



Whew! I sure am glad to get this one out!

I apologize for the minor delay getting this issue of UNDU out to everyone, but I had a flurry of article submissions right near the deadline, and they were all so good, I decided to put them in. As a result, I think you will find that this is one of the bigger issues. There are a number of product reviews as well as some announcements of new Delphi-related products that are reaching the market. I felt everything had merit and I really had no space restrictions, so I figured "why not?"

Actually, I had planned on blaming the delay on Windows 95... I have been running the betas and preview for the last 6 months or so, and was having nothing but problems. Then, cringing a bit, I installed the release version, and... it worked perfectly. Whatever problem I was having seemed to go 'poof' on this last version (whew...)

Speaking of Win95, the big topic these days seems to be the approaching release of the 32-bit version of Delphi. From what I can see, it will be just as impressive as we would all expect. Borland has assured developers that essentially all it will take to make your application a 32-bit app is simply to pick **Run|Compile**. Of course you will have the ability to add Win95/NT features to your applications as well. One of the things I am most looking forward to is a pretty mundane one: I want all the various Delphi-related windows to register themselves as "toolboxes" so they don't all sit on the Win95 task bar. I know that this is indeed going to be the case.

I plan on reporting as much as I am able about the 32-bit version of Delphi in upcoming issues, so keep tuned to this channel! And keep those articles coming in!

[Product Announcements](#)

[Product Reviews](#)

[What's In Print](#)

[The Beginners Corner](#)

[Ole Amigos!](#)

[Delphi Study Group Schedule](#)

[Tips & Tricks](#)

[Component Cookbook](#)



## The Beginners Corner: Simple Ways to Read and Write

**Paul H. Comitz - CIS 102565,3167**

Long ago in a land far far away I taught *Introduction to Computer Science* at a local community college in China Lake California. We used a product called Borland Turbo Pascal. We used Pascal because it was a "student" language, easy to learn, easy to teach. We wrote our programs for the DOS command line and we were happy to do it. It was a simpler time. Class was fun for both teacher and student.

The folks at Xerox had a computer called the Xerox Star. This system used desktops, icons, file folders, and stuff like that. Sound familiar? It should, it changed our lives. Unfortunately, Turbo Pascal was dead. Gone the way of the slide rule and the vacuum tube. Replaced by C and C++ application frameworks (remember the 15 pound box of stuff) MFC, and OWL. In order to write a program one needed a prerequisite in buttons, dialog boxes and general GUI-ology. For some of us it was just plain tedium.....Welcome back . Don't you just \*LOVE\* Delphi!!!! Programming is fun again!

I used to assign my Turbo Pascal students a program called **Palindrome**. This was a maddening program on par with the **Game of Life** and the **Tower of Hanoi**. The program took an integer as input and converted it to a palindrome (extremely useful in global thermo nuclear blenders and dishwashers). As an exercise I decided to convert this program to Windows using Delphi (and why not I never really forgot all the Pascal I used to teach).

The entire Palindrome Program is too long or too maddening (or too both) for our newsletter. Instead we'll look at one of the procedures from Palindrome. The procedure accepts an integer as input and reverses it. Huh? The procedure accepts an integer like 12345 and converts it to 54321. There that wasn't so bad. My students generally handed in code that looked like this:

```
Program Palindrome;

Var
  Original,Reversed: Integer;           (* Global *)

Procedure ReverseANumber (VAR Reversed: Integer; CopyOfOriginal: Integer);
Var
  Last:Integer;                         (* Local *)
Begin
  Reversed := 0;                         (* ReverseANumber *)
  While CopyOfOriginal <> 0 Do
  Begin
    Last := CopyOfOriginal Mod 10;      (* ReversedANumber *)
    Reversed := Reversed * 10;          (* Get Last *)
    Reversed := Reversed + Last;        (* Build Reversed *)
    CopyOfOriginal := CopyOfOriginal Div 10; (* Discard Last *)
  End;
  End;

Begin
  Write('Enter a Number  :');          (* Palindrome *)
```

```

Readln(Original);
ReverseANumber(Reversed,Original);
Writeln('The Reversal of', Original:3, 'is ', Reversed:3);
End.                                     (* Palindrome *)

```

Simple... right? The work loop is **While CopyOfOriginal <> 0 Do**. The actual algorithm for reversing a number is left to the reader as an exercise (don't you just hate that), because this is supposed to be a column about Delphi and not the Pascal I used to teach.

When I sat down to do the conversion, I realized I was like the emperor with the fancy new duds. Why is that? I didn't know how to read and write. Stop laughing. For many of us we spent our entire lives using things like *read*, *readln*, *write*, *writeln*. Our assembly language friends used *in*, *out*, and *DIOC* (direct input output control). Some of us moved on to *printf* and *scanf*. Well, none of that stuff works any more. You've got to let it go and move on. Life is like that.

A great tool for writing is *FmtStr*. You won't find *FmtStr* if you do a topic search in delphi.hlp. (credit where credit is due: I read about *FmtStr* in DDJ August 1995). DDJ compares *FmtStr* to *printf* because it allows the insertion of a numeric value into a string. I used *FmtStr* like this:

```

FmtStr(Tmp,'The reversed number is %d',[Reversed]);
AnswerLabel.Caption := Tmp;

```

In the example above *Tmp* is of type string and *reversed* is of type LongInt. *AnswerLabel* is a Delphi TLabel component. If the value of *reversed* is 2345, *AnswerLabel.Caption* would be displayed as:

```
The reversed number is 2345
```

Isn't that easy. Instantly we have a painless way to insert data into text strings. I didn't try this with real numbers or any other data type, but I'll bet a nickel (which is a bet of honor) that they work too.

What about read? Use the Delphi TEdit component. TEdit objects accept keyboard data from the user when the TEdit component has focus. Like most Delphi components there's lot's properties we programmers can use. In the Palindrome program, I decided to use the *modified* property. The *modified* property gives the programmer a way to tell if the text in the TEdit component has changed. I used the *modified* property to tell if the user had entered a new number in the TEdit component. Here's the Delphi version of the main work loop in the original DOS Procedure ReverseANumber (see above).

```

if (Edit1.Modified) then
begin
  Reversed := 0;
  Original := StrToInt(Edit1.Text);           {Does not require carriage return
etc}
  AdviseLabel.Caption := '';
  while Original <> 0 Do
  begin                                       (* ReversedANumber *)
    Last := Original Mod 10;                 (* Get Last *)
    Reversed := Reversed * 10;              (* Build Reversed *)
    Reversed := Reversed + Last;
    Original := Original Div 10;            (* Discard Last *)
  end;                                       (* while *)
  {Now display it }
  FmtStr(Tmp,'The reversed number is %d',[Reversed]);
  AnswerLabel.Caption := Tmp;
  Edit1.Clear;                               {Clear the edit box}
end                                           {then}
else
  AdviseLabel.Caption := 'Enter a new number';

```

The initial test on *Edit1.Modified* is required since we don't wait for a carriage return like we did with *read*

or readln (yes, yes, homage is paid here to the endless number of other string manipulation routines that could have been used... lighten up this is a beginners column). Once we determine that new data is present, we convert the string to an integer using StrToInt ( ) (note that there is no error checking on data entry...perhaps another day). From there the code is remarkably like its DOS ancestor until we get to the part where we need to write the result. Then we use the FmtStr as described above.

The complete listing and DFM for the converted Delphi version appears below. Feel free to E-mail me with any comments or questions. I'll try to answer them if I can. Until next month, try to remember that its OK ask questions, its OK start slowly, and that Microsoft doesn't really HAVE to rule the world.

[Project Source Code](#)

[Project Form Data](#)

[Return to Front Page](#)



## Beginners Corner Project Form (DFM File)

```
object Form1: TForm1
  Left = 335
  Top = 477
  Width = 435
  Height = 215
  Caption = 'Reverse a Number - PHC 8/13/95'
  Font.Color = clWindowText
  Font.Height = -17
  Font.Name = 'System'
  Font.Style = []
  PixelsPerInch = 120
  TextHeight = 20
  object AnswerLabel: TLabel
    Left = 32
    Top = 152
    Width = 4
    Height = 20
  end
  object EditCaption: TLabel
    Left = 32
    Top = 32
    Width = 129
    Height = 20
    Caption = 'Enter an integer'
  end
  object AdviseLabel: TLabel
    Left = 152
    Top = 110
    Width = 4
    Height = 20
  end
  object ReverseButton: TButton
    Left = 32
    Top = 104
    Width = 97
    Height = 33
    Caption = 'Reverse it!'
    TabOrder = 0
    OnClick = ReverseButtonClick
  end
  object Edit1: TEdit
    Left = 32
    Top = 56
    Width = 121
    Height = 29
    Ctl3D = True
    ParentCtl3D = False
    TabOrder = 1
  end
end
```

[Return to The Beginners Corner](#)

[Return to Front Page](#)

## **Light Lib Images VCL for Delphi**

### **DFL Software**

DFL Software has announced that their much anticipated Light Lib Images VCL for Delphi is now shipping. Light Lib Images is the first native professional imaging VCL available for Delphi. It provides Delphi programmers with the ability to add high-quality document and image management capabilities to their applications.

*"We've been integrating advanced object-oriented programming techniques into our product development for the past three years and the benefits are really starting to show. Combining our object-oriented VCL's with Delphi's object-oriented architecture is truly an incredible match."* said company Vice President Brian Loesgen.

Light Lib Images is the first imaging library to support the PNG format (public domain replacement for GIF), and support for this format is included in the Delphi VCL.

Light Lib Images supports all Windows printers and video resolutions and features industry leading TWAIN scanner support. Scanning, displaying, saving, retrieving, printing, zooming, rotating, flipping, scaling, converting, gray-scaling and dithering are fully supported for BMP, PCX, TIF, GIF, JPG, TGA, PNG file formats. It also supports Huffman, RLE, LZW, CCITT 1D/Group 3 and 4 fax and JPEG compression systems.

#### **For further information please contact:**

Brian Loesgen	Voice	(416) 789-
DFL Software Inc.	2223	
1712 Avenue Road	Fax	(416) 789-
Box 54616	0204	
Toronto, ON, M5M 4N5	Cellular	(416) 917-
CANADA	3573	
	BBS	(416) 784-
	9712	
	CIS	74723,3321

**Readers: Look for a review next issue! - Editor**

[Return to Product Announcements](#)

[Return to Front Page](#)



## Sending Messages

by Bert Evans - CompuServe: 74457,340 - Internet: [bevans@usit.net](mailto:bevans@usit.net)

Recently I was writing a small application which would read a text file containing a list of graphic files and display them on the screen. One picture would be displayed at a time and the user would have the ability to press a command button to display the subsequent picture or to scroll back to the previous picture. One of the things I was particularly keen on investigating was Delphi's exception handling capabilities, namely the try..except blocks. Development progressed very rapidly until I hit a bit of a roadblock. I wanted Delphi to check to see if the datafile containing the list of bitmaps could be found. If Delphi could not find the file, I wanted it to display a messagebox saying the file could not be found and exit. I was handling this processing during the OnCreate event of the form with a procedure that looked basically like this:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  try
    if not FileExists('FILELIST.TXT') then
      raise EFileListNotFound.Create('Unable to locate FILELIST.TXT!');
  except
    on E: EFileListNotFound do begin
      MessageDlg(E.Message, mtError, [mbOK],0);
      Close;
    end;
  end;
end;
```

where EFileListNotFound is an exception I declared earlier in the unit. Unfortunately, this code doesn't quite work as expected. The exception is raised properly, but the call to Close is completely ignored. What actually happens is the messagebox is displayed and the form opens as it normally would. I couldn't find anything in the documentation concerning this being a problem, but I've verified it several times and have read messages both on CompuServe and the Internet where others have experienced the same problem (boy don't you wish there was some \*good\* documentation for Delphi!).

At this point I hypothesized that there was nothing particularly wrong with the Close method itself, there just must be some problem calling Close during the OnCreate event. I had run into this kind of problem several times before as a PowerBuilder developer. There are several instances in PowerBuilder (and in Windows programming in general) where you simply cannot call a particular function at a particular time. Sometimes, simply for logical reasons you want to delay a particular piece of code from processing immediately. In PowerBuilder, I would have gotten around this fairly simply by declaring a user message for the window and then coding what I wanted to happen in the event. As it turns out, you can do the same thing in Delphi and it's not really all that difficult!

First, perhaps we need a brief discussion of why I felt a user message was the solution to this problem. The Windows operating system relies on messages to inform applications of events that have occurred which pertain to that application. Basically, Windows monitors the user for any input and relays the input to the application in the form of messages. It does so by placing the messages in a message queue

designated for that application. The application reads the messages in the queue one by one and handles the input as necessary. This is, of course, a simplistic explanation of what's going on, but a complete description of the windows messaging system is chapter, if not book size material (the WINAPI.HLP file contains a discussion of Windows messages under the heading, you guessed it, Messages). The point is, by placing a user message in the queue, I can allow Delphi to finish whatever it is doing that is keeping the Create method from executing. After it finishes processing the OnCreate event, Delphi will grab the next message off the queue, which should be my message. At that point I can tell the application to close.

There are three steps to the process of creating your own user message and handler. First, you need to declare your message as a constant. To do this, I created a constant section right after the uses clause in my form unit and declared my message like so:

```
const
  UM_CLOSEAPP = WM_USER + 25 ;
```

Several things jump out for discussion immediately on review of this single declaration. First, windows messages typically follow the convention of a prefix followed by an underscore followed by the particular message. In this case, UM represents User Message, and CLOSEAPP is just an indicator to me what this particular message is all about (again, browse through the WINAPI.HLP file for examples of how Windows declares messages, or, if you have access to a C/C++ compiler, browse through the windows.h file where all the windows messages are declared). WM\_USER is a message Windows defined especially for this purpose, allowing users to define their own messages. The reason I added 25 to WM\_USER is that some components utilize WM\_USER messages to implement their own additional behavior, and you would not want to declare your message with the same number. As a matter of habit, to avoid this conflict, I typically add 25 or 50 to WM\_USER.

The next step is to declare a procedure on your form which will act as the message handler. This is as simple as declaring any other procedure, with a single additional bit of code:

```
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
    procedure UMCloseApp(var message: TMessage); message UM_CLOSEAPP;
  public
    { Public declarations }
  end;
```

Note the message UM\_CLOSEAPP tacked on to the end of the declaration. This tells Delphi that it should execute the UMCloseApp procedure when it receives the UM\_CLOSEAPP message. I typically name the procedure after the user message sans the underscore. Also, one interesting quirk-the TMessage variable, message, is bold faced. Delphi recognizes message as a keyword, but still allows it to be used as a variable. Nothing showstopping, I just thought it was weird.

Finally, you need to write the procedure to handle the message when it is received. In this case, the procedure is pretty trivial:

```
procedure TForm1.UMCloseApp(var message: TMessage);
begin
  Close;
end;
```

Note that it is not necessary here to add the message clause at the end of the declaration. In fact, if you do, you will get an error message. Also, don't forget the TForm1 prefix (I \*always\* forget!).

Well, now we have everything in place! All we have to do is send the message. There are two ways of handling this, or rather two functions provided by windows to send messages to an application,



SendMessage and PostMessage. SendMessage sends a message to the identified window and waits until that window has processed the message before returning. PostMessage puts a message in the identified window's message queue and returns immediately. In our case, we want to use PostMessage. SendMessage would put us right back in the same boat we were in before in that the Close method would be executed right smack in the middle of our OnCreate procedure. PostMessage, however, will just stick our message in the queue for processing after the OnCreate event has completed. This is how the code should look now:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
try
  if not FileExists('FILELIST.TXT') then
    raise EFileListNotFound.Create('Unable to locate FILELIST.TXT!');
except
  on E: EFileListNotFound do begin
    MessageDlg(E.Message, mtError, [mbOK], 0);
    PostMessage(Handle, UM_CLOSEAPP, 0, 0);
  end;
end;
```

Execute this and yes! Finally, the window closes as expected. PostMessage accepts 4 arguments, the handle to the window your trying to send a message to, the message you're trying to send, an lparam and a wparam argument. Every window that's completely instantiated has a Handle. The handle is the primary way windows distinguishes one window from another. Delphi encapsulates the handle as a property of the form for easy access. The lparam and wparam arguments are typically used to pass additional information to the application, like a pointer to a particular structure, or the identity of the key that was just pressed, etc. We had no need to send any additional information so a zero suffices.

User messages can be used in many different ways. I hope this brief article may help you add them to your personal arsenal of windows programming techniques!

### **Two technical notes:**

The more astute of you will note that I did not need to declare my own user event to tell the application to close; windows already has a message for that, WM\_QUIT. Replacing Close in the original code with a PostMessage(Handle, WM\_QUIT, 0, 0) would have done the same thing.

As to why Close cannot be called during the OnCreate event- I imagine it has something to do with the fact that the window is not yet completely instantiated, ie the Handle does not yet exist. The presumes, however, that Close uses the handle to close the application. Perhaps someone can come up with a more detailed technical explanation as I do not have access to the VCL code (yet!).

[Return to Tips & Tricks](#)

[Return to Front Page](#)



## Review of Delphi Unleashed

by Ian Cornish - CompuServe: 100064,3326

Author: Charles Calvert  
Publisher: SAMS  
ISBN: 0672 30499 6  
Price: UK £35.50  
US \$45.00  
Can \$61.95

This book has already been given a brief review in UNDU, but it's about time it was given an in-depth review. I personally had never programmed in Pascal before Delphi, but am experienced with SQLWindows and Access. I had been tasked to develop a relatively complex application in Delphi with less than 4 months to do it in. Also, I have never written a book review before, so please bear with me!

I looked at the Borland Delphi documentation that came in the box, and felt I was missing a lot. I needed something to teach me a lot, and to ask as a reference, in as short a time as possible. I found Delphi Unleashed and instantly liked the style and range of material covered.

The book covers Delphi in a manner that allows you to start programming as quickly as possible. I was writing some code, and developing small applications within a few days of buying the book.

Part 1 introduces Delphi, the environment, the principles, and the things that makes Delphi special. The coverage of the IDE, and the Visual tools is very detailed, and there are lots of examples, which are also on the included CD.

Part 2 gives a more detailed 'analysis' of the language elements, and approaches the language from an angle that allows you to also use it as a reference. It handles variables, loops, numbers (floats & integers have their own separate chapters) and branching. I have read this section several times, and still refer to it on occasion.

Part 3 is where things get a little harder, since complex data types are introduced, along with Pointers, PChars and file I/O. Being used to languages in which a string can be up to 64K, it is a little frustrating to have two types of string. The book makes use of the 'Memory Theatre' idea for visualizing the use of memory, and this gets star billing in this part of the book. I must admit I had some fun in this part of the book, since I was also getting to grips with the Windows 95 memory system at the same time!

At this stage we are about halfway through the book, and I was merrily writing the code segments for my application. I now needed to put the information into a database. Cue Part 4, which talks about Data access, and Data aware controls. You are introduced to BDE and the Database Desktop, and then given a small Database app to work through. The examples are excellent, and relevant to what you might want to do with Delphi anyway! Following this simple application, the user is then moved on into a section talking about how to use the data controls programmatically. I had finally hit the jackpot, achieving what I can do in SQLWindows, and Access, and exactly what I wanted to do with my major application. Part 4 also covers Interbase, and a useful chapter on designing databases for use out there in the big bad world.

The final part covers OOP and 'Advanced Topics'. The coverage of OOP is significant, and good. I have yet to see a better OOP mechanism than in SQLWindows (Which is quite visual), and unfortunately Delphi does not match this visual approach. However, the 'Advanced Topics' include Multimedia, DDE, OLE, DLL's, VCL, and exceptions. I was surprised to find Exception handling at the end of the book,

rather than in the middle, where it might have had more significance.

All in all, I have enjoyed this book. It hasn't become a doorstop, as some of the earlier attempts at a Delphi book. It has brought me from novice to reasonably competent Delphi programmer in a short space of time, and has let me get a working application out of the door at the same time. At the beginning I said I had 4 months to do a project. I've done it in 2. Thank you Delphi.

The only thing I didn't like about this book is the lack of coverage of Windows 95 specific topics, but then I can't really expect that in such a complete book anyway. If you need, or want to learn Delphi quickly, then my advice is to put the Borland books away, and get Delphi Unleashed. The book is very complete, and the CD contains all the source code, and a number of shareware & demo applications, which may or may not be useful.

[Return to What's In Print](#)

[Return to Front Page](#)



## Gupta SQLWindows vs. Borland Delphi

by Ian Cornish - CompuServe: 100064,3326

As a professional SQLWindows developer, and an amateur Delphi user, I felt it was about time someone did a comparison of the two applications. I have been using SQLWindows for over a year now, and Delphi for about 2 months. Both products have a lot of good things, and a number of bad things. I am comparing Delphi Version 1 with SQLWindows version 4 & 5. The following table is a summary of what I like and hate about each package.

<u>ITEM</u>	<u>Gupta SQLWindows</u>	<u>Borland Delphi</u>
<b>Editor</b>	I love Gupta's Folding editor mechanism, as it makes reading large modules easy. However, it is very inflexible in that you can only modify the colours of the text.	Very easy to use, and very flexible. Obviously buying Brief helped Borland here.
<b>Language</b>	SAL is a pretty big language, but Gupta has the 'Options Bar' from which you can build up your command. This point + click idea makes it easier for novices.	You need to learn Object Pascal before you can use Delphi, but having said that Pascal is pretty simple.
<b>Price</b>	Aimed at corporate users, the application starts at £100, and goes up to £3000, depending on the options required.	£199 for the standard version, but the street price is a bit lower.
<b>Runtime Requirements</b>	You need to ship a whole stack of supporting DLL's (about 1.5MB min). The SQLWindows compiler	The executable, and that's it. OK, the executable is at least 200K, but that's a lot better than 1.5M.
<b>Performance</b>	SLOW.	FAST!
<b>Database Access</b>	Using SQL with SQLWindows is probably the strongest	This is my main problem with Delphi. It took me

point. Simply set up the Username, password, and database name, and then fire the SQL.

over a week to work out how to fire a bit of SQL at a database, and then I wasn't impressed. I needed 2 components, and a lot of code to fire a sequence of SQL.

**Reliability**

I have had a lot of problems with corrupt source code caused by bugs in SQLWindows. However, it's VBX support is good, and you don't need to recompile a library just to try a VBX. Also, if you get into an infinite loop, you have to reset the machine, and wave bye-bye to your changes.

I've crashed Delphi twice now, and those are because I passed a bad variable to a WinAPI function.

**Support (Technical)**

Excellent. The CIS forum is very good, as is the Hotline support.

Poor. OK the CIS forum is not bad, but I have posted questions, and not had a reply.

**Support (Third Party)**

Not very good.

Getting better.

[Return to Product Reviews](#)

[Return to Front Page](#)



## Crystal Reports & UCRPE15

by Robert Pullan - President: Lighthouse Technologies - CIS: 102162,2711

For those of you who are cynical and don't believe in the goodness of man, or the ability of non-vendors to develop significant Delphi compliant products, have I got a surprise for you...

### UCRPE

UCRPE15a.ZIP contains a Delphi wrapper VCL that exposes the Crystal Reports Paradox Engine (CRPE.DLL - a 1992 ancestor to BDE) to the developer and was NOT developed by Crystal. UCRPE was developed by John Murphy III (CIS: 72623,2075) .

UCRPE exposes more than 40 properties of a Crystal Reports report to the developer, about triple the number of properties TReport allows. Most of the major features a Crystal Report user can perform can be controlled programmatically at design time from within Delphi or permanently fixed in the base report. If you feel constrained by the limits or abilities of TReport, this is a solution for you.

UCRPE has been tested with xBase, Paradox, MS Access and SQL/ODBC and works well. It is fast, stable and powerful. Because CRPE is using a BDE ancestor, it is important to remember to set the **ShareLocal** to **YES** in the [Paradox Settings] section of your Win.Ini file if you are using Paradox ... otherwise CRPE and BDE will go to war. Obviously, UCRPE requires that the developer own a copy of Crystal Reports Professional (at least v 4.0.1.3). Crystal's tech staff recommends UCRPE to Delphi users who prefer an alternative to the out-of-the-box report writer and I do too.

A real tip-o-the-hat goes to John, hope he continues to develop this first class VCL.

### Crystal Reports Professional v.4

What can I say that hasn't already been printed? Crystal Reports is a report writer of the first magnitude. It handles every database format supported by Delphi and can and can display data as text or as graphics. It has more than 100 functions, which can be included in the base report or stipulated at design or runtime. A well designed Crystal Report can transform raw data into eyepopping reports guaranteed to get attention. Crystal also ships with a 16-bit OCX and advertisements for the 4.5 version (which looks to be a native Win95 variant). Crystal claims they will include a 32-bit OCX too - suggesting this is a product with a Delphi32 future.

Crystal Reports offers two additional features: Licensed end-users can configure reports and compile and distribute the compiled reports (royalty-free). This offers unparalleled flexibility in distribution of production reports to local PC's, as well as the flexibility of additional user designed production reports for Delphi apps.

Crystal offers a C/S style report server variant which expands Crystal Report's capabilities manyfold and delivers even more production, distribution and access controls.

Crystal Reports comes in two flavors: a Standard "End User" version and the Professional version. The Pro version includes the CRPE engine which is needed to make UCRPE operate and is distributable royalty free. The Pro version also includes a VBX version of Crystal which works to a more limited extent in Delphi, and which I would recommend Delphi developers skip.

While Crystal does not seem to be doing much to develop a VCL, it is clearly adapting the CRPE to be a Delphi compliant product.

**Synopsis:**

<b>C</b> -----							
<b>Object Files</b>	<b>Vendor</b>	<b>Style</b>	<b>Loads?</b>	<b>Functionality</b>	<b>Code Examples</b>	<b>Demos</b>	<b>PAS</b>
<i>UCRPE15a</i>	<i>John Murphy 72623,2075</i>	<i>VCL</i>	<i>Yes</i>	<i>Terrific</i>	<i>Help File</i>	<i>None</i>	<i>VCL</i>
<i>Crystal Reports</i>	<i>Crystal 800-877-2340</i>	<i>DLL</i>	<i>Yes</i>	<i>Terrific</i>	<i>None</i>	<i>None</i>	<i>n/a</i>

[Return to Product Reviews](#)

[Return to Front Page](#)

## Formula One

*by Robert Pullan - President: Lighthouse Technologies - CIS: 102162,2711*

In a word, the Delphi compliant version (2.0.3) of **Formula One** is: *WOW!*

If you are looking for a Spreadsheet VBX component, Formula One is a serious product worthy of serious consideration. It is a first class Excel 4 work alike. Make no mistake about it, it is NOT Excel, but it does offer end users more spreadsheet than probably 99.9% of them will use.

The Delphi Formula One product comes as a two class system, spreadsheet and edit box, requiring both a VBX and a DLL to function meaningfully. New buyers will get the needed Pascal DLL headers and a Delphi code sample included with the product. It's developers, Visual Components, promise a Delphi specific maintenance release for current owners at some point in the future. Registered users can download the needed Delphi DLL headers today by calling tech support and getting the password required to download the header (and I would recommend this). The header (VTSSH.PAS) is needed if one wants to access the considerable and necessary functionality resident in the DLL (like printing). There also is a truly amazing MDI example on the BBS, in which First Impression (VC's graphics product) is interactively integrated into the app, if one has the as-yet-to-be officially released First Impression DLL headers... ask for them too if you own First Impressions, but remember they are pre-release.

Formula One is not a perfect Excel 4 clone. It supports its own spreadsheet format: \*.VTS. The VTS format supports several non-Excel compliant/enhanced features, like the ability to include proprietary objects and functional Buttons, Check, and Drop Down List boxes on a spread and integrate First Impression graphics. If an end-user attempts to import an Excel spread that contains unsupported Formula One features (like Excel graphics), Formula One will nuke those features if it re-writes the file. On balance, this is not a bad trade-off, but rather a caveat to users.

Formula One is fast. My test of a 3000 line with about 12,000 unique calculations Excel spread, revealed on recalculation it was faster than Excel 5! While on the topic of Excel 5, Formula One doesn't support Excel 5's worksheet format ... but few other products which read spreadsheets do either, Excel 4 is the spreadsheet compatibility standard. It certainly beats the heck out of an OLE link to Excel ...

Formula One is a V3 compliant VBX, which gives it considerable data awareness in other programming environments. Unfortunately, this power is not available to Delphi users, because Delphi only supports V1 VBXs - which precludes data aware VBXs (among other limitations). While less than ideal, Formula One's cell content aware functions (set and get - numbers, text and formulae) do work well. It can read and write two dimensional arrays too, so Delphi database integration is still possible. You can also create a database of spreadsheets, but absent is any database functionality under Delphi. You will need to write the contents of the BLOB field containing the spreadsheet to a file then read the file into Formula One.

If control is what you seek, Formula One gives the programmer more than 350 functions to manipulate spreads and the VBX. It also gives end users more than 125 worksheet functions!

Documentation is extensive. However, it contains no Delphi-specific support. The reader needs to know a bit of VC++ or VB if it is to be of any value. Additionally, since Formula One is a V3 VBX, not all functions listed in the book function ... welcome to trial and error land. This is perhaps the greatest shortcoming of this otherwise fine product.

However, looking beyond the data aware limitation and lack of Delphi manuals Formula One is chock-full of features and functionality, a real powerhouse of an add on.

Formula One version 1 is a component of the Borland Visual Solutions Pack. Since BVSP's release predated Delphi the proper Pascal DLL headers for Formula One were not included. Without the proper headers version 1's functionality within Delphi is trivial. However, VC is offering special pricing for anyone seeking to buy Formula One now. Version 2 adds substantial increases on almost twenty major enhancements in power, functionality and sophistication over the BVSP sample product and is an upgrade worthy of the price.

Formula One V2 is also a component in the Developers Visual Suite Deal (DVSD). It joins Visual Writer as the only components that operate meaningfully in Delphi. It is my understanding that the developer



plans to release Pascal DLL headers for First Impression (a charting product) and Visual Speller at some time in the future.

Visual Components, the developer, is deeply committed to producing 32-bit OCX variants of their products... which is good news for future Delphi32 development. This commitment seems to have consumed a large amount of their limited resources and has taken precedence over fulfilling Delphi-related promises. Since Formula One is the only major component in the Developer's Visual Suite Deal that functions meaningfully in Delphi, I would at this time advise readers to defer purchases of DVSD until the other components become a truly Delphi compliant products.

While Visual Components is not alone in over committing resources, they need to work harder at fulfilling the expectations of the many buyers who have been disappointed in Delphi promises unfulfilled in a timely manner.

I would not hesitate for a minute to recommend Formula One, it is here (at last), it is real, it functions wonderfully in Delphi and it is a truly powerful tool.

### Synopsis:


----- D E L P H I   S P E C I F I							
C -----							
<b>Object Files</b>	<b>Vendor</b>	<b>Style</b>	<b>Loads?</b>	<b>Functionality</b>	<b>Code Examples</b>	<b>Demos</b>	<b>PAS</b>
<i>Formula One</i>	<i>Visual Components</i> <i>800-884-8665</i>	<i>VBX DLL</i>	<i>Yes</i>	<i>Substantial</i>	<i>None</i>	<i>Demo</i>	<i>Yes</i>

[Return to Product Reviews](#)

[Return to Front Page](#)

## Faster String Loading

by Ray Lischner - Internet: [lisch@tempest-sw.com](mailto:lisch@tempest-sw.com)  
Tempest Software, Corvallis, Oregon, USA

The June 26th issue (#5) of the Unofficial Newsletter of Delphi Users contains a tip for faster string loading . When reading a lot of strings into a control such as a list box or memo, it suggests using a temporary **TStringList** instead of loading directly into the control's string list. The performance difference can be as great as 500%. Unfortunately, this tip misses the real reason for the performance difference...

Using a separate **TStringList** and then assigning that to the control's string list is a circuitous way of using *BeginUpdate* and *EndUpdate*. When loading many strings into any string list, call *BeginUpdate* before adding any strings, and call *EndUpdate* when you are finished:

```
Memo.Lines.BeginUpdate;  
try  
    {add lots of strings to Memo.Lines...}  
finally  
    Memo.Lines.EndUpdate;  
end;
```

By using *BeginUpdate* and *EndUpdate* directly, you avoid the overhead of creating a temporary **TStringList** and copying all yours strings twice. Try this out, I'm sure you will see the difference!

[Return to Tips & Tricks](#)

[Return to Front Page](#)



## OLE, Amigos

*by Robert Pullan - President: Lighthouse Technologies - CIS: 102162,2711*

As I have wandered through the third party support world for Delphi I have been struck by how important OLE technology will become and how inadequate support of OLE will limit 16-bit Delphi in a meaningful way. The good news, OLE will cause the third party market to explode in the next couple of years and that will help improve selection and pricing. The bad news is that it may not explode in ways that specifically take advantage of Delphi's power.

Object Linking and Embedding (OLE) is the Windows feature which allows two programs to interact with each other at a very deep level. OLE 2.0 offers three modes of interaction:

1. **Client** - (which is the only feature Delphi Supports), allows users to launch or initiate an OLE session.
2. **Server** - is the application which operates the launched OLE application.

OLE Automation which is related the OLE Server except the user can gain direct access to specific parts of an otherwise larger server application, e.g.. the spell checker function of MS Word, or perhaps operate the Access database engine and avoid ODBC.

OLE version 2 is currently implemented in Windows 3.1x and is also implemented with Windows 95. Microsoft has made a huge bet on this technology and many worry about them delivering on its promises for OLE. OLE has many real world weaknesses, among the most critical has been its memory intensive nature. OLE takes significant resources as well as the OLE Client and OLE Server, which typically add up to problems on a less than industrial strength PC's... that's why most OLE examples use the Windows Paintbrush program as an example... its a small OLE server.

The backbone of 16-bit third party programming tools has been a specialized DLL called a VBX. It offers developers the opportunity to write programming components which can operate in a wide range of programming environments. Further, since VBX's are DLL's, they can operate with both compiled and interpreted products. They are as close to universal components as we have yet to see.

In 32-bit land the VBX paradigm will not exist, and it will be replaced with an OLE based OCX component. The advantage of OCXs, in theory is that if a program can access OLE as a Client and operate OLE programmatically, it can use OCXs. This expands the world of "universal" third party tools from strictly development environments to more end-user products like Access, Approach, Paradox and even other components of 32-bit "office suites".

Additionally, OCXs may partially solve the OLE memory problem, by being generally smaller applications than most current OLE Server enabled apps, they require fewer resources and can be addressed serially if more functionality is needed.

Many vendors are also bringing out 16-bit variants of the OCX. In theory, programs written using OCX accessing code can be operated as either 16 or 32 bit products, provided the user has the appropriate 16 and 32 bit OCXs. This too could prove appealing to developers who want to offer 16 and 32-bit application support, and believes in the "one set of universal code compiles to everything" fairy.

Back to reality. It is clear that many third party vendors are currently inundated with trying to convert everything they have to 32-bit OCXs (and 16-bit OCX) as quickly as possible. Since most third party

companies are small, they lack the depth of programming bench strength to handle more than a few big projects at once. This forces a need to focus upon the "biggest bang for the buck". If one looks at the list of major compiled programming versus interpreted products that represent potential markets for third party vendors you can see the reason for such activity.

<u>Major Compiled</u>	<u>Major Interpreted*</u>
MS Visual C++	MS Visual Basic
Borland Visual C++	PowerBuilder
Borland Delphi	Borland dBase
	MS FoxPro
	MS Access
	Lotus Approach   - "Suite products"
	Borland Paradox
	MS Excel

\* They all may not currently support OCX, but probably will be forced to support it in the near future.

Not only do major interpreted products outnumber compiled products, but a closer look at the total number of users... Interpreted product users far outnumber compiled product users ... OCX is clearly the wave of the future.

The immediate market for 32-bit apps seem unclear to me. Microsoft would have us all believe 16-bit development will halt after August 24th... given the 54 million PC's that have NEVER had Windows installed, the considerable hardware requirements of Win95, and the significant continuing business demand for delivery of Win3x in lieu of Win95 on new hardware deliveries, I believe reports of 16-bit development's demise have been greatly exaggerated.

What this all means for Delphi users is:

1. 32-bit/16-bit OCX development is a major cause for delay of Delphi specific VCLs or needed modifications to existing VBXs to make them Delphi compliant.
2. The market for 32-bit OCXs may become crowded quickly, and the market for Delphi VCLs may be under-served for some time... which represents an opportunity for some third party developers, (hint, hint).
3. The lack of 16-bit OCX support, and the "off-center" VBX implementation by Borland will limit Delphi developer's ability to use third party products in the future, (especially as third party developers discover it is easier to maintain 16 and 32-bit variants of OCXs, than to develop, enhance and maintain separate OCX, VBX , VCL, MFC and OWL variants.)
4. As support of the OCX increases, and OLE improves, the lack of support for OLE server or OLE automation in 16-bit Delphi will also hurt (it is promised for Delphi32).
5. While VCLs certainly offer substantial technical advantages, the advantages only work for Delphi users. This means they suffer from a small(er) market potential than OCXs, albeit less crowded, market.

Virtually every third party vendor I have spoken to, has heard from Delphi users and recognize the market for Delphi compliant products is substantial. As a group they are struggling to develop OCX based variants, and promise Delphi is the next horizon.

The problem of bogus Delphi compliant claims for many components still exists and is widespread. It is clear from my e-mail that my piece on Delphi compliant standards hit a nerve. Hopefully, this will shed some light on the underlying causes for delay.

As users we need to support those vendors that are providing truly Delphi compliant products, and continue to encourage those vendors who offer unsupported claims and promises, with our promises that we will buy Delphi compliant products when they deliver Delphi compliant products.

[Return to Front Page](#)



## Study Group Schedule

For those of you who are not aware a Electronic Delphi Study Group is now being conducted on the Informant Communications forum of CompuServe. Thomas J. Theobald (75662,2073) is moderating the group and by the amount of message traffic involved, it appears to be generating a lot of interest.

Basically, you can think of the SG as a kind of electronic "classroom". There is assigned reading from the groups textbook (Delphi Unleashed), and in between there are open discussions about the material considered. In addition, the class will be doing a project starting a bit later in the year. The project will be the joint development of a Personal Information Manager in Delphi.

Printed below is the groups study schedule. If you want to observe or participate, you can go to the forum (GO ICGFORUM) and see what's going on. For further information on the study group, please contact Tom Theobald on CompuServe at 75662,2073.

### Study Group Schedule:

8/12	Ch. 1-3	
8/19	Ch. 4,5	
8/26	Ch. 6	
9/2	Ch. 7	
9/9	Ch. 8	
9/16	Ch. 9	
9/23	Ch. 10	
9/30	Ch. 11	
10/7	Ch. 12	
10/14	Ch. 13	
10/21	Ch. 14	
10/28	Ch. 15	
11/4	Ch. 16	
11/11	Ch. 18	
11/18	Ch. 19	
11/25	Ch. 20	
12/2	Ch. 21	I think about here we'll be ready to start a module or three for the PIM project. Probably the initial interface, and the calendar. Maybe something else too. We'll see.
12/9	Ch. 22	
12/16	Ch. 23	
12/23	Ch. 24	
1/6	Ch. 25	
1/13	Ch. 26	About here we'll start on the address book of the PIM. Maybe work on preparing some basic reports (like mailing labels for people in the address book and stuff like that).
1/20	Ch. 27	
1/27	Ch. 28	
2/3	Ch. 29	

- 2/10 Ch. 30 About here we should have a fully-functional PIM on hand, and we will be into bug-catching by now.
- 2/17 Ch. 31
- 2/24 Ch. 32
- 3/2 Ch. 33
- 3/9 Ch. 34 Final touches to PIM, zip it up and post it as freeware for everyone to have a gander at. We will include both the final EXE set and the source code, for future reference by other students. Don't forget to keep a copy in your own personal portfolios!!
- 3/16 Ch. 35 We're Done!!!! Wow. This'll feel good by now.

[Return to Front Page](#)



## Creating .DBF and .MDX Files On The Fly

by Dr Charlotte Gamsu - CompuServe: 100556,1107

Sooner or later a database developer will want to write an application which can create databases and indexes. For example, I usually write software which, on first being installed, automatically creates all the data files and necessary indexes for itself. This means I can keep the size of the application I ship to a minimum - and it ensures that data files are created just where I want them to be! And, as an additional facility I usually include a menu option to allow the user to re-index all data files at will.

I am so accustomed to this way of producing software that it took me quite by surprise to find that Delphi and its developers seem to have overlooked providing this functionality for Delphi. The Help files shipping with Delphi concentrate on Paradox files and much of the necessary information to enable you to write programs to create dBase files and indexes on the fly is either wrong or totally missing! Even the user manual for the Borland Database Engine does not fill in the gaps.

### Tricks and Techniques

Once you know how, however, it is quite a simple task to write self generating programs. Listing 1 provides examples of all the tricks and techniques you will need to create databases with the full range of data types. Once you've been shown how, it really is a piece of cake! To try the code out, simply create a blank form. Plop a button on it and attach the following code to the OnClick event.

```
procedure TInitDbf.Button1Click(Sender: TObject);
begin
  with TTable.Create(Self) do
  begin
    DatabaseName := 'c:\steve\inidbf';
    TableName := 'Register.dbf';
    TableType := ttDBase;
    with FieldDefs do
    begin
      Add('CLIENT_NO', ftString, 7, false);
      Add('DATE', ftDate, 0, false);
      Add('PAIDFEES', ftFloat, 0, false);
      Add('INACTIVE', ftBoolean, 0, false);
      Add('APPOINTMENTS', ftInteger, 0, false);
      Add('notes', ftMemo, 0, false);
    end;
    CreateTable;
    with IndexDefs do
    begin
      clear;
      AddIndex('Register', 'CLIENT_NO', []);
      AddIndex('CASENOTES', 'Client_No'
        + Str(year(Date), 4) + ' '
        + Str(month(Date), 2) + ' '
        + Str(day(Date), 2), [ixExpression]);
    end;
  end;
end;
```



Let me walk you through the key bits of the code. The code starts off just as you'd expect:

```
with TTable.Create(Self) do
begin
  {This is the directory name}
  DatabaseName := 'c:\steve\inidbf';
  {This is my file name }
  TableName := 'Register.dbf';
  {I like being explicit. If you want you can use 'ttDefault' here.
  If your TableName has the '.dbf' extension the 'ttDefault' will
  still result in a .dbf file being created}
  TableType := ttDBase;
end;
```

The syntax of the statements to create fields must list 4 terms:-

1. Name of the field.
2. Type of the field.
3. Length of the field - for a large number of the possible data types length is not an issue. The compiler knows the length of an Integer, Float, & Boolean. As far as I can work out, you only need to specify the length of ftString fields.
4. A Boolean value. According to the documentation **True** flags the fields as an 'essential' field which must be filled. I don't use the facility in my applications so I always define my fields as **False**.

The syntax of the statements to create indexes caused me much grief. But I eventually got it working. And it is not quite as Borland advises! According to the Borland Database Engine, on page 182: *"Using CREATE INDEX is the only way to create indexes for dBASE tables."* Well this doesn't work for me! The on-line help shipping with Delphi is not much better. It gives an example using **IndexDefs.Add**. After much trial and error and many a loud howl, I can confirm this just does not work - its AddIndex not just Add!

To create an index in an MDX file, you need to specify 3 terms:

1. The name of the index - e.g. REGISTER,
2. The field to be indexed - e.g. Client\_No **or** a valid dBASE Expression for an Expression index, such as my CASENOTES index. When creating an expression use 'standard dBase' syntax and function calls.
3. If the index is just a simple, one-field index, leave the brackets of the third term blank! The enticing options like **ixUnique** etc are for Paradox files. Leave them well alone. If you have created an Expression index then you **MUST** put **ixExpression** in the brackets. Don't try to look for this in the on-line help, it's not there.

## Conclusion

Creating dBase files and indexes on the fly is something most of us will want to do. The advice in this article is: Ignore the written documentation. Don't read the manual! Keep your sanity. Take the easy route. Use the listing provided in this article as a template.

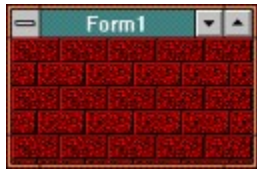
## Author Information

Charlotte Gamsu is director of a software house specializing in software for the legal market. She has written applications for this market as well as for oil industry, smoked fisheries, motorcar distributors and educational establishments. She can be contacted on 100556.1107@compuserve.com and cgamsu@taynet.co.uk

[Return to Tips & Tricks](#)

[Return to Front Page](#)





## Wallpapering A Window

by Bert Evans - CompuServe: 74457,340 - Internet: bevans@usit.net

Sometimes I just don't know what I'm getting myself into...

I surf the Delphi forum on CompuServe and the Delphi usenet groups daily. Primarily I'm looking for tips, tricks, or answers to questions I've previously posted. Now and then, when I find a question I can answer myself, I do. Suddenly, just when I don't suspect it, someone posts a message with an unobtrusive looking question, one that's probably been asked more than once, one that no one has really completely answered. 'Gee,' I think to myself. 'That can't be too hard to do.' That's when it begins. That's when I get sucked into a maelstrom of acronyms and features I've only heard hinted at before - hooks, subclassed windows, and API calls, oh my! If I'm lucky, just when I've slogged through about as much windows muck as I can handle, it all suddenly becomes clear, my prototype program works, and I brilliantly answer the original posters question. Well at least it seems brilliant to me. Its why I like working with Delphi so much! Every time I turn around I'm learning something new.

This time the question was about how to 'wallpaper' a window. Windows makes it very easy to put wallpaper on the desktop, its just a setting on the control panel. Many developers have noted that it would be a neat thing to add to MDI or SDI windows as well, and indeed, many developers have already figured out to do it. It's a neat, simple way to dress up your application a bit, or maybe get in a little free advertising by tiling the company's logo unobtrusively in the background. The problem is there's no quick and dirty way to do it: No miraculous API call for this one. So how do you create wallpaper for your window? What I wanted to create was a solution typical of Delphi - a component I could just drop on the form, set a bitmap, and viola! Wallpaper!

What immediately springs to mind is that Delphi already comes with a pretty neat component for handling bitmaps, TImage. I suppose you could dynamically create TImage components and tile them in the background, but aside from many other problems I saw in that approach, you would end up using far too much memory and resources, especially if your bitmap was small. So the question is how get the bitmap image onto the form.

Fortunately, there's a pretty easy answer to this question. All TForm components come supplied with a Canvas component, which is really a wrapper around the device context for the window. Quoting from the help documentation from Delphi, "a Device Context (DC) is a link between a Windows application, a device driver, and an output device...." Suffice to say, whenever you're doing any graphics work in Windows, you're working with a device context. To me, a device context is analogous to a sheet of paper. If I'm gonna draw a picture, I need something to draw on. To get back to the point, the easiest way to create wallpaper for a window is to load the bitmap into memory and then copy it as many times as necessary on the Canvas to cover the client area on the window.

So the first thing we need in our component is a method to replicate a bitmap across the client area of the window based on height and width. The problem then, is that the background needs to be redrawn every time something happens that changes the appearance of the background, like resizing the window, moving the window, placing another window on top of the window, etc. So, another essential part of any wallpaper component we might write is a means to detect changes to the window and act accordingly. Since we know that individual windows are notified of changes by the system through messages, what we need is some method of catching those messages as or before they are received by the window.

To do this, I'll use a technique called 'subclassing.' All windows (and most visible components) have a window procedure which the operating system uses to send messages to the window. The window procedure examines each message that comes through and decides whether it needs to do anything for that message. Subclassing is where you replace the primary window procedure with your own window

procedure. The advantage is that all messages will now be sent to the procedure you designate. This allows you to examine each message and act on it, or pass it on to the main window procedure before the window procedure ever receives the message.

Creating the initial code for the component is pretty easy. Just pull down File|New Component from the main Delphi menu and follow the prompts. I decided to name my component TWallPaper, and inherit from the TImage component. Also, I designated that the component should be listed as a Custom component. Hit the OK button and Delphi generates a basic component template for me to work with. The first thing I wanted to tackle was setting up the subclassing system so that my component could intercept the appropriate messages. The message we're looking for in particular is the WM\_ERASEBKGD message, which tells the window it needs to redraw its background. The subclassing system consists of two main functions, InitWallPaper() and WPWindowProc(). WPWindowProc is the window procedure I want to use to replace the form's window procedure, InitWallPaper is the function I'll use to set it up.

Subclassing is accomplished by making a call to the SetWindowLong() API call using GWL\_WNDPROC as an argument to say we're trying to replace the window procedure. One problem, however, is that Windows is not expecting the method of a VCL object as the replacement window procedure. In fact, the compiler will not allow you pass an object method as the argument to SetWindowLong. Fortunately, Delphi includes a function called MakeObjectInstance especially for this purpose. MakeObjectInstance creates a wrapper function that windows can call which in turn calls the object method. That problem solved, the code that sets up the new window procedure looks like this:

```
FWPWindowProc := MakeObjectInstance(WPWindowProc);
FOldWindowProc := Pointer(GetWindowLong(FDrawToHandle, GWL_WNDPROC));
SetWindowLong(FDrawToHandle, GWL_WNDPROC, LongInt(FWPWindowProc));
```

where FWPWindowProc and FOldWindowProc are of type TFarProc, and FDrawToHandle is of type HWND. FWPWindowProc now stores a pointer to my window procedure. We use FOldWindowProc to hold a reference to the original window procedure so that we can call it for those messages we don't want to handle. FDrawToHandle, is a handle to the window I want to subclass.

One interesting problem which cropped up while I was creating the component is that a form with the fsMDIForm FormStyle actually consists of two separate windows. The reason this is a problem is that the form's handle is not the handle to the window which actually displays the client area. So if we were to snag the form handle of an MDI form and use it to get our context for painting, we would never actually see the bitmap as it would be hidden behind the client area window.

Again, Delphi provides us a way to get around this via the ClientHandle property of the form. As such, prior to subclassing the window, we need to check whether the form is an MDI form or not. If it is, we set FDrawToHandle to the ClientHandle, otherwise FDrawToHandle is simply set to Handle. This is going to cause us another problem, however, down the road when we get around to painting the bitmap.

WPWindowProc consists of a single case statement which checks for WM\_ERASEBKGD. If the WM\_ERASEBKGD message is received, WPWindowProc calls PaintWallPaper() which is the function which actually paints the wallpaper (boy that was hard to guess), and it modifies the message by setting the result to 1, which informs windows that our application has handled the WM\_ERASEBKGD message. If any other message is received, WPWindowProc passes the message along to the original window procedure using CallWindowProc. Note: in a later version of the component, I modified WPWindowProc to capture WM\_SIZE as well. I found that WM\_ERASEBKGD was not always being called when the window was resized, so I wanted to explicitly repaint the background when the window was resized.

The only major piece of code left to write at this point was the procedure which actually painted the background. This turned out to be slightly trickier than anticipated because of the problem mentioned above (where an MDI form actually consists of two different windows). Had that not been the case, we could have simply determined the dimensions of the bitmap and the visible canvas and used CopyRect or Draw to paint multiple copies of the bitmap on the canvas until the background was filled. Indeed, this is still feasible with FormStyles other than fsMDIForm. Since I wanted to write this component so that it could handle any FormStyle, however, I needed a method to draw to the client area window. Since we already have a reference to the appropriate window via FDrawToHandle, the easiest way to do this is to

use the handle to get a reference to the device context and then use BitBlt() to paint the bitmap. That solved, the algorithm for painting the background is relatively trivial-just paint the bitmap across the background in rows and columns until the entire background is filled.

After a bit of testing, and some minor polishing, I added my brand new component to the library using Options\Install Component. After some more testing and a lot more polishing, its reached the state you see it in the code listing. To use TWallPaper (after you've added it to your library) drop it on your form and double click on it to select a bitmap (don't select an icon or metafile, that won't work!). Then, in the OnCreate event of your form, add this line:

```
WallPaper1.InitWallPaper(Self);
```

Then run your application! As far as I've been able to test it runs bug free. The only drawback I've noticed is that with large bitmaps, scrolling the form tends to be a bit herky jerky.

All in all, the component turned out pretty well. There are several other pieces I'd like to add to it in the future. One, and I've pretty much worked out how to do this, I'd like to get rid of the need to explicitly call InitWallPaper. You should be able to just drop the component on the form, select a bitmap and go! Also, I'd like to add several functions to allow changing the wallpaper programatically, or to disable wallpaper altogether.

On reflection, I don't think TImage was the best component to inherit from. I works quite nicely, but there's a lot of extra baggage that comes along with it I'd like to get rid of. An early beta tester (hehe) suggested it would be nice to see the background at design time. I'd like to hear your ideas and suggestions! Try it out and let me know what you think.

[Wallpaper Source](#)

[Return to Tips & Tricks](#)

[Return to Component Cookbook](#)

[Return to Front Page](#)



## Wallpaper Source Code

```
unit Wpaper;

interface

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

type
  TWallPaper = class(TImage)
  private
    { Private declarations }
    FWPWindowProc, FOldWindowProc: TFarProc ;
    FDrawToHandle: HWND;
    FDrawToDC: HDC;
    FDrawForm: TForm;
    FDrawHeight, FDrawWidth: Integer;
    procedure WPWindowProc(var message: TMessage);
  protected
    { Protected declarations }
    procedure PaintWallPaper;
  public
    { Public declarations }
    procedure InitWallPaper(AOwner: TComponent) ;
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    { Published declarations }
    property AutoSize;
    property Height;
    property Left;
    property Name;
    property Picture;
    property Stretch;
    property Tag;
    property Top;
    property Width;
  end;

  procedure Register;

implementation

procedure TWallPaper.PaintWallPaper;
var
  Row, Column : Integer;
begin
  if FDrawForm.FormStyle = fsMDIForm then
    begin
      FDrawHeight := FDrawForm.ClientHeight;
      FDrawWidth := FDrawForm.ClientWidth;
    end
  else
    begin
      FDrawHeight := FDrawForm.Height;
      FDrawWidth := FDrawForm.Width;
    end;

  for Row := 0 to FDrawHeight div Picture.Height do
```

```

    for Column := 0 to FDrawWidth div Picture.Width do
        BitBlt(FDrawToDC, Column*Picture.Width, Row*Picture.Height,
            Picture.Width, Picture.Height,
            Picture.Bitmap.Canvas.Handle, 0, 0, SRCCOPY);
    end;

procedure TWallPaper.WPWindowProc(var message: TMessage);
begin
    with Message do
        case Msg of
            WM_ERASEBKGD:
                begin
                    PaintWallPaper;
                    Result := 1;
                end;
            WM_SIZE:
                begin
                    PaintWallPaper;
                    Result := CallWindowProc(FoldWindowProc, FDrawToHandle, Msg, wParam,
lParam);
                end;
            else
                Result := CallWindowProc(FoldWindowProc, FDrawToHandle, Msg, wParam, lParam);
            end;
        end;

procedure TWallPaper.InitWallPaper(AOwner: TComponent);
begin
    FDrawForm := TForm(AOwner);
    if FDrawForm.FormStyle = fsMDIForm then
        FDrawToHandle := FDrawForm.ClientHandle
    else
        FDrawToHandle := FDrawForm.Handle;
    FDrawToDC := GetDC(FDrawToHandle);
    FWPWindowProc := MakeObjectInstance(WPWindowProc);
    FoldWindowProc := Pointer(GetWindowLong(FDrawToHandle, GWL_WNDPROC));
    SetWindowLong(FDrawToHandle, GWL_WNDPROC, LongInt(FWPWindowProc));
end;

constructor TWallPaper.Create(AOwner: TComponent);
{constructor for the TWallPaper class}
begin
    inherited Create(AOwner);
    AutoSize := True ;
    Visible := False;
end;

destructor TWallPaper.Destroy;
var
    CurrentWndProc: TFarProc;
begin
    FreeObjectInstance(FWPWindowProc);
    SetWindowLong(FDrawToHandle, GWL_WNDPROC, Longint(FoldWindowProc));
    inherited Destroy;
end;

procedure Register;
begin
    RegisterComponents('Custom', [TWallPaper]);
end;

end.

```

[Return to Wallpaper Article](#)

[Return to Component Cookbook](#)

[Return to Front Page](#)





The Unofficial Newsletter of Delphi Users - Issue #7 - August 31st, 1995

## TCompress Component Set V1.0 File and Database Compression Components for Delphi

**Peter Hyde - Internet: [Peter@spis.equinox.gen.nz](mailto:Peter@spis.equinox.gen.nz)**

TCompress Component Set V1.0 File and Database Compression Components for Delphi is now released. Copies are now on Delphi-related Web and ftp sites worldwide. These components are fully functional SHAREWARE (registration \$NZ 70, approx. \$US 45 or UKL29).

Unsolicited comments emailed to us within 18 hours of release:

*"I'm very impressed with it. It is the first component I've found that does what I want it to... and more."*

*"I didn't realize you could compress image and memo fields - that's also a huge advantage for the program I'm working on."*

*"I think using Delphi demands the need for a multi-file compression VCL, because you've got so many database files... So your compression VCL is a life-saver as you can then simply copy one file instead of 20 when you want to exchange data with someone else using the program."*

### Description:

TCompress provides easy-to-use support for multi-file compressed archives, as well as a wide range of database/file/in-memory compression capabilities using Delphi streams. Two compression methods (RLE and LZW) are built in, with "hooks" for the easy addition of custom compression formats.

TCompress also includes drop 'n play components for automatic database blob, image and memo compression, based on Delphi's TDBMemo and TDBImage components. Images compress by up to 95% when using LZW, hence there is a massive saving in disk space & disk/network access using these components.

TCompress comes with an extensive demonstration (source) and Delphi-compatible help and keyword files.

### Key features:

- \* Multi-file compressed archives
- \* Database BLOB (memo, image) compression
- \* In-memory compression using streams (file/memory/blob to file/memory/blob)
- \* Event hooks for customizable user interaction
- \* Built-in RLE (Run-Length Encoding) and LZW (Lempel-Ziv-Welch) compression
- \* Custom compression routines can be added
- \* Full HLP and KWF files included, plus source of an extensive drag-n-drop demo program

### Includes:

```
COMPRESS.DCU:
  TCompress -- Customizable data compression component
COMPCTRL.DCU:
  TCDBImage -- DBImage component with optional compression
  TCDBMemo -- DBMemo component with optional compression
  TCBlobField -- Compression-enabled version of TBlobfield class
  TCBlobstream -- Compression-enabled version of TBlobstream class
```

**Peter Hyde, South Pacific Information Services Ltd, Christchurch, NZ**  
**Author of TCompress: File & Database Compression Component Set**

[Return to Product Announcements](#)

[Return to Front Page](#)



The Unofficial Newsletter of Delphi Users - Issue #7 - August 31st, 1995

## The Delphi Magazine

You can get a free sample issue of *The Delphi Magazine* if you send an e-mail message with your name & postal address to the editor Chris Frizelle at 70630,717 on CIS or 70630.717@compuserve.com on the internet. You can also call or fax him at +44-181-460-0650.

The publisher of TDM, **iTec**, is also the publisher of *The Pascal Magazine*, so if you're interested in (Object) Pascal, ask for your free sample issue of *The Pascal Magazine* as well! Here are some more details on the past few issues:

[Issue #1](#)

[Issue #2](#)

[Issue #3](#)

[Return to What's In Print](#)

[Return to Front Page](#)



The Unofficial Newsletter of Delphi Users - Issue #7 - August 31st, 1995

## HEADCONV 1.0 - C DLL HEADER CONVERTER EXPERT FOR DELPHI

**Bob Swart - Internet: [drbob@pi.net](mailto:drbob@pi.net) - CompuServe: 100434,2072**

Targeted at the serious Delphi developer, the HeadConv Expert will assist in the process of converting C DLL header files to Delphi import units, giving all Delphi users access to the huge world of third-party C DLLs.

HeadConv has full support for functions and procedures, argument and return types (128 custom type conversions) and generates implicit Delphi import units. The expert is integrated in the Delphi IDE, making the conversion very easy as the converted file is opened up in the IDE automatically. At this time there is limited (non-complex) support for typedefs, structs, unions, enums and conditional compilations. HeadConv is not targeted to do the conversion 100% on its own, rather it will assist in converting the C DLL header files.

*"The whole idea of a C header translator is really great. I think in the long run it will prove more useful than a VB-Delphi translator because so much of the useful low level code that is suitable for a compiled language like Delphi's ObjectPascal is written in C."* said George Watson, a tester.

Another beta tester was equally excited. *"It already saved us two days of monkey work, even when not everything was converted 100% correctly."*

The shareware version of HeadConv 1.0 will be available directly from LIB 22 of the DELPHI forum. Registrations can be made (US\$ 25) in the SWREG forum (id #6533). All registered users will receive source code for the expert and stand-alone EXEs (source code for the parser is not provided!), a detailed WinHelp file, the capability for generating explicit import units, and the chance to send new wishes and receive regular updates by CompuServe mail when available.

Bob Swart (aka Dr.Bob) is a professional software developer and free-lance technical writer using Borland Delphi, Pascal and C++. Dr.Bob writes for several computer magazines and is columnist for The Delphi Magazine and The Pascal Magazine.

Robert E. Swart, Rijnlaan 66, 5704 JG Helmond, THE NETHERLANDS  
[bobs@dragons.nest.nl](mailto:bobs@dragons.nest.nl) & [100434,2072] at home, work: [bob@bolesian.nl](mailto:bob@bolesian.nl)  
Cap Volmac Bolesian, POBox 799, 5702 NP Helmond, fax +31-4920-33985

[Return to Product Announcements](#)

[Return to Front Page](#)

 The Unofficial Newsletter of Delphi Users - Issue #7 - August 31st, 1995



## Product Announcements

[Light Lib](#) - Images VCL for Delphi

[HEADCONV 1.0](#) - C DLL Header Converter Expert for Delphi

[TCompress Component Set V1.0](#) - File and Database Compression Components for Delphi

[Return to Front Page](#)



## **Product Reviews**

[Crystal Reports & UCRPE](#)

[Formula One](#)

[Gupta vs. Delphi](#)

[Return to Front Page](#)

 The Unofficial Newsletter of Delphi Users - Issue #7 - August 31st, 1995



## What's In Print?

[The Delphi Magazine](#)

[Review: Delphi Unleashed](#)

[Return to Front Page](#)



## Tips & Tricks

[Sending Messages](#)

[DBase on the Fly](#)

[Wallpapering Your Windows](#)

[Limiting A Forms Size](#)

[Even Faster String Loading](#)

[Using the ChartFX Component](#)

[Return to Front Page](#)





## Corrected Source Code from Issue #6

```
unit DebugBox;

interface

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

type
  TPositions = (poTopLeft, poBottomLeft, poTopRight, poBottomRight);

  TDebugBox = class(TComponent)
  private
    DebugForm : TForm;
    DebugList : TListBox;
    FPosition : TPositions;
    FVisible : Boolean;
    FWidth : Integer;
    FHeight : Integer;
    FCaption : String;
  procedure SetPosition(A: TPositions);
  procedure SetVisible(A: Boolean);
  procedure SetWidth(A: Integer);
  procedure SetHeight(A: Integer);
  procedure SetCaption(A: String);
  protected
    { Protected declarations }
  public
    constructor Create(AOwner: TComponent); override;
    procedure Add(A: String);
    procedure Clear;
    procedure LoadLines(FName: String);
    procedure SaveLines(FName: String);
  published
    property Caption: String read FCaption write SetCaption;
    property Position: TPositions read FPosition write SetPosition default
poTopRight;
    property Visible: Boolean read FVisible write SetVisible default True;
    property Width: Integer read FWidth write SetWidth default 250;
    property Height: Integer read FHeight write SetHeight default 200;
  end;

  procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Dialogs', [TDebugBox]);
end;

constructor TDebugBox.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FPosition := poTopRight;
  FVisible := False;
  FWidth := 250;
  FHeight := 200;
  FCaption := 'Debug Box';
end;
```

```

if not (csDesigning in ComponentState) then
  begin
    DebugForm := TForm.Create(Application);
    with DebugForm do
      begin
        Visible := FVisible;
        Caption := FCaption;
        FormStyle := fsStayOnTop;
        BorderStyle := bsSizeable;
        BorderIcons := [biSystemMenu];
      end;
    DebugList := TListBox.Create(DebugForm);
    with DebugList do
      begin
        Parent := DebugForm;
        Align := alClient;
        Sorted := False;
        Font.Name := 'Small Fonts';
        Font.Size := 7;
      end;
    end;
  end;

procedure TDebugBox.SetPosition(A: TPositions);
begin
  FPosition := A;
  if not (csDesigning in ComponentState) then with DebugForm do
    case A of
      poTopLeft      : SetBounds(0,0,Width,Height);
      poBottomLeft   : SetBounds(0,Screen.Height-Height,Width,Height);
      poTopRight     : SetBounds(Screen.Width-Width,0,Width,Height);
      poBottomRight  : SetBounds(Screen.Width-Width,Screen.Height-
Height,Width,Height);
    end;
  end;

procedure TDebugBox.SetVisible(A: Boolean);
begin
  FVisible := A;
  if not (csDesigning in ComponentState) then
    begin
      DebugForm.Hide;
      if A then
        begin
          Width := Self.Width;
          Height := Self.Height;
          SetPosition(FPosition);
          DebugForm.Show;
        end;
      end;
    end;
  end;

procedure TDebugBox.SetWidth(A: Integer);
begin
  FWidth := A;
  if not (csDesigning in ComponentState) then
    begin
      DebugForm.Width := FWidth;
      SetPosition(FPosition);
    end;
  end;

procedure TDebugBox.SetHeight(A: Integer);

```

```

begin
  FHeight := A;
  if not (csDesigning in ComponentState) then
    begin
      DebugForm.Height := FHeight;
      SetPosition(FPosition);
    end;
end;

procedure TDebugBox.SetCaption(A: String);
begin
  FCaption := A;
  if not (csDesigning in ComponentState) then
    DebugForm.Caption := FCaption;
end;

procedure TDebugBox.Add(A: String);
begin
  DebugList.Items.Add(A);
  {This makes sure the item just added is visible}
  DebugList.ItemIndex := DebugList.Items.Count-1;
end;

procedure TDebugBox.Clear;
begin
  {Remove all items from the list box}
  DebugList.Items.Clear;
end;

end.

```

[Return to Component Cookbook](#)



## Component Cookbook

*by Robert Vivrette*

Last issue I offered some source code for a Debug Box component. In my rush to get that issue out, I inadvertently included an old version of the source that did not include a method to add items to the DebugBox. Made it kind-of worthless... Anyway, I have included the correct, revised source for last issue here, and then have gone and extended the source later to include the ability to write out the lines to a disk file. Also, some people commented that the debug box was not initially visible. This was done intentionally. That way, it will not be initially displayed, and you can then show it only when you want it.

In addition, we have an **excellent** article from Bert Evans on a component for Wallpapering your windows. If you have a component that you would like to share, please let me know!

[Correct DebugBox Source for Last Issue](#)

[Extending the DebugBox](#)

[Wallpapering Your Windows!](#)

[Return to Front Page](#)



## Modified Source for Debug Box

```
unit DebugBox;

interface

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

type
  TPositions = (poTopLeft, poBottomLeft, poTopRight, poBottomRight);

  TDebugBox = class(TComponent)
  private
    DebugForm : TForm;
    DebugList : TListBox;
    FPosition : TPositions;
    FVisible : Boolean;
    FWidth : Integer;
    FHeight : Integer;
    FCaption : String;
    FStamp : Boolean;
    procedure SetPosition(A: TPositions);
    procedure SetVisible(A: Boolean);
    procedure SetWidth(A: Integer);
    procedure SetHeight(A: Integer);
    procedure SetCaption(A: String);
  protected
    { Protected declarations }
  public
    constructor Create(AOwner: TComponent); override;
    procedure Add(A: String);
    procedure Clear;
    procedure LoadLines(FName: String);
    procedure SaveLines(FName: String);
  published
    property Caption: String read FCaption write SetCaption;
    property Position: TPositions read FPosition write SetPosition default
poTopRight;
    property Visible: Boolean read FVisible write SetVisible default True;
    property Width: Integer read FWidth write SetWidth default 250;
    property Height: Integer read FHeight write SetHeight default 200;
    property TimeStamp: Boolean read FStamp write FStamp default False;
  end;

  procedure Register;

implementation

  procedure Register;
  begin
    RegisterComponents('Dialogs', [TDebugBox]);
  end;

  constructor TDebugBox.Create(AOwner: TComponent);
  begin
    inherited Create(AOwner);
    FPosition := poTopRight;
    FVisible := False;
    FWidth := 250;
  end;
end;
```

```

FHeight := 200;
FCaption := 'Debug Box';
if not (csDesigning in ComponentState) then
  begin
    DebugForm := TForm.Create(Application);
    with DebugForm do
      begin
        Visible := FVisible;
        Caption := FCaption;
        FormStyle := fsStayOnTop;
        BorderStyle := bsSizeable;
        BorderIcons := [biSystemMenu];
      end;
    DebugList := TListBox.Create(DebugForm);
    with DebugList do
      begin
        Parent := DebugForm;
        Align := alClient;
        Sorted := False;
        Font.Name := 'Small Fonts';
        Font.Size := 7;
      end;
    end;
  end;

procedure TDebugBox.SetPosition(A: TPositions);
begin
  FPosition := A;
  if not (csDesigning in ComponentState) then with DebugForm do
    case A of
      poTopLeft      : SetBounds(0,0,Width,Height);
      poBottomLeft   : SetBounds(0,Screen.Height-Height,Width,Height);
      poTopRight     : SetBounds(Screen.Width-Width,0,Width,Height);
      poBottomRight  : SetBounds(Screen.Width-Width,Screen.Height-
Height,Width,Height);
    end;
  end;

procedure TDebugBox.SetVisible(A: Boolean);
begin
  FVisible := A;
  if not (csDesigning in ComponentState) then
    begin
      DebugForm.Hide;
      if A then
        begin
          Width := Self.Width;
          Height := Self.Height;
          SetPosition(FPosition);
          DebugForm.Show;
        end;
    end;
  end;

procedure TDebugBox.SetWidth(A: Integer);
begin
  FWidth := A;
  if not (csDesigning in ComponentState) then
    begin
      DebugForm.Width := FWidth;
      SetPosition(FPosition);
    end;
  end;

```

```

procedure TDebugBox.SetHeight(A: Integer);
begin
    FHeight := A;
    if not (csDesigning in ComponentState) then
        begin
            DebugForm.Height := FHeight;
            SetPosition(FPosition);
        end;
    end;
end;

procedure TDebugBox.SetCaption(A: String);
begin
    FCaption := A;
    if not (csDesigning in ComponentState) then
        DebugForm.Caption := FCaption;
    end;
end;

procedure TDebugBox.Add(A: String);
begin
    if TimeStamp then
        DebugList.Items.Add(TimeToStr(Time)+' - '+A)
    else
        DebugList.Items.Add(A);
        {This makes sure the item just added is visible}
        DebugList.ItemIndex := DebugList.Items.Count-1;
    end;
end;

procedure TDebugBox.Clear;
begin
    {Remove all items from the list box}
    DebugList.Items.Clear;
    {This makes sure the item just added is visible}
    DebugList.ItemIndex := DebugList.Items.Count-1;
end;
end;

procedure TDebugBox.LoadLines(FName: String);
begin
    DebugList.Items.Clear;
    DebugList.Items.LoadFromFile(FName);
end;
end;

procedure TDebugBox.SaveLines(FName: String);
begin
    DebugList.Items.SaveToFile(FName);
end;
end;
end.

```

[Return to DebugBox Article](#)

[Return to Component Cookbook](#)

[Return to Front Page](#)



## Extending the DebugBox

by Robert Vivrette - CompuServe: 76416,1373 - Internet: RobertV@ix.netcom.com

This month, let's add to the project. It would be nice if we were able to load and save the lines to a disk file. This is very simple to do. First, we will make the methods necessary...

```
procedure TDebugBox.LoadLines (FName: String);
begin
  DebugList.Items.Clear;
  DebugList.Items.LoadFromFile (FName);
end;

procedure TDebugBox.SaveLines (FName: String);
begin
  DebugList.Items.SaveToFile (FName);
end;
```

And then include these methods in the "public" section of the component definition so the user can call them:

```
public
  procedure LoadLines (FName: String);
  procedure SaveLines (FName: String);
```

Next, let's add a property that will control the inclusion of a time stamp on each line:

```
private
  FStamp : Boolean;

published
  property TimeStamp: Boolean read FStamp write FStamp default False;
```

And now modify the Add method to do the appropriate work:

```
procedure TDebugBox.Add (A: String);
begin
  if TimeStamp then
    DebugList.Items.Add (TimeToStr (Time) + ' - ' + A)
  else
    DebugList.Items.Add (A);
  {This makes sure the item just added is visible}
  DebugList.ItemIndex := DebugList.Items.Count - 1;
end;
```

Next issue, we will add a Font property to allow the user to control the type face used. **If anyone has any other ideas for the DebugBox component, please let me know and we can add them!** Some ideas I am thinking of:

1. Font property
2. Items colored individually
3. 'Categories' of list items. (would allow removal of ones you aren't interested in).

[Complete DebugBox Source](#)



[Return to Component Cookbook](#)

[Return to Front Page](#)

## **The Delphi Magazine Issue #1**

### *News*

All the latest Delphi news, including new information on new products you can use to make Delphi even better

### ***The Delphi Idiom***

Steve Teixeira gets us into the Delphi groove...

### ***Optimising Display Updating***

Mike Scott shows how to make your screen refresh really fast

### ***Under Construction: Build Your Own Components***

Bob Swart kicks off his regular column by showing us how to get started on writing Delphi components

### ***Moving Up: Borland Pascal***

Dave Jewell gives the Borland Pascal developers amongst us the low down on moving up to Delphi

### ***Introducing Client/Server***

Sundar Rajan cuts through the jargon with a beginner's guide to what Client/Server development is all about and the benefits offered by Delphi.

### ***Delphi Internals: Using and Writing DLLs***

Dave Jewell begins his regular column on low-level Delphi issues by showing us how to use and write dynamic link libraries in Delphi

### ***Book Reviews***

Bob Swart reviews 'Inside Windows 95' by Adrian King

### ***Using The Borland Visual Solutions Pack***

Jeroen Plumers reviews Borland's low-cost pack of VBX custom controls and shows how they can be used to advantage in your Delphi projects.

### ***Animation Made Easy***

Xavier Pacheco knows all about UFOs - well, at least he can tell you how to get them scudding gracefully about your screen! Sprite animation doesn't come much easier than this...

### ***The Delphi Clinic***

### ***Tips & Tricks***

### ***Review - The Chief's Installer Pro***

Bob Swart looks into this shareware Windows installation program which is ideal for installing your Delphi applications - even better, it will be included on the free disk you'll receive with issue 2 of The Delphi Magazine (when you subscribe)

## **Delphi Magazine Issue #2**

### **News**

All the latest Delphi news, including new product information and updates from Borland

### **Book Review**

Bob Swart reviews 'Delphi Programming for Dummies' by Neil Rubenking

### **Delphi User Groups**

#### **SubClassing Windows**

Brian Long shows how to customise your windows and controls using various subclassing techniques

#### **Under Construction: Tic-Tac-Toe**

Bob Swart's component-building column focuses on encapsulating DLLs into Delphi components, by means of a tic-tac-toe game example (better known to our British readers as noughts-and-crosses!)

#### **Moving Up: Visual Basic**

Dave Jewell with helpful advice for those moving to Delphi from VB

#### **Building Quality Help Systems**

Robert Palomo shows how to keep your users happy by designing and writing better Help systems for your applications

#### **Drag & Drop is easy...**

Stuart Lunn shows how to make your applications accept files dragged and dropped from Windows File Manager, with other handy tips along the way

#### **Delphi Internals**

Undocumented Secrets, Part 1 -- Dave Jewell delves under the bonnet of Delphi to look at form resources and how to implement unions

#### **The Delphi Clinic**

#### **Tips & Tricks**

## **Delphi Magazine Issue #3**

### **News**

Updates from the Delphi world

### **Book Reviews**

Bob Swart and Steve Troxell review 'Delphi Programming Explorer' and 'Delphi Developer's Guide'

### **Preview: 32-Bit Delphi**

What you've all been waiting for! It won't be here for a while yet, but Chris Frizelle has details on what to expect in Borland's new version of Delphi for Windows95 and NT

### **Surviving Client/Server: Getting Started With SQL, Part 1**

Steve Troxell kicks off a brand new column for Client/Server developers with the first of a 2-part series on the basics of the SQL language

### **Custom Clipboard Formats**

Xavier Pacheco explains how to put anything you like onto the Windows clipboard

### **Under Construction: Customising Controls**

Bob Swart shows how to derive new components from existing ones, using a right-aligned edit control as an example

### **Moving Up: C And C++**

Dave Jewell has some good advice for C/C++ developers trading up to Delphi

### **Typecasting Explained: Part 1**

Brian Long sheds much-needed light on what even causes experienced developers to trip up!

### **Writing Your Own Experts**

Bob Swart shows that creating Delphi experts isn't that hard, as he takes us through the steps of building a DLL Skeleton Generator Expert

### **Delphi Internals: Through The Language Barrier**

Dave Jewell continues his explanation of creating DLLs with Delphi, showing how to put Delphi forms into DLLs which can then be called from other languages

### **The Delphi Clinic**

### **Tips & Tricks**

## Limiting A Forms Size

by Fredrik Haglund - CompuServe: 100277,347

This is a demonstration how to limit the form size during resize and the forms position and size when it's maximized.

1. Add this line to the private section of the form declaration.

```
procedure WMGetMinMaxInfo( var Message :TWMGetMinMaxInfo ); message
WM_GETMINMAXINFO;
```

2. Add this to the implementation part:

```
procedure TForm1.WMGetMinMaxInfo( var Message :TWMGetMinMaxInfo );
begin
  with Message.MinMaxInfo^ do
  begin
    ptMaxSize.X := 200;           {Width when maximized}
    ptMaxSize.Y := 200;           {Height when maximized}
    ptMaxPosition.X := 99;        {Left position when maximized}
    ptMaxPosition.Y := 99;        {Top position when maximized}
    ptMinTrackSize.X := 100;      {Minimum width}
    ptMinTrackSize.Y := 100;      {Minimum height}
    ptMaxTrackSize.X := 300;      {Maximum width}
    ptMaxTrackSize.Y := 300;      {Maximum height}
  end;
  Message.Result := 0;           {Tell windows you have changed minmaxinfo}
  inherited;
end;
```

NOTE! In windows 95 the screen size can change during runtime. If the Scaled property of the form is true then component size and position can change.

The WM\_GETMINMAXINFO message is sent to a window whenever Windows needs the maximized position or dimensions of the window or needs the maximum or minimum tracking size of the window. The maximized size of a window is the size of the window when its borders are fully extended. The maximum tracking size of a window is the largest window size that can be achieved by using the borders to size the window. The minimum tracking size of a window is the smallest window size that can be achieved by using the borders to size the window.

Windows fills in a TMINMAXINFO data structure, specifying default values for the various positions and dimensions. The application may change these values if it processes this message. An application should return zero if it processes this message.

```
type
  TPoint = record
    x: Integer;
    y: Integer;
  end;

  TMinMaxInfo = record
    ptReserved: TPoint;
    ptMaxSize: TPoint;
    ptMaxPosition: TPoint;
    ptMinTrackSize: TPoint;
    ptMaxTrackSize: TPoint;
  end;
```

```
TWMGetMinMaxInfo = record
  Msg: Cardinal;
  Unused: Integer;
  MinMaxInfo: PMinMaxInfo;
  Result: Longint;
end;
```

The TWMGetMinMaxInfo type is the message record for the WM\_GETMINMAXINFO message. The TMINMAXINFO structure contains information about a window's maximized size and position and its minimum and maximum tracking size.

<i>ptReserved</i>	Reserved for internal use.
<i>ptMaxSize</i>	Specifies the maximized width (point.x) and the maximized height (point.y) of the window.
<i>ptMaxPosition</i>	Specifies the position of the left side of the maximized window (point.x) and the position of the top of the maximized window (point.y).
<i>ptMinTrackSize</i>	Specifies the minimum tracking width (point.x) and the minimum tracking height (point.y) of the window.
<i>ptMaxTrackSize</i>	Specifies the maximum tracking width (point.x) and the maximum tracking height (point.y) of the window.

The TPOINT structure defines the x- and y-coordinates of a point.

I hope you find this information useful. Please send a mail if you there is something I've missed.

[Return to Tips & Tricks](#)

[Return to Front Page](#)



```
begin {PHC}
end.  {Newvrse}
```

[Return to The Beginners Corner](#)

[Return to Front Page](#)



## Using The ChartFX Component

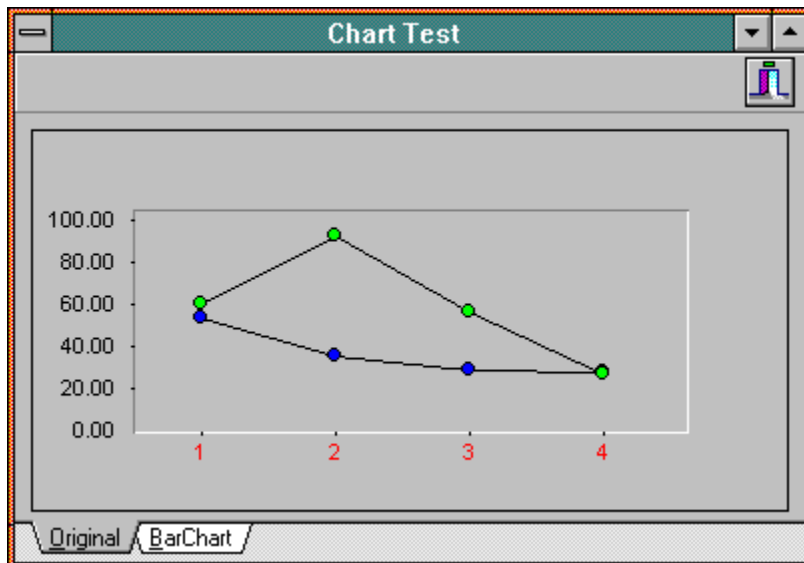
by John Braga - CompuServe: 70134,201

The ChartFX component has some very powerful capabilities, but the online documentation does it a good deal less than justice!

A native Delphi chart component would no doubt be greatly preferable, but ChartFX merits close study. I see that Chart2FX and others are available, but I felt that the ChartFX provided would do all I wanted to do if I devoted a bit of time to experimenting.

In the following single-form project, I took a SDI form, and dropped on it a notebook (NB), a tabset (TB) and 2 chartfx components (Chart1 and Chart2). Chart1 is deliberately left exactly 'as found'. All the changes are to Chart2.

The notebook NB has 2 pages '&Original' containing Chart1, and '&Barchart' containing Chart2.



The code is commented so that you can see the changes I made to create a very acceptable bar chart. Robert Wittig's Reporter component (available as shareware from Delphi 3rd party library) makes it easy to print chartFX components with other items such as data tables. Check it out!

I have no doubt I have still much to learn on barcharts, let alone all the other types available. If anyone has any comments on any of the code or the approach in general, I would welcome comments.

[ChartFX Sample Code](#)

[Return to Tips & Tricks](#)

[Return to Front Page](#)



## ChartFX Sample Code

```
unit TstChart;

interface

uses
  WinTypes, WinProcs, Classes, Graphics, Forms, Controls, Menus,
  Dialogs, StdCtrls, Buttons, ExtCtrls, Tabs,
  ChartFX, {It seems to be necessary to include this for certain constants
  such as COD_VALUES}
  VBXCtrl, Chart2fx;

type
  TF_Chart = class(TForm)
    SpeedPanel: TPanel;
    ExitBtn: TSpeedButton;
    NB: TNotebook;
    TB: TTabSet;
    Chart1: TChartFX;
    Chart2: TChartFX;
    procedure ExitItemClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure TBClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    Procedure Build1( Ch : TChartFX );
    Procedure Build2( Ch : TChartFX );
  end;

var
  F_Chart: TF_Chart;

implementation

{$R *.DFM}

procedure TF_Chart.ExitItemClick(Sender: TObject);
begin
  Close;
end;

procedure TF_Chart.FormCreate(Sender: TObject);
begin
  TB.Tabs := NB.Pages;
  NB.PageIndex := 0;
  Build1( Chart2 );
  Build2( Chart2 ); {adds values, legends etc to Chart2}
end;

procedure TF_Chart.TBClick(Sender: TObject);
begin
  NB.PageIndex := TB.TabIndex;
end;

Procedure TF_Chart.Build1( Ch : TChartFX );
begin
  {This procedure alters the properties that can be set at design
```

*time or run time. The Design method of doing things is shown in the comments}*

**with** Ch **do begin**

Adm[ CSA\_GAP ] := 25.0;  
*{Design: Use the AdmDlg property to change the Y Gap}*

pType := BAR or CT\_LEGEND;  
*{Design: Use the ChartType property to change from 1 - line to 2 - bar.}*

DecimalsNum[ CD\_YLEG ] := 0;  
*{Design: Change the Decimals Property from 2 to 0}*

Stacked := CHART\_STACKED;  
*{Design: Change the Stacked property from 0 - None to 1 - Normal}*

RightGap := 20;  
*{Design: Same}*

OpenData[ COD\_COLORS ] := 2;  
Color[ 0 ] := clBlack;  
Color[ 1 ] := clYellow;  
CloseData[ COD\_COLORS ] := 0; *{Ugh!!}*  
*{Design: To change the colors of the 2 series:  
1) Make sure ThisSerie is set to 0. Change ThisColor to clBlack.  
2) Set ThisSerie to 1. Change ThisColor to clYellow.}*

Title[ CHART\_TOPTIT ] := 'Articles vs Titles';  
Title[ CHART\_LEFTTIT ] := 'CCM';  
Title[ CHART\_BOTMOTIT ] := 'Cards';  
*{Design: click on the TitleDlg property and set Top, Left and Bottom titles}*

**end;**

**end;**

**Procedure** TF\_Chart.Build2( Ch : TChartFX );  
*{This procedure sets properties that cannot (as far as I can determine) be set at design time}*

**const**

XAbbrevs : **array**[ 0..4 ] of **string**[ 4 ] =  
( 'Acc', 'Bar', 'Mas', 'Amex', 'Din' );  
SeriesTitles : **array**[ 0..1 ] of **string**[ 8 ] =  
( 'Articles', 'Titles' );  
XTitles : **array**[ 0..4 ] of **string**[ 20 ] =  
( 'Access', 'Barclaycard', 'Mastercard', 'American Express',  
'Diners' );

*{of course you would normally read xtitles and values from a database}*

Values : **array**[ 0..1, 0..4 ] of **double** =  
( ( 50, 60, 70, 80, 90 ),  
( 30, 35, 25, 37, 42 ) );

**var**

i, SerieNo : **integer**;

**begin**

**with** Ch **do begin**

LegendWidth := 120;

```

    {Set Number of series, number of values *****}
    OpenData[ COD_INIVALUES ] := MAKELONG( 2, 5 );
    CloseData[ COD_INIVALUES ] := 0;
    {*****}

    OpenData[ COD_VALUES ] := 2;
    {if you omit the above statement, (in which you enter the
    number of SERIES not VALUES), and the CloseData below,
    the assignment to Values does not create an error, but
    does not work!
    Assigning Values to Legend, KeyLeg works without an
    OpenData/CloseData}
    ThisSerie := 0;
    for i := 0 to 1 do
        SerLeg[ i ] := SeriesTitles[ i ];
    for i := 0 to 4 do
        begin
            Legend[ i ] := XTitles[ i ];
            KeyLeg[ i ] := XAbbrevs[ i ];
        end;
    SerieNo := 0;
    for SerieNo := 0 to 1 do
        begin
            ThisSerie := SerieNo;
            for i := 0 to 4 do
                Value[ i ] := Values[ SerieNo, i ];
            end;
            CloseData[ COD_VALUES ] := 0;
        end;
    end;

    procedure TF_Chart.FormResize(Sender: TObject);
    var
        w, h : longint;
    begin
        w := NB.Width;
        H := NB.Height;
        {enlarge/reduce chart size if necessary}
        Chart1.Width := W - 18;
        Chart1.Height := H - 12;
        Chart2.Width := W - 18;
        Chart2.Height := H - 12;

        {move Exitbutton close to right edge}
        ExitBtn.Left := SpeedPanel.Width - 32;
    end;

    end.

```

[Return to ChartFX Article](#)

[Return to Front Page](#)



