler out of the blue, though, it's an improved version of the compiler in Borland Pascal 7. So, although Delphi is a new product, it's an evolution of one of the oldest and most trusted PC compilers.

The code generated when you compile a program in Delphi is on a par in terms of speed with that generated by a C or C++ compiler. In short, Delphi isn't a prototyping or "front-end" tool, but an all-round tool. Using it only for prototyping would be like buying a Ferrari and never pulling it out of your drive!

It's Pascal

If you're new to Delphi, you should find it easy moving up to Object Pascal. It's a sort of happy medium of languages, so whether you prefer the structure and verbosity of Basic or the flexibility and power of C++, you'll soon feel at home.

One language feature that can take some getting used to is Object Pascal's strongly typed nature. When you call a procedure, the compiler will make sure that you pass correct types, array sizes and pointers. If you come from a C/C++ background, this will probably annoy you at first, but I'll bet you will soon appreciate the number of bugs it prevents from entering your code. After all, wouldn't the ideal be to have the compiler find all your bugs? Strict type-checking is a step in that direction.

If you're moving to Delphi from Borland Pascal, nearly 100% of your code will compile without problem in Delphi. I've found the best approach to porting an Object Windows Library (OWL) application to Delphi is to redo the User Interface portions in Delphi's IDE, and hook the new UI into your application's existing back-end.

Visual Component Library

Visual Component Library, or VCL, is Delphi's object-oriented framework. In this rich library, you'll find classes for Windows objects such as windows, buttons, etc, and you'll also find classes for custom controls such as gauge, timer and multimedia player, along with non-visual objects such as string lists, database tables, and streams.

Each VCL class usually has a set of properties – such as color, size, position, caption – that can be modified in the Delphi IDE or in your code, and a collection of events – such as a mouse click, keypress, or component activation – for which you can specify some additional behavior. You'll spend most of your time in the Delphi IDE interacting with components' properties and events.

VCL is also remarkably platform-independent. VCL encapsulates even low-level Windows concepts such as Device Contexts, bitmaps, and timers. It is for this reason that the code you write in 16-bit Delphi will recompile with little or no changes in 32-bit Delphi when it is released. It's best to keep with VCL as much as possible to give your code maximum portability. You'll be surprised how much you can do without calling the Windows API.

Message Handling

Although VCL's events account for most of your needs, directly handling Windows messages is a piece of cake with Delphi's new `message` keyword. Simply create a method that takes one parameter of type `TMessage` (or other message record), and use the magic word. Let's say, for example, you want to write a handler for the `wm_Paint` message. The code would look like:

```
procedure WMPaint(
  var M: TWMPaint);
  message wm_Paint;
```

`TWMPaint` is a record type based on a `TMessage` whose fields are defined specifically for a `wm_Paint` message. Delphi defines a record type like this for every Windows message. The record type is always the same as the message name with a "T" in front and without the underscore.

Windows At Your Fingertips

Unlike some similar products, Delphi offers you the flexibility to easily call any Windows API procedure. Actually, Delphi doesn't just support this feature, but it makes it a trivial task: you just call the API procedure as if it were a procedure defined in your program.

Delphi also allows you to call procedures out of any other DLL, no matter what language it's written in. Although Windows DLLs generally use the Pascal calling convention for parameter passing, Delphi supports the C calling convention too: just use `cdecl` on your function declaration.

Exception Handling And Runtime Type Information

You can handle error conditions in your Delphi code using C++-like Exception handling. Exception handling enables you to gracefully

handle specific or general error conditions by enclosing potentially dangerous parts of your code in a `Try..Except` or `Try..Finally` block. The general structure of an exception handling block looks like this:

```
try
  {some stuff }
except
  {if an exception occurs... }
  on ESomeException do
    Something;
    {Handle the exception }
end;
```

Exceptions provide a significant advantage over general error procedures in that protection can be placed where it's needed in your code. Instead of trying to handle a whole variety of possible error conditions in one place, you can tailor `Try` blocks to your needs. It's also built into the Win32 API, so you'll need to get used to it!

Runtime Type Information (RTTI) is the ability to obtain information on class instances while your program is running. RTTI is perhaps the single most important feature in Object Pascal. In VCL, most classes are passed between functions and procedures as the `TObject` base class (from which all classes are derived), which satisfies the compiler's type-checking, and RTTI is used inside functions and procedures to determine the type of and typecast these classes.

Client/Server

Delphi comes with the Borland Database Engine, a high-performance database-access layer that transparently connects you to different data sources: Paradox, dBASE, ODBC, or servers like Oracle, Interbase, Sybase, and Informix. The buzzword here is scaleability: you can start off with Paradox or dBASE tables and then transparently move to Oracle or Sybase with very little change in your application. Back-end independence means productivity, and it is the wave of the future.

Component Design

Delphi was written in Delphi. Delphi components are written in Delphi. That's not to say that Delphi is xenophobic – Delphi also allows you to use VBX controls and other Windows custom controls. Simply stated, Delphi does it all, and because of that, you can do it all. There's no line between the application developer and the component writer. All components are extensible Object Pascal classes, so you can crank out a custom component any time you need to.

Power

Because Delphi is a true compiler, you have no limits. Need to write a DLL? No problem, it's hardly any different to writing a regular unit. Callback functions? Not a problem either. Just tag that procedure with the `export` directive and you're on your way. How about inline assembly language? Sure, Delphi's Built-in Assembler makes it a snap.

Whether you want to write applications, controls, or database front-ends, Delphi is your tool. Now that you know the score, what are you waiting for? Go forth and hack.

Steve Teixeira is a Senior Technical Support Engineer for Delphi at Borland International. He can be reached via the internet at steixeira@wpo.borland.com or via CompuServe at 74431,263

Don't forget the Subscription Form on Page 41!

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*

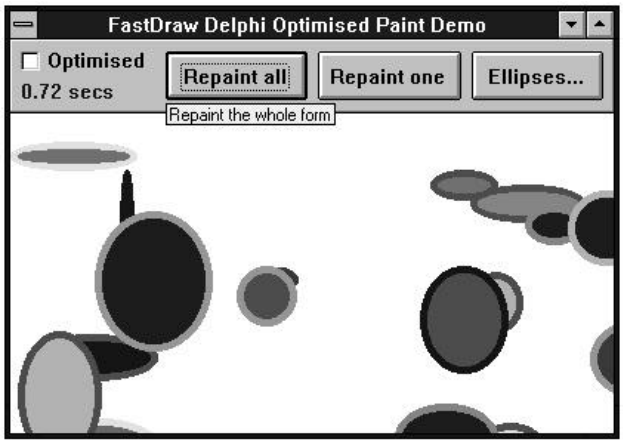
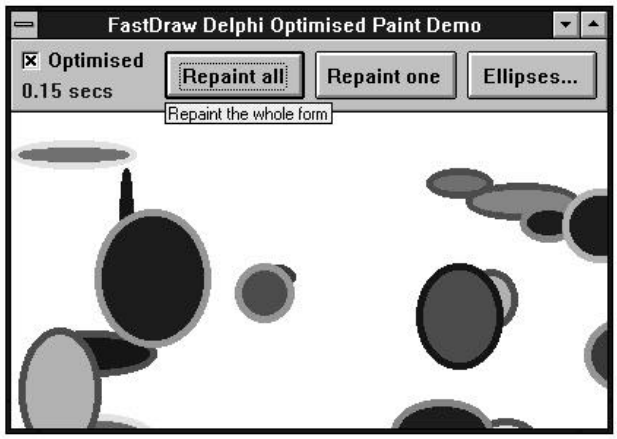


Figure 2

*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.

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.

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;
```

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.

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...!

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;
```

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

Don't forget the Subscription Form on Page 41!

Under Construction: Build Your Own Components

by Bob Swart

While Delphi is a great tool for Client/Server and Rapid Application Building, I think the most important feature of Delphi is the ability to write components and add them to the component palette of the environment itself. I strongly believe that a Delphi Component is The Object of the '90s. In fact, I believe so strongly in Delphi Components, I've asked the Editor to let me devote a regular column to Component Building: Under Construction!

In this column, I will show you how to create new visual (interface) and non-visual (engine) components for Delphi. The complete source code of all components will always be available on the accompanying disk of The Delphi Magazine (available when you subscribe!).

Examples for components that will be built in the coming months include a Date component, a small Agenda Component, a right-aligned Edit control, Spin buttons, a Tic-Tac-Toe game and a file UUEncode/UUDecode component. This time, we mainly focus on non-visual components to get a good grasp on the underlying component class architecture.

OOP

Object Oriented Programming is based on (and extends) established ideas of Structured Programming, and involve three basic principles: encapsulation, inheritance and polymorphism. Encapsulation is the concept of placing data and routines that operate on that data together and combining them to create a structure (object) that contains both. Inheritance is the concept of deriving new objects from existing objects. This is the main feature that leads to re-use of existing code. Finally, polymorphism is the concept that causes different types of objects derived from the same parent object to be able to behave differently when instructed to perform a method with the same name but a different implementation.

Although OOP has always been related to claims of code re-use and faster development cycles, in practice this has proved more often false than not. Two main problems with OOP code-reuse are determining and finding which object(s) to use for a particular problem, and organising these objects in a usable and accessible architecture. Delphi solves these problems by placing the re-usable objects or components in a structured Component Palette, where components can be logically grouped together by type. Furthermore, with Delphi we are able to develop components and applications in the same environment.

Throughout this first column, we'll see that Delphi Components truly support OOP and component re-use!

Component Building

Delphi and Delphi Components are built upon the Visual Component Library (VCL) application framework. VCL is already a very rich framework, which becomes clear if we take a look at Delphi's Component Palette: dozens of standard Windows controls like edit boxes, static lines, combo and list boxes, but also several advanced custom controls like grid controls, tab controls, notebook controls and an outliner. The list goes on and on, and will go on and on, since Delphi includes the ability to include new components in the Palette!

Creating a new component requires writing a .PAS unit file containing the source code of the component, which will be compiled to a .DCU file. Additionally, we could include a .DCR palette bitmap (a renamed .RES file with a 28x28 bitmap with the same name as the component, to appear in the component palette), a .HLP help file and a .KWF keyword file. For now, however, we will concentrate on the .PAS source file.

Before we can start writing a component ourselves, we have to identify the relevant classes from VCL. The following class hierarchy shows the most important seven classes for this task:

```
TObject
  TPersistent
    TComponent
      TControl
        TGraphicControl
        TWinControl
          TCustomControl
```

VCL is no longer based on the old object model of Borland Pascal, where Virtual Method Tables (VMTs) are stored in the Data Segment. Instead, VCL is based on a new class model (with the emphasis on `class`).

In this new model, all object instances are automagically dynamically allocated on the heap. Automagically, because Delphi now assumes that each class we reference is in fact a pointer to that class. It is no longer necessary to explicitly declare a pointer type or to use the dereference symbol (^). This greatly reduces the syntax complexity of ObjectPascal.

Every class in VCL is derived from the `TObject` root. The class `TObject` contains the `Create` and `Destroy` methods that are needed to create and destroy instances of classes. The class `TPersistent`, derived from `TObject`, contains

methods for reading and writing properties to and from a form file.

TComponent is the class to derive all components from, as it contains the methods and properties that allow Delphi to use TComponents as design elements, view their properties with the Object Inspector and place these components in the Component Palette. If we want to create a new non-visual component from scratch, then TComponent is the class we need to derive from.

Visual component classes are derived from the TControl class, that already contains the basic functionality for visual design components, like position, visibility, font and caption. Derived from TControl are TGraphicControl and TWinControl. The difference between a TGraphicControl and a TWinControl is the fact that a TWinControl contains an actual Window Handle, while a TGraphicControl does not. Therefore, derived from a TWinControl we will find classes like the standard Windows controls, while controls like TBevel, TImage, TSpeedButton and TShape are derived from TGraphicControl. Finally, the class TCustomControl is much like both TWinControl and TGraphicControl together.

In this first column we will primarily focus on non-visual components, derived from the TComponent base class.

Properties, Methods And Events

Components consist of encapsulated properties, methods and events. Properties are slots that give the component user the illusion of reading or writing the value of a variable in the component, while the component writer can use properties to hide the implementation details.

In a sense, properties are the user interface to a component. Methods are procedures and functions that are encapsulated with properties in a component. Events are like reactions (event handlers) to messages (events) that occur during execution of the component. Examples of events are OnClick and OnEnter events. Both methods and events can be made dynamic, which gives us the polymorphic ability of component classes.

Properties have to be read and written, and hence contain a read (Get) and (optional) write (Set) method. The Get method is a function that returns the property value, while the Set method is a procedure that takes as parameter the new property value. The Set property method makes a great place to include some data validation rules. An example of a property Day declaration is:

```
private
  FDay: Word;
protected
  function GetDay: Word;
  procedure SetDay(
    NewDay: Word);
published
  property Day: Word
    read GetDay write SetDay;
```

Where GetDay and SetDay are property methods that have to be implemented in the implementation section of the unit.

The new keyword private leaves the internal field FDay only visible to the current instances of the same class. The keyword protected ensures us that only classes derived from the current class can call or override the property methods GetDay and SetDay. The new keyword published tells Delphi that the property Day should be visible in the Object Inspector.

First Example

Enough talk for now, let's start with a simple non-visual component derived from TComponent that encapsulates the current system date.

This component will consist of three properties: Day, Month and Year, which will be read-only, since we wouldn't want to change the system date this way.

Note that each component can be placed in its own unit (which I think is a good convention), so we get component Date1, as shown in Listing 1.

As we can see, we have a lot of new things here. The class TComponent from which we derive our TDate1 from must be obtained from the unit Classes, hence the uses Classes statement. I used three internal fields to hold the values of the three properties Day, Month and Year. The naming convention is to give property field names a prefix "F" (hence FDay, FMonth and FYear).

I supply the three properties with only read methods, which are in fact the internal fields themselves. In general, if the read or write method is just a reference to an internal field itself, it's much faster to access the field (in this case FDay, FMonth or FYear) itself, rather than to call a method that returns this internal field.

The initialisation of the TDate1 component is done in the constructor Create. I used the SysUtils global variable Date (SysUtils.Date, so I also need to use the SysUtils unit) to initialise FYear, FMonth and FDay with the current system date.

Finally, in order to add this little component to the Delphi's Component Palette, we have to play by the rules and register ourselves with

Listing 1

```
unit Date1;
interface
uses Classes, SysUtils;
Type
  TDate1 = class(TComponent)
    private
      FDay:   Word;
      FMonth: Word;
      FYear:  Word;
    published
      property Day: Word
        read FDay;
      property Month: Word
        read FMonth;
      property Year: Word
        read FYear;
    public
      constructor Create(AOwner:
        TComponent); override;
  end {TDate1};
  procedure Register;
implementation
  Constructor TDate1.Create(
    AOwner: TComponent);
  begin
    inherited Create(AOwner);
    DecodeDate(SysUtils.Date,
      FYear,FMonth,FDay);
  end {Create};
  procedure Register;
  begin
    RegisterComponents('Dr.Bob',
      [TDate1])
  end {Register};
end.
```

Delphi using the procedure `Register`, which is used by Delphi to determine if a component is present in a unit. If a `Register` procedure is found, Delphi expects it to register all components that exist in the unit, by calling `RegisterComponents`, with the name of the Component Palette (Dr.Bob in this case) as the first argument and as the second argument a list of the component types that are included in this unit (`TDate1`).

Second Example

If we install the component `TDate1` in the Component Palette (using `Options|Install`), and drop it on a new form, we don't see the properties Day, Month or Year in the Object Inspector. Why is that?

It seems that only read/write properties are used in the Object Inspector – after all, if we can't change the value of a property, what use would displaying its value have? I disagree, and I would like to find a way to view read-only properties as well!

Of course, I could use a `SetDay` method that does nothing, but it seems an awful overhead to use this just to view the property in the Object Inspector.

Another quick-and-dirty way to show the property is to define a 'dummy' field with the same type as the property `Day` and assign this dummy to the write property. We can even re-use this dummy field for `Month` and `Year` (as they are all of the same type).

With the addition of a new enumerated property called `DayOfWeek`, with values a set of `Sunday..Saturday`, this yields the code for component `Date2` shown in Listing 2.

If we compile and install this component in the Component Palette and try to view the properties with the Object Inspector they're visible!

At first, it seems we can actually change the values of `Day`, `Month`, `Year` and `DayOfWeek`, but as soon as we leave the field in the Object Inspector, the old value pops back. Note that `Dummy` itself is not published and hence not visible in the Object Inspector.

Third Example

As small and elegant as it seems, a read-only `TDate` component that only contains the current date is not very useful. It's time to make it a read/write component.

If we do this, we face the problem of updates on one property that may have impact on other properties as well. What if we change the `Day` property value? Surely, the `DayOfWeek` will change, too. The same with `Month` or `Year`.

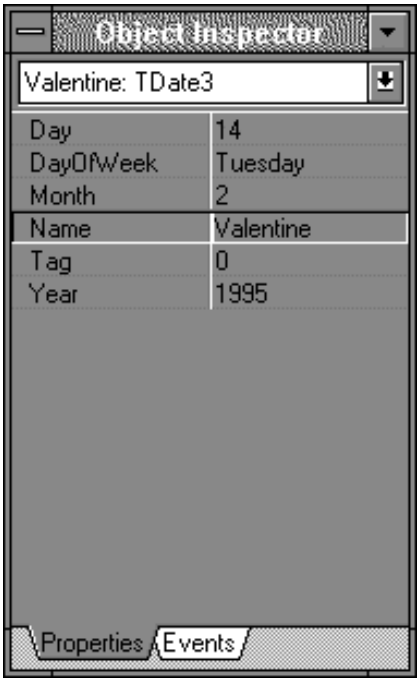
Therefore, we keep `DayOfWeek` as a read-only property, and recalculate the values of `Day`, `Month`, `Year` and `DayOfWeek` immediately after receiving a new value of either `Day`, `Month` or `Year`. Furthermore, the current date is stored in an internal (private, hence invisible) field called `FInternalDate`. The component `Date3` is shown in Listing 3.

Now, if we check out the component in the Object Inspector we can change the values of `Day`, `Month` and `Year` while the value of `DayOfWeek` is immediately updated along the way, as shown in Figure 1.

The component `TDate3` is small, but illustrates that even small components can still be very powerful.

It isn't over, however. In fact, component building is only limited by your own imagination. What else would we need? I would like a

Figure 1
TDate3 Properties



Listing 2

```
unit Date2;
interface
uses Classes, SysUtils;
Type
  TDayOfWeek = (Sunday, Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday);
  TDate2 = class(TComponent)
    private
      FDay:   Word;
      FMonth: Word;
      FYear:  Word;
      FDayOfWeek: TDayOfWeek;
    private
      DummyWord: Word;
      DummyTDayOfWeek: TDayOfWeek;
    published
      property Day: Word read FDay write DummyWord;
      property Month: Word read FMonth write DummyWord;
      property Year: Word read FYear write DummyWord;
      property DayOfWeek: TDayOfWeek read FDayOfWeek write DummyTDayOfWeek;
    public
      constructor Create(AOwner: TComponent); override;
  end {TDate2};
  procedure Register;
implementation
  Constructor TDate2.Create(AOwner: TComponent);
  begin
    inherited Create(AOwner);
    DecodeDate(SysUtils.Date, FYear,FMonth,FDay);
    FDayOfWeek := TDayOfWeek(Pred(SysUtils.DayOfWeek(SysUtils.Date)));
  end {Create};
  procedure Register;
  begin
    RegisterComponents('Dr.Bob', [TDate2])
  end {Register};
end.
```

property that gives me the date as a string, in the DD/MM format. But perhaps other people would like to get the DateString in YY/MM/DD or MM/DD/CCYY format. This could be implemented by introducing two new properties: the read-only DateString (but visible in the Object Inspector, hence with a DummyString field as well) and the read/write property DateFormat that contains examples like DD/MM, MM/DD, DD/MM/YY and so on. Depending on the value of DateFormat, the property DateString would use the Day, Month and Year properties to generate the correct value.

Listing 3

```
unit Date3;
interface
uses SysUtils, Classes;
Type
  TDayOfWeek = (Sunday, Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday);
  TDate3 = class(TComponent)
    private
      FInternalDate: TDateTime;
      FDay:   Word;
      FMonth: Word;
      FYear:  Word;
    protected
      procedure SetDay(NewDay: Word);
      procedure SetMonth(NewMonth: Word);
      procedure SetYear(NewYear: Word);
      function GetDayOfWeek: TDayOfWeek;
    private
      DummyTDayOfWeek: TDayOfWeek;
    published
      property Day:   Word read FDay   write SetDay;
      property Month: Word read FMonth write SetMonth;
      property Year:  Word read FYear  write SetYear;
      property DayOfWeek: TDayOfWeek read GetDayOfWeek
        write DummyTDayOfWeek;
    public
      constructor Create(AOwner: TComponent); override;
  end {TDate3};
  procedure Register;
implementation
  Constructor TDate3.Create(AOwner: TComponent);
  begin
    inherited Create(AOwner);
    FInternalDate := SysUtils.Date;
    DecodeDate(SysUtils.Date,FYear,FMonth,FDay);
  end {Create};
  procedure TDate3.SetDay(NewDay: Word);
  begin
    if (NewDay in [1..31]) then begin
      { Ok, so we could use some more date checking here. Volunteers? }
      FDay := NewDay;
      FInternalDate := EncodeDate(FYear,FMonth,FDay);
    end;
  end {SetDay};
  procedure TDate3.SetMonth(NewMonth: Word);
  begin
    if (NewMonth in [1..12]) then begin
      FMonth := NewMonth;
      FInternalDate := EncodeDate(FYear,FMonth,FDay);
    end;
  end {SetMonth};
  procedure TDate3.SetYear(NewYear: Word);
  begin
    FYear := NewYear;
    FInternalDate := EncodeDate(FYear,FMonth,FDay);
  end {SetYear};
  function TDate3.GetDayOfWeek: TDayOfWeek;
  begin
    GetDayOfWeek := TDayOfWeek(Pred(SysUtils.DayOfWeek(FInternalDate)));
  end {GetDayOfWeek};
  procedure Register;
  begin
    RegisterComponents('Dr.Bob', [TDate3])
  end {Register};
end.
```

This component is not included here, nor is it on the code disk. I have included the DateString property in the Date component on the disk, but I invite you to experiment yourself with the Date and other components. Once you've started, you'll see that the sky is the limit!

A Real Example

Enough about dates, let's talk about another (real world!) non-visual example. A great way to use non-visual components is as converters.

Has anyone read the old book on *The C Programming Language* by K&R? (Personally, I expect a lot of C/C++ programmers to move over to Delphi). Remember the Fahrenheit-to-Celsius conversion programs on page 8 of this book? This functionality can be put in a Delphi component very easily!

Imagine a converter component that holds a value. You have three (or more) properties with which to access that value: Decimal (most useful), Hexadecimal and Roman.

The crux is that these properties are in fact all the same, as they all correspond to the internal state of the component itself. If I do a SetDecimal of 42, the GetHex will automatically return 002A, as will the GetRoman return XLII. Setting the Roman property to VII will likewise yield a Decimal (and Hex) value of 7. Instant conversions!

You never have to write them again, and you also don't have to look for them, as they are always right here on your Component Palette, just a mouse-click away! The code is in Listing 4.

Because the properties are all read/write, we can test the component using the Object Inspector (another reason why we would want read-only properties to be visible in the Object Inspector: easier testing). Setting one property automatically updates the other properties (see Figure 2).

One more thing about this component. Statistically, a Get property method will be called more often than a Set property method (or at least I think so). This means that the Get methods are the first to optimise when we have a

component with one internal state and several properties that return a translation of that state.

Therefore, I've written the procedure SetRoman using BASM (Built-in ASseMbler) and the GetRoman in plain ObjectPascal. Just a minor detail, but it might be a consideration whenever you're planning on expanding this component or write other state conversion components yourself.

What's On Your Agenda?

So, we have just built two non-visual components: a nice TDate and a handy TConvert component. What can we do with them? Well, we could use them to write a third component, this time a visual one.

Derived from TMemo, a standard VCL class, I created TAgenda. TMemo is just a big visual edit field with optional word wrapping and scrollbars. The lines of a TMemo (and hence also our derived TAgenda) can be easily accessed by the Lines property, which is a component itself. The Lines component has two important methods: Clear to clear all lines, and Add which takes a string argument, and just adds a line to the collection of lines.

The TAgenda component modifies the TMemo component by setting the property ScrollBars to ssVertical, meaning we always want to have a vertical scrollbar.

Other than that, we only include an instance of TDate (with the current date) and a string field called Agenda. This last field can be assigned a filename, of which parts will be shown in the memo field. An Agenda file for TAgenda consists of lines with the following format (the DD/MM field starts in column 1):

```
DD/MM This meeting is important
DD/MM Another thing to do
```

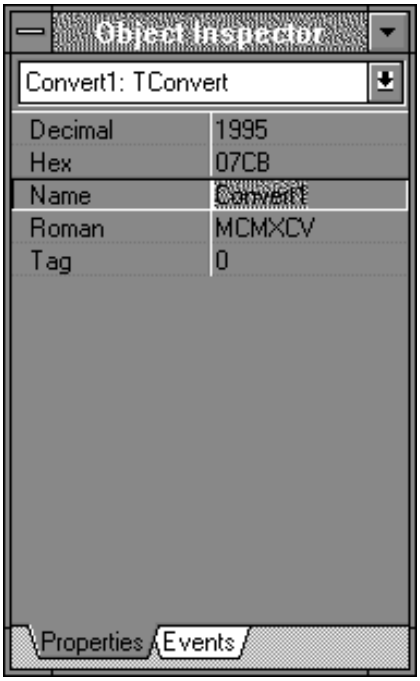
Only lines where the DD/MM part equals the current date (that is where the DD/MM equals the TDate.DateString property) are displayed. The result is a little agenda that shows all your appointments for today! The code is in Listing 5.

As you see, the TAgenda component can be activated at design time if we give the property Agenda a valid value (ie a filename that exists). The TAgenda memo field will contain the current date (note the year in Roman digits) and all lines from the Agenda file that have the current date at position 1. Since not all people will use MM/DD, this is precisely the reason why you may want to enhance the TDate component we designed earlier. To see the TAgenda component in action, just take a look at the screen shot on the cover!

Listing 4

```
unit Convert;
interface
uses SysUtils, Classes;
Type
  TConvert = class(TComponent)
    private
      FValue: Word;
    protected
      function GetHex: String;
      function GetRoman: String;
      procedure SetHex(Const Value: String);
      procedure SetRoman(Const Rom: String);
    published
      property Decimal: Word read FValue write FValue;
      property Hex:   String read GetHex write SetHex;
      property Roman: String read GetRoman write SetRoman;
    public
      constructor Create(AOwner: TComponent); override;
  end {TConvert};
  procedure Register;
implementation
  constructor TConvert.Create(AOwner: TComponent);
  begin
    inherited Create(AOwner);
    FValue := 0;
  end {Create};
  function TConvert.GetHex: String;
  Const Digits: Array[0..$F] of Char = '0123456789ABCDEF';
  begin
    GetHex[0] := #4;
    GetHex[1] := Digits[Hi(FValue) SHR 4];
    GetHex[2] := Digits[Hi(FValue) AND $F];
    GetHex[3] := Digits[Lo(FValue) SHR 4];
    GetHex[4] := Digits[Lo(FValue) AND $F];
  end {GetHex};
  procedure TConvert.SetHex(Const Value: String);
  var code: Integer;
  begin
    if (Value[1] <> '$') then
      Val('$'+Value,FValue,code)
    else
      Val(Value,FValue,code)
  end {SetHex};
  { see the Issue 2 disk for source of GetRoman and SetRoman
   -- too long to include here }
  procedure Register;
  begin
    RegisterComponents('Dr.Bob', [TConvert])
  end {Register};
end.
```

Figure 2
TConvert Properties



Don't forget the Subscription Form on Page 41!

Next Time

The second Under Construction article will be about components encapsulating a DLL (even if you have no source code), putting more Windows controls than just one in a new visual component (in other words not just deriving a component from an existing component like our TAgenda), and how to make little games with Delphi, all resulting in a Tic-Tac-Toe game component.

Also next time, we'll learn how to create .DCR files with our new components, so we no longer have all the same default bitmaps on the Component Palette.

Bob Swart (email: CompuServe 100434,2072) is a freelance technical author and professional programmer for Bolesian BV in The Netherlands.

Listing 5

```
unit Agenda;
interface
uses SysUtils, Classes, StdCtrls, Date, Convert;
Type
  TAgenda = class(TMemo)
    private
      FAgenda: String;
      FDate:   TDate;
    protected
      procedure SetAgenda(Const FileName: String);
    published
      property Agenda: String read FAgenda write
SetAgenda;
    public
      constructor Create(AOwner: TComponent); override;
  end {TAgenda};
  procedure Register;
implementation
  constructor TAgenda.Create(AOwner: TComponent);
  begin
    inherited Create(AOwner);
    ScrollBars := ssVertical; { set vertical scrollbars }
    FDate := TDate.Create(self);
  end {Create};
  procedure TAgenda.SetAgenda(Const FileName: string);
  var f: System.Text;
      Str: String;
      FConvert: TConvert;
  begin
    {$I-}
    System.Assign(f,FileName);
    Reset(f);
    if IOResult = 0 then begin
      FConvert := TConvert.Create(self);
      FConvert.Decimal := FDate.Year;
      FAgenda := FileName;
      Lines.Clear; { clear contents }
      Lines.Add(FDate.DateString + ' ' + FConvert.Roman);
      Lines.Add('——-');
      while not eof(f) do begin
        readln(f,Str);
        if System.Pos(FDate.DateString,Str) = 1 then
          Lines.Add(Str)
      end;
      FConvert.Destroy;
      System.Close(f)
    end
    {$I+}
  end {SetAgenda};
  procedure Register;
  begin
    RegisterComponents('Dr.Bob', [TAgenda])
  end {Register};
end.
```

Don't forget the Subscription Form on Page 41!

Moving Up: Borland Pascal

by Dave Jewell

In this article, Dave looks at some of the issues involved in moving to Delphi from Borland Pascal

Not including Delphi itself, there are four different Windows programming languages which might be termed 'mainstream' development environments. These are C, C++, Pascal and Visual Basic. Yes, I know that C++ is a superset of C, but it's rather a dangerous over simplification to lump C and C++ together. I've heard it said recently that programmers can pick up C++ more easily if they haven't had any previous exposure to C, and from my own experience there's certainly some truth in that point of view!

In the coming months, I'll be discussing Delphi from the viewpoint of a developer coming from one of these four camps. This time we kick off with Pascal and in the next issue we'll be looking at Delphi from the viewpoint of a Visual Basic Developer.

Delphi For Pascal Programmers

You might find it surprising that I've included Pascal in the above list. After all, Delphi is just Pascal hiding behind a pretty user interface isn't it ? Well, no – not really.

For starters, Delphi uses Borland's Object Pascal, an object-oriented Pascal dialect that offers much of the power of C++ in a far simpler, more manageable language. If you happen to be a seasoned Borland Pascal developer, and know the Windows API like the back of your hand, then you're in good shape for getting into Delphi. You'll find, however, that Borland have made a number of changes and enhancements to the language, making the new Pascal dialect even more powerful than it was previously.

If you're at all familiar with the Borland Pascal development system, you should have little difficulty in getting to grips with Delphi. Beneath Delphi's friendly visual development environment lurks the same compiler that you're familiar with. If you want to merely use Delphi as a "straight" Pascal compiler, there's nothing to stop you doing so. You can use it to compile and build your existing Pascal projects. However, it goes without saying that this approach misses out on all the major productivity benefits that come from using a visual development tool and the feature-rich component library.

This article is primarily concerned with the language changes that Borland incorporated into Delphi's particular version of the Pascal compiler. Although backwards-compatible with existing code, the new compiler incorporates a number of important language enhancements that we'll be looking at here.

New Language Features

With Delphi, Borland have introduced a number of new language features, many of which relate to the Object Browser interface. These language features generate additional categories of run-time information which are read by the Object Browser and used to fill in the browser window with properties and events that relate to the currently selected object.

The Class Declaration

The single most important language enhancement in Delphi's Pascal implementation is the introduction of the `class` declaration. Let's take a look at the `class` declaration of the `Shape` component, reproduced in Listing 1.

You can see that, superficially, it looks very much like the old-style `object` declaration used in previous versions of the compiler and, in fact `object` declarations are still supported. You must, however, use the new style `class` declaration when creating Delphi components.

The declaration starts off with the name of the new class, an equals sign, ("="), the reserved word `class` and the name of the parent class in parentheses. Like `object` declarations, a `class` declaration can contain both `private` and `public` sections.

In essence, there are now four different levels of protection within Delphi Pascal. In order of increasing accessibility, these are:

Listing 1

```
TShape = class(TGraphicControl)
  private
    FShape: TShapeType;
    FReserved: Byte;
    FPen: TPen;
    FBrush: TBrush;
    procedure SetBrush(Value:
      TBrush);
    procedure SetPen(Value:
      TPen);
    procedure SetShape(Value:
      TShapeType);
  protected
    procedure Paint; override;
  public
    constructor Create(AOwner:
      TComponent); override;
    destructor Destroy; override;
  published
    procedure StyleChanged(
      Sender: TObject);
    property Brush: TBrush
      read FBrush write SetBrush;
    property DragCursor;
    property DragMode;
    property Pen: TPen
      read FPen write SetPen;
    property Shape: TShapeType
      read FShape write SetShape;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
  end;
```

- `private` – Only accessible within the defining unit.
- `protected` – Accessible to derived classes
- `public` – Full run-time accessibility
- `published` – Available to the Object Inspector for design-time manipulation

Property Definitions

The part that's really interesting is the new `published` section. The `published` part of a `class` declaration effectively corresponds to the Object Browser interface for that object. This enables the Object Browser (or anyone else who's interested) to retrieve information on an object, its properties and event handlers. There's nothing magical about the way that the Object Browser does this, it simply makes use of the designer interface unit, DSGINTF. You can find the source code to this unit in the DSGNINTF.PAS source file.

Let's look in detail at one of the property declarations in the above class definition:

```
property Brush: TBrush
  read FBrush write SetBrush;
```

This defines a property called `Brush`. Because the property is in the `published` section of the class, it will automatically be made available to the Object Inspector. Additionally, since it's a property, it can be manipulated just like any normal `public` element of the class – property elements are inherently `public`.

In the above declaration, the `Brush` property is defined as being of type `TBrush` – a handle to a Windows brush object. This is followed by information which tells the compiler how to access the property.

When reading the value of the `Brush` property, the compiler simply references the private `FBrush` field. However, when changing the value of the property, the `SetBrush` procedure (also a private element). This approach allows us to protect the private elements of a class from direct access, while at the same time presenting a convenient user interface that's just as convenient to use as if we had direct access to public elements.

Doing things in this way also allows us to perform other actions behind the scenes. We've just seen that the mere action of assigning to a `Shape` component's `Brush` property will invoke a routine called `SetBrush`. Internally, the `SetBrush` routine will not only store the new brush handle, but also typically redraw the component to reflect the brush change.

Exception Handling

With today's increasingly sophisticated applications, exception handling becomes less of an option and more of a must-have feature when designing any serious programming language. Exception handling allows you to localise error handling and recovery to one area and eliminates the need for repetitive checking for error conditions before and after every operation. Let's see how this works in practice.

The Try-Except Block

Consider this code:

```
function SafeDivide(
  A, B : Integer): Integer;
begin
  try
    {Point 1}
    SafeDivide := A div B;
  except
    {Point 2}
    on EDivByZero do
      SafeDivide := 0;
  end;
  {Point 3}
end;
```

The simple routine shown above is responsible for dividing two integers together and returning the result.

Pascal veterans will immediately spot two oddities here – the appearance of the `try` and `except` keywords. These new keywords are used to implement the exception handling mechanism.

In the case we're looking at, the `try` and `except` statements define a `try-except` block of code. Here, there's actually only one statement (the division statement) between these two keywords but there can potentially be many. Statements within this block of code execute completely normally, starting from Point 1, but if an exception occurs, control is immediately transferred to the `except` part of the block at Point 2. If no exception takes place, then once the `except` keyword is reached, execution continues at Point 3.

The net effect, of course, is that instead of the user being presented with a run-time error, this routine will silently return the value zero whenever a run-time error occurs.

In a more real-world situation, there would typically be a lot more code between the `try` and `except` keywords – code which would normally be full of lots of messy error-checking stuff.

By using an exception handling mechanism, the error checking can be done after the `except` keyword and things become very much neater.

This concept will perhaps be more familiar to Microsoft BASIC (including Visual Basic) programmers. BASIC provides a mechanism called `ON ERR`, which allows control to be transferred to a certain point in a routine whenever a run-time error takes place. The try-except mechanism is very similar in operation.

You'll also have noticed the `on` statement at Point 2 in the above source code. There are a considerable number of different exception types that can be tested for. In this case, we're testing for a divide by zero condition, but you can also test for floating point math errors, file I/O errors, and more.

The Try-Finally Block

In addition to `Try-Except` blocks, Delphi's Pascal language also provides `Try-Finally` constructs. These are particularly useful for Windows programming where it's often necessary to perform a certain amount of 'clearing up', such as de-allocating temporary memory buffers, deleting custom brushes and pens, closing files and so on. Here's how it works:

```
procedure TForm1.Button1Click(
  Sender: TComponent);
var
  pMem: Pointer;
begin
  GetMem (pMem, 2048);
  {...}
  FreeMem (pMem, 2048);
end;
```

In the above example, a 2Kb block of memory is allocated at the beginning of the routine and deallocated at the end. That's fine, but what would happen if an exception (such as a floating point error) were to occur before the `FreeMem` call was executed? In this case, the memory would remain allocated.

Of course, if the run-time error resulted in the program's termination, there'd be no real problem since Windows would deallocate the memory anyway. However, if you were allocating large bitmaps, pens, or brushes, these items would remain allocated even after the program terminated. When programming with Delphi, the correct approach is to use a `Try-Finally` block, which looks something like this:

```
procedure TForm1.Button1Click(
  Sender: TComponent);
var
  pMem: Pointer;
begin
  GetMem (pMem, 2048);
  try
    {...}
  finally
    FreeMem (pMem, 2048);
end;
```

With this approach, the statement(s) following the `finally` keyword will be executed even if the routine terminates with a run-time error. This guarantees that the allocated resource will be freed no matter what happens.

The AS, IS And IN Keywords

The `as` and `is` keywords are used to implement run-time type checking and typecasting. For example, the following statement will determine whether an object, `xObj`, is of a given type:

```
if xObj is TForm then ....
```

This statement will return true if `xObj` is a Form component, ***or*** if it is of a type that's descended from a Form component.

Similarly, the `as` keyword can be used to perform run-time typecasting, like this:

```
with xObj as TForm do begin
  {...}
end;
```

In this example, the `xObj` object is treated as a Form component within the block. The `as` keyword will perform internal checking to ensure that it's valid to treat the `xObj` as if it were a Form component (specifically, that it is a Form component, or is derived from one). If not valid, then a `EInvalidCast` exception will be raised.

The `in` keyword will be familiar to most Pascal programmers as a test of set membership:

```
if TheInt in [1,3,5,7,9] then
  ...
```

However, this particular keyword now has a new meaning within the context of a `USES` clause inside Delphi project files:

```
uses
  Forms,
  Sdimain in 'SDIMAIN.PAS'
    {SDIAppForm},
  About in 'ABOUT.PAS'
    {AboutBox};
```

Changes To The Language

The version of Borland Pascal on which Delphi is based incorporates a number of useful language enhancements. In most cases, these are backwards-compatible. This means that they won't break any existing code.

However, there are a few pitfalls for the unwary so read the following sections with care.

The Result Variable

When developing a Pascal function, it's often useful to be able to "look" at the return result. Previously, it wasn't possible to do this, since specifying the name of the function in an expression was interpreted as a recursive call:

```
function GetFileHandle(
  fName: PChar): Integer;
begin
    GetFileHandle :=
      _lopen(fName, 0);
    if GetFileHandle = -1 then
      MessageBox(0,
        'Can''t open file',
        'Error', mb_ok);
end;
```

The reference to `GetFileHandle` in the `if` statement will be interpreted by the compiler as a recursive call which obviously isn't what's wanted. The compiler will fail to compile the code anyway, complaining that no arguments have been supplied for the (supposed) call to `GetFileHandle`. In order to get around this, most Pascal programmers use a local variable like this:

```
function GetFileHandle(
  fName: PChar): Integer;
var  fd: Integer;
begin
  fd := _lopen (fName, 0);
  if fd = -1 then
    MessageBox (0,
      'Can''t open file',
      'Error', mb_ok);
  GetFileHandle := fd;
end;
```

There's nothing wrong with this, of course, provided that you don't mind the unnecessary tedium of defining a local variable and (more importantly) remembering to set up the function result at the end!

The new `Result` variable does away with these considerations. It behaves as a predefined local variable but it also happens to correspond to the function result. Unlike the actual function name, you can use it anywhere in an expression without implying a recursive call. Here's how you'd recode the above example:

```
function GetFileHandle(
  fName: PChar): Integer;
begin
  Result := _lopen (fName, 0);
  if Result = -1 then
    MessageBox (0,
      'Can''t open file',
      'Error', mb_ok);
end;
```

This gives the best of both worlds; concise and elegant yet without irrelevant variables.

Note: There's an obvious caveat here. When porting old code to Delphi, it's a good idea to rename any local or global variables named `Result` or potential ambiguities may arise.

Function Result Types

While on the subject of function results, Borland have relaxed the previous restrictions on permissible function result types. In the words of the on-line help documentation:

"Functions can now return any type, whether simple or complex, standard or user-defined, except old-style objects (as opposed to classes), and files of type text or 'file of'. The only way to handle objects as function results is through object pointers."

Open Array Construction

Some time ago, Borland introduced Open Array parameters, which allow you to pass an array type as a parameter to a function or procedure. Inside the called routine, you can use the built-in `Low` and `High` operators to obtain the lower and upper array bounds of the array. In this way, you could, for example, pass an arbitrarily large array to a function which would then return the average value of all the elements of the array.

Delphi's version of Pascal makes this facility even more flexible, by letting you build an array and pass it to a routine in a single operation:

```
Average := CalcAverage([5, 7,
  9, 14, 234, 86]);
```

The corresponding function declaration would be:

```
function CalcAverage(Nums:
  Array of Integer): Integer;
```

Delphi includes a new routine, `Format`, which takes a pointer to a destination character array, a format string, and an open array parameter. In essence, this gives all the power and flexibility of the C language's `sprintf` statement, something that's sure to be good news for Pascal programmers.

Note: Since the elements of the array are enclosed in square brackets, this can look just like a set. Take care not to confuse the two.

Case Statement Optimizations

Borland have made two changes to the way `case` statements operate. Firstly, it's no longer possible to have overlapping ranges in a `case` statement. For example:

```
case Errcode of
  7: Writeln(
    'Disk is write protected');
  1..100: Writeln(
    'Unknown error');
end;
```

This code will compile fine under previous versions of the Pascal compiler but won't be accepted by Delphi since 7 obviously overlaps with the range 1..100.

The second change concerns the way in which the compiler generates code for `case` statements. Basically, if the various `case` constants are sorted in ascending order, then the compiler converts the `case` statement into a number of jumps.

On the other hand, a non-sorted ordering of `case` constants will result in multiple calculations being carried out. It's therefore better to sort your `case` constants into ascending order if possible. For example:

```
case ErrCode of
  1: Writeln("This is case 1");
  2: Writeln("This is case 2");
  5: Writeln("This is case 5");
  {...}
```

Using Your Old Code

Delphi is perfectly capable of using your old code, integrating it into a new-style Delphi project. If the old code is in the form of a DLL, then you can just call the DLL from Delphi. Existing units can also be easily integrated into Delphi programs. Of course, old source code won't have any knowledge of Delphi's component library and VCL framework, but provided that the DLL has been well structured, it should be relatively easy to move it across.

But what about OWL, I hear you cry? Well, admittedly, this could be something of a problem. You can certainly use Delphi to compile all your existing OWL library source code and applications if you wish to continue using the OWL application framework. It should go without saying, though, that you can't readily mix OWL code with the new VCL library. At the time of writing, there's been no commitment from Borland as regards the implementation of a 32-bit OWL library. (Because of the differences between the Win32 and Win16 APIs, it's not just a simple matter of recompiling OWL with a 32-bit compiler).

My personal advice would be to bite the bullet and port your applications to VCL. Not only will you be able to use all Delphi's user interface components, (giving your program a much nicer user interface), but you'll also be assured of portability to the world of 32-bits, be it Windows/NT or Windows 95.

This is probably a good place to point out the importance of 'decoupling' the user interface of an application from the nuts and bolts of the program code. Whatever sort of application you're writing, always make a clear distinction between what the program ***does*** and what the program ***displays*** on the screen. If you always bear this in mind, then you can put the essence of your program into units or even DLLs, completely distinct from whatever user interface and application framework you might be using. If you've adopted this sort of approach with your OWL applications, then you will have greatly simplified the job of moving across to Delphi and the VCL library.

This article is based on an extract from Dave's new book, "Instant Delphi", published by Wrox Press. Dave Jewell is a freelance consultant/programmer, specialising in systems-level work under Windows and DOS. You can contact Dave on the internet as djewell@cix.compulink.co.uk

Don't forget the Subscription Form on Page 41!

Introducing Client/Server

by Sundar Rajan

If you are a PC based database developer you might well be wondering what all the Client/Server hoopla is about. At first glance Client/Server development does not seem to be a whole lot different from conventional PC based application development. If you are puzzled, you are not alone. Client/Server is one of the most over-used buzz-phrases in the industry. It is also one of the most misunderstood. This confusion comes from many sources, the most prominent being the application vendors themselves. It seems that too many products nowadays are labeled Client/Server.

Unlike relational database technology (with Dr Codd's 12 rules), object orientation and the Windows GUI (with Charles Petzold's treatise) there is no single source definition of Client/Server. However, there are certain common traits shared amongst commercial software products conventionally known as Client/Server products.

What Is A Client/Server Application Developer?

A Client/Server developer may be developing applications on PCs just like a PC programmer. The difference is this: the 4GL/3GL programmer does not actually have to create code that runs on a server. Why? Because the underlying database (Paradox, dBASE, etc) provides the database mechanism. A Client/Server application developer is one who creates an application whose components run on both the *client* and the *server*. The Client/Server architecture makes full use of the processing power of both the client and the server, distributing parts of the application to both – this requires expertise on both the client and server sides.

A simple definition of Client/Server is that *server* software accepts requests for data from *client* software and returns the results to the client.

A client is any program that requests processing. Due to performance, data sharing and hardware resource sharing considerations you will find that that clients and servers typically run on different machines. Because the client (typically a PC) is not performing all the heavy database work it can be used for manipulating the data and providing timely and valuable information to the end user.

A server is any program that delivers processing. For example, LANs usually have both file and print servers. Another example is the SYBASE SQL Server, which responds to many simultaneous requests to perform database work.

Client/Server computing is thus any application system where separate autonomous programs request and deliver processing to and from each other. That is, an orchestration of various hardware, networking and software components to provide end users with enriched information, on demand.

Two Main Principles

There are two traits which distinguish Client/Server from both mainframe and PC computing.

➢ *Distributed Resources*

This implies that a single client application is not limited to its own resources. Let's consider a warehouse inventory example with the local inventory levels stored in tables on a local SQL server, such as an Interbase server, and company wide information stored in tables on an enterprise SQL server. When the user queries the availability of a particular product, the client application on the user's workstation first queries the local database and if there is no stock left in the local warehouse, the application calls for additional data (inventory levels) on other warehouse servers located anywhere in the world. The client is not just a dumb terminal – some data storage and processing (eg calculating daily sales) does need to be done by the client.

➢ *Intelligent Communication*

Client/Server software must have a higher degree of intelligent communication capability – the server does not send the whole list of inventory items when a client queries inventory levels. Instead the server will be smart enough to understand the client's requests. The server performs the necessary work and provides the client just what it requested. Contrast this with the typical file-server PC databases such as FoxPro or Paradox. In a file-server situation, the whole inventory table may come back to the client workstation across the network, regardless of how much data is involved. Such a request could easily bog down the network (and often does!).

Architecturally, the key difference between Client/Server and the traditional centralized mainframe is that with Client/Server the client is intelligent and performs some processing, whereas mainframe terminals are usually dumb.

2-Schema And 3-Schema

The key to understanding Client/Server is in realizing that it is a *logical* concept. The client and server parts may, or may not, exist on distinct physical machines. More precisely, Client/Server technology is a model for the interaction between concurrently executing software processes.

A 2-Schema approach (Figure 1) is generally adopted when MIS shops first move to a Client/Server environment. The application logic is moved to the client and data is put on the server. SQL is used to obtain data from the server and return it to the client.

While the traditional Client/Server model does distribute processing, the 2-Schema architecture provides no explicit home for business rules. Many Client/Server theorists have proposed a third layer for the business rules. This 3-Schema architecture seems to be the emerging trend, with three separate functional components:

- **The User Interface** – screens, reports and so on, all parts of an application which the business users get their hands on, or see;
- **The Business Logic** – the formal policies and procedures automated in the software as logic or code;
- **Database Access** – storage and retrieval of information from one or more database systems.

The 3-Schema architecture provides the means by which to distribute these three application components across various hardware and software platforms, such as PCs, workstations, local area networks and even mainframes. Thus application designers and developers can optimize for better access to data, better performance and better integration with other applications.

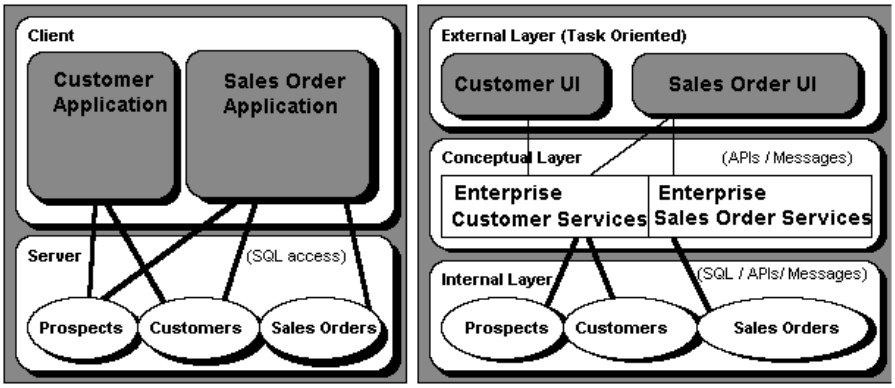


Figure 1: 2-Schema and 3-Schema approaches

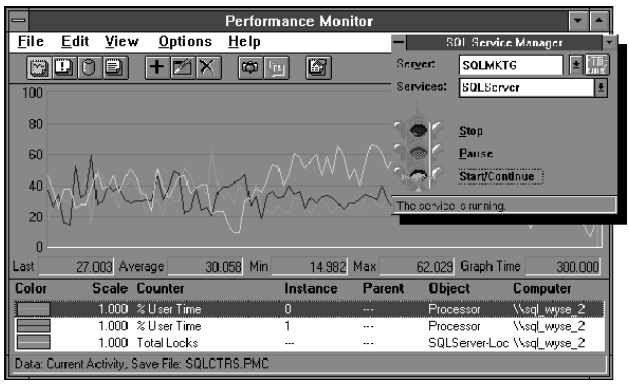


Figure 2 Sybase SQL Server performance monitor: one example of the way SQL server databases provide the ability to track and tune performance

Benefits Of Client/Server

- Client/Server gives a more efficient division of labor. Both the client and server can be dedicated to the tasks for which they are best suited: the client for presentation of a pleasant interface, the database server for high-performance data processing, with security, integrity and concurrency control.
- Client/Server architecture provides an opportunity for both horizontal and vertical scaling of resources to do the job. For example, you can distribute the work of processing data requests, such as queries or updates, to multiple processors on the same network (horizontal scaling), or you can move the database to a larger, more powerful computer (vertical scaling).
- Users can stay with the same familiar tools they've grown accustomed to on the PC.
- Increased access to Corporate Data. Unlike traditional systems which often used proprietary data formats, Client/Server SQL databases have opened up data access. Database middleware services such as IDAPI or ODBC provide a single interface to multiple SQL data sources.
- Data is safeguarded against loss or improper access. Database administration is centralized, so security, data integrity, concurrency and backup/recovery are in the control of MIS.
- Cheaper and more powerful PCs are providing hardware and software solutions that are cheaper to implement.
- Client/Server technology offers many performance advantages: reductions in network traffic, memory and CPU usage.
- Cost benefits: expensive servers with more processing power can be shared amongst many clients. This means that corporations can preserve their investment in existing client technology and incorporate more expensive, high-performance server computers to process the additional workload.
- User participation: adding processing power to user workstations is also beneficial. Traditional mainframe environments executed all application programs on a central computer, but Client/Server applications migrated from host systems enjoy the increased capacity of these distributed processing units. Scalability of distributed networks is a large potential performance advantage for Client/Server.

Servers have tremendous advantages because they centralize data and act on that data at one machine. This Divide and Rule philosophy in Client/Server has resulted in strong growth, not only in front-end tools technology but also in database server technology. SQL servers these days are remarkably different from simplistic PC databases. They abound in advanced features like built-in recovery facilities, performance tuning, parallel query processing and the like. The ability to track and tune performance (Figure 2) is a big benefit over file-based dBASE or Paradox databases, for example.

Don't forget the Subscription Form on Page 41!

What Delphi Brings To The Table

Jesse Berst, publisher of the *Windows Watcher* newsletter, observes that: *"..for the short term, at least, Client/Server means Windows/Server..."*

In this regard, Borland's Delphi tool is eminently suited for the task. It is a highly versatile Windows based Rapid Application Development tool that caters to a potentially huge audience.

With Delphi, you can drop down to assembly language and write device drivers and the like, or develop sophisticated Client/Server database applications without writing hardly any code. This breadth is unmatched in any other Windows development tool.

Client/Server technology has been around for a few years and there are some powerful tools already on the market for Windows Client/Server development. So what is unique about Delphi?

To understand Delphi's uniqueness you have to look at how MIS departments approach development. Typically, application designers hold design sessions with end-users and draft out a rough data model based on these interviews. Then, a simple prototype of the application is attempted with user involvement. Once the prototype is approved, the application is constructed and deployed.

The problem with many of the first generation tools such as Visual Basic is that while they may be good for prototyping, the application performance is not very good due to the interpreted nature of the final application.

Also, VB is not intrinsically a database tool: its native controls, such as grids, are not data aware. A true prototype cannot be produced without writing a lot of code – making changes would require considerable work.

Traditionally, database applications and reports were generated by MIS staff and might take many months to complete. Usually, there have been huge backlogs of requests for specialized entry, control and reporting systems. Now, data modelling and report production is being off-loaded to power users on the LANs.

This is where I think Delphi's architecture excels. With its live data display at design time, templates for forms development and high code re-use because of its object orientation, end-users can get involved in the design and prototyping process.

Tools such as PowerBuilder and VB are also very good at prototyping. The problem is that production applications lag in performance because they are interpreters not true compilers.

As Client/Server systems become more complex performance starts to matter significantly. Delphi's ability to generate true machine code and speed of application creation means the same tool can be used for both prototyping and production.

Delphi's OOP framework encourages the creation of templates as well as re-usable components. This reduces the amount of code required considerably. With Delphi even fairly sophisticated Client/Server applications can be built which require little or no code to be hand-written.

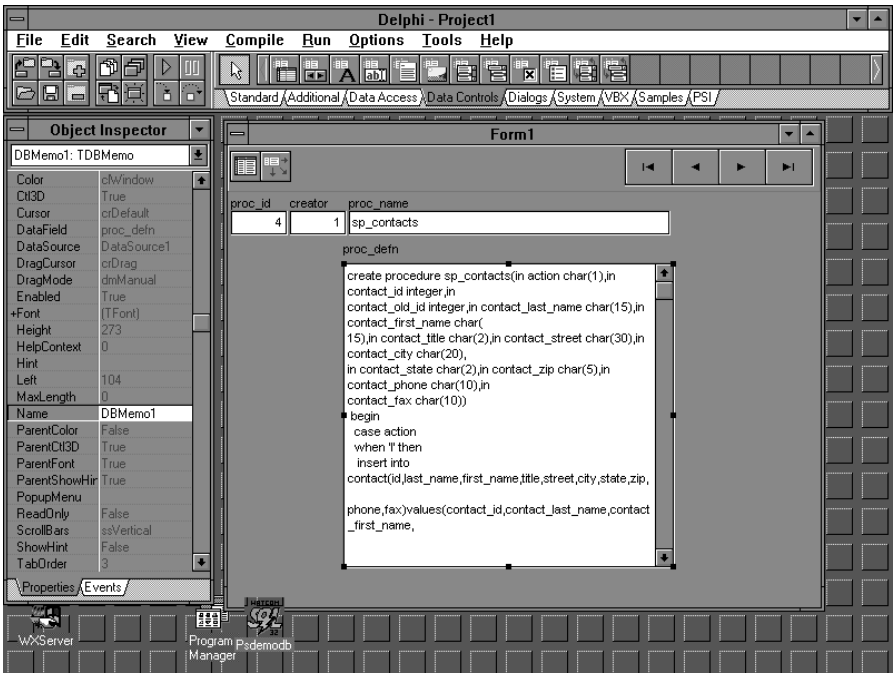
Summary

We took a quick-tour of Client/Server and Delphi's role in application development. Delphi provides an attractive framework for making the transition to Client/Server.

While the opportunities are immense, new skills have to be acquired – especially in the areas of application design, SQL server know-how and SQL. There are exciting times ahead for Delphi developers willing to take on the Client/Server challenge.

Sundar Rajan is a Consultant with On-Line Resources Inc, a consulting firm specializing in Client/Server development based in Longwood, Florida. He is currently developing Client/Server applications for a large Japanese semi-conductor manufacturer in Northern California. Before moving to the US in 1993, Sundar founded and operated SunSoft (NZ), a software consulting firm in Wellington, New Zealand. He can be reached on CompuServe at 72774,1030.

Figure 3: Client/Server inDelphi: displaying the contents of stored procedures from a Watcom SQL 4.0 database – note the LIVE data at design time!



Don't forget the Subscription Form on Page 41!

Delphi Internals: Using And Writing DLLs

by Dave Jewell

Over the coming months, the ***Delphi Internals*** column is going to peer under the hood of Delphi, examining many of the low-level aspects of Delphi programming. Some of the issues we're going to cover include dynamic link libraries, debugging Delphi applications, the inside story behind exception handling, how to interface to other programming languages and much more!

If there's anything specific that you'd like to see covered, then please write to me *c/o The Delphi Magazine* and tell me what you're interested in. Do bear in mind though, that my mission in life is to cover low-level techie programming. I may not react favourably if asked how to go about designing an accountancy package in Delphi! Alternatively, you can send email to me via the Internet at djewell@cix.compulink.co.uk

Using DLLs With Delphi

OK, enough of the social niceties – let's roll up our sleeves and get down to business. In this issue we're going to look at Dynamic Link Libraries, or DLLs for short.

I'm assuming that you've got no previous experience of using Borland Pascal, but that you are familiar with the basic concepts behind DLLs and that you've had some exposure to Windows programming. We'll therefore cover the essentials of using and creating DLLs from a Pascal developer's point of view before looking in more detail at how you would build a DLL in Delphi.

Whenever you examine a source file that's been created by Delphi, you'll see a `USES` statement near the top. This identifies which Pascal units are required to compile the file. Almost invariably, the first two units specified in the `USES` statement are `WINTYPES` and `WINPROCS`. These files contain the type definitions and routine declarations needed by Pascal in order to call the Windows API. Since the Windows API is implemented using DLLs, we need actually look no further than the routine declarations in the `WINPROCS` unit – this tells us all we need to know about calling a DLL from Delphi's Object Pascal.

Calling An External Routine In A DLL

If you look at the `WINPROCS.PAS` source code, you'll see that there are a huge number of routine declarations there – so many that it's difficult to see the wood for the trees. In order to make things clearer, I've written a small unit called `TINYAPI.PAS` which declares just a single routine, `SetRect`. The code for this unit is shown below. Let's see how it works.

```
unit TinyApi;
interface
uses WinTypes;
procedure SetRect(
  var Rect: TRect;
  X1, Y1, X2, Y2: Integer);
implementation
procedure SetRect; external
  USER  index 72;
end.
```

The interface part of the unit is followed by a `USES` clause for `WINTYPES`. This is then followed by the procedure declaration for `SetRect` itself. The important part, of course, is the actual implementation of this routine in the implementation part of the unit. The first thing you'll notice is that you don't need to repeat the list of parameters required by the `SetRect` routine. You can repeat the parameter list if you wish, but if you do so, then you must make sure that it exactly matches the previous definition. For example, if you refer to the four integers as `X1, Y2, X2, Y2` in the interface declaration, then the compiler won't allow you to refer to those same integers as `left, top, right, bottom` in the implementation part of the unit, even if these are perfectly acceptable names for the parameters.

The `EXTERNAL` keyword is more interesting. This tells the compiler that the actual code for the interface routine isn't in the unit being compiled, but is somewhere else. DOS-based Pascal developers frequently use the `EXTERNAL` keyword to link a program with separately compiler assembler code. However, thanks to the magic of dynamic linking, we can tell the compiler that the code isn't going to be statically linked at all.

The next part of the statement specifies the name of the DLL containing the `SetRect` routine – in this case, it's the `USER` DLL, one of the core components of Windows itself. The compiler doesn't need to verify this information at compile time. It doesn't care at this point whether the `USER` DLL exists, or whether it really contains the routine we're saying is there, all these checks take place at run-time.

The final part of the `implementation` statement specifies an ordinal value for the routine. You should know that all routines exported by a DLL have an associated name called the ordinal number. When you import a DLL routine (as we're doing here), you need to somehow tell Windows which routine you're interested in. By specifying an ordinal number, our application will end up asking Windows for routine number 72 in the `USER` library. This is called linking by ordinal. Alternatively, you could omit the `INDEX` keyword

and the ordinal number itself. You'd then be linking by name. In general, linking by name is easier (since you don't have to mess about with ordinal numbers) but it results in a slightly larger executable file and is fractionally slower at run-time. If you want to know what routines are exported by a particular DLL, and what ordinal values they have, you can use a Microsoft utility such as EXEHDR to dump the list of exported routines.

Import Units And Custom DLLs

The WINPROCS unit and the TINYAPI unit that we looked at earlier were examples of import units. An import unit has only one mission in life – its job is to take a set of DLL routines and make those routines available to the application in which it is linked. For example, when using the FlashWindow API call inside a Delphi routine, you can just call FlashWindow as if it were a local routine. You don't know, and don't care, that it's actually implemented inside the USER library. That's what an import unit is for: to make DLL routines more immediately accessible.

At this point, we've only discussed how to interface to the standard DLLs, but naturally, you can call custom DLLs just as easily. These DLLs might have been written using C, Pascal, assembler or Delphi itself – it really doesn't matter. As an example, here's a couple of procedure declarations used in one of my own programs, referring to routines in a custom DLL called TFRAME.DLL:

```
procedure FixLibrary; far;
  external 'TFRAME' index 1;
procedure UpdateTopLevelWindow(
  fDraw: Boolean); far;
  external 'TFRAME' index 2;
```

There are two important things to notice about these two procedure declarations. Firstly, you'll see that they include the procedure parameters along with the EXTERNAL keyword, DLL name and index information. That's because these declarations aren't part of an import unit. If you want to call a custom DLL, you don't have to create an import unit if you don't want to. Using the approach shown here, you can just go right ahead and put the DLL procedure declarations after the USES clause of your main program. Alternatively, you could put these same declarations at the beginning of the implementation part of a unit – that way, you'd be providing the unit with access to private routines that it needs to do its job, but you wouldn't be making the existence of those routines known to any other parts of your program.

The second point to notice is the use of the FAR keyword. When you declare routines in the interface part of a unit, they're always far. Any routine exported by a Delphi unit is a far routine, and can only be accessed by a far call. However, when you're declaring DLL routines outside of an import unit, you must use the FAR keyword. Not doing so will result in a compile error.

Avoiding The Windows API

Having just explained in some detail how it is that Delphi calls the Windows API, let me stress that you shouldn't go overboard on using the API. In fact, if you can find an equivalent call or method in Delphi's Visual Component Library (VCL) which will do the same job, then you should use the VCL call in preference to calling the API. What's the reason for this API-phobia? In one word – portability.

Borland went to a lot of trouble to make the VCL library as portable as possible. The great majority of your 16-bit Delphi applications will be able to move effortlessly across to 32-bit Windows/NT and Windows 95, provided that you've minimised the use of calls to the Windows API. In particular, you should avoid using API calls which send or receive messages. This is because, under 16-bit Windows, Microsoft often packed more than one quantity into the 32 bits of the lParam field. Under Win32, however, window handles are now 32-bits wide and many Windows messages have therefore had to adopt a different arrangement of values in the wParam and lParam fields of a message. C/C++ programmers get around this by using message cracker macros which pack and unpack the field of a message while retaining portability between the two different APIs. Under Delphi, the proper approach is to use VCL wherever possible.

Writing DLLs With Delphi

One of the many interesting things about Delphi is its ability not only to use DLLs but to create them. Using Delphi, you can create a DLL that's used by more than one of your programs, thus reducing the amount of disk space required when building a suite of programs.

Alternatively, you can use DLLs to provide functionality to your users in bite-sized pieces. For example, you might decide to sell an application which views and processes graphics files. You could build the capability to read and write some common graphics file formats into the application itself, but you might also want the flexibility to sell add-on packs which operate on even more formats. By packaging these add-on packs as DLLs, you can easily arrange for the main application to detect the presence of these add-ons and use them in a seamless fashion. Incidentally, much of Windows operates in this way; device drivers, Control Panel applets, even fonts, are specialised forms of DLL.

When news of Delphi first began to leak out, rumours were rife that you could just plug Delphi components into C/C++ applications – one American programming journal even enthused about this in its editorial. Of course, this just isn't possible – a Delphi component is at heart an Object Pascal unit. The Pascal compiler inside Delphi generates DCU files whereas C/C++ development systems use .OBJ files. It's really a case of never the twain shall meet. However, thanks to the magic of DLLs, it's possible to write a sophisticated user interface using Delphi components and call it from a C/C++ application. Actually, you could equally well call it from a straight Pascal program, or even from Visual Basic – a DLL completely breaks down

Don't forget the Subscription Form on Page 41!

the language barriers and can be used from any development system that supports DLL calls.

The Structure Of A DLL

The remainder of this article will demonstrate how to create a simple DLL using Delphi. In Delphi, or Borland Pascal, a DLL is structured somewhat as shown below:

```
library MyDLL;
uses WinTypes, WinProcs;
procedure MyFirstProc; export;
begin
  MessageBeep (0);
end;
exports MyFirstProc index 1;
begin
end.
```

If you look in the project file (the file with extension .DPR) of any Delphi application, you'll see that it starts off with the reserved word PROGRAM. By contrast, DLLs always begin with the reserved word LIBRARY. This is then followed by a USES clause for any needed units. In this simple example (probably the simplest DLL that it's possible to make), there then follows a procedure called MyFirstProcedure which does nothing except sound a beep.

You'll notice the procedure declaration uses the EXPORT specifier. This tells the compiler that the procedure is going to be called from another module – the compiler will then generate the special prologue and epilogue code which ensures that the processor's data segment register is properly set up on entry to the procedure. Any routines exported from a DLL must include the EXPORT specifier. All call-backs such as window procedures, hooks and enum procedures also need this specifier.

Finally, at the end of the source code we find an EXPORTS statement. This lists the routines that are actually exported from the DLL and assigns a unique ordinal value to each routine. This ordinal value can then be used to call routines in a DLL as discussed earlier. At this point, you're probably thinking why do we need a separate EXPORTS statement when we've already told the compiler that MyFirstProc is exported by using the EXPORT specifier? That's a good question! Remember that the EXPORT specifier is used for all call-backs and exported routines. However, we don't actually want to make call-backs and window procedures visible outside the DLL. It's the EXPORTS statement at the end of the library source file which tells the compiler what is visible and what isn't.

If you compile this simple DLL, you can then call it using the techniques I described earlier. However all that happens when you call the MyFirstProcedure routine is a beep sound – big deal. In the next installment, we'll conclude this discussion of DLLs by looking at the more exciting stuff – how to write DLLs that contain Delphi forms and components, and how to call those DLLs from other applications and development systems.

Dave Jewell is a freelance consultant/programmer, specialising in systems-level work under Windows and DOS. You can contact Dave on the internet as djewell@cix.compulink.co.uk

Review: "Inside Windows 95" *by Adrian King*

Adrian King has worked very closely with Microsoft in writing this book. In fact, he's a former MS employee and managed in 1987/8 the project that produced Windows/386, so he knows what he's writing about.

This book provides a lot of insight into the thinking behind the design and implementation of Windows 95 – or at least BETA-1 (1994). The author himself warns us that the information is based on a pre-release version. When it was announced (as Chicago), Microsoft wanted to release it by the end of the year – 1993. By the time the book was written, Windows 95 was expected by the end of the year – 1994. And by the time you read this review Windows 95 is still expected to ship at the end of the year – 1995!

The intention is to provide a technical introduction to Windows 95, including enough detail to satisfy power users and developers. The emphasis is on what Windows 95 can do, how it does it, and why features were designed and implemented in particular ways (though there'll be changes in the final release).

Among the topics covered are: the 32-bit protected mode environment, the revised user interface and new system shell, the new device-independent color and display drivers, the new enhanced file system with long filenames, plug and play, network support, built-in internet access utilities and so on.

The only things not included in the book are pen support and the multimedia support of Windows 95, like WinG. The book contains several charts and some example code (in C), but still the most interesting parts are those where design choices are made and motivated against possible alternatives. This makes you feel almost part of the Windows 95 team itself.

The book does certainly not claim to cover it all, but only to provide pre-release knowledge of Windows 95, for people that want to prepare before it's there (people who don't (or do) have access to a Beta version themselves). I think everyone should read *Inside Windows 95* – especially power users and developers. I can recommend this book to anyone who wants to prepare **now** for Windows 95 (and we all know 32-bit Delphi will ship just shortly after Windows 95 itself)...

For more (free) information on Windows 95, you can subscribe to Microsoft's electronic newsletter WinNews, by sending an e-mail message with the contents SUBSCRIBE WINNEWS to the address enews@microsoft.nwnet.com .

Reviewed by Bob Swart

"Inside Windows 95" by Adrian King is published by Microsoft Press, ISBN: 1-55615-626-X, 480 pages, soft cover, US price $24.95, UK price £21.95.

Don't forget the Subscription Form on Page 41!

Using The Borland Visual Solutions Pack

by Jeroen W Pluimers

Delphi has enormous potential for using components. Not only does it have its own sophisticated VCL-based components, but it also supports many of the VBX controls widely available today.

An introduction to what VBXs can do for you is the Borland Visual Solutions Pack (BVSP). Although not expensive, it contains a number of useful controls. Note that when considering other VBXs you do need to remember that Delphi only supports *Version 1.0* VBXs – if in any doubt, ask your retailer or (perhaps even better) the manufacturer.

Installation

Installing VBX controls in Delphi is easy. For the sake of simplicity, I'll assume you are using the default directory structure. Thus, you have a directory `C:\BVSP` which contains the BVSP files. Also, create a subdirectory `C:\BVSP\DELPHI`, where we will store the Delphi component wrappers around the VBXs.

Let's start installing. From the Delphi menu bar, choose `Options | Install Components`. Figure 1 shows the dialog box which is displayed, listing the components that have already been installed. Click the `VBX` button to enter a file selection dialog. Navigate to wherever the VBXs are installed (usually `WINDOWS\SYSTEM`) and select the VBX file you want to install.

After selecting the file, you end up in the dialog shown in Figure 2. The defaults are filled in, so you can just press OK, but you can also change the name of the unit file or the classnames of the controls in the VBX. In this case, I've decided to place the Delphi wrapper unit files in `C:\BVSP\DELPHI`. Delphi automatically generates a Pascal source unit containing a VCL component wrapper around the VBX.

A VBX can contain more than one component – just like regular Delphi units, which can contain more than one component as well.

Although you can install all VBX files from the BVSP, you do not need to do so. Some of the VBX components duplicate functionality that is already included in Delphi. For instance, the `SQC.VBX` (containing Database Controls) is completely covered by Delphi. Also you do not need the `SAXTABS.VBX` as the Delphi Tabs and Notebook components give better functionality.

A summary of the VBXs in the BVSP which are of use to Delphi developers is shown in Figure 4. The column with glyphs shows you the bitmaps which will appear on the VBX page of the component palette. The classname is the hint text you see when you move the mouse over the corresponding glyph. The VBX filename is included so you can easily select the VBX files you need.

Using The BVSP VBXs

The BVSP can be split conceptually into two categories. The first group consists of VBXs that encapsulate alone, or in combination with others, large parts of an application. The second group consists of gadgets: VBXs that perform only a tiny part of an application, but give it a specific look and feel.

The charting, spreadsheet and word-processing VBXs are clearly part of the first group. Depending on its usage also the SaxComm VBX can be added to it. All the other VBXs are part of the second group. You will often need to do a lot of coding yourself to create a fully functional application around these components.

With the first group of VBXs, it is possible to create complete working applications with almost no code. For instance, using the `TX4VB.VBX` and only a few lines of code, you can create a working word-processor that can import and export rich text format (RTF) files, with multiple fonts etc.

Gadgets

Not all components from the second group need much programming. For instance, the animated button can be used to quickly create a multi-state button. The card deck is ideal for setting up a card game, (it's a pity the decks lack animation like Solitaire and Hearts). With a good playing strategy, you could write your own black jack game! Combined with the clock, you can stress the player by limiting playing time.

The dice can be easily configured. With `AutoSize` disabled and a lot of colours on the dice sides, you could write a program to teach counting to children.

A combination of gauges, sliders and spin buttons could be used to wrap up the user interface for a scientific program. Then the marquee control can be used to show floating text on the screen (although its performance is unfortunately not very good).

Fully Fledged Controls

This group contains the really useful VBXs. Although most of the controls are not the most recent versions, you get a good impression of what is possible with them. If you make heavy use of any of the more complete controls, I'd recommended you buy the full current version.

For instance, the SAX communications control lacks certain protocols (like Z-Modem) that are used very widely nowadays. However, it is a good starting point if you want to see what communications could do for your application.

Another useful combination is the charting control and the spreadsheet control. This way, you could show a graphical representation of the data a user enters. Remember, though, that Delphi itself has a more powerful `ChartFX` VBX control.

Figure 3 shows a sample application written using the BVSP. With seven components and about 90 lines of hand-written code, I ended up with a complete word processor using the RTF file format. It supports multiple fonts and pages, text with attributes, paragraphs, search & replace, etc.

A large part of the code (almost 20 lines) is to make sure the `TextControl` and its bars resize within the Form. In contrast with Delphi's native components, the VBX controls lack an alignment property, so you have to do the aligning yourself.

The rest of the code is for the `File|Open` and `File|Save/SaveAs` logic. Only a tiny bit of code is needed to link the menu to the actions – one line per menu action suffices.

The resulting application .EXE file itself is only about 200Kb in size. The additional files are much larger: the `BIVBX11.DLL` (see next section) is about 80Kb and the `TX4VB.VBX` and its support files add up to 240Kb, giving just over half a megabyte in total.

The source for this example will be on the free disk with Issue 2 of The Delphi Magazine.

Distribution

When distributing an application you need to pay special attention if it uses VBXs. You will need to distribute the VBX files with your application and be careful to ship the correct support files. Also, you will need to include the BIVBX11.DLL, which is the Borland support DLL that interfaces between 16-bit applications and VBX files.

The reason behind the complexity is twofold. First of all, VBX files are external DLLs that in turn can use other external files. Second, VBXs need to have a means to distinguish between design-time

Figure 1 Installing a new VBX into Delphi

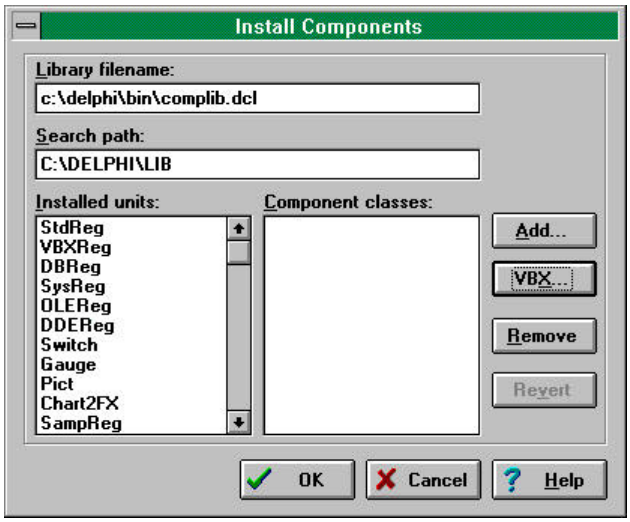
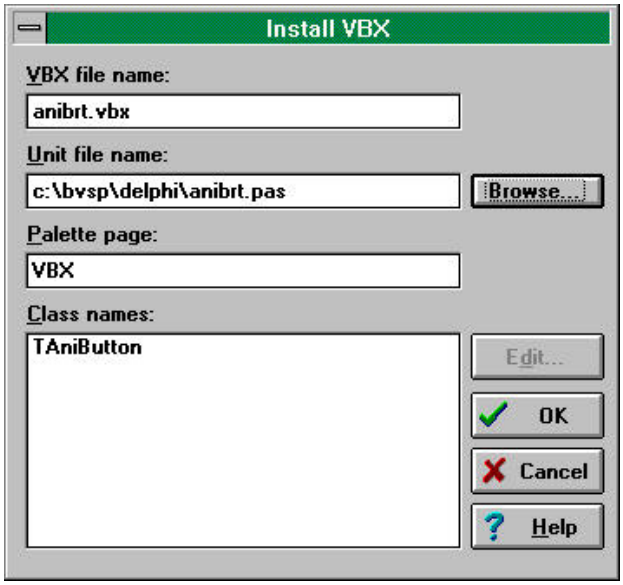


Figure 2 Specifying a Delphi Unit file name and Class names for a new VBX



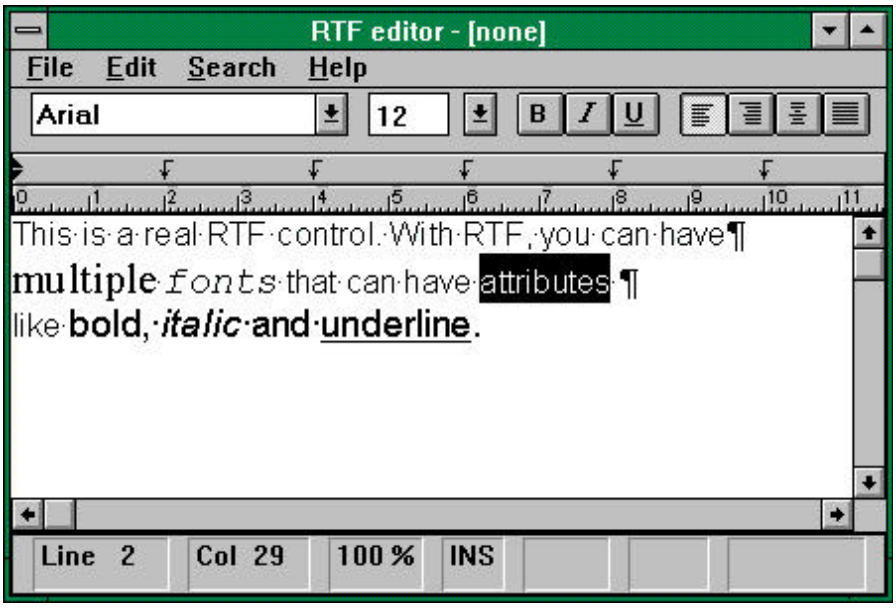


Figure 3
A fully functional rich text format word processor built in Delphi using controls from the Visual Solutions Pack

and run-time behaviour (otherwise anyone with a VBX could use it to develop a new application, without purchasing it).

The technique behind this is called licensing. Because the VBX standard was not well designed, it lacked standard support for licensing. As a result, most VBX vendors invented their own licensing mechanism. Mostly, they need special design-time files – which are called licensing files – to be present in the Windows System directory. Another way is to have separate VBX files for design-time and run-time.

`KNIFE.VBX` is an example of this approach – you should be very careful to copy `KNIFERUN.VBX` onto your installation disks and *rename it* to `KNIFE.VBX` at installation time.

All the other installable files are explained fully in the BVSP documentation. The file `REDIST.TXT` contains more information.

Jeroen Pluimers has been a Pascal programmer since 1983. He lives and works in The Netherlands and may be contacted by email as jeroenp@dragons.nest.nl or on CompuServe as 100013,1443

Figure 4: Useful VBXs from the Visual Solutions Pack

Glyph	VBX File	Class Name	Functionality
	ANIBRT.VBX	AniButton	Animated button with multiple bitmaps. Different frame states allow simulation of multistate buttons.
	KNIFE.VBX	PicBuf	Picture buffer for editing and showing bitmaps.
	MHAL200.VBX	MhIAlarm	Alarm clock with alarm interval and sound.
	MHCD200.VBX	MhCardDeck	Playing card that mimics the cards used in Solitaire and Hearts. It supports non-animated card-backs only.
	MHCL200.VBX	MhClock	Digital or analogue clock showing current time, or time offset to a specific value.
	MHDC200.VBX	MhDice	Playing dice with pictures showing top, left and right sides of dice. Properties for colours and bitmaps.
	MHGA200.VBX	Mhgauge	Gauges that can be horizontal, vertical, circular with pointing needles.
	MHMQ200.VBX	MhMarque	Label-like component with moving caption and moving bitmaps. Attracts attention – ideal for running demos.
	MHSL200.VBX	MhSlide	Slider control useful for simulating audio and industrial equipment.
	MHSN200.VBX	MhSpin	Spinbutton with embedded value component. Buttons are either shown horizontally and vertically.
	SAXCOMM.VBX	Comm	Serial communications with TTY and ANSI emulation. Supports X-modem file transfers.
	TKCHART.VBX	Chart	Chart drawing component. Both application supplied data and database supplied data.
	TX4VB.VBX	TextControl	Rich text component for editing texts with multiple fonts. Has hooks for the ruler, buttonbar and statusbar.
	TX4VB.VBX	TXRuler	Ruler component. For showing positional information of the TextControl.
	TX4VB.VBX	TXButtonBar	Speedbar with buttons and comboboxes to change appearance of text in the TextControl.
	TX4VB.VBX	TXStatusBar	Status line showing positional information of the TextControl.
	VTSS.VBX	Sheet	Spreadsheet component. Automatically links to an SSEdit component for editing if it is available.
	VTSS.VBX	SSEdit	Editor portion of the Sheet spreadsheet component.

Don't forget the Subscription Form on Page 41!

Animation Made Easy

by Xavier Pacheco

This article demonstrates how you can achieve simple sprite animation using Delphi and the Object Pascal Language. It also shows how Delphi simplifies what is usually considered a tedious process since Delphi automatically manages device context for you.

The example that I've created illustrates how you would display a background image (the universe) and draw a sprite image (the UFO) at different locations on the background.

The project's source code is shown in Listings 1 and 2: `ANIMATE.DPR` and `UNIT1.PAS`. These files and the required bitmaps will be included on the free disk which will come with Issue 2 of The Delphi Magazine.

This simple animation example uses three Windows .BMP files: `BACK.BMP` to serve as the main form's background, with `AND.BMP` and `OR.BMP` to make up the sprite image – both are 64x32 pixel bitmaps of a UFO.

A `TSprite` class that I have defined contains the sprite's properties that maintain its location on the form and the `Create()` and `Done()` methods.

`TSprite.Create` creates two `TBitMap` classes, `FAndImage` and `FOrImage`, and reads in the two bitmap files using the `TBitMap.LoadFromFile()` method. It then sets its properties `Top`, `Left`, `Width` and `Height` accordingly. `TSprite.Done` frees the memory used by `FAndImage` and `FOrImage`.

The main form has the variables `BackGnd1`, `BackGnd2` of type `TBitMap` and `Sprite` of type `TSprite`. `BackGnd1` is our original bitmap that we use for our background. `BackGnd2` is the copy of `BackGnd1` to which we perform the `BitBlt()`ing of the sprite image.

The reason we do all the drawing to `BackGnd2` instead of the form's canvas is because to achieve animation we must call `BitBlt()` three times: once to erase the sprite on the form's canvas, once to `AND` `FAndImage` to the form's canvas, and once to `OR` `FOrImage` to the form's canvas. All this drawing to the form's canvas results in a horrible flicker when the image is drawn continuously.

The example program running, with the spaceship scooting across a starry sky! It's in full colour of course and this print doesn't do it full justice.

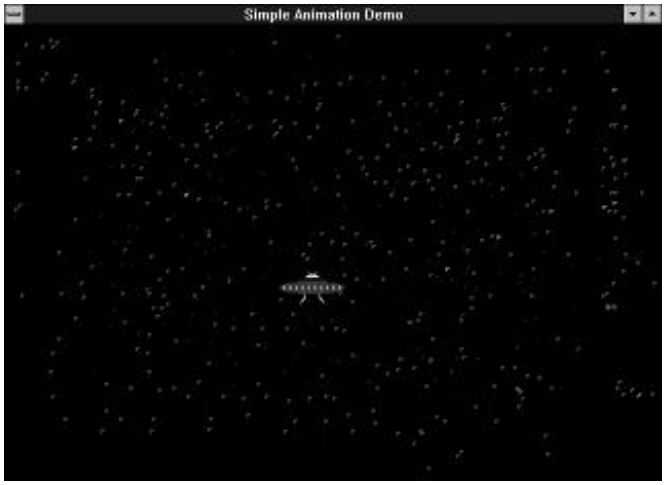


Figure 1

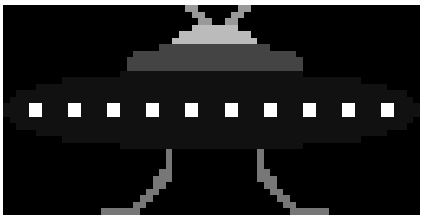


Figure 2

By performing the grunt work on `BackGnd2`, we can copy a rectangle surrounding the old sprite location and new sprite location from `BackGnd2` to the form's canvas with one `BitBlt()` call to eliminate flicker. Therefore, the overhead of maintaining a separate copy of the form's canvas is justified.

`FAndImage` (see Figure 1) effectively creates a black hole where the sprite is to be displayed on the background and preserves the background colors where the sprite does not appear by using the `BitBlt()` function with the `SRCAND` operation.

As you can see from the Figure 1, the sprite is shown where the pixel color is black. Since each black pixel has the value 0 and each white pixel has the value 1, when performing an `AND` operation of `FAndImage` to the destination background the resulting color is preserved where `FAndImage`'s color is white. Where `FAndImage` is black, the result is black.

```
BackGround      1001  some color
Image      AND  0000  black
Result          0000  black

BackGround      1001  some color
Image      AND  1111  white
Result          1001  some color
          (same as Destination)
```

Once I create this black hole, I draw the actual image, still preserving the background's original colors, by `BitBlt()`ing `FOrImage` using the `SRCPAINT` operation.

Notice from Figure 2 that the `FOrImage`'s sprite contains the actual colors while its background is white, or all 1s. You can see from the boolean operation below how `OR`ing the color white to a destination maintains the

destination's color. Since we are ORing the sprite to an only-black background (our black hole), the sprite's colors are maintained.

```
BackGround      1001  some color
Image       OR  1111  white
Result          1001  some color

BackGround      0000  black
Image       OR  1101  some color
Result          1101  some color
              (same as FOrImage)
```

All the drawing is performed in the `TForm1.DrawSprite` method. Here, I use some simple logic to keep the sprite within the form's client area.

I then erase the old sprite from `BackGnd2`, re-draw it in `BackGnd2` at the new location, and finally copy a rectangle from `BackGnd2` to `Form1.canvas` to effectively erase and re-position the sprite on `Form1`'s canvas.

`TForm1.MyIdleEvent` is where `TForm1.DrawImage` is called. I then assign this method to the `Application.OnIdle` event handler in `TForm1.Create`. The method `Application.OnIdle`, as the name implies, is executed when the application is in `Idle`.

`TForm1.Paint` `BitBlt()`s the original background, `BackGnd1`, to its canvas.

Notice the `TSprite` is not a component in and of itself, that is, a descendant of an original Delphi component such as `TControl` or `TGraphicControl`.

The reason I did this was because the form repaints itself whenever making changes to any child controls causing a yucky flicker on the screen. Also, the `TSprite` object was simple enough that I didn't really need any data or methods from an already existing object.

Although this example is very simple, it is possible to extend the functionality of `TSprite` to be more self contained, such as maintaining it's own direction, drawing itself, and being a non-static image, that is an image that changes as it is moved on the background.

Also, I didn't do anything special in this example to create true bounces – something I can keep for a later project!

Xavier Pacheco is a Consulting Engineer at Borland International. You can reach Xavier on CompuServe at 76711,666 or at xpacheco@wpo.borland.com

Listing 1 ANIMATE.DPR

```
program Animate;
uses
  Forms,
  Unit1 in 'UNIT1.PAS' {Form1};
{$R *.RES}
begin
  Application.CreateForm(TForm1,
    Form1);
  Application.Run;
end.
```

Listing 2 UNIT1.PAS

[Sorry about the small text size, it's the only way we could get it all in I'm afraid, but the code will be on the disk with Issue 2. Editor]

```
unit Unit1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages,
  Classes, Graphics, Controls,
  Forms, Dialogs, Menus, Stdctrls;
type
  TSprite = class
  private
    FWidth: integer;
    FHeight: integer;
    FLeft: integer;
    FTop: integer;
    FAndImage, FOrImage: TBitMap;
  public
    property Top: Integer read FTop write FTop;
    property Left: Integer read FLeft write FLeft;
    property Width: Integer read FWidth
      write FWidth;
    property Height: Integer read FHeight
      write FHeight;
    constructor Create(AOwner: TComponent);
    destructor Done;
   end;
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    BackGnd1, BackGnd2: TBitMap;
    Sprite: TSprite;
    GoLeft,GoRight,GoUp,GoDown: boolean;
    procedure MyIdleEvent(Sender: TObject;
      var Done: Boolean);
    procedure DrawSprite;
  end;
const
  BackGround = 'BACK.BMP';
var
  Form1: TForm1;
implementation
{$R *.DFM}
constructor TSprite.Create(AOwner: TComponent);
begin
  inherited Create;
  FAndImage := TBitMap.Create;
  FAndImage.LoadFromFile('AND.BMP');
  FOrImage := TBitMap.Create;
  FOrImage.LoadFromFile('OR.BMP');
  Left := 0;
  Top := 0;
  Height := FAndImage.Height;
  Width := FAndImage.Width;
end;
destructor TSprite.Done;
begin
  FAndImage.Free;
  FOrImage.Free;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  BackGnd1 := TBitMap.Create;
  with BackGnd1 do begin
    LoadFromFile(BackGround);
    Parent := nil;
  end;
  BackGnd2 := TBitMap.Create;
  with BackGnd2 do begin
    LoadFromFile(BackGround);
    Parent := nil;
  end;
  Sprite := TSprite.Create(self);
  GoRight := true;
  GoDown := true;
  GoLeft := false;
  GoUp := false;
  Application.OnIdle := MyIdleEvent;
  ClientWidth := BackGnd1.Width;
  ClientHeight := BackGnd1.Height;
end;
procedure TForm1.MyIdleEvent(Sender: TObject;
  var Done: Boolean);
begin
  DrawSprite;
  Done := false;
end;
procedure TForm1.DrawSprite;
var
  OldOrigin: TPoint;
  TempRect: TRect;
begin
    With OldOrigin do begin
      X := Sprite.Left;
      Y := Sprite.Top;
    end;
    with Sprite do begin
      if GoLeft then
        if Left > 0 then
          Left := Left - 1
        else begin
          GoLeft := false;
          GoRight := true;
        end;
      if GoDown then
        if (Top + Height) < self.ClientHeight then
          Top := Top + 1
        else begin
          GoDown := false;
          GoUp := true;
        end;
      if GoUp then
        if  Top > 0 then
          Top := Top - 1
        else begin
          GoUp := false;
          GoDown := true;
        end;
      if GoRight then
      if (Left + Width) < self.ClientWidth then
        Left := Left + 1
      else begin
        GoRight := false;
        GoLeft := true;
      end;
  end;
  {Erase the old sprite in BackGnd2 }
  with OldOrigin do
    BitBlt(BackGnd2.Canvas.Handle, X, Y,
      Sprite.Width, Sprite.Height,
      BackGnd1.Canvas.Handle, X, Y, SrcCopy);
 {Draw the sprite at the new location in BackGnd2}
  with Sprite do begin
    BitBlt(BackGnd2.Canvas.Handle, Left, Top,
      Width, Height, FANDImage.Canvas.Handle,
      0, 0, SRCAND);
    BitBlt(BackGnd2.Canvas.Handle, Left, Top,
      Width, Height, FOrImage.Canvas.Handle,
      0, 0, SRCPAINT);
  end;
  {Copy a rectangle from BackGnd2 to erase and
   reposition the sprite to the form's canvas}
  with OldOrigin do
    BitBlt(Canvas.Handle, X-2, Y-2,
      Sprite.Width+2, Sprite.Height+2,
      BackGnd2.Canvas.Handle, X-2, Y-2, SrcCopy);
end;
procedure TForm1.FormPaint(Sender: TObject);
begin
  BitBlt(Canvas.Handle, 0, 0, ClientWidth,
    ClientHeight, BackGnd1.Canvas.Handle,
    0, 0, SrcCopy);
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  BackGnd1.Free;
  BackGnd2.Free;
  Sprite.Free;
end;
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  DrawSprite;
end;
end.
```

Don't forget the Subscription Form on Page 41!

The Delphi Clinic

Edited by Bob Swart

Bring your problems to our panel of experts!

If there's something puzzling you about an aspect of Delphi, just email the Delphi Clinic Editor, Bob Swart, at Compuserve 100434,2072 or write or fax us at The Delphi Magazine

Q I've just installed Delphi and I just can't get it to run right. I keep getting errors like:

```
Application Error. Exception
EDatabase Error in module
FormExpt.DLL at 0007:0EAC.
Failed to initialize IDAPI.
Error code $2C09.
```

What's gone wrong?

A Please see if you've got SHARE loaded in your AUTOEXEC.BAT. The correct line there should read (at least):

```
SHARE /F:4096 /L:40
```

Alternately, you could use:

```
device=vshare.386
```

in the [386Enh] section of the SYSTEM.INI file. Note that Delphi does ask Windows whether or not share is loaded, but Windows (incorrectly) always reports SHARE to be loaded, even if it isn't.

Q I'm building a Delphi component and want to be able to tell if it's being used in design mode or at run time. How can I do this?

A Although most Delphi components work in design mode exactly the same way they work in execution mode, you can easily make your code detect whether or not you're still in design mode or execution mode. The flag cdDesigning in the ComponentState will give us this information, for example:

```
If csDesigning in
  ComponentState then
  MessageDlg('Design Mode',
    mtInformation,[mbOk],0)
```

This very nice feature enables us to put a lot of extra (debug) code and information at design places that are no longer used then we generate a true .EXE. All with the same code!

Q Is the file type Text no longer supported? I used it a lot in Borland Pascal, but this code gives me a compiler error Error 21: Error in type:

```
procedure TForm1.Button1Click(
  Sender: TObject);
  var myFile: Text;
```

A The error message is produced because Text is also a property of the button. To get the Text file type, use System.Text to scope it to the system unit. The same is true for Close and Assign. Alternatively, use the new *Delphi only* TextFile, AssignFile and CloseFile procedures.

Q I understand that identifiers in the private section of a class or object are of course private and identifiers in the public section are public, but what about the component declarations added automatically by Delphi ***above*** the private section. What access rights do they have?

A It seems that (unless specified otherwise) the fields, methods and properties of objects are all published: that is they are both public and automatically appear in the Object Inspector.

Published parts have run-time type information generated for them which is available to the Object Inspector. You can find more information on this subject in the Delphi *Component Writers Guide*.

Q How do I go about working with Microsoft Access files in Delphi? I've heard several people say they have had difficulties.

A You can work with Microsoft Access .MDB files in a Delphi application using an ODBC driver. Delphi actually gives you all you need, but the process is not immediately obvious! Here are the steps.

What You Need

First, check you have the ODBC Administrator installed (file ODBCADM.EXE in WINDOWS\SYSTEM, you also need the file ODBCINST.DLL, for installing new drivers, and ODBC.DLL). The ODBC Administrator should be shown in Control Panel as the ODBC icon. If you didn't have it already, it should have been installed when you installed Delphi.

If you get a message to the effect that "Your ODBC is not up-to-date. IDAPI needs ODBC greater then 2.0" you have an earlier version and should replace it with the one included with Delphi.

Check you have an Access ODBC driver installed in Windows. You can do this by clicking Drivers in the Data Sources dialog which appears when you run the ODBC Administrator. Delphi installs an entry Access Files (*.mdb) or Access Data (*.mdb), which works with Access ***1.10*** files and uses the SIMBA.DLL driver (note that this DLL also needs the files RED110.DLL and SIMADMIN.DLL – all installed for you by Delphi). These files are redistributable with your programs as part of the ReportSmith Runtime.

If you want to work with Access ***2.0 or 2.5*** files, you need to obtain a different set of driver files from Microsoft. The key file is MSAJT200.DLL, also needed are

MSJETERR.DLL and MSJETINT.DLL. Ask for the ODBC Desktop Drivers, Version 2.0. The cost in the USA is $10.25. If you have MSDN Level 2 they are on the January Development Platform CD 4, in \ODBC\X86 with the ODBC 2.1 SDK.

There is apparently an update for these drivers for Access 2.5 files on the MSACCESS CompuServe forum.

Note that the Access ODBC driver included with certain Microsoft applications (eg MS Office) can only be used with other MS applications. They may appear to work but sooner or later will come back to bite you! The line to watch out for (and avoid!) in the ODBC Administrator drivers list is Access 2.0 for MS Office (*.mdb).

You can install new ODBC drivers using the ODBC Administrator in Control Panel.

Adding An ODBC Data Source

If you've got all the files you need, you are ready to proceed. The example presented here uses the Access 1.10 driver supplied with Delphi. Using the ODBC Administrator, set up a data source for your Access files:

- Click Add in the data sources dialog box to display the Add Data Source dialog, then select Access Files (*.mdb) (or whatever the appropriate entry is for the driver you have installed).
- In the ODBC Microsoft Access Setup dialog (Figure 1 top) enter a name in the Data Source Name. We'll use My Test. Enter a description of the Data Source in the Description text box.
- Click Select Database to open the Select Database dialog. Navigate to the directory where your Access .MDB files are and select one (Figure 1 bottom).
- We'll select a file TEST.MDB in a directory C:\DELPROJ\ACCESS. Click OK in the Setup dialog.

You will now have an entry in the list of Data Sources of My Test (Access Files *.mdb). Click Close to exit the ODBC Administrator. You can set up more Data Sources as required, using the same method.

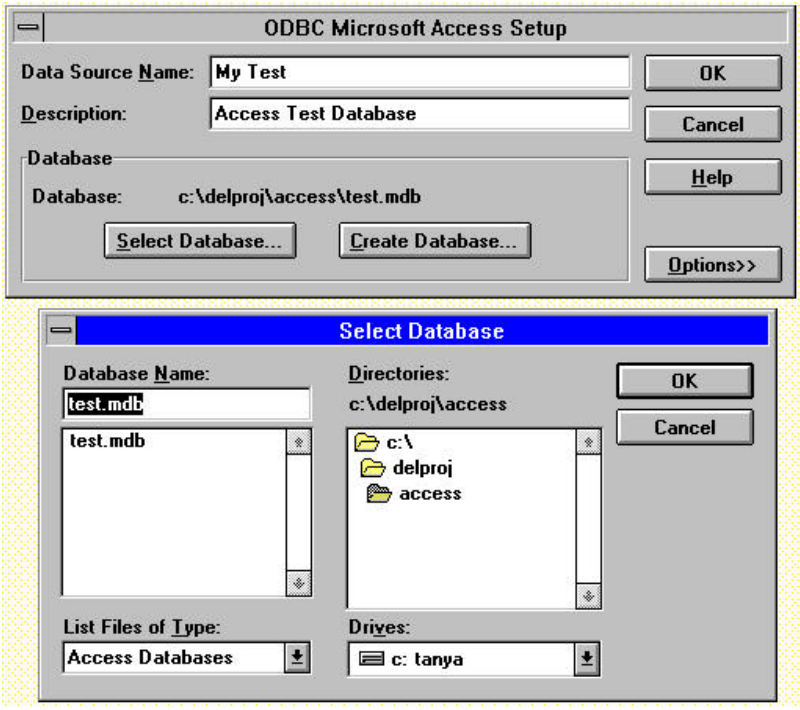


Figure 1

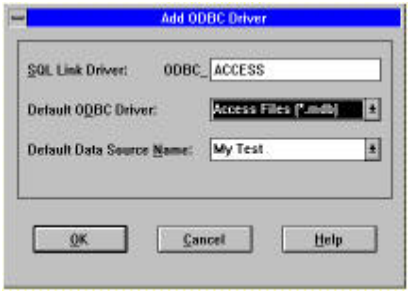


Figure 2

Figure 3

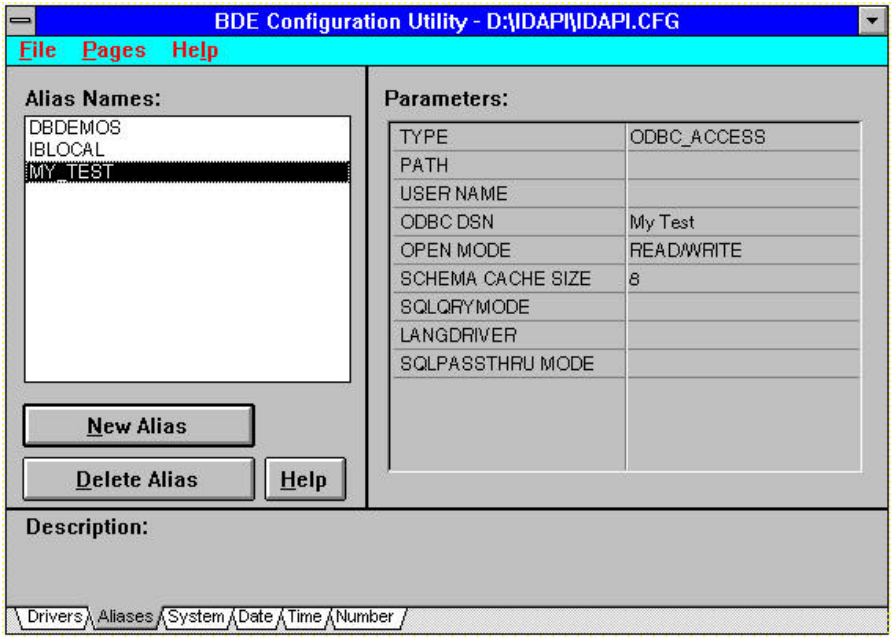


Figure 4

Don't forget the Subscription Form on Page 41!

Setting Up The Borland Database Engine

Now load the Borland Database Engine (BDE) Configuration Utility. On the Drivers page click New ODBC Driver. Note that this adds an Access driver to BDE and is a completely separate exercise to adding an Access driver to Windows using the ODBC Administrator.

- In the Add ODBC Driver dialog presented (Figure 2), enter ACCESS (or whatever you like) in the top edit box. BDE adds ODBC_ in front automatically.
- In the combobox underneath select Access Files (*.mdb). Select the Data Source in the next combobox (Default Data Source Name) – these are the Data Sources you have set up in the ODBC Administration Utility. You needn't worry which one you select, as you can change it later as we'll see. Click OK.

Once you've set up this BDE Driver, you can use it to talk to more than one ODBC Data Source, using a different **Alias** for each ODBC Data Source. To set up an Alias, switch to the Aliases page and click New Alias.

- In the Add New Alias dialog (Figure 3), enter an Alias Name of your choice. We'll use MY_TEST (note that spaces are not allowed).
- In the Alias Type combobox, select the ODBC driver name that you just created (in our case ODBC_ACCESS). Click OK.
- If you have more than one ODBC Data Source, alter the ODBC DSN Parameter ("DSN" = "Data Source Name") in the list of Parameters for the Alias to the appropriate ODBC Data Source (Figure 4), as set up in the ODBC Administrator.
- Note that you needn't add anything in the Path Parameter, as the ODBC Data Source already has this information. If you ***do*** add a Path, make sure it's correct or things won't work!

Now save the BDE configuration by selecting File|Save on the menu bar, then exit the Database Engine Configuration Utility.

In Delphi

Start a new Project and drop a Table and DataSource onto the form from the Data Access page of the Component Palette. Then drop a DBGrid onto the form from the Data Controls page.

- For the Table, set the DatabaseName in the Object Inspector to MY_TEST – the Alias you set up in the BDE Configuration Utility (Figure 5). Now go down and click the button in the TableName combobox.
- You will be asked to Log In to the Access MY_TEST database. Note that you don't need a User Name or Password unless they have been specifically set up, so just click OK.
- After a pause the list of available tables for the ODBC Data Source pointed to by the BDE Alias you have set up will appear in the combobox. Select TEST (Figure 5 again).
- For the DataSource, set the DataSet property in the Object Inspector to Table1 (Figure 6).
- For the DBGrid, set the DataSource property in the Object Inspector to DataSource1 (Figure 7).
- Return to the Table and set the Active property to True in the Object Inspector.

The data from the TEST table will now be displayed in the grid. And that's all there is to it!

One thing to beware of is that if you create an application which uses Access tables and then Run it from within the Delphi IDE, you will get an error if you attempt to modify the data in the table(s). If you run the compiled .EXE file outside of Delphi (having closed Delphi), everything will be fine.

The ODBC error messages unfortunately tend to be obscure and difficult to relate to what's going on in your application, but checking the setup in the ODBC Administrator and the BDE Configuration Utility usually sorts things out.

If you want more reference information, try the ODBC 2.0 Programmer's Reference and SDK Guide from Microsoft Press (ISBN 1-55615-658-8, US price $24.95).

As we go to press, there have been various reports of problems with using ODBC for getting at Access files. Some of these reports have been from users of Delphi field test versions, where things didn't work fully. Other reports are simply a result of users not using the correct procedure to set up ODBC links – which is why we've included this run-down. It is possible that there may still be some genuine difficulties remaining and if so we'd like to hear from you!

Finally, note that if you need to create new Access 1.10 tables you can use the Database Desktop included with Delphi.

Helping with the answers for the problems in The Delphi Clinic for this issue were Ralph Friedman (CompuServe 100064,3102), Bob Swart and Chris Frizelle. Thanks everyone! Don't forget to keep sending in your problems and queries to us...

Figure 5

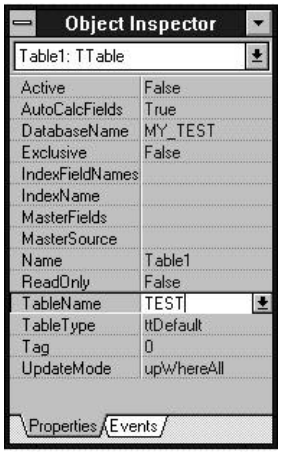


Figure 6

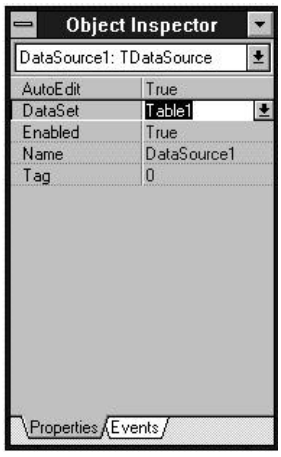
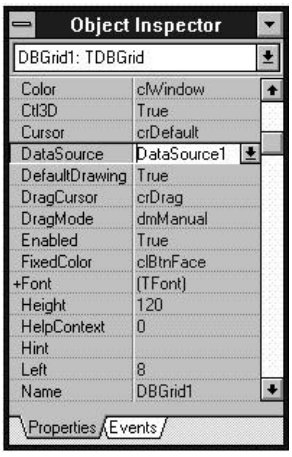


Figure 7



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. We have a similar column in our sister publication, The Pascal Magazine, and it is one which readers constantly comment on as being especially useful. However, we need your input, so please do send in your Tips & Tricks to us (preferably by email), whether large or small, on any aspect of Delphi or related issues.

Assigning Initial Property Values

Something which can seem confusing until you get the hang of what's happening is assigning initial values to properties when writing Delphi components. Consider the tiny Delphi component shown in Listing 1. If we compile the unit as a component and add it to the component palette, we see that the value assigned in the `Create` routine is not there when we try to use the component in Delphi.

All properties have a default value. If none is specified in the *property definition*, the value is whatever evaluates to zero. The streaming code does not write a property to the form file if its value is its default value. This causes the `Create` code to be executed whenever we set the value of `Info` to zero in the Object Inspector, while the streaming code is only executed if the value from the Object Inspector is set to something other than zero.

Rather than try to initialise `Info` in the `Create` procedure, we should have given the property an initial (default) value when defining the `TBob` class using the `default` keyword:

```
published
  property Info: Integer
    read FInfo write SetInfo
    default 1;
```

Auto-deleting Code

Delphi not only automatically generates code, it is also capable of deleting event handler code on request. If the event handler has no code between the begin and end statments, saving or recompiling the project will cause the method to be removed from the source. Note that an event handler does NOT disappear just because you've removed the last reference to it. You might be "unhooking" the handler from one button and getting ready to create another button and reconnect the handler to it.

Exit Windows API

The `ExitWindows` function has always been incorrectly documented. Microsoft got it wrong in their documents and everyone else followed along with this incorrect documentation. The correct definition is:

```
function ExitWindows(
  dwReturnCode: LongInt;
  Reserved: Word): Bool;
```

Actually you can define it other ways, but the important thing is that the return code itself ***must*** be in the word at `[BP+8]` (from `ExitWindows`' point of view). The definition in the API help file places the return code at `[BP+6]`.

The tips in this issue were contributed by Bob Swart (email: CompuServe 100434,2072)

Listing 1

```
unit DRBOB;
interface
uses Classes, Controls, StdCtrls, Dialogs;
Type
  TBob = class(TWinControl)
    private
      FInfo: Integer;
      procedure SetInfo(Value: Integer);
    protected
      constructor Create(AOwner: TComponent); override;
    protected
      StartButton: TButton;
      procedure StartButtonClick(Sender: TObject);
    published
      property Info: Integer read FInfo write SetInfo;
  end {TBob};
  procedure Register;
implementation
  constructor TBob.Create(AOwner: TComponent);
  begin
    inherited Create(AOwner);
    Info := 1; { Try to set FInfo to 1 at Design time }
    StartButton := TButton.Create(Self);
    StartButton.Parent := Self;
    StartButton.Visible := True;
    StartButton.Caption := 'Click me...';
    StartButton.OnClick := StartButtonClick;
    SetBounds(Left,Top,90,36)
  end {Create};
  procedure TBob.StartButtonClick(Sender: TObject);
  var InfoStr: String;
  begin
    Str(Info,InfoStr);
    MessageDlg(InfoStr, mtInformation, [mbOk], 0)
  end {StartButtonClick};
  procedure TBob.SetInfo(Value: Integer);
  begin
    FInfo := Value;
  end {SetInfo};
  procedure Register;
  begin
    RegisterComponents('BOB', [TBob]);
  end {Register};
end.
```

Don't forget the Subscription Form on Page 41!

Review: The Chief's Installer Pro

Reviewed by Bob Swart

So, you've written your first Delphi application and it's just great. Now, all you have to do is write the installation program. Or add that one missing feature that makes your great app a killer app... Unfortunately, you only have time to do one thing. Then it's time to pick up a copy of *The Chief's Installer Pro for Windows*. It will lift the burden of the install program from your shoulders, and give you more time to concentrate on your own application instead!

The Chief's Installer Pro can be used to install and optionally **uninstall** your Windows applications. It is able to copy files from up to 45 floppy disks (although I must say I haven't tested it with more than 3). Files compressed with `COMPRESS.EXE` (and the -r flag) will be expanded automatically using `LZEXPAND.DLL`.

The main program `INSTALL.EXE` can be used quite happily on its own. For foreign language support, you can create a file called `WINSTALL.DLL` with a string table for your non-English strings. The original English version is also included in .RC source form. It is very much appreciated if you send The Chief your "country specific version" of the WINSTALL.DLL (see the address at the bottom).

If the files you wish to install do not fit on one disk, you can use the `SETUP.EXE` program which is also supplied. `SETUP` is just a small loader that makes sure `INSTALL.EXE` and any optional support files are copied to the `TEMP` directory, so you're not continually prompted for Disk 1 with `INSTALL.EXE`.

The actual install instructions for your program are written in an ASCII file called `WINSTALL.INF`. The Chief's Installer Pro comes with some useful example `.INF` files and a detailed Windows Help file with information on the syntax. It's quite simple and yet powerful enough to do whatever you want, such as:

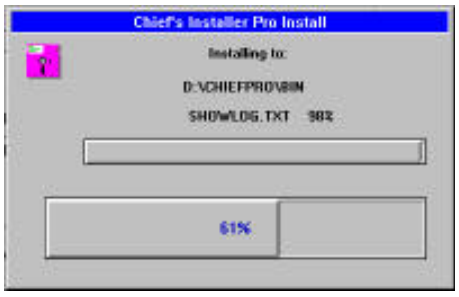
- Partial or optional installations;
- Checking available space;
- Writing to `.INI` files;
- Creating new Program Manager groups and icons for target files;
- A bitmap, banner and progress indicators during the installation;
- Messages or programs to start at the end of or during installation.

It's all there, and more too that we don't have space to discuss! The screen shots (from the install of The Chief's Installer Pro itself) give you a good idea of how your own installation will look.

And then there's the `UNINSTAL.EXE`, which can be used to uninstall any program that was installed with The Chief's Installer Pro. Note that you have to set an option in `WINSTALL.INF` to generate an `UNINSTAL.LOG` file that is needed to uninstall your application. Of course, your application will be so good that nobody would want to uninstall it, but it's a very nice feature to have available!

Main installation screen with the now obligatory colour fade background!

Display of progress information during an installation



The Chief's Installer Pro is shareware. You'll find an evaluation copy on the disk with Issue 2 of The Delphi Magazine (this is a full version, not crippled, but if you want to continue using it then you need to register it of course!). If you can't wait that long, check Library 4 of the CompuServe WINSHARE forum. The registration fee is only £20 or US$29 or equivalent. There are a number of registration sites around the world which accept local funds, or you can register on CompuServe.

Personally, I think The Chief's Installer Pro is a very good program, and well worth the modest registration fee. I've already used it on two of my applications and it enabled me to spend more time on the programs themselves! I can recommend it to anyone who wants their applications to be installed in a professional way without spending the earth. As we go to press we have learnt that it has been awarded a *Gold Award* by *PC Plus Magazine* in their April 1995 UK edition.

The Chief's Installer Pro is written (in Borland Pascal!) by Dr A Olowofoyeku (aka The African Chief), who can be reached c/o John Barton, 57 Baddeley Green Lane, STOKE ON TRENT, Staffs ST2 7JL, United Kingdom. Email to laa12@keele.ac.uk or chief@mep.com

Subscribing To The Delphi Magazine

The best way to stay at the cutting edge of Delphi development is to keep on reading The Delphi Magazine! You will receive 6 issues a year, 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 you will also receive a free disk with each issue, containing all the code from that issue, plus shareware and public domain libraries and tools.

What's the catch? Well, much as we would like to, we can't keep sending The Delphi Magazine to you free, so to keep up with all that's new that's Delphi you need to take out a subscription! We think you'll agree, though, that this represents excellent value for money and we are sure you won't want to miss a single issue. We airmail all overseas copies, too, to make sure you get it *FAST*.

Simply fill out the subscription form below and send it off by post or fax as indicated. If you prefer, you can subscribe by CompuServe™ email (we strongly recommend you do not use Internet email for security reasons). Telephone subscription orders are welcomed on 0181 460 0650 (UK callers) or +44 181 460 0650 (international callers).

The Delphi Magazine Subscription Request

Please complete this form and post or fax it to:

The Delphi Magazine, 41 Recreation Road, Shortlands, BROMLEY, Kent BR2 0DY, United Kingdom

Fax subscriptions to: 0181 460 0650 (UK) or +44 181 460 0650 (International)

TDM 1

Payment must be included, a receipt will be sent to you. Sorry, **NO** purchase orders!

Name (Mr/Ms) _____

Position _____ Company _____

Address _____

_____ City/Town _____

County/State _____ Postcode/Zipcode _____

Country _____ Email _____

Telephone _____ Fax _____

Subscription (please tick)

☐ United Kingdom £25.00 ☐ Europe £27.00 *Note:* varying costs reflect postage.
☐ USA or Canada £32.00 ☐ Rest of the world £34.00 £1 Sterling is approximately US $1.50.

Payment Method (please tick)

☐ Please debit my ☐ Visa ☐ Mastercard/Access credit card (tick as appropriate) by £_____

Card Number: _____ Expiry date: Month:_____ Year:_____

Name on Card:_____ Signature:_____

Note: Your credit card will be debited by the Sterling amount shown above, converted to your local currency by your credit card company

☐ I enclose a **Sterling** cheque, Eurocheque or bank draft drawn on a **United Kingdom** bank (ie the bank's address on the cheque/draft is in the UK), or a **Eurocheque**, for £_____

Please make your cheque payable to ITEC

Sorry, we can't accept payment by bank transfer or Giro transfer.

Note: please do not send payment in other currencies

About You

What operating systems/environments do you use? (please tick)

☐ Windows ☐ DOS ☐ OS/2 ☐ Macintosh ☐ Unix ☐ Mainframe

Which manufacturer's compiler(s) do you use? (please tick)

☐ Borland ☐ Microsoft ☐ Symantec ☐ Other:_____

What languages do you use? (please tick)

☐ C/C++ ☐ Basic ☐ PowerBuilder ☐ Assembler ☐ Pascal (non-Delphi)

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

Don't forget the Subscription Form on Page 41!

The Delphi Magazine

is an independent journal, published six times per year, for developers using Borland's Delphi product. It is published in the United Kingdom by iTec and is available by subscription only. All overseas copies are sent by air for prompt delivery.

Editor: Chris J G Frizelle

Contributors:
Ralph Friedman, Jeroen Pluimers, Bob Swart, Dave Jewell, Sundar Rajan, Steve Teixeira, Xavier Pacheco, Mike Scott

Caveat: Whilst we endeavour to ensure that what is published in the magazine is correct, we cannot accept responsibility for any errors or ommissions. If you notice something which you feel may be incorrect, please contact the Editor and we will publish a correction when possible.

Submissions: Letters, comments, ideas for articles and Tips & Tricks pieces are welcomed and should be sent to the Editor.

Contact:
The Delphi Magazine
iTec, 41 Recreation Road, Shortlands
BROMLEY, Kent BR2 0DY, United Kingdom
Tel/Fax: +44 (0)181 460 0650
Email: 70630.717@compuserve.com

Advertising:
Leigh Foster, Advertising Manager
Tel/Fax: +44 (0)1234 241454

Production:
Design and typesetting at iTec.
Printed by GNBA Design & Print, Bawtry
United Kingdom. Tel: +44 (0)1302 710576.