



Cover Art By: Doug Smith

# Face Value

*Creating an Attractive and Useful Interface*

## ON THE COVER



**7 Face Value** — Robert Vivrette  
You've heard the adage, "First impressions are everything." This is especially true when it comes to the UI of your Windows application. This month, Mr Vivrette presents the basic elements of good UI design for your Delphi programs. You'll see that controls placement, form margins, use of hints, and user-configurable options are all essentials for quick user acceptance.



**12 3-D Labels with a Twist** — Keith Wood  
Flash back: the June '95 *DI* featured the 3-D Label component that spruces up your applications. Flash forward: in this issue, Mr Wood takes that component several steps further by introducing enhanced features such as depth, shadow, and rotation capabilities. But this Label component isn't just another pretty face — it demonstrates important Delphi OOP design principles.

## FEATURES



**18 Informant Spotlight** — Michael Maloof  
Long the *bête noire* of Windows 3.x, the GDI resource beast haunts Windows 95 as well. In fact, it's even more elusive — until now. Mr Maloof provides us with a Delphi utility that monitors GDI and can be incorporated into your Delphi applications.



**24 Sights & Sounds** — Charlie Howell  
Using the *TAnimated VCL*, Delphi programmers can add animation sequences to their programs. They can even be synchronized with sound. Mr Howell shows off this extensible component and explains how your application doesn't have to be "all work and no play."



**28 Visual Programming** — Jim Callan  
Rich application functionality can sometimes lead to periodic processing delays, and developers need to let users know what's going on. To help you let them know, Mr Callan examines using progress indicators and changing the mouse pointer to manage user expectations.



**33 DBNavigator** — Cary Jensen, Ph.D.  
Delphi 2 is an even more robust environment for creating effective data-access applications. Dr Jensen demonstrates how to display multiple records with the enhanced DBGrid and DBCtrlGrid components. He also discusses the new Columns Editor and properties that give users the best views of data.



**38 The API Calls** — Karl Thompson  
How will the application you're getting ready to deliver work in a low-memory situation? How about low disk space? Low GDI? There's a utility that can answer these questions and you already own it. This month, Mr Thompson shows how stress can be good for you — and your applications.



**43 At Your Fingertips** — David Rippey  
Delphi 1 and Delphi 2 tips just for your perusal! This month, Mr Rippey explains how to: automatically set focus to a subsequent edit field in a data entry screen; have your program check for another running program; and ensure that your program points to the application's directory.

## PRODUCT REVIEW



**40 ImageLib Portfolio 3.1** — Douglas Horn  
This month, Mr Horn reviews ImageLib Portfolio 3.1 by SkyLine Tools, Inc. He finds the product well worth its cost, and notes it's the "little things" that make it a must-have third-party tool.

## DEPARTMENTS

- 2 **Delphi Tools**
- 5 **Newsline**
- 45 **File | New** by Richard Wagner

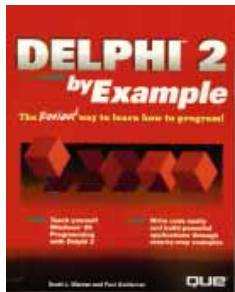


New Products and Solutions



## New Delphi Book

**Delphi 2 by Example**  
Scott Warner & Paul Goldsman  
QUE



ISBN: 0-7897-0592-3  
Price: US\$29.99 (472 pages)  
Phone: (800) 848-8199

## Systems Advisory Group Releases RingZero GDK

Systems Advisory Group Enterprises, Inc. of Amarillo, TX has released its *RingZero GDK* component suite, allowing Delphi 2 developers to access the Microsoft game SDK. The suite includes components for DirectDraw, DirectSound, DirectPlay, and DirectInput.

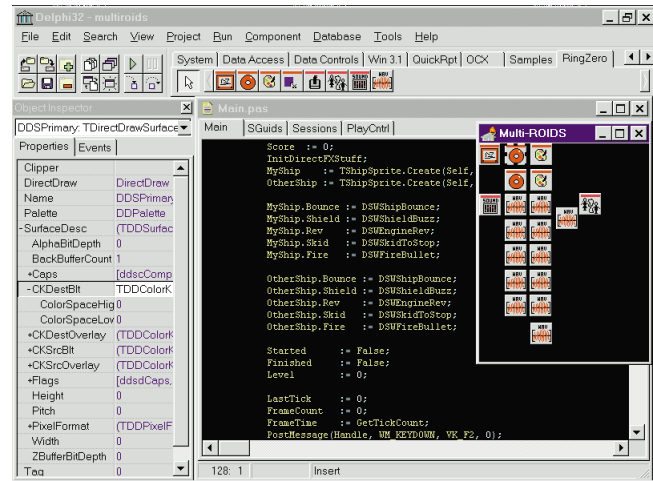
RingZero provides access to Microsoft's Common Object Model programming interface. It also includes component properties to control the variables associated with setting video modes, palettes, surfaces, sound, communications, etc., plus tools and example code.

RingZero features direct access to video memory, backbuffers, blit operations, and built-in page flipping. It also allows developers to write their own PutPixel and DrawLine specialty func-

## Data Dynamics Ships DynamiCube OCX for Windows 95 and Windows NT

Data Dynamics, Ltd. of Columbus, OH has released *DynamiCube*, a 32-bit OLE custom control (OCX) for Windows 95 and Windows NT. DynamiCube OCX provides data analysis and dynamic OLAP views within any application developed with tools supporting 32-bit OLE controls. It works with all data sources supported by ODBC, JET, and RDO.

REGION	COUNTRY	Jan	Feb	Mar	Total	±Qtr 2	±Qtr 3	±Qtr 4
Africa		290886.00	294533.00	310462.00	886281.00	878379.00	891284.00	890958.00
Europe	France	44304.00	52160.00	57335.00	153799.00	145453.00	140475.00	139646.00
	Germany	50056.00	57518.00	52934.00	160508.00	156300.00	146347.00	145184.00
	Italy	58663.00	50036.00	56154.00	165129.00	157395.00	151688.00	151280.00
	Spain	40266.00	51123.00	59620.00	150921.00	147521.00	138071.00	147440.00
	UK	54034.00	41578.00	48790.00	144702.00	156801.00	140312.00	143687.00
	Total	255329.00	253085.00	274705.00	783119.00	763470.00	716839.00	727137.00
North America	Canada	41491.00	50076.00	54953.00	146520.00	150015.00	155448.00	148951.00
	USA	48147.00	45573.00	52250.00	145970.00	151056.00	139218.00	138742.00
	Total	89638.00	95649.00	107203.00	292490.00	301071.00	294667.00	288933.00
South America		141587.00	144179.00	132588.00	418354.00	449123.00	448412.00	461575.00
Total		767440.00	787846.00	824958.00	2380244.00	2392043.00	2352202.00	2368263.00



tions. Sprite example classes are included with the demonstration files.

Sound can be added using play, stop, and position calls. For advanced sound programmers, mixing unique buffers into a primary buffer is supported.

RingZero includes Send/Receive methods for communicating during multi-player games. Players and groups can be identified and controlled. Methods for creating, joining,

DynamiCube doesn't require a server-based OLAP engine. It works directly with existing relational data sources to provide dimensional views of that data. DynamiCube allows developers to create virtual multidimensional dynamic views of tables for users, without writing code.

DynamiCube features unlimited n-dimensional views; drag-and-drop data pivoting at run time; a design time layout editor with drag-and-drop areas and live data; multiple aggregate and statistical functions, and summary levels.

It also offers drill-down and roll-up at any summary level, data filters of any dimension's data items that provide focused views of the data without requering the source database, and custom

and maintaining sessions are also included.

**Price:** RingZero Suite, US\$99 (all four components); RingZero Suite with source, US\$199. No royalty fees are required.

**Contact:** Systems Advisory Group Enterprises, Inc., 2201 Civic Circle, Ste. 1001, Amarillo, TX 79109-1853

**Phone:** (800) 580-0025 or (806) 354-8185

**Fax:** (806) 354-8366

**E-Mail:** Internet: RingZero@sage-inc.com

**Web Site:** <http://www.sage-inc.com>

methods to retrieve, graph, or print summarized data within an application.

DynamiCube supports multiple data items, and time series fields (year, quarter, month, week) can be automatically calculated from the source database date field.

DynamiCube demonstration software can be downloaded from the CompuServe MSBASIC and VBPI-FORUM forums, as well as Data Dynamics' Web site.

**Price:** US\$499, including royalty-free distribution with Delphi 2 or Visual Basic 4.0 applications.

**Contact:** Data Dynamics, Ltd., 2600 Tiller Lane, Columbus, OH 43231

**Phone:** (614) 895-3142

**Fax:** (614) 899-2943

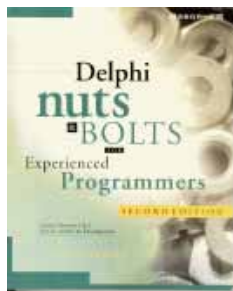
**E-Mail:** CIS: 72672,550

**Web Site:** <http://www.data-dynamics.com>



## New Delphi Book

**Delphi Nuts & Bolts For Experienced Programmers Second Edition**  
 Gary Cornel  
 Osborne/McGraw-Hill



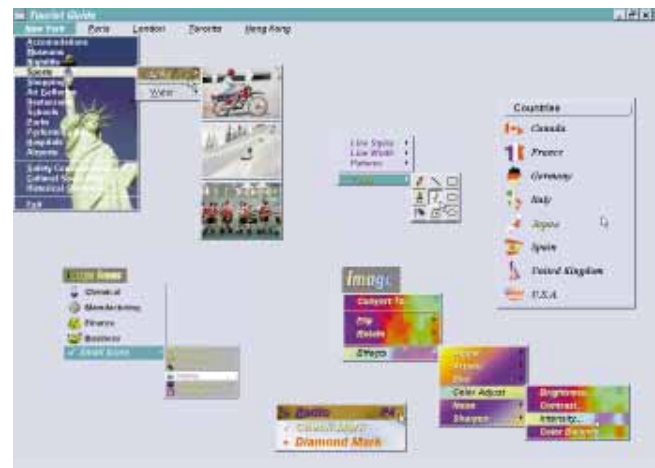
ISBN: 0-07-882203-3  
 Price: US\$24.95 (376 pages)  
 Phone: (800) 722-4726

## DFL Software Releases Light Lib Magic Menus

DFL Software Inc. of Toronto, Ontario, Canada is now shipping *Light Lib Magic Menus*, a VCL that offers Delphi developers an alternative to the standard Windows menu system.

It enables developers to add background images, textures, bitmap menu items, tool button palettes, and more to their applications.

Light Lib Magic Menus has classes which parallel the Delphi menu classes. There's a MagicMenu class and a MagicItem class. For example, to have a bitmap behind a menu, drop a MagicMenu control on a form, assign it to an existing menu, and then specify the bitmap it should use. This architecture allows developers to switch between conventional and Magic Menus at run time.



Light Lib Magic Menus allows developers to create menus featuring attributes such as bitmap or textured backgrounds, 3D text styles, or combined icons and text (such as the Windows 95 "Start" menu). They also support menu structures where entire menus are composed of bitmaps, and users select options by

selecting a bitmap rather than a text option.

**Price:** US\$99

**Contact:** DFL Software Inc., 55 Eglinton Ave. E, Suite 208, Toronto, ON, Canada M4P 1G8

**Phone:** (416) 487-2660

**Fax:** (416) 487-3656

**BBS:** (416) 487-4041

**CIS Forum:** GO DFLSW

**Web Site:** <http://www.dfl.com>

## High Gear, Inc. Announces High Gear 1.0 VCL Components

High Gear, Inc. of Brookfield, WI is shipping *High Gear 1.0*, a set of 19 native VCL components for Delphi and Delphi 2.

High Gear includes a suite of 10 controls with 3D text effects. Text can appear raised, lowered, or normal. The 3D controls include the Bitmap Button, Speed Button, Label, Data

Aware Label, Panel, Data Aware Panel, Group Box, Data Aware Group Box, Radio Group, and Data Aware Radio Group. All of the data-aware 3D controls have their caption property linked to a field in the data source, thereby displaying a value stored in the database.

High Gear also provides a progress bar that can be oriented horizontally or vertically, and has several options for appearance including pie, multi-color intensity meter, and the standard Windows 95 progress bar.

The High Gear Track Bar enables end-users to graphically specify an integral value. It can be oriented horizontally or vertically, and has the appearance of the standard Windows 95 track bar.

High Gear includes two components for customizing a Delphi form's background. The Form Gradient is a non-visual component that displays a gradient pattern on the background of a form.

The Form Tile is a non-visual component that can be used to tile a graphic file across the back of a form.

High Gear ships with online help, full source code, and is compatible with both Delphi and Delphi 2.

**Price:** US\$149, royalty-free. Includes free technical support via phone, fax, and e-mail.

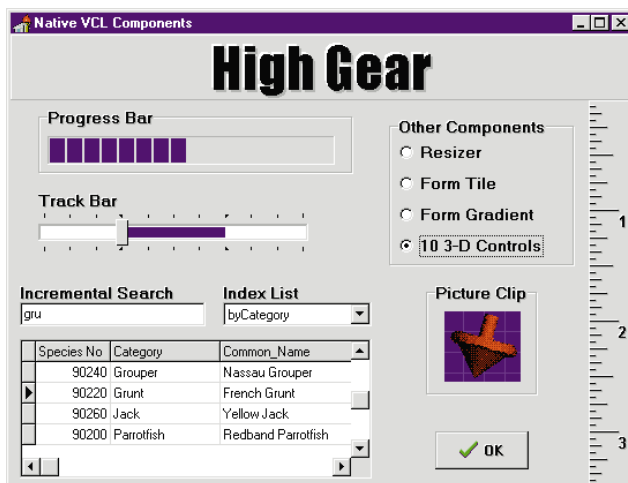
**Contact:** High Gear, Inc., 17125 C West Bluemound Rd., Ste. 114, Brookfield, WI 53008-0949

**Phone/Fax:** (414) 524-1045

**Order Number:** (800) 463-3574

**E-Mail:** Internet: [highgear@highgear.com](mailto:highgear@highgear.com)

**Web Site:** <http://www.high-gear.com/>



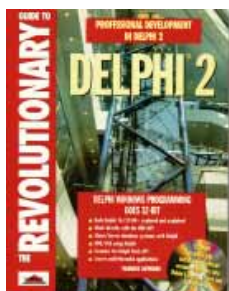


New Products  
and Solutions



## New Delphi Book

The Revolutionary Guide  
to Delphi 2  
Paul Hinks, et al.  
WROX Press



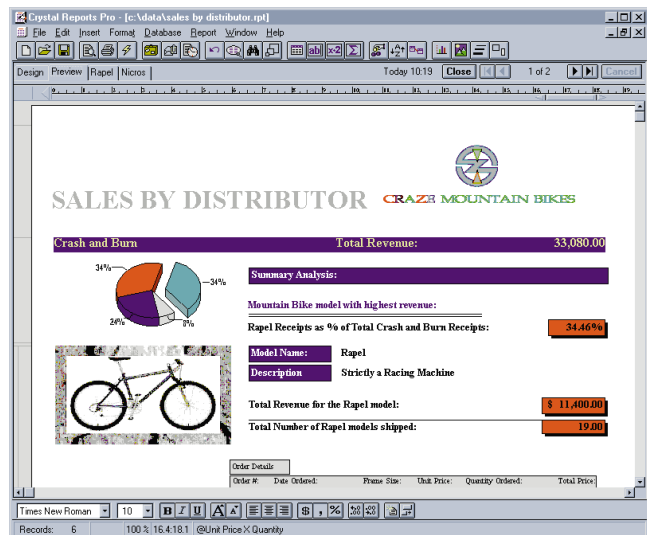
ISBN: 1-874416-67-2  
Price: US\$49.95  
(711 pages, CD-ROM)  
Phone: (800) USE-WROX

## Crystal Announces 16- and 32-Bit Delphi VCL for Crystal Reports

Crystal, of Vancouver, BC, Canada, a subsidiary of Seagate Technology, Inc., has announced a new 16- and 32-bit *Delphi Visual Component Library (VCL) for Crystal Reports Professional 4.5*. The new VCL provides developers with an interface that enables visual control over the integration of Crystal Reports technology into Delphi applications.

There are more than 80 properties in the new VCL, including existing VBX properties that allow developers to convert existing VBX applications to the new VCL. For example, an MDIChild property is included, which simplifies the creation of an MDI-Child preview window and verifies the form holding the VCL is the parent.

The new Crystal Reports VCL design doesn't require



Delphi developers to set each data table in the report individually, like the current Crystal Reports VBX. The DatafileLocation property of the new VCL enables developers to set all the data tables in one step, provided all the tables are located in the same directory.

**Price:** The updated VCL for Delphi is

available as a free download from the Crystal Web Site.

**Contact:** Crystal Inc., 1095 West Pender Street, Fourth Floor, Vancouver, BC, Canada V6E 2M6

**Phone:** (800) 877-2340 or (604) 681-3435

**Fax:** (604) 681-2934

**E-Mail:** Internet: sales@crystalinc.com

**Web Site:** http://www.seagate.com/software/crystal/

**CIS Forum:** GO REPORTS

## Nevrona Designs Releases ReportPrinter Pro 2.0 for Delphi

Nevrona Designs of Mesa, AZ is now shipping *ReportPrinter Pro 2.0*, a suite of native Delphi components for creating reports that are compiled into an application without external files.

Version 2.0 features a source code generating

report expert, a group of components for creating table style listings, a ReportSystem component with default setup, and status and preview screens. It has an integrated Help file, memo printing with word wrapping (even across multiple pages), form letter functions, as well as rotated and justified text.

ReportPrinter also includes snaking columns, full graphics (bitmaps, rectangles, ellipses, etc.), page positioning for pre-printed forms, measurements in inches or metric, scaling to any percent of the original size, print to file, printer control (paper size, orientation, etc.), and direct printer output for electronic forms.

ReportPrinter offers print

preview with zooming, panning, print-after-preview, and multiple page display. It is compatible with any database that can be accessed from within Delphi, or can be used without a database. Using ReportPrinter's components and class library, reports are created in Delphi with minimal coding.

**Price:** US\$149 (includes component source, printed documentation, and a 30-day, money-back guarantee); US\$49 upgrade for existing ReportPrinter 1.1 users.

**Contact:** Nevrona Designs, 1930 S. Alma School, Ste. C204, Mesa, AZ 85210-3043

**Phone:** (602) 491-5492

**Fax:** (602) 530-4823

**E-Mail:** Internet: info@nevrona.com;

CIS: 70711,2020

**Web Site:** http://www.nevrona.com/designs

Common Name	Image	Notes
Clown Triggerfish		Also known as the two spotted triggerfish. Inhabits outer reef areas and feeds upon invertebrates and smaller fish. Unlike other triggerfish, they are voracious eaters, and do not report seeing the clown triggerfish devour bits of coral systems.  Not edible.  Range is Indo-Pacific and East Africa to Somalia
Reef Haplochromis		Called pompano in Australia. Inhabits the more shallow lagoons and sandy bottoms.  The reef surgeon is a valuable food fish and considered a great sport fish that fights with any other hooked. The flesh of an old fish is just as tender to eat as that of the very young.
Great White Wrasse		Range is from the Indo-Pacific to East Africa.  This is the largest of all the wrasses. It is found in dense reef areas, feeding on a wide variety of invertebrates, fishes, sea urchins, crustaceans, and other invertebrates. In spite of its enormous size, does not feed at a very fast rate.  Edibility is considered poor.
Blue Tang		Range is the Indo-Pacific and the East Sea.  Habitat is around boulders, caves, coral ledges and crevices in shallow waters. Somewhat shy or nervous.  Its color changes dramatically from juvenile to adult. The mature adult fish can startle divers by producing a powerful drumming or thumping sound intended to warn off predators.





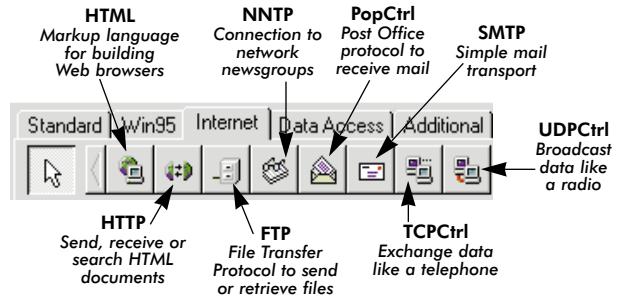
### Delphi Success

According to *Windows Watcher*, a newsletter that covers Microsoft technologies, Delphi Developer 2 is ranked first and Delphi Desktop 2 is fifth on the Windows bestseller list, as reported by Ingram Micro, a top distributor. Delphi Developer 2 outsold Microsoft Office, as well as products from WordPerfect, Intuit, and Corel. In addition, Borland has sold over 350,000 units of Delphi in the last 12 months.

## Borland Releases Delphi 2 Internet Solutions Pack

Scotts Valley, CA —

Borland has introduced the Delphi 2 Internet Solutions Pack, enabling Delphi developers to build Web-enabled applications. The Delphi 2 Internet Solutions Pack features eight ActiveX controls. These include an HTML component for building Web browsers, an NNTP component for connecting to network newsgroups, a PopCtrl component with Post Office protocol to receive e-mail, an SMTP component for simple mail transport, an



HTTP component for sending, receiving, or searching HTML documents, an FTP component for sending and receiving files, a TCPCtrl for exchanging data, and a

UDPCtrl for broadcasting data.

Because Delphi is a native code compiler, developers can write DLLs that communicate with an Internet server via ISAPI, CGI, or NSAPI. With the addition of the Internet Solutions Pack, developers can extend their client/server applications to the Internet or an intranet.

Delphi's Borland Database Engine can manage the security and process query requests from any HTML client. The query result sets are dynamically-generated virtual Web pages, and appear to end-users as a part of the Web-enabled application. Because the Borland

## Crystal Announces Crystal Reports 5.0

Vancouver, BC, Canada —

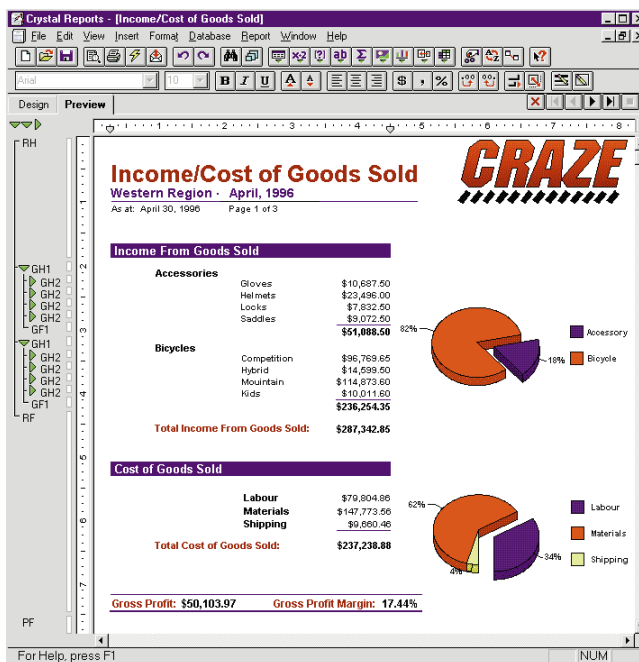
Crystal, a wholly-owned subsidiary of Seagate Technology, Inc. announced it will release Crystal Reports 5.0, Professional and Standard editions, by the third quarter of 1996.

Available for both 16- and 32-bit platforms, Crystal Reports 5.0 will feature two key components: Crystal Report Designer and Crystal Report Engine. Version 5.0 will also add broader database

support and the ability to distribute reports via communications infrastructures, such as Lotus Notes, Microsoft Exchange, and the Internet.

In addition, Crystal is planning to release a Crystal Reports 5.0 New Features Interactive Learning CD in the third quarter of 1996.

For more information, visit Crystal's Web site at <http://www.seagate.com/software/crystal>, or call (800) 877-2340 or (604) 681-3435.



July 1996



## Borland Reports Revenue in Fourth Quarter

Borland International Inc. has announced its fourth consecutive quarter of profitability since they restructured operations in January 1995. In its fourth quarter and end of fiscal year 1996, which ended March 31, 1996, Borland recorded revenues of US\$62.9 million. Net income for the fourth quarter of fiscal year 1996 was US\$8 million. Revenues for fiscal year 1996 were US\$215.2 million, and net income was US\$14.3 million.

## Borland Acquires Open Environment Corporation

Scotts Valley, CA — Borland International Inc. announced it will acquire Open Environment Corp. Based in Boston, MA, Open Environment provides scalable, multi-tier client/server products for a variety of corporations.

Under the terms of the agreement, Open Environment shareholders will receive .51 shares of Borland common stock for each share of Open Environment common stock — providing the shares issued for each Open Environment share have market value of no more than US\$25 and no less than US\$12.75. Open Environment has approxi-

mately eight million shares and vested options outstanding, giving an indicated value of approximately US\$64 million for the transaction. Borland has approximately US\$31 million shares outstanding.

According to Gary Wetsel, president and CEO of Borland, this acquisition is part of Borland's plan for increasing their presence in the client/server and Internet markets. Wetsel also stated the acquisition of Open Environment more than doubles Borland's world-wide client/server sales force and adds client/server and intranet support, consulting, and education capabilities.

Borland plans to use Open Environment's North American, European, and Asian direct sales organizations as well as their partnership program, which includes over 75 strategic client/server VARs and systems integrators.

With the acquisition of Open Environment, Borland will combine its rapid application development tools with Open Environment's scalable, multi-tier solutions.

The transaction is expected to be completed later this summer. For more information visit Open Environment's Web site at <http://www.openenv.com> or Borland Online at <http://www.borland.com>.

## DSW Plans Delphi 2 Client/Server and Java Tours

Atlanta, GA — The DSW Group Limited has announced it will offer the Borland Delphi 2 Client/Server Tour, with over 40 stops in the US. The tour includes a Java, and two Delphi seminars.

The first 2-day seminar, Delphi 2 Fundamentals, cov-

ers the Delphi architecture, creating forms, working with databases, Pascal, component creation, and more.

For advanced Delphi developers, DSW is offering Advanced Delphi 2. This 2-day seminar focuses on visual form inheritance, MAPI applications, the Object

Repository and Database Explorer, advanced SQL, advanced component creation, OCXes, and more.

Delphi seminar attendees will receive a free copy of Delphi 2 and the latest version of Delphi 1.

In addition, DSW is offering a Java seminar. This 1-day seminar will teach the fundamentals of Java, and will include instruction on Borland's new Java development tools. Seminar attendees will receive Borland's Java Debugger and the Java Development Kit.

Tour registration packages range from a five-day packet for US\$1,355; four-day packet (includes both Delphi sessions), US\$1,160; three-day package (includes either Delphi 2 Fundamentals or Advanced Delphi 2, and the Java session), US\$890; two-day package (includes either Fundamentals or Advanced), US\$695; or the Java seminar alone is US\$295.

For details call (800) OK-DELPHI [(800) 653-3574].

## Netscape Licenses Java Compiler from Borland

Scotts Valley, CA — Netscape Communications Corp. has announced plans to include Borland's AppAccelerator compiler for Java applications in Netscape Navigator.

As a result, Netscape Navigator users can access Java applications available on Web pages and corporate intranets. The non-exclusive agreement between the two companies extends Netscape's Java support by enhancing Java application performance throughout Netscape's open software platform.

Borland's AppAccelerator reads the intermediate byte code produced by Java development tools and translates it

“on-the-fly” into machine-executable instructions on the local client system. Already shipping, Borland C++ Development Suite 5.0 includes the AppAccelerator compiler, a graphical program debugger, and an integrated version of Sun's Java Development Kit. Netscape plans to incorporate Borland's AppAccelerator technology into future versions of Netscape Navigator. Specific terms of the agreement were not announced at press time. More information about Netscape is available on their Web site at <http://home.netscape.com>, or by calling (415) 937-3777.





# ON THE COVER

Delphi / Object Pascal



By *Robert Vivrette*

## Face Value

### Creating an Attractive and Useful Interface

**F**irst impressions are important. What your application says to the user is going to reflect positively or negatively on your ability to develop a functional and professional product.

This article will address issues of how to develop a professional application. Some of the topics we'll cover may seem obvious — some may even seem trivial. However, the tips and techniques presented form at least a basic set of quality standards that programmers today should consider when developing applications.

To illustrate some of the principles that we'll cover in this article, I have written a handy utility called BMPVIEW (see Figure 1) that displays all the bitmap files (.BMP) in a directory. Although it might seem a mundane application, it illustrates a number of important elements of program design.

#### Be Considerate of the Environment

These days it can almost be said that no two

PCs are the same. As a result, your application should make as few assumptions about the user's machine as possible. However, it is appropriate to make some threshold assumptions, such as "the PC must be running Windows 3.1 or higher, and should be a 386DX/33 or better to run properly." The less you assume beyond that, the better.

For example, your program should not assume a certain screen size or color depth. As much as possible, different screen resolutions should be tested. Screen resolution could vary from 640x480 up to 1280x1024, or higher. If you design a program to look good at lower resolutions, it might wind up resembling something the size of a postage stamp at higher resolutions.

In addition, PCs can have different color depths. Generally, users are running their machines in 16 or 256 color configurations. Yet some users may be running thousands (or millions) or colors. Here are some questions to consider:

- What impact will these changes have on your application?
- Are you using 256-color images that will look hideous on a 16-color machine? If this is the case, should you include 16-color versions of the graphics and have your application adapt?
- If the program must have 256 colors or better, does it test for it, then provide the user with a helpful explanation of what must be done to use the program to its full capability?

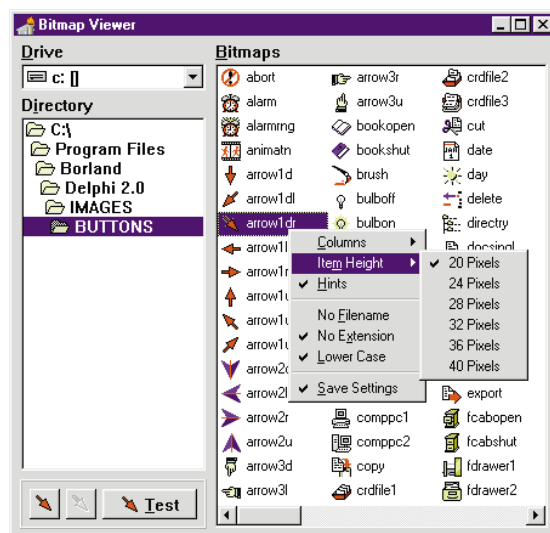


Figure 1: The BMPVIEW program.



## ON THE COVER

In addition, you should test to see what happens when you run other programs concurrently. Keep in mind that Windows must manage the palette of colors used by every application, and when you switch between programs, the visual effect could be unpleasant.

Something that may be more difficult to test would be the effect of running an application on a machine with a gray-scale monitor. Will your selected colors be too subtle to discern when viewed in shades of gray? If you have the luxury of knowing that all users of the application will have color displays, you can dismiss this issue.

Now, designing your program to *tolerate* certain resolutions or colors is one thing, but will it take *advantage* of them?

### UI Aerobics

Every time I evaluate a program, I put it through a brief regimen of calisthenics. I resize the application window with every edge, maximize it, minimize it, even see how large or small I can get it. What if the user shrinks the application window — will the edges obscure controls? Many novice users may be stumped by the appearance this might give. Alternatively, if you expand the program to fill the screen, what happens? Will all the controls retain their location and the rest of the form contain nothing but empty space? A well-designed application should adapt and take advantage of the extended screen space, moving and expanding the size of list boxes, edit boxes, memo fields, and the like.

The BMPVIEW program does just that. It can dynamically adjust its controls to handle any screen resolution on which it might be run. The key to this is to use the *TForm Resize* event (see [Figure 2](#)).

Whenever the main window of the program is resized, I make certain adjustments to the controls. The first is to move the two buttons in the bottom left of the screen. I move them to a point that is six pixels up from the bottom edge (*ClientHeight*) of the main form. Since *ClientHeight* is changing during the resize, the buttons will maintain their distance from the bottom edge.

```
procedure TBitmapViewerDialog.FormResize(Sender: TObject);
begin
  { Align the Samples panel to the bottom left }
  with SamplesPanel do
    SetBounds(Left,Self.ClientHeight-Height-6,
              Width,Height);
  { The DirList should take up all space
    remaining above }
  DirList.Height := SamplesPanel.Top-DirList.Top-6;
  { Perform Update to ensure all controls paint before
    the BmpList }
  Update;
  { BmpList should take up all space remaining on form }
  with BmpList do
    SetBounds(Left,Top,Self.ClientWidth-Left-6,
              Self.ClientHeight-Top-6);
end;
```

**Figure 2:** The *Resize* event of *TForm*.

Next, I adjust the directory list control to stop six pixels above the top edge of one of the buttons (which we just moved). We are not adjusting the width or placement of the directory list control. Thus, only its height will change.

Last, I use the remaining space for the bitmap viewer list box. I want to keep its upper left corner the same, but I want the width and height to be six pixels short of the bottom right corner. Delphi provides the ability to do most of this automatically for most controls by means of the *Align* property. However, I have found that there are many behaviors that *Align* cannot duplicate, so I often revert to using the *Resize* method.

If you're assuming the program will always be the same size, you should set the form's *BorderStyle* property to *bsSingle* to prevent resizing. Better yet, why not use the *bsDialog* style so that it will have a more attractive 3D look?

If you want to allow resizing — but only within certain limits — you should have the program respond to the *WM\_GETMINMAXINFO* message. Whenever a user resizes a form, Windows sends one of these messages. The response it receives determines the limits of how large or how small the form will be permitted to become. The nice part about this is that it is done *as the form is being resized*. By setting a minimum width and height in this way, Windows allows the user to reduce the form's size only until this limit is reached. Then the form's edge stops moving.

Again, the BMPVIEW program uses one of these message handlers. There is no point allowing the user to resize the application below a usable size. When the form shrinks past a certain point, the bitmap list box and/or the directory list box will disappear, making the application non-functional until it's enlarged. The following message handler limits the form's size to 300 pixels wide and 250 pixels high:

```
procedure TForm1.WMGetMinMaxInfo(
  var Message : TWMGetMinMaxInfo);
begin
  with Message.MinMaxInfo^ do begin
    ptMinTrackSize.X := 300;  { Minimum width }
    ptMinTrackSize.Y := 250;  { Minimum height }
  end;
  Message.Result := 0;
  inherited;
end;
```

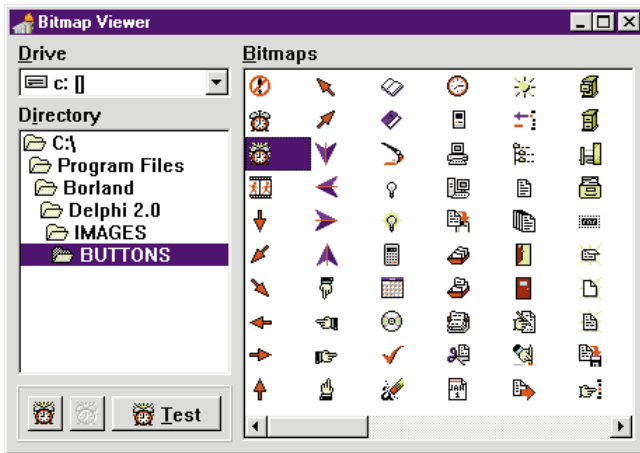
With the dynamic moving and sizing of the main form and its controls — as well as the low-end limit to the form's size — it can truly take advantage of any kind of screen real estate that we throw at it. The BMPVIEW program extends these capabilities by allowing the user to modify the number of columns displayed as well as the height of each of the items in the bitmap list box.

[Figure 3](#) shows how the program might look after a user resizes it.

### Look for Annoyances

Do you see any annoying things in the Delphi IDE? Not





**Figure 3:** Configuration settings allow the appearance and behavior of BMPVIEW to be changed. Here, you can see that six columns of bitmaps are shown, without file name extensions. Notice the abbreviated length of this figure when compared to Figure 1.

many I'll bet, and there's a good reason. Since Delphi was written with Delphi, its developers and programmers had to use elements of it (such as the IDE) every day. Since they had to use it constantly, they didn't tolerate little functional annoyances. The product is, of course, better as a result. They wouldn't put up with these problems, and they knew we wouldn't either.

When developing your application, keep this same attitude. If you see an element of the program that is annoying or unnecessarily difficult, fix it. Don't think "Well, I'll fix it in version 1.1." Just because you'll allow these annoyances does not mean that your users will. And they may not let you get to 1.1!

What kind of annoyances am I talking about? Let's address a few that come to mind.

**Save Configuration Information.** This is one of my biggest gripes. Whenever your program ends, it should write certain information to an .INI file so it can restore these defaults the next time it runs. This would include things such as form size and placement; default directories for common dialog boxes (e.g. Open and Save dialog boxes); overridden filters in common dialog boxes (e.g. the wild-card specifications for an Open dialog box), etc.

These settings should not be written to the WIN.INI file (unless there is a specific reason to do so). Rather, they should be written to a private .INI file or, better yet, to the System Registry in Windows 95. For this simple application, I'll be writing these settings to a private .INI file called BMPVIEW.INI that Delphi will automatically place in the \WINDOWS directory.

The BMPVIEW program makes full use of the *TIniFile* object to read and write configuration information. It remembers the last directory it was in, the window size and placement, as well as the various configuration settings within the program (such as the number of columns to display, or the height of each item in the bitmap list box). The procedures shown in Figure 4 control the loading and saving of this information.

The *ReadSettings* method is called when the form is created. This loads the information from the .INI file and sets the appropriate values of the various application properties. The *WriteSettings* method does just the reverse; it writes the current state of these values so the next time the program is executed, it will start with the same settings.

**A Logical Tab Order for Controls.** Make sure the tab order has been set and that it's appropriate. A user should be able to anticipate where the focus will move if [Tab] is pressed. Test this with a user who is unfamiliar with the application.

Sometimes it may be appropriate for you to move the focus for the user. For example, if you know that after clicking on a particular button the user will need to fill in an edit field, why not shift the focus there immediately so the user can begin typing?

**Support for the Keyboard.** This seems to be a commonly overlooked topic. Before you ship your program, ask yourself, "Without a mouse, could the user effectively navigate the application?" The answer should always be "Yes." It's a common thing to overlook, but many developers assume that the mouse will always be available. Not a good assumption! What if your user is running the program on a laptop and forgot the mouse? Would the application be unusable or difficult to use?

I test this by placing the mouse out of reach and then putting the program through its paces. Command buttons and menus should have hot-key shortcuts and the tab sequence should cycle through all appropriate controls. Make a note of all program functions that cannot be accessed and add keyboard support for them.

In the BMPVIEW program, I included a pop-up menu over the **Bitmaps** list box to control the number of columns to display, the height of each item, and the method of loading and displaying bitmaps (see Figure 5). It was necessary to include a way of activating this pop-up menu using the keyboard. This was done by turning on the main form's *KeyPreview* property, and then modifying its *OnKeyDown* method as follows:

```
procedure TForm1.FormKeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  if (Shift = [ssCtrl]) and (Key = VK_1) then
    PopUpMenu1.PopUp(Left+ BmpList.Left+
      (BmpList.Width div 2),
      Top+BmpList.Top+Height-ClientHeight+
      (BmpList.Height div 2));
end;
```

This procedure instructs the form to activate the pop-up menu in the middle of the **Bitmaps** list area when the user presses [Ctrl] [1].

**Overuse of Hints.** Delphi's Help Hint feature is wonderful, and it takes little or no programming to implement a fully functional hint system in your program. However, just because it's easy to include does not mean that you should add this fea-

```

procedure TBitmapViewerDialog.ReadSettings;
begin
  { Read all configuration and window settings from the INI file }
  SettingsIni := TIniFile.Create('BMPVIEW.INI');
  { Configuration Settings }
  DirList.Directory :=
    SettingsIni.ReadString('Options','Directory','C:\');
  BmpList.Columns :=
    SettingsIni.ReadInteger('Options','Columns',2);
  BmpList.ItemHeight :=
    SettingsIni.ReadInteger('Options','ItemHeight',24);
  pmHints.Checked :=
    SettingsIni.ReadBool('Options','Hints',True);
  pmSaveSettings.Checked :=
    SettingsIni.ReadBool('Options','SaveSettings',True);
  pmLowerCase.Checked :=
    SettingsIni.ReadBool('Options','LowerCase',False);
  pmNoFilename.Checked :=
    SettingsIni.ReadBool('Options','NoFilename',False);
  pmNoExtension.Checked :=
    SettingsIni.ReadBool('Options','NoExtension',True);
  { Window position settings }
  FormL := SettingsIni.ReadInteger('Window','Left', -99);
  FormT := SettingsIni.ReadInteger('Window','Top', -99);
  FormW := SettingsIni.ReadInteger('Window','Width', -99);
  FormH := SettingsIni.ReadInteger('Window','Height',-99);
  FormState := SettingsIni.ReadInteger('Window','State',0);
  { Perform SetBounds only if all parameters have
    been specified }
  if (FormL >= 0) and (FormT >= 0) and
    (FormW > 0) and (FormH > 0) then
    SetBounds(FormL,FormT,FormW,FormH);
  { Set the previous window state }
  case FormState of
    0 : WindowState := wsNormal;
    1 : WindowState := wsMaximized;
    2 : WindowState := wsMinimized;
  end;
  { Set initial menu check states }
  SetColumnCheck(BmpList.Columns);
  SetPixelCheck(BmpList.ItemHeight);
  SetHintsCheck;

```

```

  { Free the TIniFile object }
  SettingsIni.Free;
end;

procedure TBitmapViewerDialog.WriteSettings;
begin
  { Write out configuration & Window settings to INI file }
  SettingsIni := TIniFile.Create('BMPVIEW.INI');
  SettingsIni.WriteBool('Options','SaveSettings',
    pmSaveSettings.Checked);
  { Only save remaining items if SaveSettings is checked }
  if pmSaveSettings.Checked then
    begin
      SettingsIni.WriteString(
        'Options','Directory',DirList.Directory);
      SettingsIni.WriteInteger(
        'Options','Columns',BmpList.Columns);
      SettingsIni.WriteInteger(
        'Options','ItemHeight',BmpList.ItemHeight);
      SettingsIni.WriteBool(
        'Options','Hints',pmHints.Checked);
      SettingsIni.WriteBool(
        'Options','LowerCase',pmLowerCase.Checked);
      SettingsIni.WriteBool(
        'Options','NoFilename',pmNoFilename.Checked);
      SettingsIni.WriteBool(
        'Options','NoExtension',pmNoExtension.Checked);
      SettingsIni.WriteInteger('Window','Left', Left);
      SettingsIni.WriteInteger('Window','Top', Top);
      SettingsIni.WriteInteger('Window','Width', Width);
      SettingsIni.WriteInteger('Window','Height',Height);
      case WindowState of
        wsNormal :
          SettingsIni.WriteInteger('Window','State',0);
        wsMaximized :
          SettingsIni.WriteInteger('Window','State',1);
        wsMinimized :
          SettingsIni.WriteInteger('Window','State',2);
      end;
    end;
  SettingsIni.Free;
end;

```

**Figure 4:** The `ReadSettings` and `WriteSettings` procedures are responsible for reading and writing information from and to the application's .INI file.

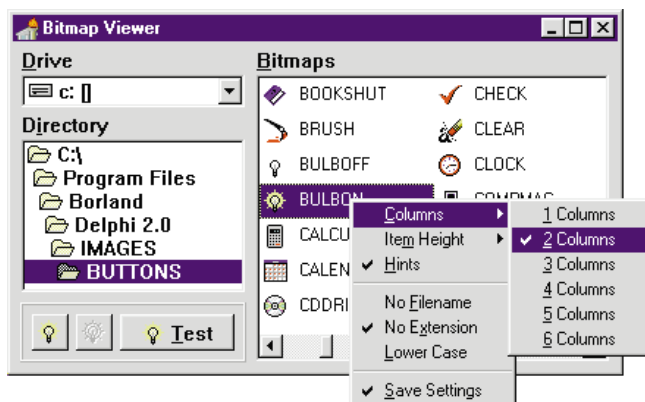
ture. I've seen programs where every on-screen control has an associated hint. This gets annoying after about 10 minutes.

The recommendation, therefore, is to only include hints on things where it's necessary. If there are a lot of hints, you might want to have a configuration option to allow the user to enable or disable them. (And, in light of our first annoyance mentioned above, you're going to save the hint status to an .INI file, aren't you?)

## Do You Feel Like You Look?

It's an overly-used phrase, but there is much to be said for a program's "look and feel." As stated at the beginning of this article, a user's first impression of an application is important. With that in mind, let's look at a few things that affect a program's "look and feel."

**Use of Color.** Windows applications inherit much of their color settings from the operating system itself. When selecting colors for forms, buttons, etc., be careful when using hard-coded color references such as `clGreen` or `clRed`. Instead, use appropriate constants such as `clButtonFace`, or `clWindow`.



**Figure 5:** Another view of the BMPVIEW program. The user configured the `Bitmaps` listbox to enable hints, display two columns of bitmaps, omit the file name extension, and save settings on exit. For ease of use and versatility, these settings can also be configured via the keyboard.

This way, your application will fit in with the color settings that the user has defined from within Windows, and can still look good after the user selects a different setting.

## ON THE COVER

**Use of Fonts.** You just purchased that cool set of 800 typefaces and are all set to jazz up your applications. Yes, these new fonts will work on your applications as long they continue to run on your machine. However, once the program is moved to another computer, these fonts will probably no longer be available.

In this situation, Windows will substitute standard fonts for your fancy ones, which will probably mess up all your careful positioning of labels and controls on the form. To avoid this, stick with fonts that you know will be on all users' machines, such as MS Sans Serif, System, Arial, Courier, Times New Roman, or Small Fonts (to name a few). This way, the fonts your program uses will be the same on any machine.

**Placement of Controls.** Make sure the program's layout is neat and organized. Consider all these questions:

- 1) Are all controls placed in a logical arrangement? Are commonly used sets of controls located near each other to minimize mouse travel? Controls that are supposed to be in a row should be exactly on the same pixel row.
- 2) Does the form have an even margin around the edges? Controls that are jammed against the side of a form show that you didn't care enough to place them correctly. There should also be an even distribution of space between controls.
- 3) Do buttons have glyphs attached to them? Plain buttons with just text are dull and flat looking. Spruce up the buttons by using the BitButton controls and assigning appropriate glyphs to them.

By paying careful attention to the details that increase your application's usability, and by allowing the user to customize settings in the program, you'll find that user acceptance will come more easily.

### The Sample Application

Figure 1 shows the BMPVIEW program in one configuration. By changing a few of the configuration settings (e.g.

disabling file names, and increasing the number of columns displayed), the application can now look more like Figure 3.

When I created this program, I thought it would make an excellent replacement for the bitmap property editor in the Delphi IDE. Once installed in the IDE, this replacement allows the developer to click on *Glyph* properties for SpeedButtons or BitButtons and use this property editor to select an appropriate bitmap.

The change-over from bitmap viewer to property editor is actually quite simple (see end of article for download information). This project is featured in Chapter 5 of a book I co-authored, entitled *Delphi In Depth* [Osborne/McGraw-Hill, 1996]. If you would like to learn more about how BMPVIEW the bitmap viewer became BMPPROP the property editor, it's covered in detail in the book.

### Conclusion

The way your program is put together reflects the kind of programmer you are. Hopefully the material we have covered here will serve to show some of the concepts and practices that go into the construction of a professional, intuitive, and attractive Windows application.

It's often the little things you do to a program that will have the greatest impact. The best part is that you aren't alone. With the help of the Delphi programming environment, it's surprisingly easy to achieve all of this, and much more.  $\Delta$

*The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\JUL\DI9607RV.ZIP.*

Robert Vivrette is a contract programmer for Pacific Gas & Electric and Technical Editor for *Delphi Informant*. He has worked as a game designer and computer consultant, and has experience in a number of programming languages. He can be reached on CompuServe at 76416,1373.





## ON THE COVER

Delphi 1 / Object Pascal



By *Keith Wood*

# 3-D Labels with a Twist

## Extending a Custom Delphi VCL

In the June 1995 issue of *Delphi Informant*, Jim Allen and Steve Teixeira introduced “A 3-D Label Component.” In this article, we’ll build on the foundation laid in that article to make the component more flexible and customizable.

This article describes changes to the 3-D Label to enhance its appearance and provide for many different effects, including rotation. Along the way, we’ll discuss interaction between component properties, plan for ease-of-use, and run the entire component production cycle — from design to implementation. Next month, we’ll build property editors and add help.

Let’s get started.

### Wish List

The 3-D Label component described previously added three new properties to the standard Delphi *TLabel* component. These included *Offset*, a property that represented the “depth” of the effect; *LabelStyle*, which indicated if the lettering was raised or lowered; and *AsButton*, a flag to indicate whether the Label should act as a button when pressed.

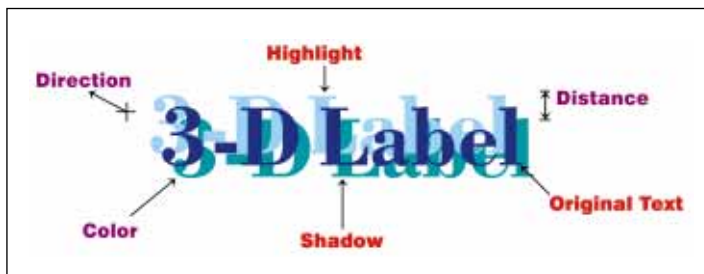


Figure 1: Anatomy of a 3-D Label component.

A 3-D Label has three parts: the highlight, shadow, and original text (see Figure 1).

The first two have three attributes we may want to alter:

- 1) the position or direction with respect to the main text
- 2) the distance from that text
- 3) the color

Ideally we would like to control each of these individually without affecting any other attribute. To make things easier, however, it would be nice to have preset combinations of these attributes that could be set with a single value. Thus we could have a style property — similar to the original 3-D Label — that sets distance and direction, and a color scheme property that changes the colors of all the parts, including the original text.

Besides all this, we’d like to rotate the text to any angle and perhaps still apply a 3-D effect to it. This can all be done without too much effort to produce a very flexible Label component for your use.

For reasons described in the earlier article by Allen and Teixeira, the component is based on the *TCustomLabel* component that comes with Delphi. This is because “it has all the functionality we want to either publish or override, and it doesn’t publish



anything of its own.” The name of the new component is *TLabelEffect*, since it does more than just 3-D Labels.

## Basic Properties

The new properties we’ve identified for this component are the direction and distance of both the highlight and shadow from the original text, and their colors. The directions (to make things a little easier without losing too much functionality) are restricted to the cardinal directions only, i.e. up, down, left, right, and their combinations.

Since a limited number of combinations exist and we want to make setting these properties as intuitive as possible, we are using an enumerated type. This allows us to restrict the range of values that may be applied to the properties, as well as provide something meaningful for the user to work with.

The **type** definition appears as:

```
type TEffectDirection =
  (edNone, edUp, edUpRight, edRight, edDownRight,
   edDown, edDownLeft, edLeft, edUpLeft);
```

The first value, *edNone*, prevents us from shifting the highlight or shadow away from the original text, thereby effectively hiding it. This allows us to produce effects with only one shadow, for example, or to rotate the text without a 3-D effect. The order of the remaining values follows around the circle clockwise from the 12 o’clock position. The names of the properties are *DirectionHighlight* and *DirectionShadow*.

Since we don’t want the shadows getting too far away from the original text, the distance properties, *DepthHighlight* and *DepthShadow*, are also restricted to a small range. If we specify these as subranges — rather than just integers — Delphi will automatically perform validity checks to ensure they remain in the prescribed range. The **type** definition for these then becomes:

```
type TEffectDepth = 0..10;
```

The colors of the highlight and shadow use the standard *TColor* type available in Delphi. The properties are called *ColourHighlight* and *ColourShadow*. The color of the original text is still controlled by the *Font.Color* property provided by the *TCustomLabel* component.

We’ll also publish several of the properties and events provided by the *TCustomLabel* component, including *Alignment*, *Color*, *Font*, *Transparent*, *Visible*, and *WordWrap*. The functionality for most of these is automatically provided by that class. Isn’t OOP wonderful?

## First Cut

These properties are enough to get us started. Each of them will directly affect the display of the text on screen while and when they are changed. This means that meth-

ods must be called to set each property so that we can force the image to be repainted. Reading the values of the properties has no side-effects and therefore can refer directly to the internal variables.

The declarations and writing methods for each of these properties are very similar, so only one will be shown here. For the *DirectionHighlight* property we need an internal variable — that is hidden outside the component — to hold the current value. By convention this variable is called *FDirectionHighlight* and its declaration appears in the **private** part of the component definition.

Then, since we want to be able to access this property through the Object Inspector, we have the following declaration in the **published** section:

```
property DirectionHighlight:
  TEffectDirection read FDirectionHighlight
  write SetDirectionHighlight
  default edUpLeft;
```

To enable us to set this value we must declare the writing method in the **private** section, again, hiding it from outside the component, and place the code for it in the **implementation** section of the unit:

```
procedure TLabelEffect.SetDirectionHighlight(
  EdDirection: TEffectDirection);
begin
  if FDirectionHighlight <> EdDirection then
    begin
      FDirectionHighlight := EdDirection;
      Invalidate;
    end;
end;
```

By checking that the value has actually changed before saving its new value, you’ll reduce the number of times the image must be redrawn. This redrawing is caused by the *Invalidate* method.

The default value for the property defined in its declaration does not actually set the value. Instead, the **default** directive indicates when the component should save the property value when writing to a stream.

If the property is set to the default value, then this value need not be saved. The setting of the actual initial value is done in the *Create* method. Also in *Create* we set the initial values of some other properties we’re using, including the font name and size.

For reasons that will become obvious later, you should use a TrueType font. Note also that the size is increased since this component is probably going to be used as a heading or to grab the user’s attention.

We also set the *AutoSize* property to *False*. Since this property is neither public nor published it cannot be altered by anything else:

## ON THE COVER

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

    { Initialize default values for internal fields }
    FDirectionHighlight := edUpLeft;

    ...

    AutoSize      := False;
    Height        := 33;
    Width         := 142;
    Transparent   := True;
    Font.Name     := 'Times New Roman';
    Font.Size     := 20;
end;
```

### Painting the Label

Of course, none of this will do anything until we alter the painting of the text to reflect the properties we've defined. For this we must override the *Paint* method of the *TCustomLabel* component. Instead of just painting the text, we must draw each component that comprises our 3-D Label. To ensure the effect appears as expected, this is done in the order of shadow, highlight, and original text.

Figure 2 shows the Object Pascal code for the *Paint* method. First, we must work out the minimum depth of a highlight or shadow from the original text so we can correctly position each relative to each other and have all of them inside the canvas. This value cannot be greater than zero since that is the offset for the original text. To make this easier, a *Min* function is defined to return the minimum of two integer values:

```
function Min(I, J: Integer): Integer;
begin
    if I < J then
        Min := I
    else
        Min := J;
end;
```

A constant array is declared in the unit to consider the effects of differing directions on the depth values. It has one entry for each of the direction values listed earlier, with each entry having a multiplier for the X and Y directions:

```
type TDirXY = (drX, drY);
const IOffsets:
    array [TEffectDirection, TDirXY] of -1..1 =
    (( 0, 0), ( 0, -1), (+1, -1), (+1, 0), (+1, +1),
     ( 0, +1), (-1, +1), (-1, 0), (-1, -1));
```

We must also set the font of the canvas to that specified by the user, and paint the background for the Label if the *Transparent* property is *False*.

For each of the shadow, highlight, and original text we then set the canvas' font color appropriately, define a rectangle in which to draw the text, and draw it. The rectangle must be offset by the depth value modified by the direction requested for the shadow and highlight, adjusting all this by the minimum offset found above. Using the

API procedure *DrawText* allows for the alignment of the text to be handled automatically.

The component could now be compiled and put into use. Each of the new properties can be set individually to achieve many different 3-D effects.

### Making It Easier

We don't want to have to set each property separately to get the effect we're after. It would be a lot nicer to package combinations of these properties into single, easy-to-use properties.

The first of these combines the depth and direction properties for the highlight and shadow. Called *EffectStyle*, it takes a limited range of values, again making it ideal for an enumerated type:

```
type TEffectStyle = (esNone, esCustom, esRaised,
                    esSunken, esShadow, esFlying);
```

The *esCustom* value is used to indicate that the user has altered one of the preset values. This is to avoid confusion where the *EffectStyle* wouldn't reflect the current appearance of the Label.

To achieve this, the write methods for the constituent properties are modified to alter the *EffectStyle* to *esCustom* whenever they are changed. To avoid an infinite loop — where setting the *EffectStyle* sets the constituent property that then sets the *EffectStyle*, etc. — a flag is introduced to indicate when a change originates in the altering of the overall style:

```
procedure TLabelEffect.SetDirectionHighlight(
    EdDirection: TEffectDirection);
begin
    if FDirectionHighlight <> EdDirection then
        begin
            FDirectionHighlight := EdDirection;
            { Default to custom style when changed }
            if not BChangingStyle then
                SetEffectStyle(esCustom);
            Invalidate;
        end;
end;
```

Setting the *EffectStyle* property will have side-effects requiring the use of a write method. The method in Figure 3 sets the appropriate values into the depth and direction properties defined above. Note that the affected properties are set through their associated methods rather than setting the internal variables directly. This ensures that any later changes to their implementation will have minimal impact on this method.

The flag is first set to stop an infinite loop. Then the constituent properties are set based on the style's new value. Note that there is some color property processing in here as well, since the "Flying" style is defined to have two shadows. Also, the *AsButton* property is reset if the style is not "Raised" or "Lowered." Finally the flag is reset so that any further changes to the con-

```

procedure TLabelEffect.Paint;
const
  WAlignments: array [TAlignment] of Word =
    (DT_LEFT, DT_RIGHT, DT_CENTER);
var
  IMinOffset: Integer;
  RctTemp: TRect;
  StrText: array [0..255] of Char;
begin
  { Find minimum of all offsets,
    including the font itself }
  IMinOffset := Min(
    Min(Min(
      IOffsets[DirectionHighlight,drX] *
      DepthHighlight,
      IOffsets[DirectionShadow,drX] *DepthShadow),
      IOffsets[DirectionHighlight,drY] *
      DepthHighlight),
      IOffsets[DirectionShadow,drY] *
      DepthShadow),0);
  { Ensure canvas font is set }
  Canvas.Font := Self.Font;

  with Canvas do begin
    { Fill in background }
    if not Transparent then
      begin
        Brush.Color := Self.Color;
        Brush.Style := bsSolid;
        FillRect(ClientRect)
      end;
  end;

  { Don't overwrite background above }
  Canvas.Brush.Style := bsClear;

  { Get label's caption }
  GetTextBuf(StrText, SizeOf(StrText));
  Canvas.Font.Color := ColourShadow;

  { Create a rect that is offset for the shadow }

```

**Figure 2:** The first cut of the *Paint* method for *TLabelEffect*.

stituent properties will alter the style to *esCustom*. Similarly, the colors of all three parts of the Label are controlled by the *ColourScheme* property. Again the valid values come from an enumerated type:

```

type TColourScheme = (csCustom, csText, csWindows,
  csEmbossed, csGold, csSteel);

```

The *csCustom* value functions as for *esCustom*, indicating a change to a preset value.

The color schemes are held in an array, indexed by the *ColourScheme* property itself. This array cannot be made a constant since the *csCustom* entries are updated whenever the colors change, so that the rest of the component works easily. Therefore the array is initialized in the *Create* method of the Label, and the *csCustom* entries are updated in the write methods for the color properties.

As for the *EffectStyle* property, a flag is used for control when the color scheme is reset to *csCustom*. This is set at the beginning of the *SetColourScheme* method and reset at its end.

There are no changes to the *Paint* method for the Label since the new properties only allow us to alter the original

```

RctTemp := Rect(
  ClientRect.Left - IMinOffset +
  IOffsets[DirectionShadow,drX] * DepthShadow,
  ClientRect.Top - IMinOffset +
  IOffsets[DirectionShadow,drY] * DepthShadow,
  ClientRect.Right - IMinOffset +
  IOffsets[DirectionShadow,drX] * DepthShadow,
  ClientRect.Bottom - IMinOffset +
  IOffsets[DirectionShadow,drY] * DepthShadow);
  { Draw shadow text with alignment }
  DrawText(Canvas.Handle, StrText, StrLen(StrText),
    RctTemp, DT_EXPANDTABS or DT_WORDBREAK or
    WAlignments[Alignment]);
  Canvas.Font.Color := ColourHighlight;
  { Create a rect that is offset for the highlight }
  RctTemp := Rect(
    ClientRect.Left - IMinOffset +
    IOffsets[DirectionHighlight,drX] * DepthHighlight,
    ClientRect.Top - IMinOffset +
    IOffsets[DirectionHighlight,drY] * DepthHighlight,
    ClientRect.Right - IMinOffset +
    IOffsets[DirectionHighlight,drX] * DepthHighlight,
    ClientRect.Bottom - IMinOffset +
    IOffsets[DirectionHighlight,drY] * DepthHighlight);
  { Draw highlight text with alignment }
  DrawText(Canvas.Handle, StrText, StrLen(StrText),
    RctTemp, DT_EXPANDTABS or DT_WORDBREAK or
    WAlignments[Alignment]);
  { Restore original font colour }
  Canvas.Font.Color := Font.Color;
  { Create a rect that is offset for the original text }
  RctTemp := Rect(
    ClientRect.Left - IMinOffset,
    ClientRect.Top - IMinOffset,
    ClientRect.Right - IMinOffset,
    ClientRect.Bottom - IMinOffset);
  { Draw original text with alignment }
  DrawText(Canvas.Handle, StrText, StrLen(StrText),
    RctTemp, DT_EXPANDTABS or DT_WORDBREAK or
    WAlignments[Alignment]);
end;

```

properties as a group, without adding any new variants to the final appearance.

### Rotating Text

The next step is to allow the text to be rotated. The *Angle* property is added for this purpose, and represents the number of degrees through which the text will be rotated counter-clockwise from the x-axis. If this property is left at zero — its initial value — the component will function exactly as before.

To limit the range of values available for this property, it's declared as a subrange:

```

type TAngleRange = 0..360;

```

Again, this allows Delphi to enforce valid values for us. Note that normally 0 and 360 degrees would be regarded as the same thing, but for our purposes they are different. If the *Angle* is 0 then the Label behaves exactly as before, with full alignment and word wrap. If any other value is specified, then the Label is centered in a circle with a diameter equal to the width of the text. Allowing a value of 360 degrees means that we can still display the Label

```

procedure TLabelEffect.SetEffectStyle(
    EsStyle: TEffectStyle);
begin
    if FEffectStyle <> EsStyle then
        begin
            { So it doesn't reset to custom }
            BChangingStyle := True;
            BChangingScheme := True;
            FEffectStyle := EsStyle;
            SetColourHighlight(ClrSchemes[ColourScheme,
                cpHighlight]);
            case FEffectStyle of
                esRaised:
                    begin
                        SetDirectionHighlight(edUpLeft);
                        SetDirectionShadow(edDownRight);
                        SetDepthHighlight(1);
                        SetDepthShadow(1);
                    end;
                .
                .
                .
            else
                SetAsButton(False);
            end;

            { So further changes set style to custom }
            BChangingStyle := False;
            { So further changes set colour scheme to custom }
            BChangingScheme := False;
        end;
    end;

```

**Figure 3:** The *SetEffectStyle* method (abbreviated).

horizontally but also based at the same point as for other angles (run the accompanying demonstration form to see the effects of this).

Normally fonts are only available in a horizontal aspect, since this is what is commonly required. We don't want to have to go through complicated programming steps to convert a standard font into one sloped appropriately. Fortunately, Windows provides a simple solution to this in the form of the API procedure *CreateFontIndirect* (see online Windows API help for more details).

*CreateFontIndirect* takes a definition of the desired font, in the form of a *TLogFont* structure, and finds or generates a matching font for our use. One of the attributes of the desired font is the *escapement*, or the angle between the base line of a character and the x-axis (measured in tenths of degrees).

Thus to get the font we want, we simply load the *TLogFont* structure with the current settings of the canvas' font, change the escapement angle to the required value, and ask Windows for a font similar to the one shown in [Figure 4](#). The new font is then set as the default for the canvas and we can draw text as usual.

In addition, here's an important point: The output precision attribute of the *TLogFont* structure should be set to:

```
OUT_TT_ONLY_PRECIS
```

```

procedure TLabelEffect.SetTextAngle(Cnv: TCanvas;
    AAngle: TAngleRange);
var
    { Storage area for font information }
    FntLogRec: TLogFont;
begin
    { Get the current font information. We only want to
      modify the angle. }
    GetObject(Cnv.Font.Handle, SizeOf(FntLogRec),
        Addr(FntLogRec));
    { Modify the angle. "The angle, in tenths of a degrees,
      between the base line of a character and the x-axis."
      - Windows API Help file. }
    FntLogRec.lfEscapement := AAngle * 10;
    { Request TrueType precision }
    FntLogRec.lfOutPrecision := OUT_TT_ONLY_PRECIS;
    { Delphi will handle the deallocation of
      the old font handle }
    Cnv.Font.Handle := CreateFontIndirect(FntLogRec);
end;

```

**Figure 4:** Creating the sloping font.

This will ensure that a TrueType font is chosen, allowing it to be more easily rotated.

To ease the processing required to position the text on the canvas, the *Alignment* property of the text is ignored (effectively *taLeftJustify*). In fact, it's set to this value as a side-effect of setting the *Angle* to a value other than 0.

We need to use trigonometry to determine the starting point for the sloping text. Remember that the text is centered in a circle of appropriate diameter. This is all handled in the code and is left to those interested enough to examine it. To reduce the amount of calculation required, the cosine and sine of the angle are calculated once when the *Angle* property is set, and are then saved for later use. Similarly, the conversion factor from degrees to radians is calculated once during the *Create* method of the Label, and then saved. All of Delphi's trigonometric functions work on angles expressed as radians (there are  $2\pi$  radians in 360 degrees).

Having computed the starting point, it's simply a matter of calling the *TextOut* method of the canvas to display the text. Note that any 3-D effects that may be applied do not alter their direction with respect to the canvas; that is, the effects are not rotated with the text.

Further trigonometry is required if the background for the Label is not transparent. In this case, the bounding rectangle must be tilted by the same amount as the text, and then filled with the background color. Delphi makes this easy with the *Polygon* method of the canvas. It takes an array of points, plots the lines between them (joining the last back to the first), and then fills the interior with the current brush style and color. The hardest part is working out where the points are that form the rotated rectangle.



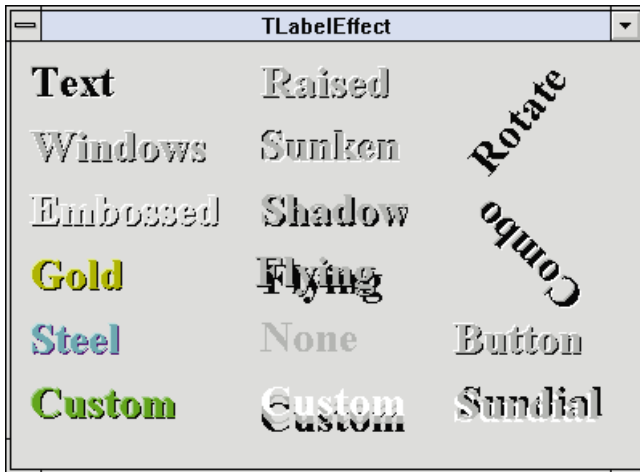


Figure 5: TLabelEffect's capabilities.

### Conclusion

That completes the *TLabelEffect* component, allowing for an almost unlimited range of 3-D effects, with the common ones just a click or two away. It also allows for the Label to be rotated, with or without a 3-D effect. Along the way we saw how properties can affect one another and interact to work as a whole.

To install the component into Delphi, copy the files LBLEFFCT.PAS and LBLEFFCT.DCR into the library directory (DELPHI\LIB). Then select **Options | Install Components** from Delphi's menu. Add LblEffect to the list of units and click **OK** to start the installation. The

new LabelEffect component will appear on the Standard page of the Component Palette. Check out the demonstration form that accompanies the component (shown in Figure 5), especially the "Sundial" example, to see its capabilities.

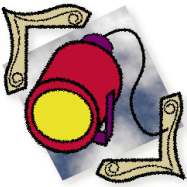
In our next article we'll look at providing property editors for the two preset properties, *EffectStyle* and *ColourScheme*, and providing help links for the component. Until then, enjoy playing with your new Label.  $\Delta$

I wish to thank Bill Murto and Curtis Keisler for their RotateLabel component and TextRotation examples, respectively (available on the Internet). These provided most of the code for the text rotation in this component. I hope I have added something more to their efforts, and spurred someone else to improve it further.

*The demonstration files referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\JUL\DI9607KW.*

Keith Wood is an analyst/programmer with CSC Australia, based in Canberra. He started using Borland's products with Turbo Pascal on a CP/M machine. Although not working with Delphi currently, he has enjoyed exploring it since it first appeared. You can reach him via e-mail at [kwood@nla.gov.au](mailto:kwood@nla.gov.au) or by phone (Australia) 6 291 8070.





## INFORMANT SPOTLIGHT

Delphi 2 / Object Pascal



By *Michael Maloof*

# An Eye on GDI

## A Delphi 2 Object for Monitoring the Wily GDI Beast

**I**t's no secret: most programmers know that Windows 95 is not a true 32-bit operating system. Everyone seems to appreciate that some compromises were necessary to ensure compatibility with existing 16-bit applications. What may surprise you is that one of those 16-bit compromises can seriously affect the stability of your Delphi 2 applications.

Tracking your application's GDI (Graphics Display Interface) usage, and the available GDI resources, are common tasks for Windows 3.1 and Delphi 16-bit development. It's all too easy to build spectacular forms with layers of panels, tabs, and grids. Running out of memory is just as easy. Often the GDI is exhausted, with symptoms ranging from partially displayed screens to a completely frozen machine.

### Wobegone Days

With Windows 95, Delphi 2, and the multi-gigabyte, flat-address-space world of 32-bit programming, those GDI woes are gone forever, right? Not quite. Windows 95 has done much to improve the limitations of Windows 3.1 programming, but the GDI remains true to its 16-bit heritage.

The bad news is the GDI is still a 16-bit resource and remains limited to 64K. However, the good news is that Windows 95 uses less GDI memory than Windows 3.1 by shifting items such as TrueType font support to other areas of the system.

Is a 64K GDI region still a serious issue? It can be if left unmonitored. The point is that you need to know your program's impact on the GDI, and you should keep an eye on the system-wide GDI usage. This is easily done using one of the commercial utilities on the market. For example, Norton Utilities for Windows 95 contains a sophisticated resource monitor that

can be set to warn you of impending doom. There are even shareware and freeware monitors available (one is included in the Accessories folder with Windows 95).

### Delphi: An Ideal Custom Solution

While these external monitors can do the job, the ideal solution is to create a Delphi-based GDI resource monitor that can be linked to your application. The Delphi approach allows you to track your GDI usage with much finer granularity. During development, you can observe the GDI impact of specific control creation events by simply calling your GDI monitor before and after the event.

A Delphi-based GDI monitor could even be used to protect your application from other GDI-eating applications. Simply check the available GDI memory at critical points in your application, and warn the user to close a few applications or windows before continuing.

Building a GDI resource monitor in Delphi should be simple. In fact, a few 16-bit versions of Delphi-based resource monitors already exist. The problem is that the one function used to measure the GDI, namely *GetFreeSystemResources*, is no longer available. (At least, it's no longer part of the published Win32 API.)

From the Microsoft point of view, *GetFreeSystemResources* is no longer support-

ed. Its replacement, *GlobalMemoryStatus*, is a useful function that returns a wealth of information, but absolutely nothing about GDI usage. This is unfortunate considering the importance of this resource.

## Get Me Some Free Resources

Of course, the *GetFreeSystemResources* function isn't really gone. It still exists as part of the Windows 95 support for 16-bit applications. You'll find the function in the file USER.EXE. The question is, how do you call this 16-bit function from your 32-bit Delphi application?

There are three ways this could be accomplished. The Microsoft approved method for calling a 16-bit routine from a 32-bit Windows 95 application is to use a *flat thunk*. The process is actually uglier than its name, and involves the use of a *thunk* compiler and separate 16- and 32-bit DLLs to perform this 16-to-32-bit magic. In spite of being the approved method, this only works under Windows 95, meaning that your code must compensate for running under Windows NT.

The undocumented method — which happens to be the one Microsoft uses — involves the use of the *LoadLibrary16*, *FreeLibrary16*, and *GetProcAddress16* functions found in the KERNEL32.EXE. Matt Pietrik discovered these undocumented functions, and described their use in his "PC Tech" column in the September 26, 1995 issue of *PC Magazine*. Considering Microsoft's reliance on these functions, they're probably safe to use, but like the flat thunk, they'll work only on Windows 95.

The easy method — which happens to be the one I use — takes advantage of the fact that Delphi 2 includes the original Delphi 16-bit program. You can use the 16-bit version to build a very small, non-visual program that does nothing but call *GetFreeSystemResources* and return the GDI utilization. This 16-bit application is called from inside your 32-bit Delphi application, and the return value can be processed any way you see fit.

This technique works equally well under Windows 95 and Windows NT. (It should be noted that Windows NT, as a true 32-bit operating system, does not have a GDI usage problem. The fact that this technique works under NT simply means that your code can safely call the GDI monitor without worrying about the current platform.)

Credit for the easy method must be given to Lou Grinzo, who presents the technique in his book *Zen of Windows 95 Programming* [The Coriolis Group, Inc., 1996].

## Getting Started

The first step is to create the 16-bit application that will call *GetFreeSystemResources* and return the current GDI utilization. Let's step through the process:

- Launch the 16-bit version of Delphi and start a new project. Then use the Project Manager to remove the *Unit1* file from the default *Project1*. The resulting project source code is shown here:

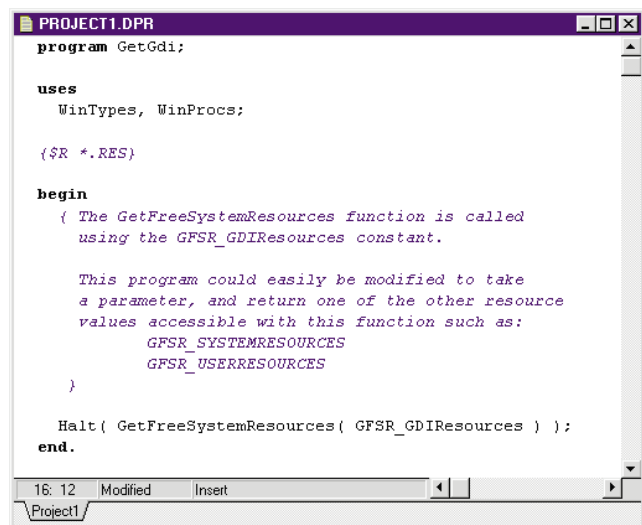
```
program Project1;

uses
  Forms;

{$R *.RES}

begin
  Application.Run;
end.
```

- This leaves a project with only one unit: *Project1*. Edit *Project1* and remove the *Forms* unit from the *uses* clause. By removing the *Forms* unit, you'll find that Delphi can produce a very small executable.
- Add the *WinTypes* and *WinProcs* units to the *uses* clause. These units give us the access we need to the Windows 16-bit API and the associated constants.
- Finally, remove the *Application.Run* statement, and replace it with a *Halt* statement that calls the *GetFreeSystemResources* function and passes our request for the GDI utilization. The result is shown in Figure 1. The *Halt* statement tells an application to terminate, but it can also pass back an exit code to the application that launched it. In this case, our exit code is the amount of free GDI resources.



```
PROJECT1.DPR
program GetGdi;

uses
  WinTypes, WinProcs;

{$R *.RES}

begin
  { The GetFreeSystemResources function is called
    using the GFSR_GDIResources constant.

    This program could easily be modified to take
    a parameter, and return one of the other resource
    values accessible with this function such as:
      GFSR_SYSTEMRESOURCES
      GFSR_USERRESOURCES
  }

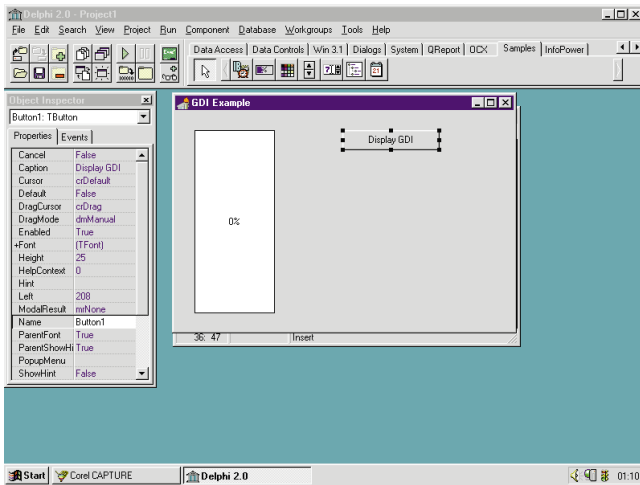
  Halt( GetFreeSystemResources( GFSR_GDIResources ) );
end.
```

**Figure 1:** This simple Delphi 1 project makes a call to the Windows 3.x API function, *GetFreeSystemResources*, and returns the result to the program that calls it.

## A Simple Example

The next step is to integrate this executable into a 32-bit Delphi application. To illustrate the basic principle, we'll create a simple form that contains a Gauge and a Button. When the Button is pressed, the Gauge progress value will be set to the available GDI.

Using a default project and form, place a Gauge component on the form. In our example, the Gauge's *Kind* property was set to *gkVerticalBar*. Place a Button on the form, and change the Button's *Caption* property to **Display GDI** (see Figure 2). Once the Gauge and Button have been placed on the form, double-click on the Button to create the framework for the *ButtonClick* procedure and enter the code shown in Figure 3.



```

procedure TForm1.Button1Click(Sender: TObject);
var
  StartupInfo: TStartupInfo;
  ProcessInfo: TProcessInformation;
  GdiValue: DWORD;
  cp: Boolean;
begin
  FillChar(StartupInfo, SizeOf(TStartupInfo), 0);

  // Get GDI resource utilization by calling
  // our GETGDI.EXE 16-bit Delphi application.
  cp := CreateProcess(nil, 'GETGDI.EXE', nil, nil, False,
    NORMAL_PRIORITY_CLASS, nil, nil,
    StartupInfo, ProcessInfo);

  if cp then
    begin
      WaitForSingleObject(ProcessInfo.hProcess, 1000);
      GetExitCodeProcess(ProcessInfo.hProcess, GdiValue);
      CloseHandle(ProcessInfo.hProcess);
      Gauge1.Progress := GdiValue;
    end
  else
    Gauge1.Progress := 0;

end;

```

**Figure 2 (Top):** Building a simple GDI monitor with Delphi 2 that calls the Delphi 1 application shown in Figure 1.

**Figure 3 (Bottom):** The sample application's *ButtonClick* procedure.

You'll notice we're using a call to *CreateProcess* to launch our 16-bit application, GETGDI.EXE. Windows 95 will go through the following specific search sequence to find this executable:

- The directory from which the main application was loaded.
- The current directory.
- The Windows System directory.
- The Windows directory.
- The directories that are listed in the PATH environment variable.

When you run this simple example and press the Button, the current GDI-free percentage is returned from our GETGDI executable. The Gauge reflects this percentage. Assuming that Delphi is still running, you may notice that your GDI-free percentage is already in the 80s. Considering that you were probably in the high 90s when Windows 95 started, you've already lost a fair amount of memory.

## A GDI Monitoring Object

Where did the GDI go? While prototyping a new Delphi 2 application, I was rudely awakened by Norton's System Doctor warning me that my GDI resources were dangerously low. This led to the discovery of the 16-bit nature of the Windows 95 GDI, but that was only the beginning.

Knowing that the GDI is still a limited resource and knowing how to preserve that resource are two distinct issues. Which form consumed the most GDI? Which components or controls could be eliminated? Which forms should be created during program initialization, and which ones should be created only on demand? How many forms can be opened before the application approaches critical mass?

The search for these answers resulted in the creation of a *TTimeMem* object. Our lead Delphi programmer, Lynn Settle, constructed the *TTimeMem* object to measure the elapsed time and the memory consumed between two events. [*TTimeMem* is available for download from Library 7 in the Delphi Forum on CompuServe. The file name is TIMEMEM.ZIP.] It was during the creation of the *TTimeMem* object that we discovered there was no apparent way to track the GDI.

Using the technique presented in this article, we now have a *TTimeMemGDI* object that adds GDI tracking to the original object's capabilities. This object can be integrated into your application in many ways.

To illustrate the use of the object, it's included in the main project source file shown in Figure 4. By incorporating *TTimeMemGDI* in this fashion, we can now allow Delphi to create the sample form, and measure the form's impact on the GDI, as well as other system resources (see Figure 5). The complete .PAS file for *TTimeMemGDI* is shown in Listing One, starting on page 21.

```

program Project2;

uses
  Forms,
  TimeMemG,
  Unit2 in 'Unit2.pas' {Form1};

{$R *.RES}

var
  TimeMem : TTimeMemGDI;
begin
  Application.Initialize;
  // Create an instance of the object.
  TimeMem := TTimeMemGDI.Create;
  // Capture starting values.
  TimeMem.Start;
  // Create the sample form.
  Application.CreateForm(TForm1, Form1);
  // Capture the ending values.
  TimeMem.Stop;
  // Show the impact of creating this form.
  TimeMem.DisplayInfo('Creating Sample Form');

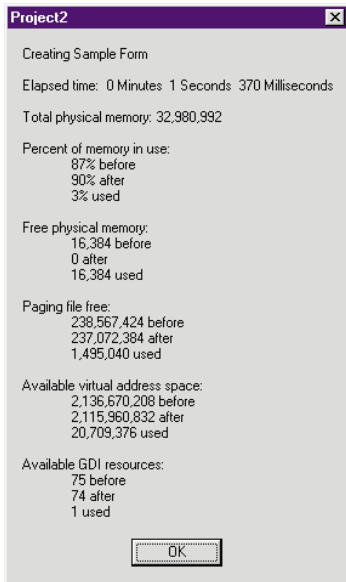
  Application.Run;

end.

```

**Figure 4:** Using the *TTimeMemGDI* object.





**Figure 5:** An example of the GDI Monitor at run time.

By creating an instance of the *TTimeMemGDI* object and calling the *Start* method, we have an accurate picture of the initial environment. The *Stop* method is called immediately following the *CreateForm* procedure, and the *DisplayInfo* method gives a complete picture of what has changed between the two events.

You can use the *TTimeMemGDI* object by calling sequences of *Start*, event of interest, *Stop*, and *DisplayInfo*. You can also measure the increasing impact of a

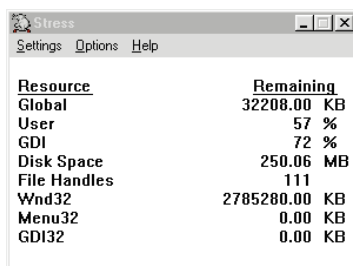
series of events using a sequence such as, *Start*, event #1, *Stop*, *DisplayInfo*, event #2, *Stop*, *DisplayInfo*. By continually calling only the *Stop* and *DisplayInfo* methods, you'll see the cumulative effect of the events.

### Conclusion

We concentrated our GDI resource reduction efforts on descendants of the *TWinControl*. We eliminated virtually all of our *TPanel* components using Eric Uber's *TGraphicPanel*, which descends from *TBevel*, but has the same visual properties as a *TPanel*. We also substantially reduced the number of *TDBNavigator* components. Furthermore, we carefully monitor the GDI situation as we create forms on demand. [Eric's *TGraphicPanel* is found in the file, GRAPHPNL.ZIP, located in Library 9 of the Delphi Forum on CompuServe.]

By using this technique we reduced the GDI usage in our application from an unbelievable 90 percent to about 30 percent, or roughly about twice as much as Delphi itself. That's not bad for a major vertical market application that relies heavily on Grid, Page, and Tab Controls. This reduction is especially significant because we'll soon be applying for Windows 95 Logo Certification.

One element of the certification process is a general stability test. **Figure 6** shows the application being exercised while running Microsoft's Stress utility (available on the MSDN CDs). Among other things, the Stress utility randomly consumes between 40 and 60 percent of the GDI. An application that consumes more than 40 percent of the GDI may not survive in that environment. Will



**Figure 6:** The Windows Stress utility at run time.

your application? [For more information about the Windows 3.x Stress utility, see Karl Thompson's article "Gimme Some Stress!" beginning on page 38.]

It's unfortunate that the GDI beast still haunts Windows 95, but with a little effort you can prevent your Delphi applications from becoming its next meal.  $\Delta$

*The demonstration files referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\JUL\DI9607MM.*

Michael Maloof is CEO of Smart Shop Software, Inc., a Coeur d'Alene, ID-based publisher of software for the manufacturing industry. Following Jeff Duntemann's software publishing advice, Michael married his best friend, moved to a beautiful place, and brings his golden retriever to work every day. You can reach Michael through CompuServe at 76050,1607.

### Begin Listing One: TTimeMemGDI

```

unit TimeMemG;

interface

uses SysUtils, Windows, Dialogs, Classes, Forms;

type
  TTimeMemGDI = class(TObject)
  private
    StartTime: TDateTime;
    UsedTime: TDateTime;
    // Bytes of physical memory.
    TotalPhys: Integer;
    // Percent of memory in use.
    StartMemoryLoad: Integer;
    EndMemoryLoad: Integer;
    // Bytes physical memory available.
    StartAvailPhys: Integer;
    EndAvailPhys: Integer;
    // Bytes available in paging file.
    StartAvailPageFile: Integer;
    EndAvailPageFile: Integer;
    // Bytes available in the user mode portion of
    // the virtual address space.
    StartAvailVirtual: Integer;
    EndAvailVirtual: Integer;
    // Percent GDI resources available.
    StartAvailGDI: Integer;
    EndAvailGDI: Integer;

  public
    // Capture start time, memory stats, and GDI free.
    procedure Start;
    // Capture stop time, memory stats, and GDI free.
    procedure Stop;
    // Display information.
    procedure DisplayInfo(InfoTitle: String);
  end;

implementation

procedure TTimeMemGDI.Start;
var
  SysMemoryStatus: TMemoryStatus;
  StartupInfo: TStartupInfo;
  ProcessInfo: TProcessInformation;
  GdiValue: DWORD;
  cp: Boolean;

```

```

begin
  FillChar(SysMemoryStatus, SizeOf(TMemoryStatus), 0);
  //SysMemoryStatus.dwLength := SizeOf(TMemoryStatus);
  GlobalMemoryStatus(SysMemoryStatus);

  { Get starting memory info. }
  StartMemoryLoad := SysMemoryStatus.dwMemoryLoad;
  StartAvailPhys := SysMemoryStatus.dwAvailPhys;
  StartAvailPageFile := SysMemoryStatus.dwAvailPageFile;
  StartAvailVirtual := SysMemoryStatus.dwAvailVirtual;

  FillChar(StartupInfo, SizeOf(TStartupInfo), 0);

  // Get starting GDI resources percentage.
  cp := CreateProcess(nil, 'GETGDI.EXE', nil, nil, False,
    NORMAL_PRIORITY_CLASS, nil, nil,
    StartupInfo, ProcessInfo);

  if cp then
    begin
      WaitForSingleObject(ProcessInfo.hProcess, 1000);
      GetExitCodeProcess(ProcessInfo.hProcess, GdiValue);
      CloseHandle(ProcessInfo.hProcess);
      StartAvailGDI := GdiValue;
    end
  else
    StartAvailGDI := -1;

  // Before I go, brother could you spare the time?
  StartTime := NOW;
end;

procedure TTimeMemGDI.Stop;
var
  StartupInfo: TStartupInfo;
  ProcessInfo: TProcessInformation;
  GdiValue: DWORD;
  cp: Boolean;
  SysMemoryStatus: TMemoryStatus;
begin
  // How long has it been?
  UsedTime := NOW - StartTime;

  SysMemoryStatus.dwLength := SizeOf(SysMemoryStatus);
  GlobalMemoryStatus(SysMemoryStatus);

  // Get ending memory info.
  TotalPhys := SysMemoryStatus.dwTotalPhys;
  EndMemoryLoad := SysMemoryStatus.dwMemoryLoad;
  EndAvailPhys := SysMemoryStatus.dwAvailPhys;
  EndAvailPageFile := SysMemoryStatus.dwAvailPageFile;
  EndAvailVirtual := SysMemoryStatus.dwAvailVirtual;

  FillChar(StartupInfo, SizeOf(TStartupInfo), 0);

  // Get ending GDI resources percentage.
  cp := CreateProcess(nil, 'GETGDI.EXE', nil, nil, False,
    NORMAL_PRIORITY_CLASS, nil, nil,
    StartupInfo, ProcessInfo);

  if cp then
    begin
      WaitForSingleObject(ProcessInfo.hProcess, 1000);
      GetExitCodeProcess(ProcessInfo.hProcess, GdiValue);
      CloseHandle(ProcessInfo.hProcess);
      EndAvailGDI := GdiValue;
    end
  else
    EndAvailGDI := -1;
end;

{ The DisplayInfo procedure allows you to visually
inspect the values captured by the Start and Stop
methods. The InfoTitle parameter allows you to
display a meaningful title, such as 'After Order Entry
Form Creation.' This approach was adequate for our
needs, but certainly leaves room for significant
enhancements.}

```

```

procedure TTimeMemGDI.DisplayInfo(InfoTitle: string);
const
  CRLF = #10+#13;
  TAB = #9;
var
  Hour, Min, Sec, MSec: Word;
var
  TimeMemMessage,
  StrUsedMemoryLoad,
  StrUsedAvailPhys,
  StrUsedAvailPageFile,
  StrUsedAvailVirtual,
  StrUsedAvailGDI,
  StrStartMemoryLoad,
  StrStartAvailPhys,
  StrStartAvailPageFile,
  StrStartAvailVirtual,
  StrStartAvailGDI,
  StrEndMemoryLoad,
  StrTotalPhys,
  StrEndAvailPhys,
  StrEndAvailPageFile,
  StrEndAvailVirtual,
  StrEndAvailGDI: string;
begin
  DecodeTime(UsedTime, Hour, Min, Sec, MSec);

  StrUsedMemoryLoad :=
    IntToStr(EndMemoryLoad - StartMemoryLoad);
  StrUsedAvailPhys :=
    FloatToStrF((StartAvailPhys - EndAvailPhys),
      ffNumber, 10, 0);
  StrUsedAvailPageFile :=
    FloatToStrF((StartAvailPageFile - EndAvailPageFile),
      ffNumber, 10, 0);
  StrUsedAvailVirtual :=
    FloatToStrF((StartAvailVirtual - EndAvailVirtual),
      ffNumber, 10, 0);
  if (StartAvailGDI = -1) or (EndAvailGDI = -1) then
    StrUsedAvailGDI := 'Unable to determine'
  else
    StrUsedAvailGDI :=
      FloatToStrF((StartAvailGDI - EndAvailGDI),
        ffNumber, 10, 0);

  StrTotalPhys :=
    FloatToStrF(TotalPhys, ffNumber, 13, 0);
  StrStartMemoryLoad :=
    FloatToStrF(StartMemoryLoad, ffNumber, 13, 0);
  StrStartAvailPhys :=
    FloatToStrF(StartAvailPhys, ffNumber, 13, 0);
  StrStartAvailPageFile :=
    FloatToStrF(StartAvailPageFile, ffNumber, 13, 0);
  StrStartAvailVirtual :=
    FloatToStrF(StartAvailVirtual, ffNumber, 13, 0);
  if StartAvailGDI = -1 then
    StrStartAvailGDI := 'Error accessing GetGDI.exe'
  else
    StrStartAvailGDI :=
      FloatToStrF(StartAvailGDI, ffNumber, 13, 0);
  StrEndMemoryLoad :=
    FloatToStrF(EndMemoryLoad, ffNumber, 13, 0);
  StrEndAvailPhys :=
    FloatToStrF(EndAvailPhys, ffNumber, 13, 0);
  StrEndAvailPageFile :=
    FloatToStrF(EndAvailPageFile, ffNumber, 13, 0);
  StrEndAvailVirtual :=
    FloatToStrF(EndAvailVirtual, ffNumber, 13, 0);
  if EndAvailGDI = -1 then
    StrEndAvailGDI := 'Error accessing GetGDI.exe'
  else
    StrEndAvailGDI :=
      FloatToStrF(EndAvailGDI, ffNumber, 13, 0);

  TimeMemMessage :=
    'Elapsed time: ' + IntToStr(Min) + ' Minutes ' +
    IntToStr(Sec) + ' Seconds ' + IntToStr(MSec) +

```

```
' Milliseconds' + CRLF+CRLF +

'Total physical memory: ' + StrTotalPhys +CRLF+CRLF+
'Percent of memory in use:' +CRLF+
TAB + StrStartMemoryLoad + '% before' +CRLF+
TAB + StrEndMemoryLoad + '% after' +CRLF+
TAB + StrUsedMemoryLoad + '% used' +CRLF+CRLF+

'Free physical memory:' +CRLF+
TAB + StrStartAvailPhys + ' before' +CRLF+
TAB + StrEndAvailPhys + ' after' +CRLF+
TAB + StrUsedAvailPhys + ' used' +CRLF+CRLF+

'Paging file free:' +CRLF+
TAB + StrStartAvailPageFile + ' before' +CRLF+
TAB + StrEndAvailPageFile + ' after' +CRLF+
TAB + StrUsedAvailPageFile + ' used' +CRLF+CRLF+

'Available virtual address space:' +CRLF+
TAB + StrStartAvailVirtual + ' before' +CRLF+
TAB + StrEndAvailVirtual + ' after' +CRLF+
TAB + StrUsedAvailVirtual + ' used' +CRLF+CRLF+

'Available GDI resources:' +CRLF+
TAB + StrStartAvailGDI + ' before' +CRLF+
TAB + StrEndAvailGDI + ' after' +CRLF+
TAB + StrUsedAvailGDI + ' used';

  ShowMessage(InfoTitle + CRLF+CRLF + TimeMemMessage);
end;

end.
```

**End Listing One**





## SIGHTS & SOUNDS

Delphi 1 / Object Pascal



By *Charlie Howell*

# Animation Made Simple

## Using the *TAnimated* Component to Integrate Animation and Sound into Applications

**A**nimation has become an integral part of user-friendly applications. Since the release of Windows 95, who among us has resisted the temptation to delete files — or even whole directories — just so we could watch them fly into the recycle bin? An hourglass that remains inactive for more than a few seconds is a sure way to induce a yawn or a reboot. However, animated audio/visual features hold a user's attention, inform them of ongoing processes, and can be downright entertaining.

Potential uses for animated sequences during time-consuming processes include file management operations, establishing online connections, SQL processing, and last — but certainly not least — ODBC connections. Each of these operations may leave users wondering what their computer is doing, unless the developer provides some meaningful visual cues.

David Baldwin's freeware *TAnimated* component makes it a snap for developers to include animation in Delphi applications. *TAnimated* offers the ability to display a sequence of bitmaps at specific intervals, respond to program events, allow continuous looping or start-stop display, and synchronize sounds with animation. In short, the possibilities of *TAnimated* applications are limited only by the developer's creativity. As a native Delphi VCL component, all the capabilities of *TAnimated* become a part of the developer's executable file.

### What is *TAnimated*?

*TAnimated* is a non-windowed descendant of *TGraphicControl*. The component is a virtual movie projector that displays bitmap "filmstrips" like the file transmission example shown in [Figure 1](#). Key properties provide a tremen-



**Figure 1:** A file transmission "filmstrip" using the PCs from the Windows 95 desktop. This bitmap is shown one section at a time to create an animated effect.

dous amount of flexibility for the developer. For fluid animation, individual frames of the filmstrip may be displayed in sequence, at a preset interval ranging from one millisecond to 32 seconds. Although the full range of positive integers may be specified by the interval property, Baldwin's Help file explains actual display intervals are in multiples of 55 milliseconds.

Animation is "live" at design time, once the number of frames in the filmstrip and the display interval is specified via the Object Inspector, and *Play* is set to *True*. This provides a useful way to test bitmap filmstrips before compiling a program or writing code to control what users will see. After setting the *Loop* property to *True*, a *TAnimated* component will loop or replay to display continuous movement. Alternatively, a filmstrip may be played through once in response to an event. Additionally, animated sequences may be set to play in forward or reverse, a feature that makes a single filmstrip useful for two-way operations, like uploading and downloading files.

*TAnimated* also offers the option to set *Play* to *False* and display specific filmstrip frames at run time via the integer *Frame* property. This provides an easy way to use state change graphics like the trash can shown in [Figure 2](#).

*TAnimated* includes a *TransparentColor* property that allows specification of a color in the



bitmap to appear transparent. The *TransparentColor* property is useful for blending the component into a form, but it also offers the possibility of layering two or more *TAnimated* components to create a complex animation.



**Figure 2:** A two-frame state change filmstrip.

Besides standard drag-and-drop mouse events, a type of *TNotifyEvent*, *OnChangeFrame*, was added by the component developer to allow sounds to be synchronized with specific filmstrip frames. The sample application included with *TAnimated* includes the source code required to test for the presence of MMSYSTEM.DLL and then play a .WAV file. Of course, the *OnChangeFrame* event could also be used to start any event.

### An Issue of Scale

The size of the *TAnimated* component must remain equal to a single frame of the filmstrip. If the form's *Scaled* property is set to *True* at design time (default setting), and the screen resolution is altered to enlarge or reduce the screen font, the size of the component will change. The result is like watching Grandma's old television, i.e. the individual frames slide across the component, rather than being displayed frame-by-frame.

If *TAnimated* components are the only components on the form, simply set the form's *Scaled* property to *False* at design time. The size of the form and the components will remain unchanged at different screen resolutions. However, if other font-based components are included on the form, they may not fit at lower screen resolutions. An alternative is to accept the form's default *Scaled* property at design time. If the form and its components must be resized, be sure to set the form's *AutoScroll* property to *False*. Then, reset the size of the *TAnimated* component at run time (upon activation of the form), as shown below:

```
procedure TAboutBox.FormActivate(Sender: TObject);
begin
  { Reset size of TAnimated component to 32 pixels
    x 32 pixels and play animation }
  Animated1.SetBounds(Left,Top,32,32);
  Animated1.Play := True;
end;
```

Using this approach, the form will rescale to accommodate any font-based components, and the size of the *TAnimated* component will reset to a single frame size to assure proper playback.

### Maintaining Fluid Animation

Animation may slow down or fail to play altogether if other tasks are allowed to dominate the processor. It may be necessary to interrupt processing to allow frames to change, or process a task in the background to maintain fluid animation in the foreground. Delphi provides an easy way to do both, but there are considerations to help decide which approach to implement. For tasks processed in a loop, an *Application.ProcessMessages* command may be inserted inside the loop. This gives the *TAnimated* component an opportunity to change frames each time the loop is repeated, if the timing coincides with the interval property setting.

If a task cannot be interrupted or broken into small chunks (e.g. a SQL query), it may be impossible to maintain fluid animation in a 16-bit application. In this case, the *Application.ProcessMessages* command may be used to interrupt processing and change frames before and after performing the task.

We'll demonstrate this approach in an example SQL application. First however, let's discuss background computing.

If a task can be broken into small blocks, or if it requires little processing time, fluid animation may be played in the foreground while the task is processed in the background. A task can be assigned to run in the background using the *OnIdle* event of *TApplication*. To do this, declare the *TIdleEvent* method in the **private** part of the **type** declaration of the form. Then, assign the name of the method to the *OnIdle* event and write an event handler (see [Figure 3](#)).

Delphi online help states that a complete *TApplication* event handler should be written and declared as a method of the main form, and the code should be executed as soon as the application is run. However, declaring and handling the *OnIdle* event in a secondary form works just fine. Specify and run the *TIdleEvent* in the secondary form, and after the process is complete, use the **nil** pointer to disable the method.

### A Practical Application for TAnimated

Delphi shipped with an example application named "Stocks Version 1.0". By default, the application resides in the directory \DELPHI\DEMOS\DB\STOCKS.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  { Specify TIdleEvent }
  Application.OnIdle := TimeConsumingProc;
  ShowAnimation;
end;

procedure TForm1.ShowAnimation;
begin
  { Show TAnimated form }
  AnimatedForm2.Visible := True;
  { Start animation }
  AnimatedForm2.Animated1.Play := True;
  { Display continuous animation }
  AnimatedForm2.Animated1.Loop := True;
  { Start TIdleEvent method }
  TimeConsumingProc;
end;

procedure TForm1.TimeConsumingProc(Sender: TObject;
                                     var Done: Boolean);
begin
  { Insert the time consuming task here with statement
    to break out of task }

  { Disable OnIdle event using nil pointer }
  Application.OnIdle := nil;
  { End animation by hiding form }
  AnimatedForm2.Visible := False;
end;
```

**Figure 3:** These three procedures demonstrate how to assign a task to run in the background using the *OnIdle* event of *TApplication* while displaying an animated sequence.

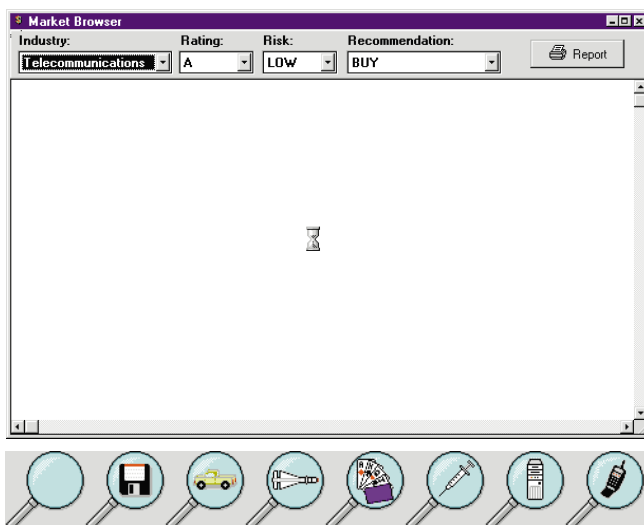
## SIGHTS & SOUNDS

Compile the application. Then, from the Broker's Command Center menu, select **View | Market Browser**. The Market Browser form includes four combo boxes that allow a user to filter a data set by industry, bond rating, risk-level, and a buy-sell-hold recommendation. The underlying code in the procedure *UpdateGridQuery* is a marvel of dynamic SQL. However, the Market Browser form is pretty boring (see **Figure 4**), and an hourglass is displayed while the query is processed.

We'll add a *TAnimated* component during processing to provide a graphical cue that confirms the user's selection from the **Industry** combo box. When a user selects an industry and the SQL statement begins processing to update the grid, a "search" form will appear that displays a magnifying glass; a split second later, an icon representing the selected industry will appear in the magnifying glass. The form caption will show that a search for matching records is underway, and will signal when the search is complete. Then, the requested data will appear in the Market Browser grid and the search form will disappear.

Here's how to do it. Add a new form to the project; name it *fmSearch* and save it in **SEARCH.PAS**. From the Object Inspector set the *BorderStyle* to *bsDialog* and *Position* to *poScreenCenter*. Also change the form's *Scaled* property to *False*. This ensures the size of a *TAnimated* component will remain unchanged at different screen resolutions. Add *Search* to the **uses** clause of the Browser unit. For this example, everything will be controlled from the *UpdateGridQuery* procedure in the Browser unit to make it easy to follow the sequence of events.

Next, select a *TAnimated* component from the Component Palette and place it on *fmSearch*. This assumes you have already installed the component (following guidance in the *Delphi User's Manual* or the *TAnimated* "readme" file). From the Object Inspector, select the *TAnimated BitMap* property and load the filmstrip shown in **Figure 5**. Set the *FrameCount* property to 8, and the height and width to 85 pixels and 87 pixels, respectively (i.e. the size of a single frame of the filmstrip).



**Figure 4 (Top):** The Market Browser in search or query mode.  
**Figure 5 (Bottom):** The state change filmstrip used to confirm a combo box selection.

The *UpdateGridQuery* procedure is called from each of the four combo boxes on the Market Browser form, and upon activation of the form. To enable the search form to appear only in response to a user updating the combo boxes, add an **if** statement to test whether the *fmMktBrowser* form is already visible. Within the same statement, add code to change the search form caption, set the *TAnimated* component to the first frame or blank magnifying glass (frame 0), and make the form visible. Insert an *Application.ProcessMessages* command so that *TAnimated* will be displayed before the SQL statements take over the processor.

By using the integer combo box index (*DropIndustry.ItemIndex*), it is simple to add code to select the integer frame value of *TAnimated* that matches the selected industry. The industry icons shown in the filmstrip in **Figure 5** were arranged in an order corresponding to the industries listed in the combo box. If a user picks the first selection, or **All**, the blank magnifying glass will remain visible until the query is processed. After the *TAnimated Frame* property is changed, an *Application.ProcessMessages* command again interrupts processing to allow the frame to change before the SQL query is opened. Add code to change the search form caption, and then hide the form after the grid is updated.

The modified *UpdateGridQuery* procedure is shown in **Figure 6**. The original developer's comments are enclosed in brackets **{ }** and the new comments are enclosed in parentheses **( )**. The new and improved Market Browser is shown in search mode in **Figures 7** and **8**. It works!

So, how does the addition of the component, bitmap, and code affect performance? The updated version of the Stocks executable file is a mere 20KB larger than the original, and there's no noticeable difference in the time required to process the queries. In fact, the queries seem to take *less* time, because the animation is distracting.

### Last Minute Tips

Unlike a mechanical movie projector, *TAnimated* is guaranteed to not break down. If you load a bitmap, set a few properties, and nothing happens, be sure that the *FrameCount* and the *Interval* properties are set to values greater than 1 and 0, respectively. Also, be sure that *Play* is set to *True*.

There are no rules that require a developer to set the *FrameCount* property to the actual number of frames in a bitmap filmstrip. A single filmstrip may be used to play fluid animation to depict an ongoing process, and then subsequent frames may be displayed to depict a completed process. At design time simply set the *FrameCount* property to the number of frames corresponding to the fluid animation sequence. Then, after completing the process at run time, set *Play* to *False*, increase the frame count, and jump to the frame that depicts the completed process.

### Conclusion

The possibilities for integrating animated sequences into applications are virtually unlimited using the *TAnimated* component. Remember to ensure the component will not be

```

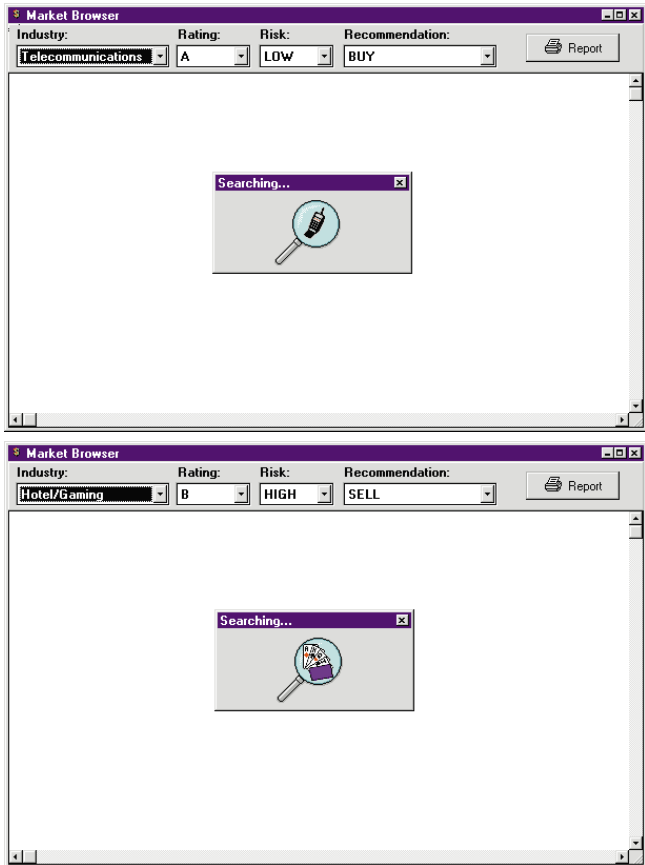
procedure TfmMktBrowser.UpdateGridQuery;
var
  WhereUsed: Boolean;

  function SQLPrefix: string;
  begin
    if WhereUsed then
      SQLPrefix := 'and '
    else
      begin
        WhereUsed := True;
        SQLPrefix := 'where '
      end;
  end;

begin
  (* Show Search form only if Browser is already open *)
  if fmMktBrowser.Visible then
    begin
      fmSearch.Show;
      fmSearch.Caption := 'Searching...';
      (* Set magnifying glass to first frame
      (no industry icon) *)
      fmSearch.Animated1.Frame := 0;
      (* Interrupt processing to allow animation to appear *)
      Application.ProcessMessages;
    end;
  Screen.Cursor := crHourGlass;
  WhereUsed := false;
  with GridQuery do begin
    Close;
    SQL.Clear;
    SQL.Add('select * from MASTER');
    if DropIndustry.ItemIndex > 0 then
      begin
        (* Set magnifying glass frame to selected industry *)
        fmSearch.Animated1.Frame := DropIndustry.ItemIndex;
        (* Interrupt processing to allow frames to change *)
        Application.ProcessMessages;
        { Lookup the industry code from the INDUSTRY table }
        TempQuery.Close;
        TempQuery.SQL.Clear;
        TempQuery.SQL.Add('Select IND_CODE from INDUSTRY');
        TempQuery.SQL.Add(
          'where LONG_NAME ' +
            DropIndustry.Items[DropIndustry.ItemIndex] + '');
        TempQuery.Open;
        { Store the industry code; used for RS report }
        IndustCode := TempQuery.Fields[0].AsString;
        { Now add the clause to the Grid SQL }
        SQL.Add(SQLPrefix+FldIndustry + ' = ' + IndustCode);
      end;
    if DropRating.ItemIndex > 0 then
      SQL.Add(SQLPrefix + FldRating + '=' +
        DropRating.Items[DropRating.ItemIndex] + '');
    if DropRecommend.ItemIndex > 0 then
      SQL.Add(SQLPrefix + FldRecommend + ' = ' +
        DropRecommend.Items[DropRecommend.ItemIndex] + '');
    if DropRisk.ItemIndex > 0 then
      SQL.Add(SQLPrefix+FldRisk + '=' +
        DropRisk.Items[DropRisk.ItemIndex] + '');
    SQL.Add('order by CO_NAME');
    Open;
  end;
  (* Close the Search form *)
  if fmMktBrowser.Visible then
    begin
      fmSearch.Caption := 'Search Complete';
      (* Interrupt processing to allow form caption to
      change before closing form *)
      Application.ProcessMessages;
      (* Hide the search form *)
      fmSearch.Hide;
    end;
  Screen.Cursor := crDefault;
end;

```

**Figure 6:** The modified *UpdateGridQuery* procedure.



**Figure 7 (Top):** The modified Market Browser in search mode after selecting **Telecommunications**. **Figure 8 (Bottom):** The modified Market Browser in search mode after selecting **Hotels/Gaming**.

resized at different screen resolutions, and that a fluid animation sequence will not be bogged down by other processing tasks. Your users will be impressed!

You may have noticed that I have carefully avoided one important question about *TAnimated* — if *Play* and *Loop* are set to *True*, and the *Visible* property is set to *False*, do the pictures continue to move?  $\Delta$

## References

Baldwin, L. David, *TAnimated* Help file, 1995.  
Cantù, Marco, *Mastering Delphi* [Sybex Inc., 1995].

David Baldwin's freeware *TAnimated* component, source code, and his example application may be found in the VCL library of the Borland Delphi Forum on CompuServe (GO DELPHI), and on the CD-ROM that ships with Marco Cantù's book *Mastering Delphi*.

*The demonstration applications referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\JUL\DI9607CH.*

Charlie Howell is an environmental scientist employed by the US Environmental Protection Agency, a free lance writer, and a wannabe programmer. He has developed and distributed database applications and spreadsheet templates for analyzing and sharing environmental data and information. You can reach Charlie at (214) 506-0812 or via e-mail at [chowell@cyberramp.net](mailto:chowell@cyberramp.net).





# VISUAL PROGRAMMING

Delphi / Object Pascal



By *Jim Callan*

## Worth the Wait

### Strategies for Keeping Users Patient

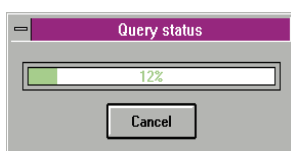
**R**ich application functionality inevitably leads to periodic processing delays. How application designers prepare users for these delays separates production applications from mere programming diversions. This article examines the most popular techniques used in software products to manage user expectations for snappy performance: progress indicators and changing the mouse pointer.

With the advent of Windows and the Delphi programming environment, even apparently simple applications can contain sophisticated features such as database access, report generation, intensive numeric computations, glitzy graphics, and telephony access to online information sources like the Internet.

After basic functionality, most users look at an application's response time when judging quality. In this era of instant gratification — when *now* is never fast enough — processing delays can be disappointing for unsuspecting users. Since it is inevitable that processing delays will occur in such feature-rich applications, the challenge becomes how to set and manage user expectations when processing delays do occur.

#### Setting Expectations

Changing the mouse cursor (a.k.a. mouse pointer) and using graphical progress indicators (a.k.a. meters or gauges) are the two most common methods employed in Windows applications to notify users of impending delays. Progress indicators are typically used to indicate percentage complete and percentage remaining during lengthy operations. Almost every standard Windows application uses a progress meter during installation, and many applications employ them during file transfers or lengthy screen updates. **Figure 1** illustrates a typical progress indicator as employed by Borland's Database Desktop during a lengthy database query operation.



**Figure 1:** The Database Desktop's progress indicator showing a query with 88 percent to go.

Use of the hourglass mouse pointer has become a consistent method of setting user expectations, and is just as ubiquitous as the progress indicator. Although use of progress meters tends to be application specific, changing the mouse cursor to the hourglass pointer has literally been standardized by Microsoft. (See section 3.6.1.1 on Graphical Feedback in *The Windows Interface: An Application Design Guide* [Microsoft Press, 1992], which promotes both the progress indicator and the hourglass mouse pointer as recommended techniques for preparing users for lengthy delays.)

#### Progress Indicators

Although Delphi component vendors have been flourishing since Delphi's introduction, and many are offering some value-added meter components, Delphi ships with everything you need to build acceptable progress indicators for your applications. You very well may find the *TGauge* component on the Samples page contains all the functionality you require for many applications.

To build an example progress indicator, start a new project. On the default form, drop the following components:

- A *TButton* from the Standard page,
- a *TTimer* from the System page, and
- a *TGauge* from the Samples page (see **Figure 2**).

Using the Object Inspector, set the *Enabled* property for *Timer1* to *False* and the



Interval property to 100 (see Figure 3). Double-click the *Timer1* component and add the following code to the *OnTimer* event procedure:

```
Gauge1.Progress :=
Gauge1.Progress + 1;
if Gauge1.Progress >= 100
then
  Timer1.Enabled := False;
```

Every tenth of a second the progress meter will advance by one percent. In 10 seconds the meter will have advanced to 100 percent and will turn off the timer.

First, however, we need a way to enable the timer. Double-click the *Button1* component and add the following code to the button's *OnClick* event:

```
Gauge1.Progress := 0;
Timer1.Enabled := True;
```

In this way, when the button is clicked, the timer starts ticking off tenths of seconds for a period of 10 seconds. Your unit should match Figure 4. Run the program and click the button to display your sample progress indicator as shown in Figure 5.

### Using Meters in Production Programs

Although the simple progress meter example in Figure 5 makes use of a *Timer* component, timers are not generally required for most programs. You will typically want to embed progress meter updates inside the traditional *for..do*, *do..while*, and *repeat..until* loop constructs that contain your processing logic. You can use a timer when the processing logic you are encapsulating behind the progress meter involves multiple loops, each of which has somewhat irregular processing periods. Although this use of timers often makes users wait longer than the processing will require, it does provide a consistent delay. It is best to profile your program and place code to update the meters appropriately. Note that timers tend to be scarce Windows resources; careful programmers use them sparingly.

In a production program you would probably want to use the *TGauge* component with a *TPanel* to give your progress meter a chiseled, 3-D effect. Since progress indicators are application specific, you may want to experiment with the *TGauge's Kind* property to determine if dials or vertical meters are more appropriate for the situation.

If you find you need a fancier meter, you can add functionality to *TGauge* through the miraculous powers of inheritance, or you could use the source code for the *TGauge*

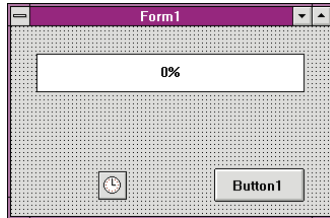


Figure 2: The sample form at design time.

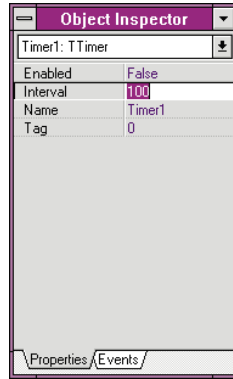


Figure 3: The *Timer1* object is shown in the Object Inspector. Set its properties as shown here.

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    Timer1: TTimer;
    Gauge1: TGauge;
  procedure Timer1Timer(Sender: TObject);
  procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Gauge1.Progress := Gauge1.Progress + 1;
  if Gauge1.Progress = 100 then
    Timer1.Enabled := False;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Timer1.Enabled := True;
end;

end.
```

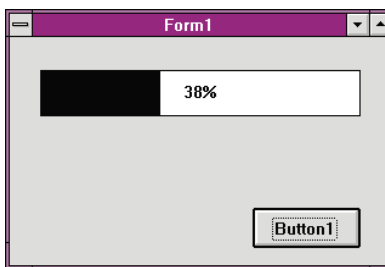


Figure 4 (Top): The code for your *Unit1*.

Figure 5 (Left): The sample progress indicator at run time.

component (source for the Samples page components ships with Delphi) as a model and build your own. If you would rather buy than build, you can always purchase third-party meter components like the Visual Progress component from Shoreline Software, the *TOvcCustomMeter* component in Turbo Power's Orpheus suite, or the Gauge control from Borland's RAD Pack for Delphi, just to name a few.

Before we leave the subject, some cautions regarding progress indicators are in order. The quality and appropriateness of a progress indicator is judged by its accuracy. If your meter says that you are 50 percent complete, users would like to be able to rely on it. Avoid jumpy and discontinuous meter updates; don't race out to 90 percent complete in the first 10 seconds, only to spend five minutes completing the last 10 percent.

## Cursors for Everyone

All controls in the Visual Component Library that ships with Delphi contain a *Cursor* property that specifies which pointer to display when the mouse is moved over the control. If you search the online help under the topic of “mouse pointer” and choose the topic “Cursor Property” you will be directed to two important help resources for changing the mouse cursor: “Cursor Property for All Controls” and “Cursor Property for Screen Objects.” The first topic provides examples of each of the predefined system mouse pointers, and is not only helpful for lengthy delay processing, but for remembering the constants used for selecting any of the predefined system pointers that you may need for other applications.

The second topic provides a simple `try..except` block example of changing the Screen object’s cursor. Delphi applications have a predefined *TScreen* object that encapsulates the screen. Changing the Screen’s cursor amends it for the application. Changing the *Cursor* property of a component simply defines the cursor that will appear when the mouse is over that component. In this case we want the change to be system-wide, so we use the Screen’s *Cursor* property. (Note that the initial Delphi release does not document the *crMultiDrag* cursor [value 16] in its online help. You’ll likely find it useful for drag-and-drop applications.)

## Changing the Mouse Pointer

As an example of how to change the mouse cursor to the hourglass pointer during a processing delay, start a new project with an empty form. As in our first example, place a *TButton* component from the Standard page and a *TTimer* component from the System page on your form. Using the Object Inspector, set the *Interval* property to 5000 (five seconds) and the *Enabled* property to *False* for *Timer1*. To *Button1*’s *OnClick* event, add:

```
Timer1.Enabled := True;
Screen.Cursor := crHourGlass;
```

and to *Timer1*’s *OnTimer* event add:

```
Timer1.Enabled := False;
Screen.Cursor := crDefault;
```

Now, run the application and click the mouse to start the timer. After five seconds the mouse pointer will revert back to the default pointer. Similar to our previous program, the button’s *OnClick* procedure changes the Screen cursor to the hourglass and starts the timer. Conversely, after five seconds, the timer’s *OnTimer* procedure changes the cursor back and disables the timer.

Although this example illustrates how to change the mouse cursor, we will next explore how this example lacks robustness and fails to provide the behavior many users expect.

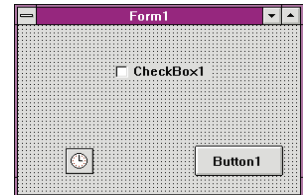
## Wild Mouse Runs Amok

Our new mouse-changing program has a flaw; it employs an

untamed mouse. What you might not have noticed is that the program permits users to continue queuing and processing Windows events. Although the mouse indicates that the program is busy (the conventional meaning of the hourglass), the program actually remains idle, i.e. it will eagerly continue processing keyboard and mouse events.

To illustrate what a wild mouse means to your users, drop a *TCheckBox* component from the Standard page on the form you just created, as shown in Figure 6. Now run the program and click the check box “on” and “off” while the mouse pointer looks like an hourglass. What went wrong?

The program in Figure 6 provides a simple contrived example of how you can change the mouse pointer, yet continue to process keyboard and mouse events. Instead of this simple example, imagine a complex database or graphics application that changed the mouse pointer to the hourglass during a particularly lengthy query or screen refresh. When the application completes the query or screen refresh, it must then run off and service each additional mouse click the user has queued for the program. If you’ve ever used a program like this, then you’ve visited the land of pure frustration. You spend most of your time waiting for the program to catch up.



**Figure 6:** Adding a *CheckBox* component to our sample application.

It’s plainly unacceptable. So, how do you tame the wild mouse?

## Taming of the Shrew

The problem with the wild mouse lies not in the method by which the mouse pointer is changed, but rather in how Windows events are processed after the pointer has changed shape. Changing the shape of the cursor in no way alters how Windows events are processed. Additional programming is required to change the way Windows events are processed by your application. Most of the time you will want to ignore both keyboard and mouse events while your application is off doing its thing. When your application completes its processing, you will then want to restore the mouse cursor and begin handling mouse events normally.

The proper way to change the mouse pointer thus becomes:

- 1) Change the mouse cursor to the hourglass pointer.
- 2) Begin filtering of mouse and keyboard events from the Windows event queue.
- 3) Begin the lengthy process.
- 4) End the lengthy process.
- 5) Remove the mouse and keyboard event filters from the Windows event queue.
- 6) Restore the mouse cursor to the previous pointer.

Luckily for us, the Delphi designers left hooks in the *TApplication* event model to make filtering of raw Windows events easy.

Before we examine the technique, however, it may be prudent to understand — at least at an abstract level — how a Delphi program receives Windows events. What’s really going on in a Delphi application?

### Fable of the Great Loop

Pretty early on one gets the idea that events are critical to a Windows program. As we know from our two example programs, Delphi makes responding to various Windows events a simple task. Windows programming — for those of us that have had to dip in to pure Pascal or C — can often be much less trivial. Why is Delphi so much easier?

The very first Windows programs were huge loops that contained nested case statements. Applications would constantly (when they received focus) request Windows events in a big loop, looking for events of interest. Windows would periodically detect a user action (like a mouse movement) and drop an event on the queue. The application would then filter this event through a large sieve of case statements, finally performing something nifty or simply ignoring the event.

Through the use of encapsulation and delegation — both key notions to object-oriented programming — we have since moved beyond the Great Loop.

### How Delphi Encapsulates Events

Delphi, through the *TApplication* object that is built into each program, encapsulates the Great Loop as depicted in Figure 7.

Windows constantly watches for user or machine activity. When an event occurs it is placed in an event queue. These events are then queued for the appropriate application by virtue of each Windows message being “tagged” by what is termed a handle. Beneath the covers of the *TApplication* object, your application sits in a tight loop. It gets a message and processes the message. The *TApplication* object encapsulates all the default behavior, so all we have to do is respond to exceptional events.

Although Figure 7’s explanation of Delphi’s event model is simplistic, it will serve our needs for the next section. Please refer to Delphi’s online help for *TApplication*, and in particular, the *ProcessMessages* procedure and the *OnMessage* event for additional information on how the event model operates. If you purchased the source code to the Visual Component Library, you may want to trace through some of the *TApplication* and *TForm* code in a simple application to examine the Delphi event model in detail.

### Keyboard and Mouse Events

One thing you’ll notice when you begin working with raw Windows events in Delphi is that *TApplication* event handlers are created differently than event handlers you might create for other components. The main difference is that, because the Object Inspector cannot be used to generate code, you must create your own procedure and map your own event handling procedure into the hook provided by Delphi’s *TApplication* object. See the topic “Handling Application

Events” in Delphi’s online documentation for a detailed discussion and examples of these event handling differences.

For our purposes, we will need to define a procedure that will “de-queue” and ignore all keyboard and mouse events while our application is involved in lengthy processing. So every time Windows places a message on our application queue, we will examine the message and remove it if it’s a keyboard or mouse event, and leave it alone if it is any other kind of event. We will be using a *Boolean* flag to determine if our application is involved in lengthy processing.

### Create the Handler

We will be extending the example program in Figure 6 to properly ignore keyboard and mouse events, so our first step is to create the *OnMessage* event handler for our *TApplication* object. *OnMessage* event handlers, defined as *TMessageEvents*, are procedures that receive two arguments. The first is an encapsulated Windows message, and the second is a *Boolean* flag that the procedure uses to indicate whether additional handling is required. Anticipating that we will require a *Boolean* flag to tell us when our application is off processing, we will also add the line:

```
HardAtWork : Boolean;
```

to the **private** section of the unit to define the flag. To define an *OnMessage* handler called *AppMsg*, we will also add the line:

```
procedure AppMsg(var Msg: TMsg; var Handled: Boolean);
```

to declare the procedure in the **private** section of the unit we defined in Figure 6.

Searching on *TMsg* in the Windows API section of Delphi’s online help provides the definition of the *TMsg* data structure. We will only need to know what kind of message Windows is sending to the application, so only the message field is of interest in this example. Referring back to our conceptual event model in Figure 7 reminds us that the *GetMessage* Windows API function is used to retrieve Windows messages from an application’s event queue. Searching the online help for the *GetMessage* Windows API provides all the information that we require to perform our message filtering.

The *GetMessage* function retrieves messages for a specific window (using the Window’s handle), and can be used to filter specific types of messages. The *MsgFilterMin* and *MsgFilterMax* arguments specifically perform this service



Figure 7: The Great Loop.

when non-zero. The most important phrase for our example will be found in the comments section of the *GetMessage* documentation. The constant pairs WM\_KEYFIRST and WM\_KEYLAST and WM\_MOUSEFIRST and WM\_MOUSELAST delineate the upper and lower boundaries of keyboard and mouse events respectively. Using these constants greatly simplifies our filtering procedure. Note: the “Handling Messages” chapter (Chapter 7) of the *Delphi Component Writer’s Guide* also includes an excellent discussion of trapping Window messages.

Armed with *GetMessage*, complete the *AppMsg* procedure definition as shown in Figure 8. The next step is to establish *AppMsg* as *TApplication’s OnMessage* handler by adding the following line to the form’s *OnCreate* event:

```
Application.OnMessage := AppMsg;
```

We have now established a custom event handling procedure to call when we receive Windows messages, as well as a filtering mechanism for keyboard and mouse events to use when our application is processing.

### One Well-Behaved Mouse

The final step in our program is to notify our message filtering procedure when to filter and when not to filter messages from Windows. By adding the line:

```
HardAtWork := True;
```

to the *OnClick* procedure from *Button1* and:

```
HardAtWork := False;
```

to the *OnCreate* and *OnTimer* procedures for our Form and Timer respectively, we have tamed the mouse. Re-run the program. Start the timer by pressing the button, then attempt to change the check box while the timer is ticking. Voilà! You have tamed the shrew.

### Conclusion

Setting and meeting user expectations is a basic principle in good application design. This article has demonstrated the two most popular techniques used in commercial applications to keep users patient during lengthy processing delays. You should find the progress indicator and the hourglass cursor two indispensable tools for keeping your users happy.

A bonus program that includes both a progress indicator and both techniques for altering the mouse cursor discussed in this article may be found in the file BONUS.ZIP. The bonus program also includes an alternative technique for changing the cursor, and an occasionally useful (in specialized applications) method of freezing the mouse cursor (and all other hardware events) during processing. May your mice never run wild. ▲

*The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\JULADI9607JC.*

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    Timer1: TTimer;
    CheckBox1: TCheckBox;
    procedure Button1Click(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    HardAtWork: Boolean;
    procedure AppMsg(var Msg: TMsg;
                     var Handled: Boolean);
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
begin
  Timer1.Enabled := True;
  Screen.Cursor := crHourGlass;
  HardAtWork := True;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Timer1.Enabled := False;
  Screen.Cursor := crDefault;
  HardAtWork := False;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  { Turn off special processing }
  bHardAtWork := False;
  { Next, assign our custom event handler }
  bApplication.OnMessage := AppMsg;
end;

procedure TForm1.AppMsg(var Msg: TMsg; var Handled: Boolean);
begin
  { Perform default handling }
  Handled := False;
  if HardAtWork then
    if ((Msg.message >= WM_KEYFIRST) and
        (Msg.message <= WM_KEYLAST)) or
        ((Msg.message >= WM_MOUSEFIRST) and
         (Msg.message <= WM_MOUSELAST)) then
      { Ignore the event completely }
      Handled := True;
end;

end.
```

Figure 8: Use this code to complete the *AppMsg* procedure.

Jim Callan, an 18-year computing veteran and former consulting director for Oracle Corporation, is currently president of Gordian Solutions, Inc., an information technology consulting provider in Cary, NC. A frequent writer and speaker on information technology and client/server computing, Jim specializes in product design. He can be reached at (919) 460-0555 or by e-mail at 102533.2247@compuserve.com.







## DBNAVIGATOR

Delphi 2 / Object Pascal



By Cary Jensen, Ph.D.

# Once Is Not Enough

## Enhanced Multi-Record Views in Delphi 2

**M**any database applications require that you simultaneously display two or more records from a single table. In Delphi 1, the DBGrid provides you with a simple component for displaying multiple records, but Delphi 2 improves your options.

First, the DBGrid in Delphi 2 is more sophisticated than that of Delphi 1. Further, Delphi 2 adds a new multi-record component — the DBCtrlGrid. This installment of DBNavigator shows you how to leverage these enhanced features.

There are three basic areas where the DBGrid component has been improved. These include greater control over the visual appearance of the DBGrid using the Columns Editor, the addition of new event properties, and the addition of a new value for the *Options* property that permits users to select multiple records in the DBGrid. Each of these are described separately in this article.

### The Columns Editor

Using the Columns Editor dialog box (see Figure 1), you can easily control the visual appearance of a DBGrid. After you have associated the DBGrid with a DataSource that points to an active DataSet, you can display the Columns Editor by double-clicking the DBGrid component, or by right-clicking it and selecting **Columns Editor** from the displayed SpeedMenu.

The Columns Editor is blank when you initially bring it up. Consequently, it is necessary to first add to the Columns list the fields you want to display in the DBGrid. (This needs to be done whether or not you have instantiated fields using the Fields Editor.) You do this by clicking the **Add All Fields** button on the Columns Editor dialog box. This will create one column entry for each field in the corresponding DataSet.

Once you have added all fields, you can selectively remove one or more of the listed columns. You do this when you want to display less than all of the DataSet fields in the DBGrid. (This can be done in both Delphi 1 and Delphi 2 by instantiating Field components for the DataSet, and then setting the individual Field components' *Visible* property to *False*.) You can also control the order that the fields appear in the DBGrid by dragging the various column names to new positions in the Columns list. The column that appears at

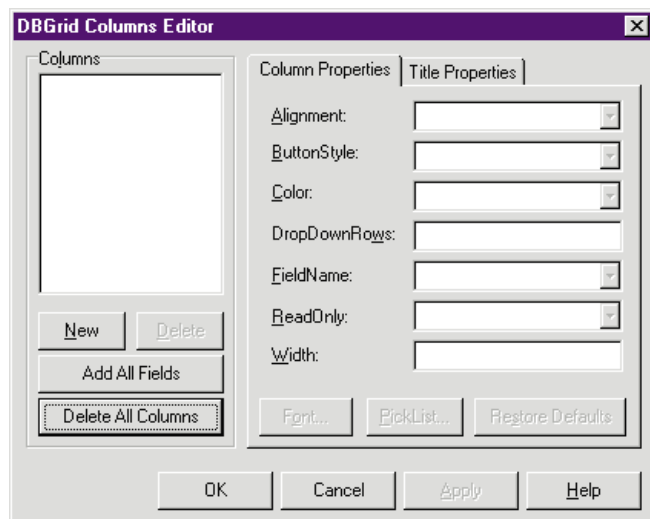


Figure 1: The Columns Editor is a new feature of Delphi 2.



<i>Alignment</i>	Alignment of text in cell
<i>ButtonStyle</i>	Type of button to display when editing cell
<i>Color</i>	Default color of cell background
<i>DropDownRows</i>	Number of rows to display in drop-down list
<i>FieldName</i>	Name of field for selected column
<i>ReadOnly</i>	Control whether field can be edited or not
<i>Width</i>	Width of cell, in pixels
<i>Font</i>	Font used to display cell contents
<i>PickList</i>	Use to create a drop-down list for a field

<i>Alignment</i>	Alignment of column caption
<i>Caption</i>	Text of column caption
<i>Color</i>	Color of column caption cell background
<i>Font</i>	Font for column caption

**Figure 2 (Top):** The fields of the Column Properties page of the Columns Editor. **Figure 3 (Bottom):** The fields of the Title Properties page of the Columns Editor.

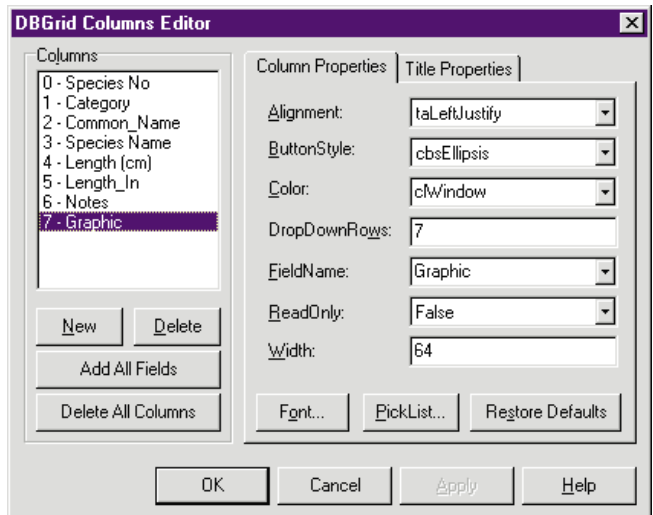
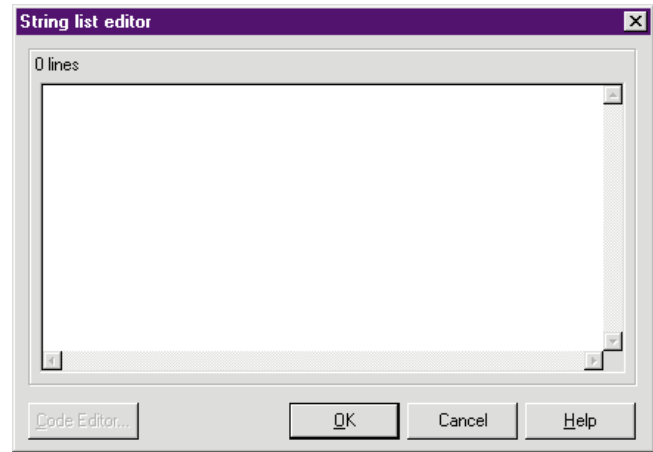
the top of the list will appear in the first column of the DBGrid, the column that appears in the second position will be the second column, and so forth.

After you have defined which columns the DBGrid will display, use the Tabbed notebook on the right side of the Columns Editor to customize the display of each column. Begin by selecting the column you want to control. Next, specify the characteristics you want for that column. The first tab, Column Properties, permits you to control how the field will look within the DBGrid, as well as how it will behave. The second tab, Title Properties, is used to control the appearance of the DBGrid column captions. **Figure 2** contains a list of the Column Property fields and their functions. **Figure 3** displays a similar list for the Title Properties tab.

One of the more interesting options on the Columns Editor dialog box is the **ButtonStyle** field. Using this field you can configure the DBGrid to display either a drop-down list or an ellipsis-style button (...) when the corresponding field is being edited in the DBGrid.

When you set *ButtonStyle* to *cbsAuto*, a drop-down button will be displayed if the corresponding field is a lookup field (defined using the Fields Editor), or if you have defined a picklist for that field. If you set *ButtonStyle* to *cbsEllipsis*, an ellipsis button — similar to the one used to signify a property editor dialog box in the Object Inspector — will be displayed when the corresponding field is being edited. If the ellipsis button is clicked, an event handler assigned to the new *OnEditButtonClick* event property will be executed, if defined. From within this event handler, you can determine which field is current, and program a custom response to the button click.

Creating the picklist that is displayed when the ellipsis button is clicked is easy. Begin by selecting the field for which you want to create the picklist from the **Columns** list. Next,



**Figure 4 (Top):** The String list editor is displayed when you click the **PickList** button on the Columns Editor dialog box. **Figure 5 (Bottom):** The **Graphic** field's column settings for the EDITBTNN.DPR project.

set the *ButtonStyle* to *cbsAuto*. Finally, click on the **PickList** button. Delphi will display a String list editor, as shown in **Figure 4**. Use the String list editor to enter the values that you want displayed in the drop-down list that will appear when the user clicks the drop-down arrow associated with the defined field.

If the values you want to appear in this list are already values that appear in another table, consider creating a lookup field as opposed to defining a picklist. [Creating a lookup field is described in Dr Jensen's article, "Elysian Fields", in the *May 1996 Delphi Informant*.] This way, your drop-down list will always be up-to-date.

Creating a custom editor that is accessed by clicking an ellipsis button is only slightly more involved. First, select the field for which you want to create the custom editor. Then set the *ButtonStyle* property to *cbsEllipsis*. Next, select the DBGrid in the Object Inspector and display the Events page. Double-click the *OnEditButtonClick* event property to create an event handler that will be executed when the ellipsis button is clicked. The code you add to this event handler should evaluate which field is selected, and provide a corresponding approach.

The use of a custom editor is demonstrated in the project EDITBTTN.DPR. Figure 5 shows the Columns Editor for a DBGrid that is associated with the BIOLIFE.DB table, a Paradox table stored in the directory pointed to by the DBDEMOS alias (this alias is created when you install Delphi). Notice in this figure that the *ButtonStyle* has been set to *cbsEllipsis*.

All this setting does is instruct Delphi to display an ellipsis next to the Graphic field when that field is being edited. In order for something to happen when this button is clicked, it is also necessary to add an event handler to the *OnEditButtonClick* event property of the DBGrid. The following is the code used for this event handler in the EDITBTTN.DPR project:

```
procedure TForm1.DBGrid1EditButtonClick(Sender: TObject);
begin
  if DBGrid1.SelectedField.Name = 'Table1Graphic' then
    Form3.ShowModal;
end;
```

This event handler uses the *SelectedField* property of the DBGrid to identify which field's ellipsis button was clicked. Although in this example project there is only one field with an ellipsis button, it is generally a good idea to test which field is selected, in case you decide to add ellipsis buttons to the DBGrid in the future. It is also important to note that in this case the Graphic field was instantiated using the Fields Editor. This was necessary to be able to refer to the *Name* property of the current field. If the current field had not been instantiated, the *Name* property would return a null string.

When the event handler determines that the Graphic field's ellipsis button was clicked, it displays a form named *Form3*. This form contains a single DBImage component associated with the Graphic field, as well as a Close button. Because *Form1* (the one on which the DBGrid is attached) and *Form3* share the same Data Module, *Form3* will display the bitmap stored in the Graphic field corresponding to the current record on *Form1*. Figure 6 shows how the screen might look when you click the ellipsis button for the second record in the BIOLIFE.DB table.

## New Event Properties

There are four new event properties for the DBGrid component in Delphi 2. The preceding section introduced one of these, the *OnEditButtonClick* event property. The three other event properties are *OnColumnMoved*, *OnDrawColumnCell*, and *OnStartDrag*.

*OnColumnMoved* is executed when the user changes column order at run time. Of course, this can only occur if the *Options* property contains the *dbColumnResize* value. The parameter list of the event handler you assign to the *OnColumnMoved* event property includes an integer that identifies which column is moved, as well as the position to which it was moved.

The *OnDrawColumnCell* event property provides you with an event handler from which you can control the painting of the cells in a DBGrid. This event property is an improved replacement for the *OnDrawDataCell* event

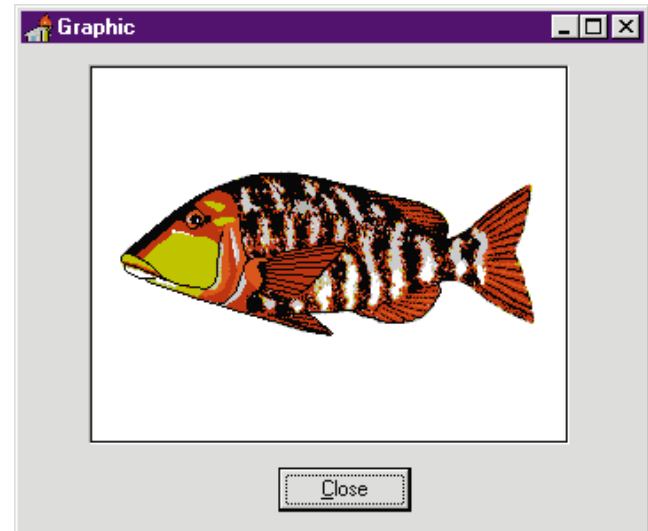


Figure 6: The form that displays the Graphic field's contents is shown modally when the ellipsis is clicked for the Graphic field in the DBGrid.

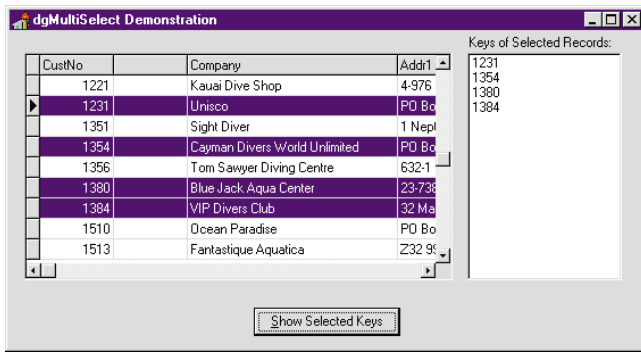
property. While the *OnDrawColumnCell* event handler parameter list includes a parameter that indicates which column is being repainted, the *OnDrawDataCell* event handler parameter list does not.

The final new event property, *OnStartDrag*, is an event property that is instantiated in the *TControl* object in Delphi 2, and inherited by all descendant objects. Use this event property to perform any necessary initialization of a drag-and-drop operation.

## Creating MultiSelect DBGrids

The Properties page of the Object Inspector of the DBGrid component includes only one enhancement in Delphi 2, but it's an important one. The *Options* set property includes one new value — *dgMultiSelect*. When this value exists in the *Options* set, the user can highlight more than one record within the DBGrid by holding down (Ctrl) while selecting records.

There is one critical property to use when working with DBGrids that permit multiple record selection. It is the *SelectedRows* property, which is of the type *TBookMarkList*. This is a poorly documented property, and until Borland ships an improved Help file for Delphi 2, you need to have the VCL source code to adequately work with it. (*TBookMarkList* is defined in the DBGrids unit.) However, there are two essential properties of the *TBookMarkList* class that you will need: *Count* and *Items*. *Count* is an integer that identifies how many records have been selected, and *Items* is an array of *TBookMarkStr* objects that can be cast as *TBookMark* objects and then be used to locate a selected record. Other notable properties of *SelectedRows* are *CurrentRowSelected* (a Boolean property that reports whether the current record is one of the selected records), *Delete* (remove an item from the *TBookMarkList*), *Clear* (remove all items from the *TBookMarkList*), and *Refresh* (remove any items from the *TBookMarkList* associated with records that have been deleted in the time since they were selected).



**Figure 7:** The MultSel project demonstrates the use of the `SelectedRows` property of the `DBGrid`. Using this property it is possible to determine which records of a `DBGrid` are selected.

The use of the `SelectedRows` is demonstrated in the project `MultSel` shown in [Figure 7](#). The main form of the `MultSel` project contains a `DBGrid` that includes the `dgMultiSelect` value in the `Options` set. When the `Show Selected Keys` button is clicked, the `Listbox` labeled `Keys of Selected Records` is populated with the `CustNo` value of each selected record. [Figure 8](#) is the code associated with the `OnClick` event handler for the `Show Selected Keys` button.

(Note: The author wishes to thank Martin Rudy of Para/Matrix Solutions, Inc., for his work in deciphering the `SelectedRows` property. The preceding example is based in part on an example presented by Martin in the book *Delphi In Depth* [Osborne/McGraw-Hill, 1996].)

## Overview of the `DBCtrlGrid`

In addition to the `DBGrid`, Delphi 2 also includes the `DBCtrlGrid` for displaying two or more records simultaneously. The `DBCtrlGrid` is a fairly simple container object that automatically replicates itself at run time. Any objects that have been placed in the main panel of the `DBCtrlGrid` at design time will automatically be displayed in every panel of the grid at run time. An example of the `DBCtrlGrid` component at design time is shown in [Figure 9](#).

One reason the `DBCtrlGrid` is so simple is because it's very limited in terms of the types of objects it can contain. Here's a list of the components that can appear in a `DBCtrlGrid`:

- `DBCkCheckBox`
- `DBCkComboBox`
- `DBEdit`
- `DBLookupComboBox`
- `DBText`
- `GroupBox`
- `Label`
- `Panel`

Other characteristics that make the `DBCtrlGrid` so simple are its properties. There are only five basic properties you need to adjust in most instances. These are the `DataSource` property (to associate the `DBCtrlGrid` with a `DataSource`), the `ColCount` and `RowCount` properties (to control how many rows and columns to display), and the `Height` proper-

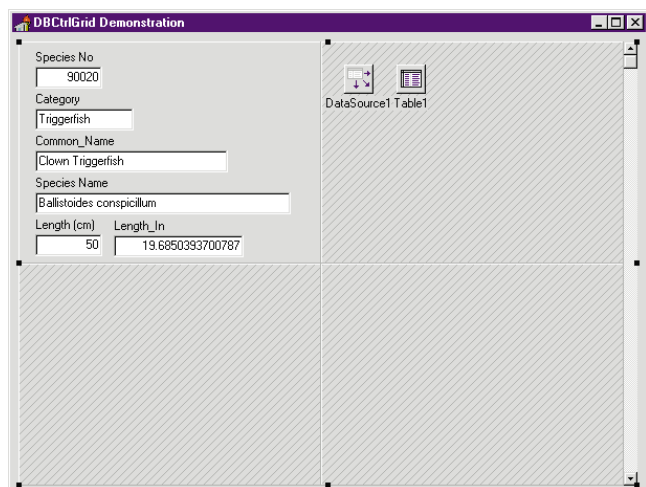
```

procedure TForm1.Button1Click(Sender: TObject);
var
    CurrentRecord : TBookmark;
    i : Integer;
begin
    // Ensure that the SelectedRows property is up to date.
    DBGrid1.SelectedRows.Refresh;
    // Do not continue as no records are selected.
    if DBGrid1.SelectedRows.Count = 0 then
        Exit;

    // If the current record has not been posted,
    // attempt to post it.
    if Table1.State in [dsEdit, dsInsert] then
        // If the following statement raises an exception,
        // the remainder of this event handler will not execute.
        Table1.Post;

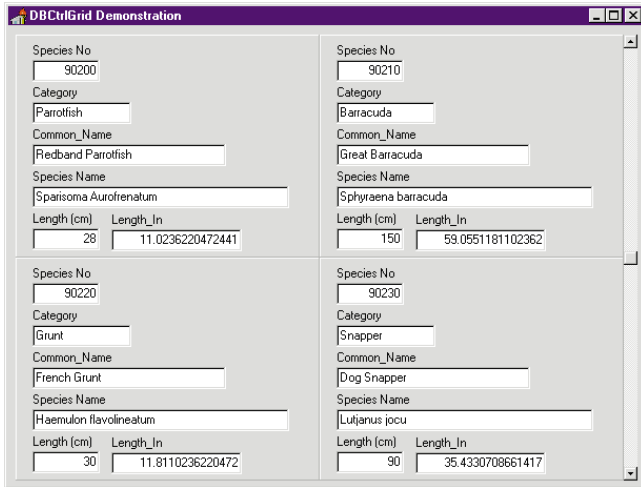
    // Store a bookmark for the current record.
    CurrentRecord := Table1.GetBookmark;
    Table1.DisableControls;
    try
        // Clear any current contents of the ListBox.
        ListBox1.Clear;
        // For each of the selected records.
        for i := 0 to DBGrid1.SelectedRows.Count - 1 do
            begin
                // Move to the record.
                Table1.GotoBookmark(TBookmark(
                    DBGrid1.SelectedRows.Items[i]));
                // Add the record's key to the ListBox.
                ListBox1.Items.Add(
                    Table1.FieldName('CustNo').AsString);
            end;
        finally
            // All done. Move back to the original current record.
            Table1.GotoBookmark(CurrentRecord);
            // Restore the DataSet.
            Table1.EnableControls;
            // Free the temporary bookmark.
            Table1.FreeBookmark(CurrentRecord);
        end;
    end;

```



**Figure 8 (Top):** The `OnClick` event handler for the `Show Selected Keys` button. **Figure 9 (Bottom):** A `DBCtrlGrid` at design time.

ty (which in conjunction with the `ColCount` and `RowCount` properties controls the size of the individual panels). The final property, `Orientation`, controls whether records are displayed in “newspaper” style (the second record will appear in the second row), or in a left-to-right orientation (the second record will appear in the second column).



**Figure 10:** Components placed into the primary panel of a DBCtrlGrid are replicated at run time.

Figure 10 depicts how the DBCtrlGrid shown in Figure 9 looks at run time. This DBCtrlGrid has its *ColCount* and *RowCount* properties set to 2. Notice that the fields placed in the main panel are repeated for each of the four panels.

However, the data displayed in these panels is associated with different records of the corresponding DataSet. This form is associated with the project GRIDDEMO.DPR.

### Conclusion

Delphi 2 gives you more control than ever before over the display of multiple records. The enhancements to the DBGrid component provide your applications with new and interesting ways to display data to your users. Likewise, the DBCtrlGrid component, although limited, offers additional flexibility in the display of data.  $\Delta$

*The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\JUL\DI9607CJ.*

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including *Delphi In Depth* [Osborne/McGraw-Hill, 1996]. He is also Contributing Editor of *Paradox Informant* and *Delphi Informant*, and is this year's Chairperson of the Paradox Advisory Board for the upcoming Borland Developers Conference. You can reach Jensen Data Systems at (713) 359-3311, or via CompuServe at 76307,1533.





## THE API CALLS

Delphi 1 / Object Pascal / Windows 3.x Stress DLL



By *Karl Thompson*

# Gimme Some Stress!

## Put Your Delphi App to the Test with the Windows Stress DLL

**O**K. You've just finished writing a major application on your new Pentium-166 PC with 32MB of RAM. Everything works fine — and boy is it fast! You're ready for commercial release, but you have some nagging questions: How will it run on more modest computers? or, How will it run on even a powerful machine when Windows' resources are stretched?

How do you test an application under such conditions?

### Enter Stress

Windows 3.x installs STRESS.DLL into the \Windows\System directory. STRESS.DLL contains functions that allow the developer to easily limit various system resources. These resources include the number of file handles, the amount of drive space, and Global, User, and GDI memory pools.

The Delphi 1 project that accompanies this article, Stress, is an interface to the Stress unit that ships with Delphi 1. The Stress unit is itself an interface to the STRESS.DLL API.

While Stress can be run as a stand-alone application, you'll only get limited functionality if you use it that way. Its greatest functionality comes when STRESSME.PAS is incorporated as a unit into a debug version of your Delphi 1 application. (More about this later.)

### Stress Functions

STRESS.DLL exports five functions that can be used to allocate system resources:

- **AllocDiskSpace** creates a file, STRESS.EAT, that eats up (if you will) disk space on a partition until only the amount of space that is passed as a parameter is left on that partition. **AllocDiskSpace** will create STRESS.EAT on one of three

partitions: the partition that Windows is installed on, the one that has a TEMP directory that Windows uses for storing temporary files, or the current partition.

- **AllocFileHandles** will limit the number of file handles available to the current instance of the program, up to a maximum of 256. Whenever a file is opened, a file handle is consumed. Therefore, you may want to test your application with only a small number of handles available. Try 40 handles or so to start.
- **AllocGDIMem** will limit the amount of GDI memory that is available to your application. Every programmer is familiar with the number that is reported in the Window's About Program Manager box as Free System Resources. There are two 64K memory segments, GDI memory and User memory, that Windows manages for the storing of resources.

The segment that has the least amount of free memory is used to report this System Resource percentage. If the GDI segment has 32K free and the User segment has 40K free, Windows will report 50 percent free System Resources. To test your application in a low resource environment, use **AllocGDIMem** and **AllocUserMem** to reduce the amount of available memory.



## THE API CALLS

- **AllocUserMem** limits the amount of memory available on the User Heap.
- **AllocMem** limits the amount of memory available on the Global Memory Heap. Global memory is the entire memory area that Windows manages, including virtual memory. This function can be used to limit the memory available to make a developer's 32MB system look as if it is a 4MB system.

In addition to these, STRESS.DLL also exports complementary functions that free the consumed resources. [For a more in-depth discussion of Windows resources, see Karl Thompson's article "A Walk on the Wild Side" in the April 1996 *Delphi Informant*.]

In all cases, a subsequent call to any of these functions will automatically de-allocate the appropriate resource allocated by a previous call. For example, if **AllocFileHandles** is called with a parameter of 40, and again with 50, the allocation of 40 file handles is de-allocated; then the allocation of 50 file handles is made. For further information on all of these functions, see the Windows API Help file that ships with Delphi.

### Using the Stress Program

If you want to use up some disk space or reduce the size of the memory pools, you may run PSTRESS.EXE as you would any other Windows program. However, if you want to reduce the number of file handles that are available to your application, you must include *StressMe* in your program's uses statement. You'll then want to put a button or menu choice on one of your program's forms that loads the *StressMe* form.

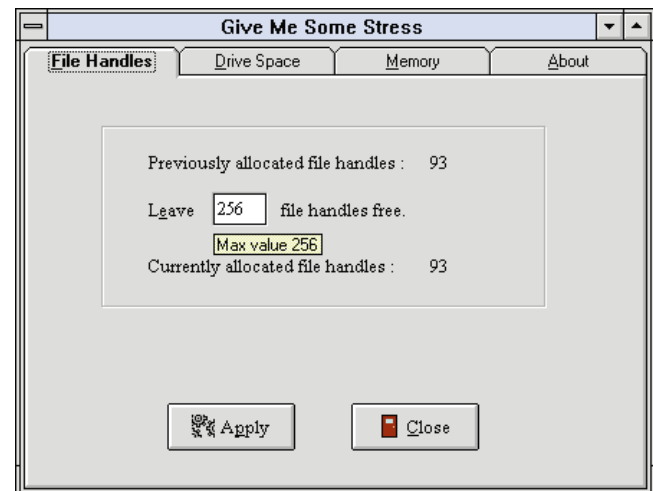
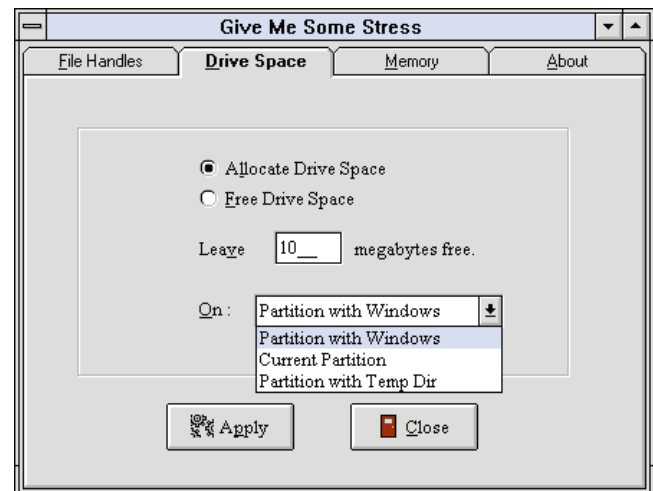
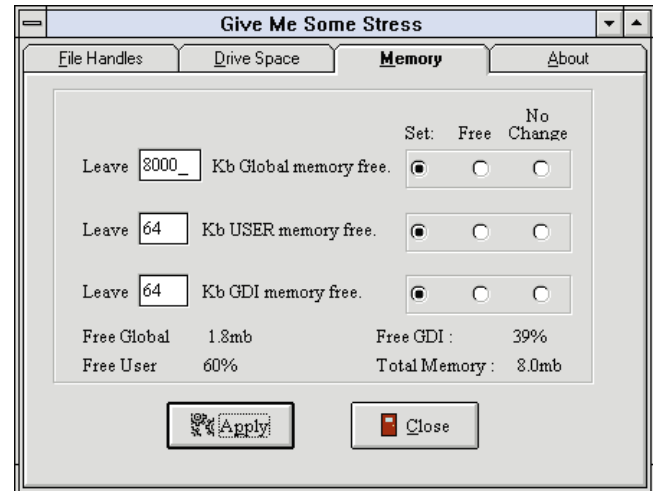
A small project file, *tStress*, has been included with this article that shows how to call the *StressMe* unit from an application. Note that the form is not created automatically (see from Delphi's menu: **Options | Projects**) and that it is explicitly freed after it is shown. By explicitly creating and freeing the form, the *StressMe* unit does not affect the way your application runs. You may want to go one step further by using a conditional define that will exclude the call to *StressMe* when the define is not declared. That way you can easily switch back and forth between a debug build and a production build.

Using Stress is straightforward. Select a tab, set a value, and then click on **Apply**. For example, if you want to limit your global memory to 8MB, click on the Memory tab or use the M accelerator key and fill in the edit box on the first row with 8000 (see **Figure 1**). After you click on the **Apply** button, the **Total Memory** should read 8MB and the **Free Global** will be something less than this amount. **Figures 2** and **3** are the other pages of the Stress program.

### Conclusion

I hope we've added some good stress to your life this month. Use the project in good health! ▲

*The demonstration Stress project referenced in this article is available on the Delphi Informant Works CD located in INFORM\96\JUL\DI9607KT.*



**Figure 1 (Top):** Using the Stress program to limit global memory to 8MB. **Figure 2 (Middle):** The Drive Space page allows you to limit the amount of disk space. **Figure 3 (Bottom):** The File Handles page.

Karl Thompson is a Delphi/Paradox/SQL developer serving clients from New York City to Philadelphia. He has been programming with Borland's Pascal language since 1984. He can be reached via the Internet at 72366.306@compuserve.com, phone at (609) 730-1430, or fax at (609) 730-1530.





## NEW & USED

By Douglas Horn

# ImageLib Portfolio 3.1

## Extras Enhance Already Worthwhile Tool

**S**urprise! ImageLib Portfolio 3.1 by SkyLine Tools, Inc. is full of interesting extras that customers probably don't realize they're getting. There's no question this library of graphics and multimedia components is worth its price based on its advertised features, but the little things Delphi developers will discover in using ImageLib make the product even more difficult to resist.

ImageLib Portfolio is available as either a Windows 3.1 16-bit or Windows 95 32-bit VCL. Each contains components for displaying, modifying, and converting graphics and multimedia files. These data-aware components handle nearly any display or conversion tasks a user could imagine. ImageLib supports .BMP, .GIF, .ICO, .JPG, .PCX, .PNG, .TIF, and .WMF graphics, and .AVI, .MID, .MOV, .RMI, and .WAV multimedia files. In addition, the tool supports multiple color depths and compression levels for the various graphics formats. Conversion is as simple as loading a file in one format and saving it in another. (.ICO and .WMF formats are read-only.)

The first extras that ImageLib users may notice are two unfamiliar file formats — .CMS and .SCM. These formats — and their editors that are included in ImageLib's cornerstone components — allow developers to incorporate vertical or horizontal scrolling text messages perfect for multimedia presentations or program credit screens.

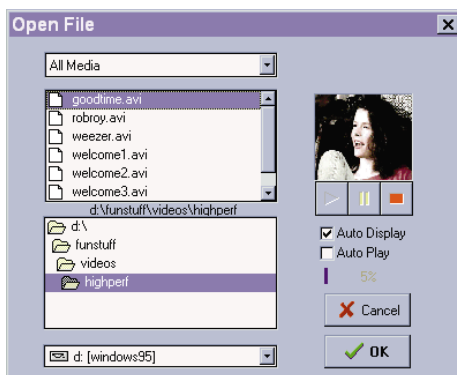
ImageLib is primarily a data-aware graphics viewing and conversion tool. ImageLib version 3.0 contained only four components to version

3.1/95's 16. For the most part, the 12 new components are thoughtful extras that make development faster and easier, rather than adding radically new features.

ImageLib has four closely related core components: *TPMultiImage*, *TPDBMultiImage*, *TPMultiMedia*, and *TPDBMultiMedia*. *TPMultiImage* is capable of displaying any of the image formats listed earlier. The component can also save files of one format to any compatible format. (.JPG and .CMS files, for example, would be incompatible because one stores images, while the other stores scrolling text.) *TPMultiMedia* is identical to *TPMultiImage*, except that in addition to image files, it plays multimedia files such as .AVI, .MOV, .MID, and .WAV. Both components are descendants of Delphi's *TImage* component.

As users would expect, the components support image zooming, rotation, printing, and Clipboard functions. As descendants of *TImage*, the components also support Delphi Canvas functions that developers can use to modify images. Many users will be surprised to find they can also capture still images from multimedia files, add formatted, rotated captions on images, and scan images from TWAIN-compliant scanners.

*TPDBMultiImage* and *TPDBMultiMedia* are data-aware versions of *TPMultiImage* and



**Figure 1:** ImageLib's multimedia open dialog box allows users to preview files before opening them.

*TPMultiMedia*. The main difference between the two sets of components is that the former work with images and multimedia in distinct files, while the latter, data-aware versions, work with images and multimedia in BLOB (Binary Large Object) database files. ImageLib's data-aware controls support Paradox and dBASE tables.

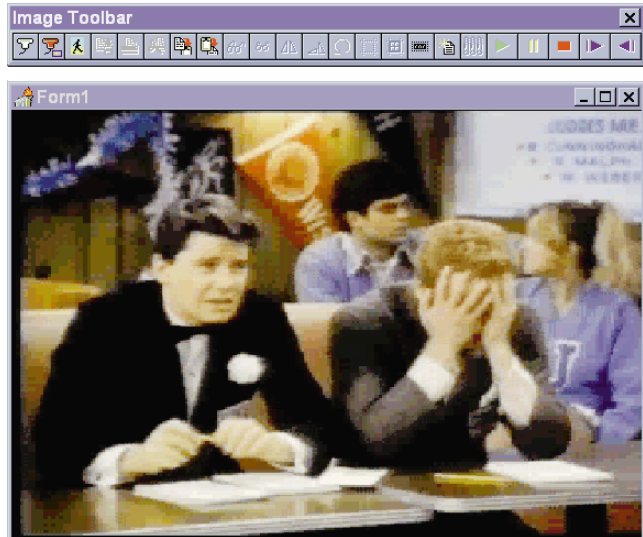
The components *TMMediaPlayer* and *TPDBMediaPlayer* control *TPMultiMedia* and *TPDBMultiMedia*. Both are little-changed descendants of Delphi's own *TMediaPlayer*. The main difference is that the custom components sense installed multimedia components and disable options that aren't supported by the computer system. For example, on a system without QuickTime for Windows installed, the custom media players would disable .MOV files, while Delphi's *TMediaPlayer* component would allow users to select the files (unless excluded programmatically), thus causing an error.

*TMMOpenDialog* and *TMMSaveDialog* are open and save dialog boxes for image and multimedia files (see Figure 1). These dialog boxes include the functions of Windows common dialog boxes (though unfortunately, they look nothing like common dialog boxes), with the ability to preview images, videos, or sounds prior to opening them. Users can enable or disable both automatic display of image files and automatic play of multimedia files. The *TMMSaveDialog* allows users to view an existing file before saving a file to the same name. This feature should help prevent users from accidentally overwriting files. And since their Delphi source code is included, they're useful not only as ready-made components, but also as learning guides for incorporating the VCL's core components into applications.

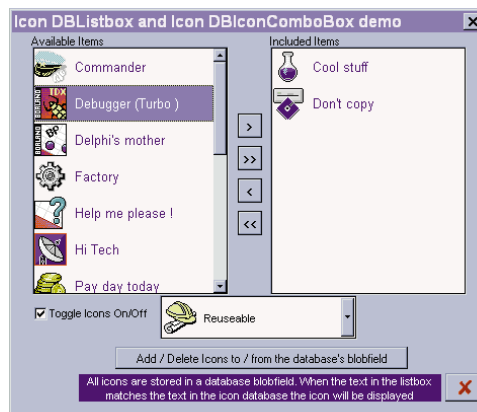
Another aid to opening files is ImageLib's *TThumbPreview* component. *TThumbPreview* creates and displays thumbnail views of images in a directory, allowing users to quickly see a catalog of the available images, rather than having to move through images one at a time. By clicking on an image a user can open the full-sized version in the *TPMultiImage*.

To make development faster and easier, the VCL also includes a toolbar corresponding to each of the display components (see Figure 2). The toolbars contain numerous buttons covering the major functions of each display component. Each includes scan, open, save, copy to Clipboard, and other familiar functions. Toolbars for data-aware components add the data navigation buttons of Delphi's *TDBNavigator*, and toolbars for multimedia components add the buttons of the *TMediaPlayer* component. Like other Delphi toolbars, ImageLib's toolbars' buttons can be displayed or concealed as the developer or user chooses.

The *TDBIconListBox*, *TDBIconComboBox*, and *TDBIconEditor* components allow developers to incorporate icon images into Delphi list boxes and combo boxes from a table's BLOB field (see Figure 3). *TDBIconEditor* allows the BLOB field icons to be edited.



**Figure 2:** The prefabricated toolbar components control ImageLib's components, such as this multimedia file.



**Figure 3:** ImageLib's icon list and combo box components allow developers to display graphics in lookup lists and combo boxes.

As a native Delphi VCL, ImageLib is compatible with Delphi and Delphi applications. The only program error this reviewer encountered was an occasional "Divide by zero" error while trying to load .GIF images. However, turning off the "break on exception" parameter via Delphi's **Options | Environment** menu command corrected the problem.

While ImageLib is a Delphi VCL, the tool's core functionality resides in a DLL file, with the VCL acting as a wrapper to ease Delphi programming. While there are many reasons to prefer pure Delphi VCLs over VCL/DLL tools, at least SkyLine Tools documents all Pascal interface calls for those developers who choose to reference the DLL directly. This is just one sign of ImageLib's fine documentation. The printed, indexed manual is over 160 pages. It not only documents all the unique properties and procedures for each component, but also includes program snippets where necessary. The online Help file is a clone of the printed manual, and can be integrated into Delphi's online Help.

ImageLib includes code documentation, in addition to the printed manual and online help file. The program includes source code for each of the components, as well as the many sample programs. All program code is well documented with

program remarks. By including this amount of documentation, the folks at SkyLine Tools have made ImageLib almost immediately understandable, even to novice programmers.

One further example of the thought put into ImageLib is the fact that the message strings of the DLL are externalized to a resource file for easier translation and internationalization of applications built using ImageLib. This is just good programming practice, but it's done so rarely that it seems like a bonus. Complete internationalization of applications created with third-party tools is often nearly impossible because developers do not follow this rule. Hopefully in the future more will follow SkyLine Tools' example.

This is not a comparative review. However, as *Delphi Informant* reviewed a similar product this year, LightLib Images by DFL Software [April 1996], this analysis would

**INFORMANT**  
**FACT FILE**

ImageLib Portfolio 3.1 is a Delphi VCL/DLL for incorporating image and multimedia files into Delphi applications, and is available in 16- and 32-bit formats. The development tools are notable for their extras such as sample toolbars, video frame capture, extensive sample code, and documentation. Performance is excellent, as is programmability. Source code is available, and distribution of compiled applications is royalty-free.

**SkyLine Tools, Inc.**  
11956 Riverside Drive, Suite 206  
North Hollywood, CA 91607  
**Phone:** (800) 404-3832 or (818) 766-3900  
**Fax:** (818) 766-9027  
**E-Mail:** 72130.353@compuserve.com  
**Web Site:** [http://theclassifieds.com/skyline\\_tools](http://theclassifieds.com/skyline_tools)  
**Price:** 16-bit version US\$139; 32-bit Delphi 2 version US\$169; and 16- and 32-bit bundled US\$199.

be remiss if it did not reference some major differences between the two products. First, while both tools use BLOB files extensively, LightLib Images uses its own BLOB format which compresses images better than Paradox BLOBs. ImageLib, on the other hand, uses standard BLOBs that are compatible with existing (and future) applications.

When testing the time required to load .JPG, .TIF, and .BMP graphics, ImageLib was faster than LightLib Images on all but very large .BMP files where LightLib Images was marginally faster. On a Pentium 100MHz computer, ImageLib was able to load a 73KB .JPG file from an IDE drive in one second, while LightLib Images took five seconds to load the same file.

For ImageLib's price, most users would be satisfied with the core *TPMultiMedia* and *TPDBMultiMedia* components alone. However, the source code, additional components, documentation, and sample programs are all useful extras that make the tools easier and more enjoyable to use. ▲

Douglas Horn is a free lance writer and Contributing Editor to *Delphi Informant*. He can be reached via e-mail at [horn@halcyon.com](mailto:horn@halcyon.com). Readers may browse a collection of his past articles at his World Wide Web site, <http://www.halcyon.com/horn/default.htm>.





# AT YOUR FINGERTIPS

Delphi 1 / Delphi 2 / Object Pascal

By *David Rippy*

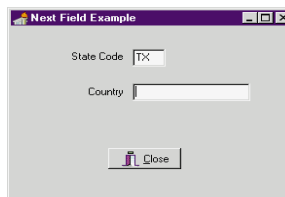
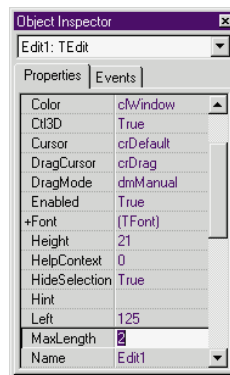


If we are to achieve results never before accomplished, we must employ methods never before attempted. — *Sir Francis Bacon*

## How can I automatically set focus to the “next” edit field in a data entry screen when I’ve finished entering the value for the current field?

For many “heads down” data entry programs, you don’t want to slow users by requiring them to use the mouse to advance to the next field. Here’s a way to automatically set focus to the next control on the screen as soon as the user enters the last character in the current edit field.

First you must enter a value for the field’s (i.e. the Edit component’s) *MaxLength* property as shown in **Figure 1**. For example, if the edit field is designed to store a two-character state abbreviation (see **Figure 2**), enter 2. Then add the code shown in **Figure 3** to the *OnKeyPress* event handler for the *Edit1* control. This code executes each time the user presses a key while in



**Figure 1 (Top):** Setting an Edit component’s *MaxLength* property. **Figure 2 (Bottom):** Focus will be automatically set to **Country** after the user enters the two-digit **State Code**.

```
procedure TForm1.Edit1KeyPress(Sender: TObject;
                                var Key: Char);
begin
  if (Length(Edit1.Text) = (Edit1.MaxLength - 1)) then
    Perform(WM_NEXTDLGCTL, 0, 0);
end;
```

**Figure 3:** The *OnKeyPress* event handler for *Edit1*.

the *Edit1* field. After each key press, the number of characters entered in *Edit1* is compared to its *MaxLength* property.

You might be surprised that the code compares the length to *MaxLength* - 1. This occurs because the code in the event handler executes *before* the key is entered in the field. Try comparing this against *MaxLength* without subtracting 1 and see what happens. If the user has entered the last required character, the message *WM\_NEXTDLGCTL* is sent to Windows, and focus advances to the next field (**Country** in this case).

Thanks to Randy White for posing this question. — *D.R.*

## How can a program check if another program is running?

I recently wrote a utility application that updated the executable of another application. Naturally, I needed to ensure the other application was not running concurrently with the utility program. Surprisingly, this is an easy task. All that is required is one call to the Windows API function, *GetModuleHandle*. You pass to *GetModuleHandle* the name of the executable file you are checking for (OTHER-APP.EXE in this case), and it returns its handle. If the program is not running, it returns a zero.

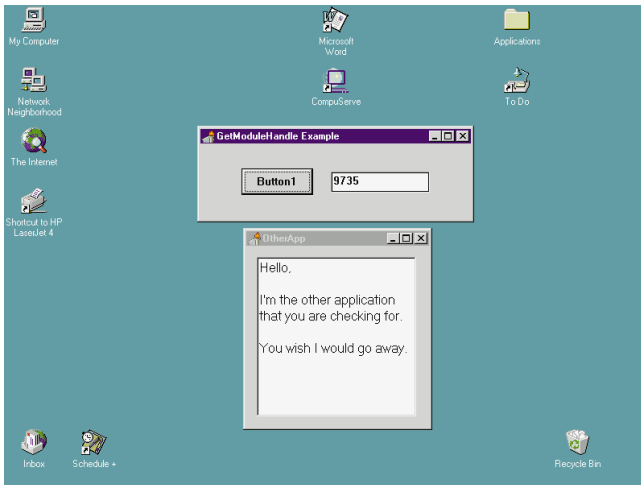
Examine the event handler for the *OnClick* method in **Figure 4**. When **Button1** is pressed, this code displays the handle for OTHER-APP.EXE in *Edit1* (see **Figure 5**). If OTHER-APP.EXE is not running, 0 is displayed.

Note: This tip applies only to Delphi 1. This is because *GetModuleHandle* is part of



```

procedure TForm1.Button1Click(Sender: TObject);
begin
    Edit1.Text := IntToStr(GetModuleHandle('Otherapp.exe'));
end;
    
```



**Figure 4 (Top):** The *OnClick* event handler for *Button1*.  
**Figure 5 (Bottom):** If *OTHERAPP.EXE* is running, its handle is displayed in the *Edit* box.

the Windows 16-bit API, and does not work the same in the 32-bit Windows 95/NT API. It would be much more difficult to implement this tip in the 32-bit world due to address spacing and registry considerations, but it just might show up as a future tip in this column. — *Fred Rahmanian, Ensemble Corporation*

**How can I ensure that my program is pointing to the application’s directory?**

By using this tip, you can assure your application always “points” to the directory where its executable (.EXE) resides. This is useful if your application performs actions that relate to file directories such as loading an image, playing a .WAV file, or setting the *Database* property of a *TTable* component. This helps make your application more directory-independent, assuming the files you need are in the same directory as the executable.

Examine the code in **Figure 6**. This code loads a bitmap image (.BMP) into the *Image1* component on the form. *PICTURE.BMP* is located in the same directory as our program, but we had to hard-code its path. This is because the current directory has not been set properly, and the application would not have found the .BMP by default in the directory it was using. The result would be a run-time error indicating that the image could not be found.

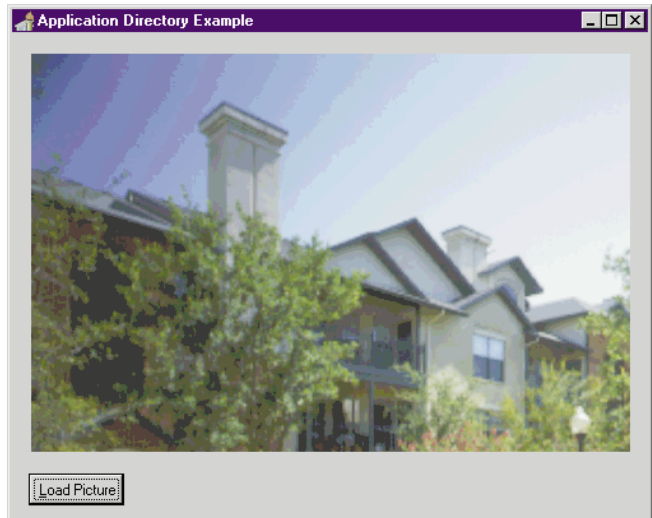
Now examine the form’s *OnCreate* event handler in **Figure 7**. By using the *ExtractPath* and *ChDir* commands, we can set the current directory to the same directory where the program’s executable resides. After this is done, we no longer need to hard-code the directory where the image is stored, and the picture loads properly (see **Figure 8**). — *D.R.*

```

procedure TForm1.BitBtn1Click(Sender: TObject);
begin
    Image1.Picture.LoadFromFile('Picture.BMP');
end;
    
```

```

procedure TForm1.FormCreate(Sender: TObject);
var
    sAppExe : string;
begin
    sAppExe := ExtractFilePath(Application.ExeName);
    ChDir(sAppExe);
end;
    
```



**Figure 6 (Top):** We had to hard-code the path for *PICTURE.BMP*.  
**Figure 7 (Middle):** The *OnCreate* event handler for *Form1*.  
**Figure 8 (Bottom):** Now the picture loads properly.

**Quick Tip for Laptop and Notebook Computer Users**

If you use a laptop as your development platform, you probably spend a great deal of time repositioning windows, showing and hiding the Object Inspector, and switching between form and code view. 640x480 simply doesn’t provide enough room to work! Here’s a reminder of a few useful key strokes that will make life with the laptop more bearable:

- **F9** — Compile and run the program
- **F11** — Toggle to show/hide the Object Inspector
- **F12** — Toggle between viewing the form and its unit
- **Ctrl+F12** — Show a list of units
- **Shift+F12** — Show a list of forms
- **Alt+O** — Show a list of all open windows

— *D.R.* ▲

*The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\JUL\DI9607DR.*

David Rippy is a Senior Consultant with Ensemble Corporation, specializing in the design and deployment of client/server database applications. He has contributed to several books published by Que, and is a contributing writer to *Paradox Informant*. David can be reached on CompuServe at 74444,415.



## Levis, Stalin, and Software

“Real world” is turning into one of those hackneyed terms in software development. User interface design is obsessed with finding real world metaphors that work in applications. Similarly, object-oriented design aims to create software objects that model their real world counterparts. As you can see, we as developers are captivated with bringing real world *content* into our software products — but how often do we look at software systems themselves and see how they are related to other systems in our world? In that light, let’s take a look at four lessons that can be gleaned from “human systems.” I believe that by looking at these truths, we can learn to design better software.

**Rugged Individualism.** *Successful systems let an individual be himself or herself.* In the retail world, companies like Levi Strauss are now using “mass customization” to provide better service to its customers — in this case, a custom-fit pair of jeans. In the political world, the history of the former Soviet Union shows us it is impossible to truly advance and prosper without a democratic society. This same need for freedom also exists in the software realm. While people worked with inflexible, controlled programs for years, they typically did their work in spite of them, rarely flourishing in such an environment. (In fact, as was the case with the thriving Soviet black market, much of the energy may be spent going *around* a rigid system, not working within it.) In other words, your users need freedom to “do their own thing.” Alan Cooper writes in his thought-provoking book *About Face: The Essentials of User Interface Design* [IDG Books, 1995], “Users really like personalization. It allows them to feel part of the computing process; to buy into the task being performed.” Windows 95 — which has an interface geared towards customization — provides a great example of this democratic ideal. Don’t design your software as Stalin would; if so, it will never produce the effects for which the software was designed.

**Balanced Approach.** *Paradoxically, successful systems also provide structure.* Too much individualism can be detrimental, and can leave people feeling lost. As a father, I have discovered the intrinsic need that children have for

structure and discipline — not unadulterated freedom. Bringing this discussion back to our software world, we can say that personalization is important, but only within a strong framework. Moreover, within this backdrop, the individual needs of users are quite different. For example, some users will change their Windows color schemes daily, while others will never stray from the default settings. Your applications must account for both types of users. Cooper concludes: “Personalization is one of those idiosyncratically modal things. People either like it or they don’t.”

**Decentralization.** *Successful systems are decentralized.* Maintaining a central focus always sounds great in principle, especially to those in leadership. But the reality is that “centralized” systems rarely work. Think of the economic world in the twentieth century: free economies prospered while centrally planned economies of communist governments floundered. This also holds true in our software domain. Server-based applications running on terminal emulators are relics. While the World Wide Web renews the call by some for centralized solutions, the fact is that the Web will be successful because there is “freedom” on the desktop, not because people will rush out to buy Internet terminals. People can accept centralization when it makes sense, but not when it limits their individualism.

**Fallen Systems.** *Successful systems are built with realistic expectations.* As much as we hate to admit it, human

systems are “fallen systems.” Realizing this, our Founding Fathers structured a divided government in such a way as to prevent a single individual from achieving absolute power. Their premise was to plan for the possibility of problems. However, the temptation in system design is often the opposite: to build a perfect world. People have always tried to achieve this — be it through government, economics, education, or philosophy — but it has never been achieved. So too in the software realm. The expectations of developers and users in software are so high that we are bound to be disillusioned when problems develop. While we should not excuse bugs, we do need to put them into the proper context. Software is a human invention, and just like other products of the human intellect, it is fallible. If we recognize this reality, we can plan for problems and put appropriate safeguards into place.

**Not in a Vacuum.** Software development is never performed in a vacuum. The products we develop are used by people living in the “real world.” Thus, the same way people react to political, economic, and commercial systems will be the way users respond to your software. ▲

— Richard Wagner

*Richard Wagner is Contributing Editor to Delphi Informant and the Chief Technology Officer of Acadia Software in the Boston, MA area. He welcomes your comments at [rwagner@acadians.com](mailto:rwagner@acadians.com).*

