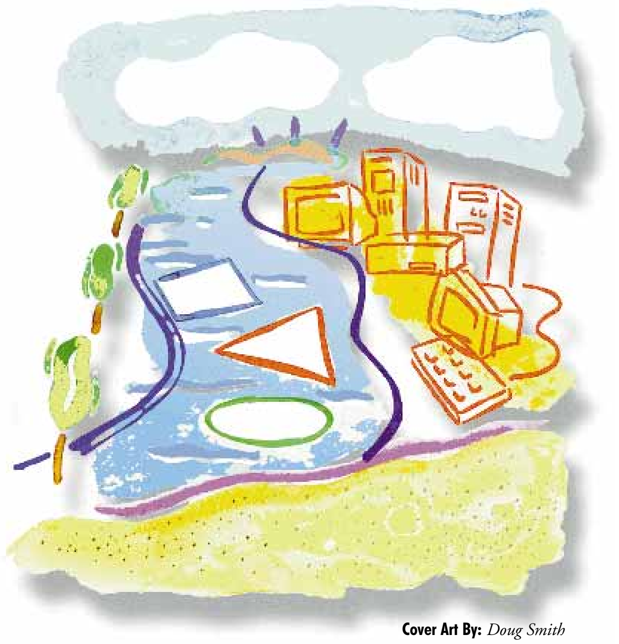
December 1995 - Volume 1, Number 8

Objects in the Stream

Building a Stream-Based Persistent Object Library



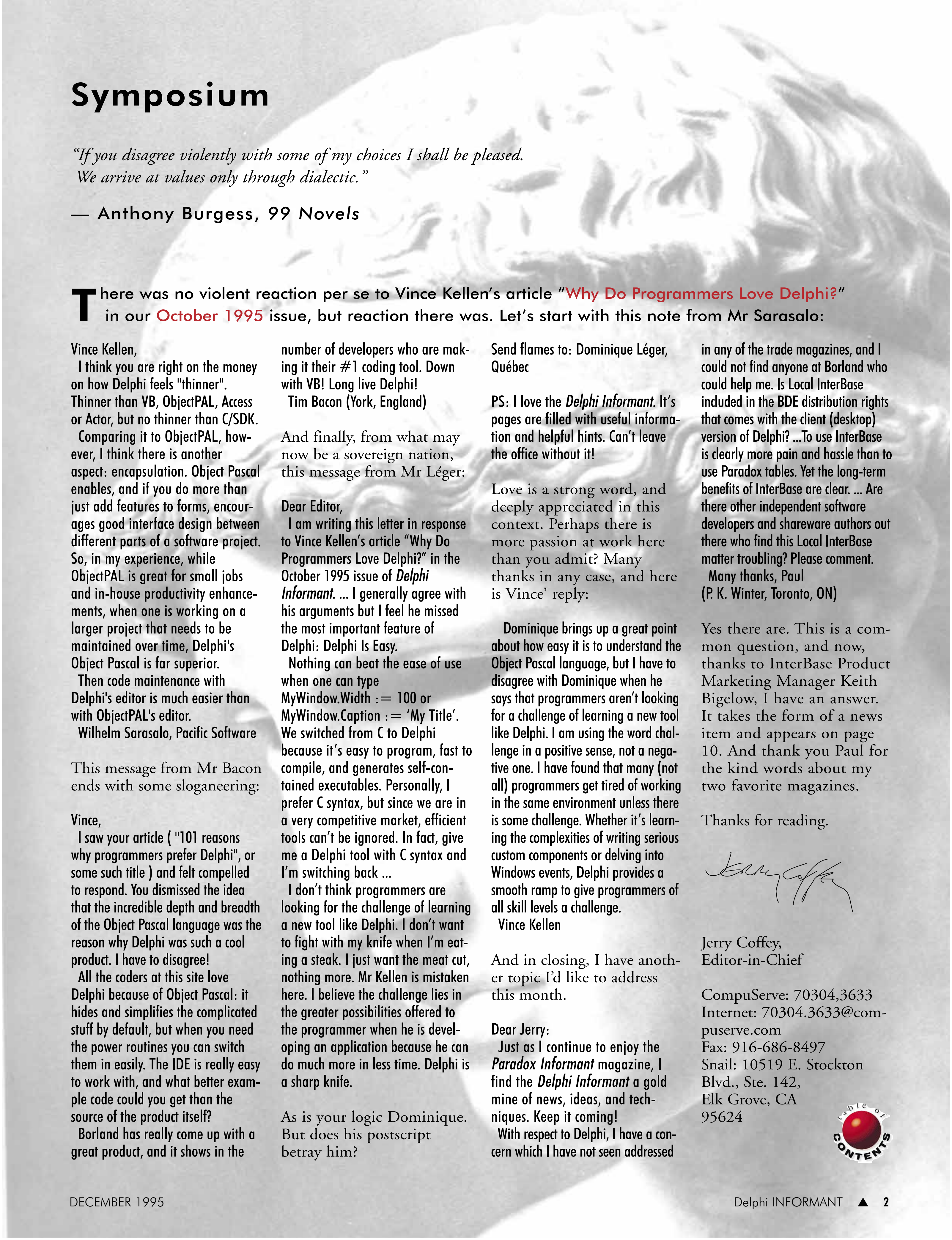
Cover Art By: *Doug Smith*

ON THE COVER

FEATURES

REVIEWS

DEPARTMENTS



Symposium

"If you disagree violently with some of my choices I shall be pleased. We arrive at values only through dialectic."

— Anthony Burgess, 99 Novels

There was no violent reaction per se to Vince Kellen's article "Why Do Programmers Love Delphi?" in our October 1995 issue, but reaction there was. Let's start with this note from Mr Sarasalo:

Vince Kellen,

I think you are right on the money on how Delphi feels "thinner". Thinner than VB, ObjectPAL, Access or Actor, but no thinner than C/SDK.

Comparing it to ObjectPAL, however, I think there is another aspect: encapsulation. Object Pascal enables, and if you do more than just add features to forms, encourages good interface design between different parts of a software project. So, in my experience, while ObjectPAL is great for small jobs and in-house productivity enhancements, when one is working on a larger project that needs to be maintained over time, Delphi's Object Pascal is far superior.

Then code maintenance with Delphi's editor is much easier than with ObjectPAL's editor.

Wilhelm Sarasalo, Pacific Software

This message from Mr Bacon ends with some sloganeering:

Vince,

I saw your article ("101 reasons why programmers prefer Delphi", or some such title) and felt compelled to respond. You dismissed the idea that the incredible depth and breadth of the Object Pascal language was the reason why Delphi was such a cool product. I have to disagree!

All the coders at this site love Delphi because of Object Pascal: it hides and simplifies the complicated stuff by default, but when you need the power routines you can switch them in easily. The IDE is really easy to work with, and what better example code could you get than the source of the product itself?

Borland has really come up with a great product, and it shows in the number of developers who are making it their #1 coding tool. Down with VB! Long live Delphi!

Tim Bacon (York, England)

And finally, from what may now be a sovereign nation, this message from Mr Léger:

Dear Editor,

I am writing this letter in response to Vince Kellen's article "Why Do Programmers Love Delphi?" in the October 1995 issue of *Delphi Informant*. ... I generally agree with his arguments but I feel he missed the most important feature of Delphi: Delphi Is Easy.

Nothing can beat the ease of use when one can type MyWindow.Width := 100 or MyWindow.Caption := 'My Title'. We switched from C to Delphi because it's easy to program, fast to compile, and generates self-contained executables. Personally, I prefer C syntax, but since we are in a very competitive market, efficient tools can't be ignored. In fact, give me a Delphi tool with C syntax and I'm switching back ...

I don't think programmers are looking for the challenge of learning a new tool like Delphi. I don't want to fight with my knife when I'm eating a steak. I just want the meat cut, nothing more. Mr Kellen is mistaken here. I believe the challenge lies in the greater possibilities offered to the programmer when he is developing an application because he can do much more in less time. Delphi is a sharp knife.

As is your logic Dominique. But does his postscript betray him?

Send flames to: Dominique Léger, Québec

PS: I love the *Delphi Informant*. It's pages are filled with useful information and helpful hints. Can't leave the office without it!

Love is a strong word, and deeply appreciated in this context. Perhaps there is more passion at work here than you admit? Many thanks in any case, and here is Vince' reply:

Dominique brings up a great point about how easy it is to understand the Object Pascal language, but I have to disagree with Dominique when he says that programmers aren't looking for a challenge of learning a new tool like Delphi. I am using the word challenge in a positive sense, not a negative one. I have found that many (not all) programmers get tired of working in the same environment unless there is some challenge. Whether it's learning the complexities of writing serious custom components or delving into Windows events, Delphi provides a smooth ramp to give programmers of all skill levels a challenge.

Vince Kellen

And in closing, I have another topic I'd like to address this month.

Dear Jerry:

Just as I continue to enjoy the *Paradox Informant* magazine, I find the *Delphi Informant* a gold mine of news, ideas, and techniques. Keep it coming!

With respect to Delphi, I have a concern which I have not seen addressed in any of the trade magazines, and I could not find anyone at Borland who could help me. Is Local InterBase included in the BDE distribution rights that comes with the client (desktop) version of Delphi? ...To use InterBase is clearly more pain and hassle than to use Paradox tables. Yet the long-term benefits of InterBase are clear. ... Are there other independent software developers and shareware authors out there who find this Local InterBase matter troubling? Please comment.

Many thanks, Paul
(P. K. Winter, Toronto, ON)

Yes there are. This is a common question, and now, thanks to InterBase Product Marketing Manager Keith Bigelow, I have an answer. It takes the form of a news item and appears on page 10. And thank you Paul for the kind words about my two favorite magazines.

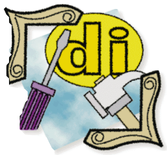
Thanks for reading.

Jerry Coffey,
Editor-in-Chief

CompuServe: 70304,3633
Internet: 70304.3633@compuserve.com
Fax: 916-686-8497
Snail: 10519 E. Stockton Blvd., Ste. 142,
Elk Grove, CA
95624

Delphi TOOLS

New Products and Solutions



Delphi Training Tour

Grumpfish Inc. has posted its training schedule through June 1996. It offers three courses: Introduction to Delphi (one day), Delphi Fundamentals (two days), and Delphi Advanced (three days). The classes are US$295 to US$995, with a discount offered for early registration. For more information call 1-800-GO-DELPHI or e-mail CompuServe: 102121,2741 or Internet: training@ZacCatalog.com.

TurboPower Releases Async Professional for Delphi

TurboPower Software of Colorado Springs, CO is shipping *Async Professional for Delphi*, a serial communications library with VCL components for Delphi.

Async Professional for Delphi (APD) is the first communications toolkit to include ready-to-use terminal windows, protocol status, modem selection, and dialing dialog boxes as components in Delphi's native VCL. All the library functions, from serial port access to file transfers, are implemented as VCL components. APD also has a unique architecture that offers automatic background functions, such as file transfers, dialing terminal updating, etc. The port component generates events under programmer-defined conditions. This approach maximizes communications throughput and responsiveness of the application to user input.

APD also offers a full selection of protocols including Z/Y/XModem, CompuServe B+, ASCII, and more. It has an event-driven dialing engine, flexible modem database with over 100 pre-configured modems, and logging and tracing tools for debugging. APD can use COMM.DRV, other COMM.DRV-compatible replacements, or FOSSIL. It includes an ANSI terminal emulator with scrollback and capture.

Price: US$179. No royalties required.

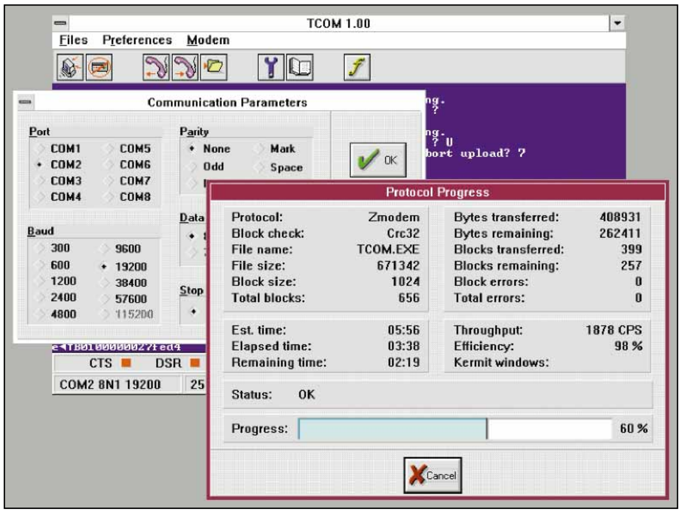
Includes source code; example programs; documentation; free support via phone, fax, and e-mail; free updates; and a 60-day money-back guarantee.

Contact: TurboPower Software, PO Box 49009, Colorado Springs, CO 80949-9009

Phone: (800) 333-4160 or (719) 260-9136

Fax: (719) 260-7151

E-Mail: CIS: 76004,2611



Port: ○ COM1 ● COM2 ○ COM3 ○ COM4 ○ COM5 ○ COM6 ○ COM7 ○ COM8

Baud: ○ 300 ○ 600 ○ 1200 ○ 2400 ○ 4800 ○ 9600 ● 19200 ○ 38400 ○ 57600 ○ 115200

Parity: ● None ○ Odd ○ Mark ○ Space

COM2 8N1 19200

Protocol: Zmodem
Block check: Crc32
File name: TCOM.EXE
File size: 671342
Block size: 1024
Total blocks: 656

Bytes transferred: 408931
Bytes remaining: 262411
Blocks transferred: 399
Blocks remaining: 257
Block errors: 0
Total errors: 0

Est. time: 05:56
Elapsed time: 03:38
Remaining time: 02:19

Throughput: 1878 CPS
Efficiency: 98 %
Kermit windows:

Status: OK

Progress: 60 %

ReportWorks 1.03 for Delphi Ships

HSoftWare of Kitchener, Ontario, has released version 1.03 of *ReportWorks for Delphi*. The ReportWorks component allows developers to create reports with few lines of code and little overhead.

ReportWorks offers left, right, or center column justification, pre-defined line sizes and widths, standard graphic shapes, and imported images such as bitmaps and metafiles. It also supports memo fields, and text strings longer than 255 characters can be output with automatic word-wrap within the report borders. With ReportWorks, page measurements can be made in inches, millimeters, or points, and the fonts, pens, and brushes are customizable.

ReportWorks can direct reports to the built-in preview screen, the printer, or to a file. The screen redraw enables the user to quickly skim or skip report pages, and zoom to any percentage. From the Object Inspector, the preview page can be set to have a black border outline.

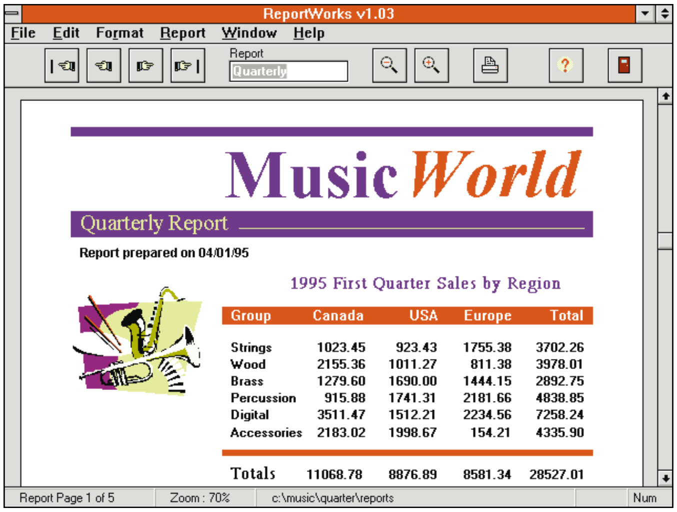
Tracking of report page lines shows the developer the report's printing status, and line height can be set by text height or lines-per-inch.

Price: US$34.95 (no royalties, source code included). Shareware is available on the Informant and Delphi CompuServe forums, filename RPTWORKS.ZIP. Registration can be made in the SWREG forum (ID# 6751).

Contact: HSoftWare, 385 Fairway Road South, Suite 4A-154, Kitchener, Ontario, Canada N2C 2N9

Fax: (519) 894-0739

E-Mail: CIS: 76741,2077



Report: Quarterly

Quarterly Report

Report prepared on 04/01/95

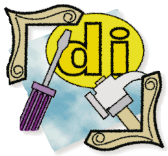
1995 First Quarter Sales by Region

Group	Canada	USA	Europe	Total
Strings	1023.45	923.43	1755.38	3702.26
Wood	2155.36	1011.27	811.38	3978.01
Brass	1279.60	1690.00	1444.15	2892.75
Percussion	915.88	1741.31	2181.66	4838.85
Digital	3511.47	1512.21	2234.56	7258.24
Accessories	2183.02	1998.67	154.21	4335.90
Totals	11068.78	8876.89	8581.34	28527.01

Report Page 1 of 5 | Zoom : 70% | c:\music\quarter\reports | Num

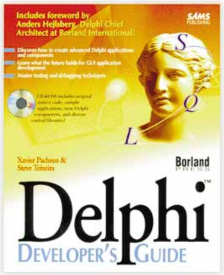
Delphi TOOLS

New Products and Solutions



New Delphi Books

Delphi Developer's Guide
Xavier Pacheco & Steve Teixeira
Sams Publishing
ISBN: 0-672-30704-9
Price: US$49.99
(907 pages, CD-ROM)
Phone: (800) 428-5331


Includes foreword by Anders Hejlsberg, Delphi Chief Architect at Borland International!
SAMS PUBLISHING
Discover how to create advanced Delphi applications and components
Learn what the future holds for GUI application development
Master testing and debugging techniques
S
Q
L
Xavier Pacheco & Steve Teixeira
Borland PRESS
Delphi™
DEVELOPER'S GUIDE

Objective Software Technology Introduces ABC for Delphi

Objective Software Technology has released *ABC for Delphi*, which extends Delphi's VCL with over 25 visual control components for building database applications, graphical systems, or general programming.

ABC for Delphi provides central error handling with messages, log files, and error locations that tie exceptions to original source code. A database version also offers table-driven messages and error logs.

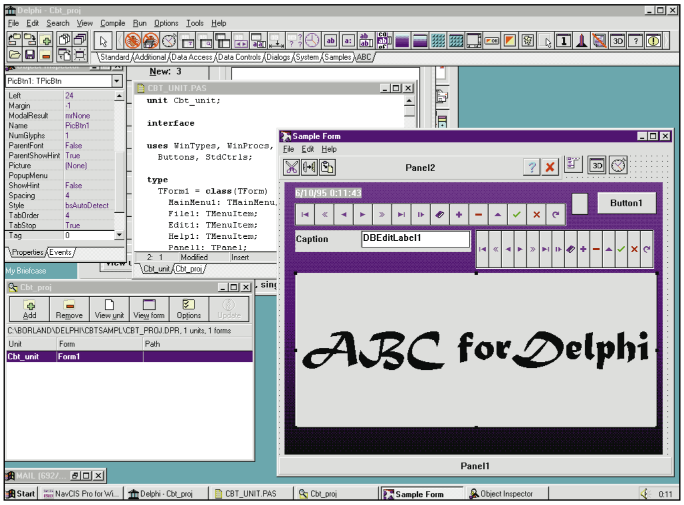
Developers can monitor reports and log program execution using the Stopwatch component. The Navigator component scrolls records up to four times faster than the standard Delphi navigator, and adds extra buttons for jumps, bookmarks, and append processing.

The DBTableGrid and DBQueryGrid components build a complete database form and require only three property settings for a live connection. This eliminates the need to link additional Table, Query, and DataSource components.

Shaded backgrounds and tiled images can easily be added to forms, and there are special components to add these effects to the background of MDI main forms. Several picture buttons provide advanced graphics, 3D labels, and repeating click events on a form. A transparent button can also detect mouse movement over other controls.

Demonstration versions are available in the Informant and Delphi CompuServe forums, filenames: ABC1DEMO.EXE, and ABC1COMP.EXE.

For orders call ZAC catalogs in the USA (1-800-GO-DELPHI), QBS Software in the UK (+44 181 994 4842), K&R Software in Germany (+49 2272 901585) or GUI Computing in Australia (+61 3 9804 3999).


Delphi - Cbt_proj
File Edit Search View Compile Run Options Tools Help
Standard Additional Data Access Data Controls Dialogs System Samples ABC
New: 3
CBT_UNIT.PAS
Sample Form
File Edit Help
Panel2
6/10/95 0:11:43
Button1
Caption
DBEditLabel1
ABC for Delphi
Panel1
Properties Events
My Briefcase
Cbt_proj
Add Remove View unit View form Options Update
MAIL (692/...
Start NavCIS Pro for Wi... Delphi - Cbt_proj CBT_UNIT.PAS Cbt_proj Sample Form Object Inspector 0:11

PicBtn1: TPicBtn

Left	24
Margin	-1
ModalResult	mrNone
Name	PicBtn1
NumGlyphs	1
ParentFont	False
ParentShowHint	True
Picture	(None)
PopupMenu	
ShowHint	False
Spacing	4
Style	bsAutoDetect
TabOrder	4
TabStop	True
Tag	0

```
unit Cbt_unit;

interface

uses WinTypes, WinProcs,
  Buttons, StdCtrls;

type
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Edit1: TMenuItem;
    Help1: TMenuItem;
    Panel1: TPanel;
```

2: 1 Modified Insert

C:\BORLAND\DELPHI\CBTSAMPL\CBT_PROJ.DPR, 1 units, 1 forms

Unit	Form	Path
Cbt_unit	Form1	

Price: Runtime Version 1.0a, US$95; Pro Version 1.0a with source code, US$184.

Contact: Objective Software Technology,
PO Box E138, Queen Victoria Terrace, ACT
2600, Australia

Phone: +61 6 273 2100

Fax: +61 6 273 2190

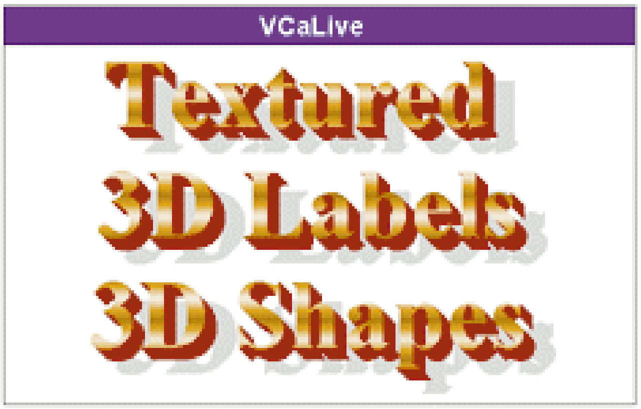
E-Mail: CIS: 100035,2441

VCaLive Offers New 3D Features

Odyssey Technologies, Inc. of Cincinnati, OH has released *VCaLive*, a VCL component pack for Delphi. These components give applications a distinctive appearance with bitmap-textured 3D labels and shapes, as well as animation effects.

This pack includes JazLabel and JazShape which extend the functionality of Delphi's Label and Shape components. Developers can add impact with 3D extrusion, shadowing, and bitmap texturing. In addition, the JazCredits and JazBanners components provide animation by scrolling information either horizontally or vertically.

VCaLive also has a Layout unit to extend Delphi's Printers unit. It features margin settings, column printing, word-wrapping, automatic font-sizing of text boxes, and grid printing.



Price: US$89.

Contact: Odyssey Technologies, Inc.,
PO Box 62733, Cincinnati, OH 45262-0733

Phone: (800) 293-7893

Fax: (513) 777-8026

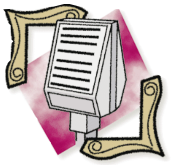
Web Site: http: //ww2.eos.net/odyssey/

E-Mail: Internet: odyssey@eos.net

News

LINE

December 1995



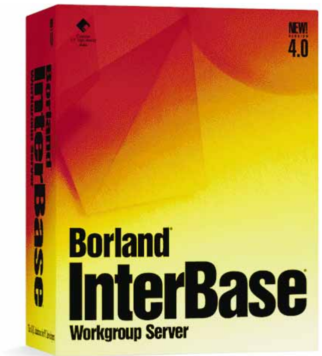
InterBase Distribution, Pricing, and Licensing Explained

Scotts Valley, CA — Every copy of Delphi and Delphi Client/Server ships with a copy of the Local InterBase Server that allows Delphi developers to design and prototype client/server applications remotely on one machine. This means that development can occur on a laptop while on the train, airplane, or at the customer site, and the ultimate database to be used can be changed when the application is ready to be deployed.

Because InterBase is an ANSI SQL 92-compliant server, all the dynamic SQL and transactions that Delphi developers write in their application against Local InterBase are immediately applicable to any multi-user SQL server such as InterBase NT, Oracle 7, Sybase 10, and Microsoft SQL Server.

InterBase workgroup and enterprise developers receive the greatest advantage by using Local InterBase. They can write the SQL their application requires as well as the server objects to make their database applications perform: triggers, stored procedures, and views. Local InterBase applications can scale up to any of the 13 versions of InterBase currently available, including Windows NT, Novell NetWare, and 10 versions of UNIX.

For example, if an application will be deployed against InterBase for NT, the developer can use Delphi and the Local InterBase Server to create a database application with SQL queries, views, and stored procedures. The resulting application, running on the Windows or Windows 95 development computer, can then be scaled up to a multi-user application by copying the database to an installed InterBase Workgroup Server for NT. Using the Windows File Manager, the developer can drag and drop the database from the laptop computer to the NT database server's data volume, without worrying about rewriting server objects such as triggers and stored procedures. Once copied to the NT server, the database and its objects automatically adopt the security of NT, the multi-user capability of NT, and the performance of a dedicated NT database server. The Delphi application, while until now used an alias to point to the database on the development laptop, requires one simple change to redirect the alias so that it now points to the InterBase NT Server's copy of the database.



To clarify, the standard Delphi package includes Local InterBase, but no distribution rights. Delphi Client/Server includes Local InterBase, and Local InterBase distribution rights. When distributed, Local InterBase is a single-user database.

To take a prototype into production, the developer would buy either the NT or NLM version (US$650), the number of user licenses (US$195 each, US$1670/10 pack), and then move the database to the server's data volume. (Use File Manager for moving from Windows 3.1 to Windows NT, use Server Manager's Back-up and Restore for going from Windows 3.1 to NetWare.)

Like Delphi, InterBase has a money-back guarantee.

UK Borland Developers Conference 1996

London — Borland International UK Limited, Desktop Associates Limited, and Dunstan Thomas Limited have planned the UK Borland Developers Conference, 1996.

The event will be held in London, England, April 28-30, 1996 at the Royal Lancaster Hotel, Lancaster Gate, London.

In the last few months, the UK Borland Developers Conference has added 10 new conference sessions. With a total of 50 sessions, these meetings will discuss all Borland products, including: Delphi, Paradox, C++, client/server solutions, InterBase, and others.

They will also address programming, solutions, tools and techniques, and methodologies.

All Borland Developers Conference delegates will receive a free copy of Borland software, documentation, and a diskette or CD containing all the conference sessions and code, and a free tote bag containing sample issues of *Delphi Informant*, *Paradox Informant*, and *Oracle Informant*.

Pricing for the UK Borland Developers Conference 1996 for both days is £495.00 + vat. Those attending only one day pay £250.00 + vat. Discounts are available for three or more attending from the same company.

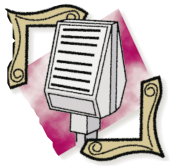
For more information phone Desktop Associates Limited at +44 (0)171 381 9995, fax +44 (0)171 381 9777, or e-mail CIS: 100016,552 (Internet: 100016.552@compuserve.com).

Borland Reports a Profit

Borland International Inc. has announced a net income of US$2.6 million (US$.08 per share) on revenues of US$51.3 million for its second fiscal quarter ending September 30, 1995. These results reflect the second consecutive quarter of profitability since Borland restructured in January 1995, and focused its strategy on software developers. The net income for the six months ending September 30, 1995 was US$5.4 million (US$.18 per share) on revenues of US$105.1 million. According to Borland's President, Gary Wetsel, this progress was particularly pleasing because Borland overcame the seasonal slowness in Europe and the uncertainty surrounding Windows 95. He also said the results reflect growth in their client/server business, including Delphi Client/Server and InterBase. Net income for the same quarter in the prior fiscal year was US$4 million (US$.01 per share) on revenues of US$81.3 million.

News LINE

December 1995



Database & Client/Server World

DCI's Database & Client/Server World is scheduled for December 5-8, 1995 at the Navy Pier in Chicago. The event will feature six conferences covering a variety of client/server and database topics. Over 300 venders are scheduled to display their products and services. For more information, contact DCI at (508) 470-3870 or e-mail 74404.156@compuserve.com.

Borland Promotes Gross, Rosenberg, and Bartelmie

Borland International Inc. has promoted Paul Gross to senior vice president of Research and Development, Jonathan B. Rosenberg to vice president of Borland C++ and Companion Products Development, and Marcia Bartelmie to vice president of Human Resources.

Informant Communications Group Launches New Web Site

Elk Grove, CA — Informant Communications Group, Inc. (ICG) has launched a new Web site on the Internet. Located at http://www.informant.com, this site offers detailed information about all Informant publications and services. The new Informant Web site replaces the Informant Bulletin Board (ICGBB) which will be discontinued December 31, 1995.

From the initial Web site, users can choose from several sections, including magazines, catalogs, CDs, files to download, advertising, ICG news, ICG apparel, and general information.

The magazine section consists of several pages covering every aspect of *Delphi Informant*, *Paradox Informant*, and *Oracle Informant*. One of its more unique features is the Table of Contents outlining the current issue's articles. Users will be able to sample content from each magazine, and have access to a listing of bookstores that carry ICG publications. A magazine article index will also be contained on the Web page.

At the Web site, customers can place subscription orders for any ICG magazine, check the status of a subscription, request a free sample issue, and place orders for back issues. Users can also order the *Delphi Informant Works* and *Paradox Informant Works* CD-ROMs.

Those interested in contributing articles to any ICG publication will have access to writer style guides and editorial calendars, and can submit letters to the editor. ICG will also have a variety of links to third-party vendors that provide add-in products for Delphi, Paradox, and Oracle.

Users can also browse the third-party catalogs produced by ICG including *Delphi Power Tools*, *Paradox Power Tools*, and *Borland C++ Power Tools*. For those interested in advertising in an ICG magazine or catalog, advertising media kits can be requested via this Web page.

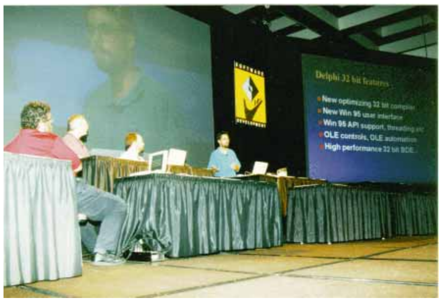
In early 1995, ICG opened the Informant forum on CompuServe (GO ICGFORUM). The files located in this forum will also be available on the Internet at ftp.informant.com. Details on the ICG ftp site were not available at press time.

Companies and user groups interested in linking their Web site to the ICG Web site should contact Carol Boosembark via CompuServe at 72702,1274 or call (916) 686-6610 ext. 16.

SD 95 East: Update

Washington, DC — At Software Development 95 East, Borland wasted no time getting Delphi32 into the spotlight. The October event hosted over 150 venders and 8,000 professionals, and the Borland booth was full during the entire 3-day show. Groups of 40 or more developers gathered for demonstrations of Delphi32, Paradox 7, and C++ 5.0.

On the eve of the show, Borland held a product demonstration and reception. C++ Product Manager Bill Dunlap began the presentation with an overview of the new version of C++. This was followed by the Dover Elevator Systems Delphi case study, and Delphi Product Manager Zack Urlocker concluded the evening with a discussion of the new features in Delphi32.


SOFTWARE DEVELOPMENT
Delphi 32 bit features
New optimizing 32 bit compiler
New Win 95 user interface
Win 95 API support, threading etc
OLE controls, OLE automation
High performance 32 bit BDE...

SD 96 West will welcome over 300 venders to the Moscone Center on March 25-29, 1996. For information call Miller Freeman at (415) 905-2702, or visit their web site at http://www.mfi.com/sdconfs.

TurboPower's Orpheus Bundles with BSS Application Framework

Fairfax Station, VA — Business Software Systems has announced an agreement to bundle TurboPower's Orpheus with their new BSS Application Framework.

The BSS Application Framework was developed to maintain consistent program design and increase programming efficiency. It can advance a development project two to three months into its development cycle, according to the company. In the bundle, the BSS Application Framework consists of three projects, 10 units, a custom DBNavigator, and a project guide.

The three projects include one for development, one master project that is the final program, and one DLL that maintains all the search windows. The units include a registration splash screen, a custom login screen, a Single Document Interface (SDI) menu panel with custom DBNavigator, and maintenance, transaction, posting, system setup, and search screens.

The application framework extensively uses Orpheus to offer features not found in many standard Delphi VCL controls.

The BSS Application Framework and Orpheus bundle is available for US$495 per user. For more information, contact Business Software Systems, Inc. at (703) 503-5600.

By Alan Ciemian

Objects in the Stream

A Framework for a Stream-Based Persistent Object Library

Delphi strongly encourages and supports object-oriented programming. However, Delphi lacks a simple mechanism for object persistence. Although the Visual Component Library (VCL) includes powerful capabilities for storing and loading components to and from streams, these capabilities are difficult to implement and are nearly undocumented. Furthermore, for managing simpler non-component persistent objects, Delphi's component streaming facilities seem like overkill.

This article presents a set of classes that provide the framework for a stream-based persistent object library. In some ways, the classes resemble the streaming facilities of Borland's OWL (Object Windows Library), but they take full advantage of Delphi's updated class model and class registration facilities, and integrate well with the VCL.

The design decisions of these classes were based on:
1) Full support of heterogeneous objects and lists of objects.
2) Simple class interfaces that "feel" like Delphi.
3) Reasonable performance and storage efficiency without compromising items 1 and 2.

Overview

Before getting into the details, I will present a quick survey of the classes, their relationships to each other, and their relationships to the VCL hierarchy (see Figure 1).

TacStreamable is an abstract class that defines the required interface for classes supporting stream-based persistence. It's a subclass of *TPersistent*, Delphi's base class for persistent classes. The *TPersistent* class is the highest class in the VCL hierarchy accepted by Delphi's run-time class registration facilities. *TPersistent* also declares the *Assign* method (and its protected implementation *AssignTo*), the VCL's interface for object assignment. The *TacStreamable* class requires both features.

TacObjStringList is a base class for lists of persistent objects. It's reasonable to suspect that *TacObjStringList* is a subclass of *TStrings* or *TList*. However, because it's extremely useful to treat an entire list as a single persistent object, I descended *TacObjStringList* from *TacStreamable*. If Delphi supported multiple inheritance, *TacObjStringList* would be a sub-

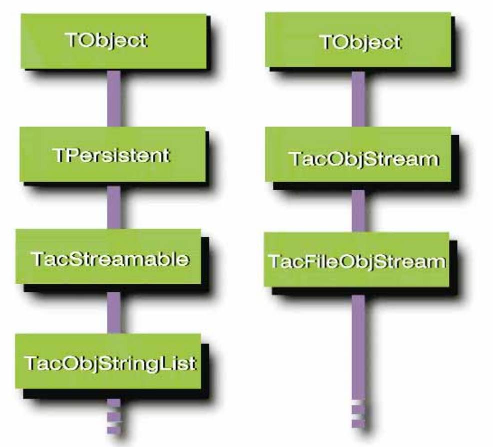


Figure 1: The custom classes presented in this article and their position in the VCL hierarchy.

class of *TStrings* and *TacStreamable*. Since it does not, *TacObjStringList* acquires its list characteristics through another form of code reuse — namely containment.

TacObjStream is an abstract class that defines the required interface for streams that read and save *TacStreamable* objects. It provides the functionality for saving and reading objects to any *TStream* subclass. It's abstract because it does not include the logic for managing the actual *TStream*.

TacFileObjStream is a *TacObjStream* subclass that implements a file-based stream for persistent objects. In another example of containment, the file stream functionality is provided by a privately managed instance of *TFileStream*.

TacStreamable: Building Persistent Objects

Objects that need persistence must be instances of a subclass of *TacStreamable*. This class defines the object's responsibilities in the streaming process. Figure 2 shows the source listing for the *TacStreamable* class.

The protected virtual methods, *SaveToStream* and *ReadFromStream,* are the most critical. Subclasses must override these two methods to provide the necessary logic for saving and reading the state of their objects. The *TacObjStream* class relies on these methods for reading and saving all persistent objects.

The protected virtual method, *InitFields,* provides for class-specific private initialization. All three constructors declared for the *TacStreamable* class call this method either directly or indirectly. Since *TacStreamable* does not contain any data fields, the default implementation of *InitFields* is empty.

The *Create* constructor creates an instance with a default state by calling the inherited constructor and the virtual *InitFields* method.

The *CreateClone* constructor creates an instance from another assignment-compatible instance. It relies on the *Create* constructor to initialize the object to a default state, and the *Assign* method, inherited from *TPersistent*, to copy the other object's state. *CreateClone* is analogous to a C++ copy constructor. Since it depends on *TPersistent.Assign*,

```
type
  TacStreamable = class(TPersistent)
  protected
    { Centralized field initialization }
    procedure InitFields; virtual;
    { Stream interface }
    constructor CreateFromStream(Stream: TacObjStream);
    procedure SaveToStream (Stream: TacObjStream);
      virtual; abstract;
    procedure ReadFromStream(Stream: TacObjStream);
      virtual; abstract;
    { Property methods }
    function GetAsString: string; virtual;
  public
    { Constructors }
    constructor Create;
    constructor CreateClone(const Other: TacStreamable);
    { Properties }
    property AsString: string
             read GetAsString;
  end;

{ TacStreamable implementation }

constructor TacStreamable.Create;
begin
  inherited Create;
  InitFields;
end;

constructor TacStreamable.CreateClone
  ( const Other : TacStreamable );

begin
  Create;
  Assign(Other);
end;

constructor TacStreamable.CreateFromStream
  ( Stream : TacObjStream );
begin
  Create;
  ReadFromStream(Stream);
end;

procedure TacStreamable.InitFields;
begin
end;

function TacStreamable.GetAsString;
begin
  Result := '';
end;
```

Figure 2: The source listing for *TacStreamable*.

TacStreamable subclasses must override either the *TPersistent.AssignTo* or the *TPersistent.Assign* method to define the copy semantic of the class.

CreateFromStream is a protected constructor called by *TacObjStream* to create an instance from a stream. After calling the *Create* constructor to initialize the instance to a default state, it calls the *ReadFromStream* method to update the state from the stream.

The public property, *AsString*, returns the string representation of a persistent object. The protected virtual method, *GetAsString*, provides the implementation. *TacObjStringList* instances use the string representation as a means of referencing and displaying their contained objects. By default, *GetAsString* returns an empty string.

TacObjStringList: Managing Persistent Objects

If your application only uses a few or a fixed number of persistent objects, saving and reading them one by one is probably sufficient. For applications that create a large or indeterminate number of persistent objects, the ability to manage objects in heterogeneous lists is critical. The *TacObjStringList* class is a base class for persistent object containers. As previously discussed, the *TacObjStringList* class is a subclass of *TacStreamable* and supports streaming the entire list as a single persistent object.

The *TacObjStringList* class acquires its list characteristics through containment. Containment is an alternative means of code reuse that allows a class to use the services of another class by maintaining a reference to an instance of that class.

Containment has advantages and disadvantages as compared to inheritance. To expose the methods of the contained object, the class must declare its own corresponding methods that eventually call the methods of the contained object. The resulting advantages include:

- control over method visibility,
- the ability to redefine the interface to the contained object's services, and
- the ability to modify the behavior of any of the contained object's methods.

The resulting disadvantages include:

- the added function call required, and
- the inability to access the contained object's protected interface.

Delphi includes two distinct types of list classes: the familiar string lists (subclasses of *TStrings*) used by numerous VCL components and the more generic *TList* class. Each item in a string list consists of a string and an associated object. Each item in a *TList* is a generic pointer.

The *TacObjStringList* class uses a contained list reference of type *TStrings*. Using the *TStrings* class instead of the *TList* class results in a more versatile class that works well in combination with Delphi's components (especially list-based controls such as *TListBox* and *TComboBox*). *TStrings* is an abstract class that defines the interface for all of Delphi's string lists, including *TStringList*, *TListBoxStrings*, and *TComboBoxStrings*. Since the list reference type is *TStrings* instead of a particular subclass, the *TacObjStringList* class can work directly with any type of string list.

Using the *TStrings* reference in this manner, *TacObjStringList* simulates a limited form of delegation. Delegation, in an object-oriented sense, refers to a form of per-object inheritance that allows different instances of a single class to behave as if they were each inherited from a different class. At runtime, each instance of the original class is associated with an object of a second class, to which it delegates, or forwards, all related method calls.

Specifically, the contained list reference determines the list behavior for *TacObjStringList* instances. Depending on the actual class of the contained list object, individual *TacObjStringList* instances can behave as if inherited from *TStringList*, *TListBoxStrings*, *TComboBoxStrings*, or any other subclass of *TStrings*.

Although the *TacObjStringList* class uses string lists internally, it does not directly expose these to its clients. Items are inserted and removed as *TacStreamable* objects only. The value of each object's *AsString* property determines its associated string. This string identifies objects in the list and visually represents objects in lists associated with visible controls.

The *TacObjStringList* class supports object ownership and also provides insertion and deletion notifications for its subclasses.

One last note before examining the implementation: the design of *TacObjStringList* facilitates its use in conjunction with list controls. Consider the following source:

```
var
  ObjList : TacObjStringList;
...
{ ListBox refers to a TListBox component placed on a form }
ObjList := TacObjStringList.Create(ListBox.Items,False);
```

This code creates a *TacObjStringList* object, *ObjList*, that references the string list associated with a specific list box, *ListBox*. Used in this manner, *ObjList* also acts as a list manager for *ListBox*. You can perform most list manipulations, including insertions, deletions, and persistent storage, directly on the *ObjList* and have the results automatically reflected in the visible list. This reduces coupling the application's logic to the interface, limiting the effects of interface changes (such as changing from a list box to a combo box) to a few lines of code.

The implementation of *TacObjStringList* can logically be divided into two parts: list management and persistence. Figure 3 shows the source listing for the class interface. The following discussion covers the major aspects of the class. Refer to the complete source for the remaining details.

The *TacObjStringList* Class

Five private fields relate to list management. The *FList* field is a reference to the contained list object. *FOwnList* is a Boolean flag that specifies ownership of the list referenced by *FList*. *FOwnObjects*, another Boolean flag, specifies ownership of the objects contained in the list. In both cases, ownership means the *TacObjStringList* object has responsibility for freeing the owned object(s). The *FOnInsert* and *FOnDelete* fields are method references for the insertion and deletion notification events.

Public properties provide access to the *TStrings* object (read-only), the count of objects in the list (read-only) and the list

```
type
  TacObjListIndex = Integer;  { Indexing into lists }
  TacObjListCount = LongInt;  { For saving list count
                                to stream. }

type { for TacObjStringList notifications }
  TacObjListNotifyEvent =
    procedure (Idx: TacObjListIndex) of object;

type
  TacObjStringList = class(TacStreamable)
  private
    { Reference to contained list }
    FList       : TStrings;
    { Flag for list ownership }
    FOwnList    : Boolean;
    { Flag for list item ownership }
    FOwnObjects : Boolean;
    { Delete notification }
    FOnDelete   : TacObjListNotifyEvent;
    { Insert notification }
    FOnInsert   : TacObjListNotifyEvent;
    procedure ResetList(const Strings : TStrings;
                        const OwnObjs : Boolean);
    procedure CloneContents(const OtherList:
      TacObjStringList);
    procedure FreeList;
    procedure FreeObjects;
    { Property access methods }
    function  GetCount: TacObjListIndex;
  protected
    { TPersistent overrides }
    procedure AssignTo(Dest: TPersistent); override;
    { TacStreamable overrides }
    procedure InitFields; override;
    procedure ReadFromStream(Stream: TacObjStream);
      override;
    procedure SaveToStream  (Stream: TacObjStream);
      override;
    { Protected properties }
    property OnObjDelete: TacObjListNotifyEvent
      read FOnDelete
      write FOnDelete;
    property OnObjInsert: TacObjListNotifyEvent
      read FOnInsert
      write FOnInsert;
  public
    { Construction/Destruction }
    constructor Create(const Strings    : TStrings;
                       const OwnObjects : Boolean);
    destructor  Destroy; override;
    { List object access }
    function  AtIndex(const Idx: TacObjListIndex):
      TacStreamable;
    function  AtName(const Name: string): TacStreamable;
    { Standard list methods }
    procedure BeginUpdate;
    procedure EndUpdate;
    function  Add(const Obj: TacStreamable):
      TacObjListIndex;
    procedure Insert(const Idx: TacObjListIndex;
                     const Obj: TacStreamable);
    procedure Move(const FromIdx: TacObjListIndex;
                   const ToIdx: TacObjListIndex);
    { Deletes delete the objects if they are owned }
    procedure DeleteIdx(const Idx: TacObjListIndex);
    procedure DeleteObj(const Obj: TacStreamable);
    procedure DeleteName(const Name: string);
    procedure DeleteAll;
    { Removes NEVER delete the objects }
    function  RemoveIdx(const Idx: TacObjListIndex):
      TacStreamable;
    function  RemoveObj(const Obj: TacStreamable):
      TacStreamable;
    function  RemoveName(const Name: string):
      TacStreamable;
    { ObjStringList specific methods }
    procedure UpdateObjectName(const Idx: TacObjListIndex);
    { Public properties }
    property Strings: TStrings
      read FList;
    property Count: TacObjListIndex
      read GetCount;
    property OwnObjects: Boolean
      read FOwnObjects
      write FOwnObjects;
    property OwnList: Boolean
      read FOwnList
      write FOwnList;
  end;
```

Figure 3: The source listing for the *TacObjStringList* interface.

and object ownership settings. Protected properties allow derived classes to install handlers for the insertion and deletion notifications.

The private methods *ResetList*, *CloneContents*, *FreeObjects*, and *FreeList* manage the list reference. *ResetList* updates the list reference and the list and object ownership flags. *CloneContents* rebuilds the list to duplicate the contents of another resulting in a list containing copies of the objects in the source list, not references to the same objects. *FreeObjects* and *FreeList* are helper methods for freeing all the objects in the list and the list itself.

The constructor, *Create*, takes two parameters — a *TStrings* object reference and an object ownership flag. The *TStrings* reference identifies an existing list object to use as the list delegate. If the reference is nil, a new *TStringList* object is created to serve as the list delegate. The object ownership flag determines whether or not the list owns the contained objects.

The *Destroy* destructor override frees owned objects and lists before calling the inherited destructor.

The public methods, *AtIndex* and *AtName*, provide access to the objects in the list. *AtIndex* returns the object at a specific index, and *AtName* returns the object with a specific string representation.

The *UpdateObjectName* method updates an object's string representation from its *AsString* property. This method should be called for any object in the list that is modified in a way that could potentially affect its string representation.

Most of the remaining public methods are concerned with list manipulation. Most of these map directly or indirectly to methods of the *TStrings* class. In some cases, the *TacObjStringList* methods augment the *TStrings* behavior with additional error checking and support for notifications and object ownership. For details, refer to the complete source code listing.

Implementing *TacObjStringList*

The implementation of *TacObjStringList* is also a good example of building a persistent class. There are five requirements for a fully functional persistent class:

1) It must be a descendant of *TacStreamable*.
2) It must override the *AssignTo* method to support assignment and copy construction.
3) It must override the *InitFields* method to perform class specific initialization.
4) It must override the *SaveToStream* and *ReadFromStream* methods.
5) It must be registered if objects will be created directly from it.

In addition, the class should override the *GetAsString* method to support string-based list access or display in list controls.

The following discussion approaches these requirements in the context of the *TacObjStringList* class. As previously stated, *TacObjStringList* is a subclass of *TacStreamable*.

Figure 4 shows the source listing of the *AssignTo* method. First, you must verify that the instance is not being assigned to itself. This is more than an efficiency issue. Allowing *AssignTo* to proceed with itself as the destination object would result in memory leaks and nasty general protection faults. The next concern is with the class of the destination object. At first glance, it might appear that the type checking performed by the **as** operator when casting the destination from *TPersistent* to *TacObjStringList* would be sufficient to guarantee assignment compatibility. However, the **as** operator only requires that the destination object's class be *TacObjStringList* or one of its subclasses. Considering the source object's class may also be *TacObjStringList* or one of its subclasses, this would allow all the assignments (see Figure 5).

Unfortunately, in cases (A) and (B), the assignment may not be proper. In both cases, the destination class may include fields and behaviors unknown to the source class. Therefore, the implementation of *TacObjStringList* performs an explicit check to restrict assignments to the type indicated by case (C). Of course, subclasses can easily override this behavior as needed. For valid assignments, the destination object is cast to a *TacObjStringList* and the *CloneContents* method is called to duplicate the source list.

```
procedure TacObjStringList.AssignTo(Dest : TPersistent);
var
  DestStringList : TacObjStringList;
begin
  if ( Dest = self ) then Exit;

  if ( (Dest is TacObjStringList) and
     (Self is Dest.ClassType) ) then
    begin  { Assigning to same or superclass }
      DestStringList := ( Dest as TacObjStringList );
      DestStringList.ResetList(DestStringList.FList,True);
      DestStringList.CloneContents(self);
    end
  else
    begin  { TPersistent will process error }
      inherited AssignTo(Dest);
    end;
end;
```

Figure 4: The *TacObjStringList.AssignTo* procedure.

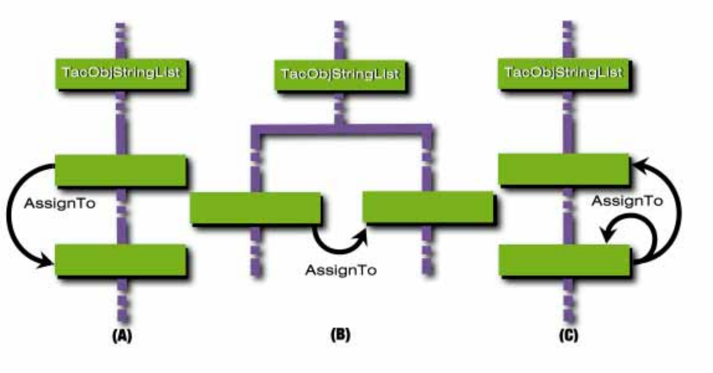


Figure 5: *TacObjStringList* assignments.

The *InitFields* implementation calls the inherited version and then initializes the private list reference to nil and the ownership flags to *False*.

The *SaveToStream* method saves the count of the objects in the list to the stream and then iterates through the list, saving each object with a call to the stream's *SaveObject* method. The *ReadFromStream* method reverses this process. It reads the count of the objects in the list from the stream and then creates each object with a call to the stream's *ReadObject* method.

TacObjStringList does not override the *GetAsString* method. It's unlikely that string representations of *TacObjStringList* objects will be required.

That's it. *TacObjStringList* is now a fully functional persistent class. I have intentionally not yet registered the class. Registration is required only for classes used to create actual objects. Because any given application may or may not create direct instances of *TacObjStringList*, the application is responsible for registration.

TacObjStream: Building a Home for Persistent Objects

TacObjStream is the base class for persistent object streams. Before examining its particulars, it's important to understand how *TacObjStream* organizes data in the stream. Conceptually, the stream consists of four areas:

1) the Stream Header
2) the User Header
3) the Object Data
4) the Class Table

Figure 6 illustrates this format. The stream and user headers are fixed length areas. More specifically, the size of the header for any particular stream must remain constant for the life of the stream, although it may vary from stream to stream. The stream header begins every stream created from the *TacObjStream* class. It includes a signature and version number for identifying the stream, and an offset to

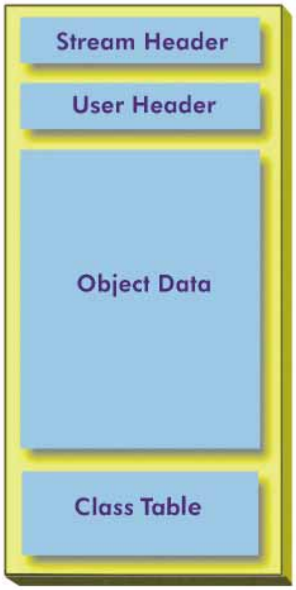


Figure 6: The *TacObjStream* file format.

the class table. *TacObjStream* subclasses can define and manage the optional user header area as needed.

The object data area contains the persistent object data. A 2-byte class ID, written by *TacObjStream*, prefixes each object's data. The class ID identifies the object's class type indirectly through the class table. The class table stores the class type information for each class represented in the stream. The class table consists of a counter, specifying the number of class entries, followed by the class entries in the form of <class name, class id> pairs.

The initial implementation of *TacObjStream* did not include a class table. Instead of writing a class ID before each object, it wrote the class name. Delphi uses a similar procedure when saving components. However, since each class name can occupy up to 64 bytes (they are 63 character strings), this scheme can be space inefficient, especially for streams containing large numbers of small objects. Performance may also suffer, as it requires a class type lookup for each object.

The class table implementation avoids these problems and is more efficient in almost all cases. The class table stores each referenced class name (just once) along with its associated class ID. As a result, the per-object overhead decreases from a possible 64 bytes to a constant 2 bytes. Also, reading from the stream requires a class type lookup only once per class instead of once per object. Although the class table does add complexity, I think the benefits more than justify its use.

Figure 7 shows the source listing for the *TacObjStream* class interface. *TacObjStream* includes three private fields. The *FMode* field stores the stream's current access mode. It's of type *TacObjStreamMode*. *TacObjStream* provides three access modes: input, output, and append. Input mode is for reading some or all the objects in a stream, output mode is for saving an entire stream, and append mode is for adding objects to the end of an existing stream.

The remaining two fields relate to the stream's on disk format. The *FHeader* field contains the stream header. The *FClassTable* field is a reference to the *TStringList* object that maintains the in-memory class table.

The *Create* constructor merely initializes the private fields. The *Destroy* destructor override ensures the stream is closed and the class table string list is freed.

The private methods, *SaveStreamHeader* and *ReadStreamHeader,* manage the stream header. After processing the standard stream header, these methods call the *SaveHeader* and *ReadHeader* methods, respectively. Subclasses override these protected virtual methods to define and manage a user header. By default, their implementation is empty.

The class table is managed by the following private methods: *PrepareClassTable*, *SaveClassTable*, *ReadClassTable*, and *AddClassRef.* The organization of the class table is dependent on the stream's access mode. The *PrepareClassTable* method is responsible for properly configuring the class table prior to its use.

For input mode streams, the class table is organized as an unsorted string list of class names ordered by class ID. The private *ReadClassTable* method reads each entry from the stream and inserts the class name into the class table at a position determined by the class ID. It also calls Delphi's *FindClass* procedure to determine the class type reference for the class name and stores that in the entry's corresponding *Objects* property.

For output mode streams, the class table is organized as a sorted string list of class names, each associated with an integer class ID. The table builds incrementally as objects are saved to the stream. Each saved object results in a call to the stream's *AddClassRef* method. A call to the object's *ClassName* method returns the class name. If the class name is already in the table, *AddClassRef* returns the previously assigned class ID. If the class name is not in the table, *AddClassRef* creates a new class table entry, assigns it the next class ID, and returns the class ID. The private method, *SaveClassTable,* appends the class table to the stream before the stream is closed.

It's important to realize that *ClassName* always returns the class name of the object's actual class, not of the reference variable's class. Specifically, even though *AddClassRef* sees all objects passed to it as *TacStreamable* references, if an object of class *TMyStreamableObject* (a subclass of *TacStreamable*) is passed, the call to *ClassName* returns "*TMyStreamableObject,*" not "*TacStreamable.*"

For append mode streams, the class table is initialized by *ReadClassTable*, but must be organized as if in output mode. This ensures the class table is properly updated if new classes of objects are appended to the stream.

As previously stated, *TacObjStream* does not implement the physical stream management. This is left to its subclasses. Instead, it performs all stream operations through a reference to Delphi's abstract stream class *TStream*. This is similar to using a *TStrings* reference in *TacObjStringList*; it makes it possible to transparently support multiple types of streams.

TacObjStream declares three abstract protected methods that define its required internal stream interface. Subclasses must override these methods to produce a fully working stream

```
type
  { Only 63 chars of identifiers are significant }
  TacStreamableClassName = string[63];
  { Identifies class of streamed objects }
  TacStreamableClassId    = Integer;
  { Index into class list }
  TacStreamableClassIdx   = Integer;

type
  TacObjStreamMode =
    (
    osmClosed,  { Stream not open }
    osmInput,   { For reading only }
    osmOutput,  { For writing only, starts with
                  empty stream }
    osmAppend   { For writing only, starts with
                  current contents }
    );
  TacObjStreamModes = set of TacObjStreamMode;

{ Standard stream header. Starts every TacObjStream. }
type
  TacObjStreamHeader = record
    Signature        : array[0..7] of Char;
    Version          : LongInt;
    ClassTableOffset : LongInt;
  end;

const
  DefaultObjStreamHeader : TacObjStreamHeader =
    (
    Signature        : 'ACSTREAM';
    Version          : $00000000;
    ClassTableOffset : $00000000
    );

type  { TacObjStream exception classes }
  EacObjStream = class(Exception)
  { Base class for TacObjStream Exceptions }
  end;

  EacObjStreamInvalid = class(EacObjStream)
  { Unexpected stream format, header unrecognized }
  end;

  EacObjStreamWrongMode = class(EacObjStream)
  { Stream is in the wrong mode for requested operation }
  end;
type
```

```
  TacObjStream = class(TObject)
  private
    FMode       : TacObjStreamMode;  { Access mode }
    FHeader     : TacObjStreamHeader;{ Stream header }
    FClassTable : TStringList;       { In-memory class
                                       lookup table }
    { Stream header management }
    procedure SaveStreamHeader;
    procedure ReadStreamHeader;
    { Class table management }
    procedure PrepareClassTable(const Mode:
      TacObjStreamMode);
    procedure SaveClassTable;
    procedure ReadClassTable;
    function  AddClassRef(const Obj: TacStreamable):
      TacStreamableClassId;
  protected
    { Abstract internal stream interface }
    function  GetStream: TStream; virtual; abstract;
    procedure OpenStream(const Mode: TacObjStreamMode);
      virtual; abstract;
    procedure CloseStream; virtual; abstract;
    { Error handling }
    procedure ValidateStreamMode(const Modes:
      TacObjStreamModes);
    procedure ObjStreamError(Exc: Exception); virtual;
    { Placeholders for user added headers }
    procedure SaveHeader; virtual;
    procedure ReadHeader; virtual;
 public
    { Construction/Destruction }
    constructor Create;
    destructor  Destroy; override;
    { Opening and closing stream }
    procedure OpenForInput;
    procedure OpenForOutput;
    procedure OpenForAppend;
    procedure Close;
    { Save and Read methods for streaming objects }
    procedure SaveObject(const Obj: TacStreamable);
    function  ReadObject(const Obj: TacStreamable):
      TacStreamable;
    { Methods used by objects to read/write their data }
    procedure SaveBuffer(const Buffer; Count: Longint);
    procedure ReadBuffer(var Buffer; Count: Longint);
    procedure SaveCStr(const CStr: PChar);
    function  ReadCStr: PChar;
  end;
```

Figure 7: The *TacObjStream* interface source listing.

class for *TacStreamable* objects. The *GetStream* method only needs to return a reference to the actual stream object. This reference provides access to the actual stream for data insertion and extraction. The other two required methods, *OpenStream* and *CloseStream*, impose a standard protocol for opening and closing the stream that is not necessarily linked to the stream's creation and destruction. This minimizes the limitations resulting from *TacObjStream's* inability to support simultaneous input and output on an open stream.

The **public** interface of *TacObjStream* includes methods for opening and closing the stream. The *OpenForInput*, *OpenForOutput*, and *OpenForAppend* open the stream and prepare it for reading, saving, and appending, respectively. The *Close* method closes any open stream.

The *SaveBuffer* and *ReadBuffer* methods are provided mainly for implementing the *SaveToStream* and *ReadFromStream* methods of *TacStreamable* subclasses. Their implementation is trivial. The calls are simply forwarded to the *TStream* object returned by *GetStream*. *SaveCStr* and *ReadCStr* are helper functions that simplify the handling of C-style, null-terminated strings.

The most interesting methods are *SaveObject* and *ReadObject*. *SaveObject* writes a persistent object to the stream, and *ReadObject* recreates a persistent object from the stream. Because of the previously discussed infrastructure, the implementation of *SaveObject* and *ReadObject* is relatively simple (see Figure 8).

SaveObject saves the class ID, returned by a call to *AddClassRef*, and then saves the object with a call to its *SaveToStream* method.

ReadObject reads the class ID and determines the associated class type by using the class ID as an index into the class table. Since the class type reference is stored as a *TObject* reference, it

```
procedure TacObjStream.SaveObject
  (const Obj : TacStreamable);
var
  ClassId : TacStreamableClassId;
begin
  ValidateStreamMode([osmOutput, osmAppend]);

  if ( Assigned(Obj) ) then
    begin
    { Get the class ID }
    ClassId := AddClassRef(Obj);
    { Save the class ID }
    GetStream.WriteBuffer(ClassId, Sizeof(ClassId));
    { Save the object }
    Obj.SaveToStream(self);
    end;
end;

function  TacObjStream.ReadObject
  ( const Obj : TacStreamable ): TacStreamable;
var
  ClassId : TacStreamableClassId;
  ObjType : TacStreamableClass;
  NewObj  : TacStreamable;
begin
  ValidateStreamMode([osmInput]);

  Result := nil;

  { Read class ID and get the corresponding class type
    reference }
  GetStream.ReadBuffer(ClassId, sizeof(ClassId));
  ObjType :=
    TacStreamableClass(FClassTable.Objects[ClassId]);

  { Create a new object of the proper class from the
    stream data }
  NewObj := ObjType.CreateFromStream(self);

  if ( Assigned(Obj) ) then
    begin { Assign created object to passed object and
            return object }
    try
      obj.Assign(NewObj);
      Result := Obj;
    finally
      NewObj.Free;
      end;
    end
  else
    begin { Just return created object }
    Result := NewObj;
    end;
end;
```

Figure 8: The *TacObjStream SaveObject* and *ReadObject* functions.

must be cast back to a *TacStreamableClass* reference before it's assigned to the local class reference variable, *ObjType*. *TacStreamableClass* is a class reference type that can reference any class descended from *TacStreamable*. This cast should always be valid since all objects saved to the stream must be instances of *TacStreamable* subclasses.

A call to the *CreateFromStream* constructor with *ObjType* as the class reference creates a new object from the stream. This call causes Delphi to create an instance of the actual *TacStreamable* subclass referenced by the *ObjType* variable. Thus, even though the *CreateFromStream* constructor is a static method of *TacStreamable*, when it calls the virtual *ReadFromStream* method to perform the actual initialization, the call occurs in the context of the desired class.

At this point, *ReadObject* could simply return a reference to the newly created object. However, the current implementation includes an added degree of flexibility. The *ReadObject* method takes a single parameter of type *TacStreamable*. If nil is passed, *ReadObject* returns the created object, as expected. If a valid object reference is passed, *ReadObject* calls the object's *Assign* method to update its state from the created object, frees the created object, and returns the passed object reference. This added capability circumvents a weakness of *ReadObject*. *ReadObject* always uses the *CreateFromStream* constructor, which in turn, relies on the default constructor, *Create*. Even if the constructor was virtual, all *TacStreamable* based classes would still be limited to a single constructor with a predefined parameter list. As implemented, if the class of the next object in the stream is known, the object can be initialized by any declared constructor before calling *ReadObject*.

TacFileObjStream: Saving Objects to Disk

In most applications involving persistent objects, the objects are stored in disk files. *TacFileObjStream* is a concrete subclass of *TacObjStream* for disk file-based streams. Since most of the functionality is provided by *TacObjStream*, its implementation is rather short. Figure 9 shows the complete listing for the class.

The class declares two private fields, *FFilename* and *FFileStream*. *FFilename* stores the name of the disk file so the file can be opened and closed as needed. *FFileStream* is a reference to the actual *TFileStream* object created to access the file associated with the filename.

As required, *TacFileObjStream* overrides the protected methods *GetStream*, *OpenStream*, and *CloseStream*. *GetStream* returns the *FFileStream* stream reference. *OpenStream* creates a *TFileStream* object and saves the reference in the *FFileStream* field. The desired *TacObjStream* access mode, passed in as a parameter, determines the file access mode of the created stream. *CloseStream* frees the *TFileStream* object referenced by *FFileStream* and resets the field to nil. The corresponding disk file is automatically closed by the *TFileStream* instance before it's destroyed.

This class has a single constructor, *Create*, that takes a filename as a parameter. It calls the inherited constructor and then saves the filename in the *FFilename* field. The inherited destructor, *Destroy*, is overridden. It calls the inherited destructor and then frees the *TFileStream* instance, if it still exists.

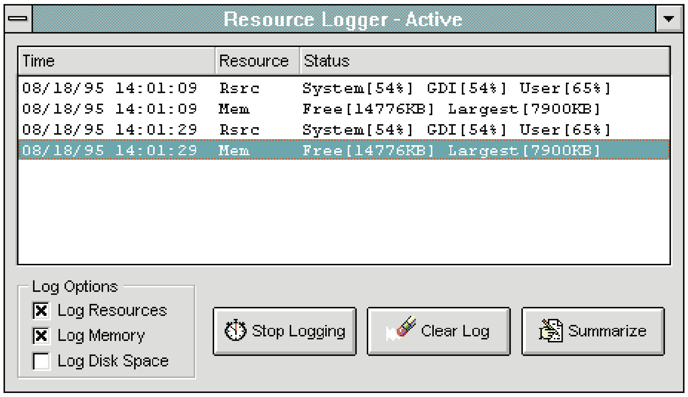
Streams and Persistent Objects in Action

The remainder of this article presents a complete, albeit somewhat trivial, application that exercises most of the library. The name of the application is Resource Logger. It exists solely for periodically logging the system's resource status. It optionally tracks free Windows resources, available memory, and available disk space.

```
type
  TacFileObjStream = class(TacObjStream)
  private
    FFilename   : TFilename;
    FFileStream : TFileStream;
  protected
    { Required internal stream interface }
    function  GetStream: TStream; override;
    procedure OpenStream(const Mode: TacObjStreamMode);
      override;
    procedure CloseStream; override;
  public
    { Construction/Destruction }
    constructor Create(const Filename: TFilename);
    destructor  Destroy; override;
    { Properties }
    property Filename: TFilename
      read FFilename;
  end;

implementation

{ ************* TacFileObjStream ****************** }

constructor TacFileObjStream.Create
  ( const Filename : TFilename );
begin
  inherited Create;
  FFilename := Filename;
end;

destructor TacFileObjStream.Destroy;
begin
  inherited Destroy;
  { Postponed stream Free so TacObjStream can close
    it up, if needed }
  FFileStream.Free;
end;

function TacFileObjStream.GetStream: TStream;
begin
  Result := FFileStream;
end;

procedure TacFileObjStream.OpenStream
  ( const Mode : TacObjStreamMode );
var
  StreamFileMode : Word;
begin
  case  Mode  of
    osmInput  : StreamFileMode :=
      fmOpenRead or fmShareDenyWrite;
    osmOutput : StreamFileMode := fmCreate;
    osmAppend : StreamFileMode :=
      fmOpenReadWrite or fmShareDenyWrite;
    end;
  FFileStream := TFileStream.Create(Filename,
                                    StreamFileMode);
end;

procedure TacFileObjStream.CloseStream;
begin
  FFileStream.Free;
  FFileStream := nil;
end;
```

Figure 9: The *TacFileObjStream* class.

The application consists of a single form (see Figure 10). A list box displays the logged entries and a group of check boxes provides control over logging for each type of resource. The three command buttons start and stop logging, clear the log, and summarize the logged entries.



Resource Logger - Active

```
Time               Resource  Status
08/18/95 14:01:09  Rsrc      System[54%] GDI[54%] User[65%]
08/18/95 14:01:09  Mem       Free[14776KB] Largest[7900KB]
08/18/95 14:01:29  Rsrc      System[54%] GDI[54%] User[65%]
08/18/95 14:01:29  Mem       Free[14776KB] Largest[7900KB]
```

Log Options
☒ Log Resources
☒ Log Memory
☐ Log Disk Space

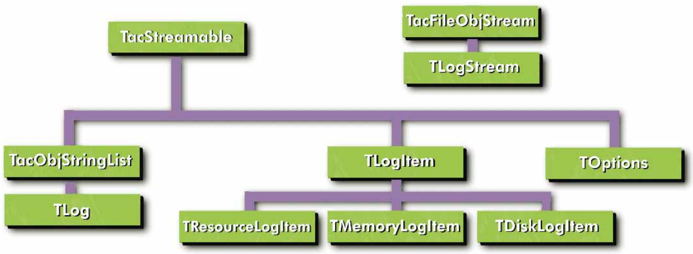


Figure 10 (Top): The Resource Logger form at run-time. **Figure 11 (Bottom):** Resource Logger class hierarchy.

Figure 11 shows the relevant class hierarchy for the application. The *TLogItem* subclass of *TacStreamable* is the base class for all types of log entries. A distinct subclass of *TLogItem* is declared for each of the three types of resources. *TOptions* is another subclass of *TacStreamable* used for saving the logging options. The *TLog* class is a simple subclass of *TacObjStringList* that adds a default array property for accessing the log items with array-like syntax. While active, the application manages the log with an instance of *TLog*, referencing the list associated with the form's list box. The *TLogStream* subclass of *TacFileObjStream* manages the application's data file.

All the application-specific persistent classes are declared and implemented in the EXAMPLE unit. The implementations are all similar. Figure 12 shows the *TMemoryLogItem* class source listing as an example.

The *TLogStream* class is an example of subclassing *TacFileObjStream* to add a user-defined stream header (see Figure 13). The *SaveHeader* method override saves the application specific signature, and the *ReadHeader* method override reads the signature and verifies it. *ReadHeader* raises an exception if the signature is invalid.

The EXAMPLE unit registers the *TResourceLogItem*, *TMemoryLogItem*, *TDiskLogItem*, *TOptions*, and *TLog* classes in its **initialization** section. Only objects of these five classes are instantiated and streamed.

The main form unit contains all the application's major functionality. Up to this point, the discussion has focused on implementing the persistent and stream classes. The remainder examines the form unit as an illustration of using the classes.

```
type
  { TMemoryLogItem handles memory usage log entries. }
  TMemoryLogItem = class(TLogItem)
  private
    FFreeSpace    : LongInt; { Amount of free memory (KB) }
    FLargestBlock : LongInt; { Size of largest available
                               block (KB) }
  protected
    { TPersistent overrides }
    procedure AssignTo(Dest: TPersistent); override;
    { TacStreamable overrides }
    procedure InitFields; override;
    function  GetAsString: string; override;
    procedure SaveToStream(Stream: TacObjStream);
      override;
    procedure ReadFromStream(Stream: TacObjStream);
      override;
  public
    { Properties }
    property  FreeSpace: LongInt
      read FFreeSpace;
    property  LargestBlock: LongInt
      read FLargestBlock;
  end;

implementation

procedure TMemoryLogItem.AssignTo(Dest: TPersistent);
begin
  if ( (Dest is TMemoryLogItem) and
    (Self is Dest.ClassType) ) then
    begin
    inherited AssignTo(TLogItem(Dest));
    with Dest as TMemoryLogItem do
      begin
        FFreeSpace    := self.FFreeSpace;
        FLargestBlock := self.FLargestBlock;
      end;
    end
  else
    begin
      inherited AssignTo(Dest);
    end;
end;

procedure TMemoryLogItem.InitFields;
begin
  { Allow TLogItem to initialize }
  inherited InitFields;

  FFreeSpace    := GetFreeSpace(0)  div 1024;
  FLargestBlock := GlobalCompact(0) div 1024;
end;

procedure TMemoryLogItem.SaveToStream(Stream:
  TacObjStream);
begin
  { Allow TLogItem chance to save }
  inherited SaveToStream(Stream);

  Stream.SaveBuffer(FFreeSpace,SizeOf(FFreeSpace));
  Stream.SaveBuffer(FLargestBlock,SizeOf(FLargestBlock));
end;

procedure TMemoryLogItem.ReadFromStream(Stream:
  TacObjStream);
begin
  { Allow TLogItem chance to read }
  inherited ReadFromStream(Stream);

  Stream.ReadBuffer(FFreeSpace,SizeOf(FFreeSpace));
  Stream.ReadBuffer(FLargestBlock,SizeOf(FLargestBlock));
end;

function  TMemoryLogItem.GetAsString: string;
begin
  { Get Timestamp string from TLogItem }
  Result := inherited GetAsString;

  { Append memory descriptions }
  AppendStr(Result,'  Mem     ');
  AppendStr(Result,Format(' Free[%dKB]',[FreeSpace]));
  AppendStr (Result,Format (' Largest[%dKB]',[LargestBlock]));
end;
```

Figure 12: The *TMemoryLogItem* class.

The form class declares three private fields. The *FOptions* field is a reference to the *TOptions* object used to save the state of the logging options. The *FLog* field is a reference to the *TLog* object used to contain and manage the active log entries. The *FLogStream* field is a reference to the *TLogStream* object used to manage the application data file. All three objects are created in the *FormCreate* method and freed in the *FormDestroy* method.

The **Start Logging** button toggles logging on and off. When logging is enabled, a Timer component triggers a periodic event, and log entries for the selected resources are created and added to the log with a call to *FLog.Add*.

The **Clear Log** button clears all entries from the log with a call to *FLog.DeleteAll*. Since the list owns the log entry objects, it automatically frees them.

The **Summarize** button reports the maximum usage for each resource. This button's handler, *BtnSummarizeClick*, demonstrates iteration of a *TacObjStringList* instance and assignment of *TacStreamable* objects.

The application saves its option settings and the active log entries into a single stream. The *SaveLog* and *ReadLog* methods are responsible for the stream management. Figure 14 shows their source listing.

ReadLog opens *FLogStream* for input, calls the *ReadObject* method twice (once for the options object and once for the list object), and then closes *FLogStream*. Since both objects were previously created, their references are passed to *ReadObject*. As discussed earlier, this alternative use updates the existing objects instead of creating new ones. For the options object, this is mainly for convenience, as it avoids having to free the current options object. Regarding the log, however, it's critical. *FLog* uses the form's list box as its storage list. Freeing the *FLog* object and letting *ReadObject* create a new one would break the association.

SaveLog is nearly identical to *ReadLog*. The only differences are that it opens *FLogStream* for output and calls *SaveObject* in place of *ReadObject*. In both cases, the list object referenced by *FLog* handles reading and saving the individual log entries, regardless of their specific class.

```
type
  { TLogStream represents the log data file }
  TSignature = string[6];
  TLogStream = class(TacFileObjStream)
  private
    FSignature : TSignature;
  protected
    procedure ReadHeader; override;
    procedure SaveHeader; override;
  end;

implementation

const
  LOG_SIGNATURE : TSignature = 'LOGDAT';

procedure TLogStream.ReadHeader;
begin
  ReadBuffer(FSignature, SizeOf(FSignature));
  { Throw exception if invalid signature }
  if ( FSignature <> LOG_SIGNATURE ) then
    begin
      raise Exception.Create('Unrecognized data file');
    end;
end;

procedure TLogStream.SaveHeader;
begin
  FSignature := LOG_SIGNATURE;
  SaveBuffer(FSignature, SizeOf(FSignature));
end;
```

```
{ ReadLog reads the options and log from the data file }
procedure TMainForm.ReadLog;
begin
  if ( FileExists(FLogStream.Filename) ) then
    begin
    with FLogStream do
      begin
        OpenForInput;
        ReadObject(FOptions);
        ReadObject(FLog);
        Close;
      end;
    end;
end;

{ SaveLog saves the options and the log to the data file }
procedure TMainForm.SaveLog;
begin
  with FLogStream do
    begin
      OpenForOutput;
      SaveObject(FOptions);
      SaveObject(FLog);
      Close;
    end;
end;
```

Figure 13 (Top): The *TLogStream* class. **Figure 14 (Bottom):** *TMainForm's SaveLog* and *ReadLog* procedures.

Conclusion

The *TacStreamable*, *TacObjStringList*, *TacObjStream*, and *TacFileObjStream* classes provide the minimum functionality for a useful persistent object library. For many applications, this functionality may be sufficient. For others, it should provide a reasonable starting point for more sophisticated implementations.

There are many enhancements to consider. *TacObjStream* could be extended to support simultaneous reading and writing, or random object access. The accompanying source code includes a *TacObjStream* subclass for memory-based streams. Support for other types of streams might also be useful. Iterators for *TacObjStringList* will be my next enhancement. I would also like to have other types of object containers, such as stacks and queues.

Whether you use the library as-is or as a starting point for your own, there are no more excuses for avoiding persistent objects and streams in your Delphi applications. Δ

References

- Booch, Grady, *Object-Oriented Analysis and Design*, The Benjamin/Cummings Publishing Company, Inc., 1994.
- Coplien, James O., *Advanced C++*, Addison-Wesley Publishing Company, 1992.

The demonstration application referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\DEC\DI9512AC.

Alan Ciemian is a contract programmer and software developer specializing in Windows development using C++, Delphi, and Paradox for Windows. He can be reached on CompuServe at 70134,611 or on the Internet at 70134.611@compuserve.com.

INFORMANT SPOTLIGHT

DELPHI / OBJECT PASCAL

By *Douglas Horn*

Delphi MRU

Adding a Shortcut to Your Most Recently Used Files

It's a familiar feature in Windows programming — a section of the **File** menu that lists the last files accessed by an application. Clicking on any of these filenames opens the corresponding file immediately, allowing users to quickly pick up where they left off. These MRU (or *most recently used*) lists are popular time savers, and are available in most commercial Windows applications (see Figure 1). Unfortunately, MRU lists are not standard in Delphi — you must create them.

In a previous issue we covered integrating .INI files into Delphi applications. [For more information regarding .INI files, see Douglas Horn's article "Initialization Rites" in the August 1995 *Delphi Informant*.] MRU lists are a natural extension of this topic because they use .INI files to store the list of most recently used files between application sessions. This article illustrates how to include MRU lists in new or existing applications with minimal programming.

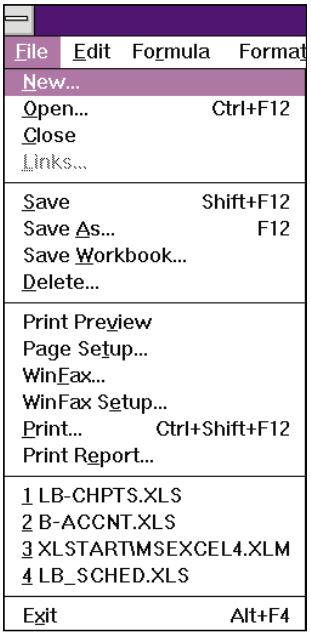


Figure 1: The Microsoft Excel **File** menu showing typical placement and appearance of MRU list items.

Startin

MRU lists can be added just as easily to new or existing applications. To illustrate this, we'll add an MRU list to one of the sample applications that ships with Delphi.

The text editor sample application, described in Chapter 10 of the *Delphi User's Guide*, is a good introduction to Multiple Document Interface (MDI) applications. It's not necessary to perform the tutorial to understand this article, but any reader with questions about programming MDI applications should use it. The completed text editor application, TEXTEDIT, is in the \DELPHI\DEMOS\DOC\TEXTEDIT directory (assuming the default directories were used at Delphi installation time).

There are five basic steps required to add an MRU list to an application:

1) Add the MRU list objects.
2) Manage the .INI file.
3) Update the MRU list.
4) Display the MRU list.
5) Open files from the MRU list.

Adding MRU List Objects

To create an MRU list, you must add two types of objects to an application: the menu items to access the list, and the list itself. The menu items are any number of blank items positioned properly on the main menu. This sample application uses four, because it's the standard number and is easy to work with. However, a list can hold up to nine items. In some Windows applications, such as Microsoft Word 6.0 for Windows, the user can configure the number of MRU items. Using more than nine is possible, but not recommended, because it makes the menus long and the program runs out of single-digit numbers to use as menu shortcut keys.

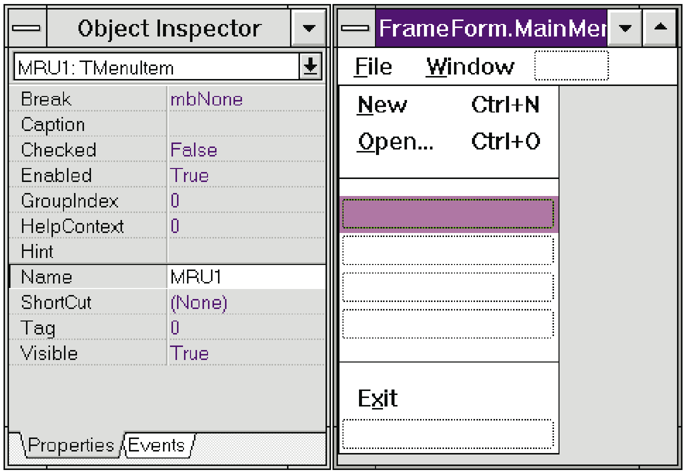
The proper position of the MRU list on a menu is under the **File** menu group, above **Exit**, in its own subsection. Mnemonic shortcut keys for MRU list items are always listed in numerical order (beginning with 1), corresponding to the number of MRU items (again, see Figure 1).

To add the MRU menu items to TEXTEDIT, first open the menu editor for the application's main (MDIParent) form, *FrameForm*. Starting just below the lowest separator (at **Exit**), insert five new menu items. From top to bottom, name these items *MRU1*, *MRU2*, *MRU3*, *MRU4*, and *MRUSeparator*. Leave the captions for items MRU 1 through 4 blank. These will be filled by code at run-time. The caption for *MRUSeparator* should be set to a hyphen (-) to create a menu separator line. Set the *Tag* property of the items as shown in Figure 2.

When the menu items on the MDI parent form (*FrameForm*) are complete (see Figure 3), repeat the process on the application's MDI child form (*EditForm*) using the same names and properties. This does not create a conflict, because although the items are merged at run-time, they exist on different menus.

Name	Caption	Tag
MRU1	*blank*	0
MRU2	*blank*	1
MRU3	*blank*	2
MRU4	*blank*	3
MRUSeparator	-	0

Figure 2 (Left): Setting the *Tag* properties for individual lines in our example.
Figure 3 (Bottom): New MRU list menu items added to the *FrameForm* main menu via the Menu Editor.



Object Inspector

MRU1: TMenuItem

Break mbNone
Caption
Checked False
Enabled True
GroupIndex 0
HelpContext 0
Hint
Name MRU1
ShortCut (None)
Tag 0
Visible True

The MRU list is a *TStringList* component that maintains a list of MRU files while the application is running. To add this list, first declare the *MRUList* variable under *FrameForm*'s **public** part of the **interface** section.

```
public
  MRUList: TStringList;
```

This allows access by the application's other form, *EditForm*.

Now that *MRUList* is declared, it must be put into action. Since it will be active for the life of the application session, it should be created in the *OnCreate* event when the application starts. *MRUList* is a global variable. It will be accessible throughout the application and automatically freed when the application is closed:

```
procedure TFrameForm.FormCreate(Sender: TObject);
begin
  MRUList := TStringList.Create;
end;
```

Managing the .INI File

The container for the MRU list (the *StringList* object, *MRUList*) only contains its contents while the application is running. Between program sessions, it stores the list to an .INI file.

The best time to load values into *MRUList* is when the list is created. Since this application uses four items, a loop should be added to the *FormCreate* procedure to read four values from the MRU section of the .INI file, TEXTEDIT.INI (in the C:\WINDOWS directory). They should then be added to the string list. If the .INI file does not exist, it's created automatically, and blank values are added to the list. Figure 4 shows the updated *FormCreate* procedure.

The *FormCreate* procedure loads the contents of the .INI file into the *MRUList* string list at startup. To have anything to load, however, the application must save the contents of the list upon closing. To accomplish this, the *FormDestroy* procedure is made similar to *FormCreate*, except *FormDestroy* (see Figure 5) copies the contents of the string list back to the .INI file for storage (see Figure 6) until the program is run again.

Finally, for Delphi to access the .INI file (and for the application to run), IniFiles must be added to the **uses** statement in the **interface** section of the *FrameForm* code:

```
uses WinTypes, WinProcs, Classes, Graphics, Controls,
  Printers, Menus, MDIEdit, Dialogs, Forms, IniFiles;
```

Updating the List

The MRU list now exists and can be opened and saved with the application. It must also be updated periodically as new files are used. Traditionally, MRU lists contain the names of all files opened or saved, with the most recently used file at the top. To keep the list current, a procedure must be created to update it, and then referenced every time a file is opened or saved.

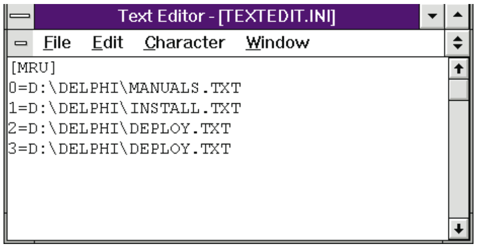
The procedure, *MRUUpdate* (see Figure 7), limits the list to four items and prevents duplicate entries. First, it loops through

```
procedure TFrameForm.FormCreate(Sender: Tobject);
var
  AppIni: TIniFile;  { Declare IniFile variable }
  Index: Integer;    { Declare loop Index variable }
begin
  MRUList  := TStringList.Create; { Create link to IniFile }
  AppIni   := TIniFile.Create('TEXTEDIT.INI');
  for Index := 0 to 3 do;
    { Add items from INI File to MRU list }
   MRUList.Add(AppIni.ReadString('MRU',IntToStr(Index),''));
  AppIni.Free;    { Free IniFile link from memory }
end;
```

```
procedure TFrameForm.FormDestroy(Sender: Tobject);
var
  AppIni: TIniFile;
  Index: Integer;
begin
  AppIni := TIniFile.Create('TEXTEDIT.INI');
  for Index := 0 to 3 do;
  { Write contents of MRU list to .INI file }
  AppIni.WriteString('MRU',IntToStr(Index),MRUList[Index]);
  AppIni.Free;
end;
```

Figure 4 (Top): The *FormCreate* procedure reads MRU information from the .INI file. **Figure 5 (Bottom):** The *FormDestroy* procedure.

the items in the *TStringList* object, *MRUList*, and compares each to the name of the file being opened or saved, *AddFileName*. If an entry matches, it is deleted from the MRU list array. When the procedure has looped through all the items in the *TStringList* object, it does some housekeeping to ensure the MRU list contains exactly three objects, then adds the new, fourth object (*AddFileName*) to the top of the list.



```
[MRU]
0=D:\DELPHI\MANUALS.TXT
1=D:\DELPHI\INSTALL.TXT
2=D:\DELPHI\DEPLOY.TXT
3=D:\DELPHI\DEPLOY.TXT
```

Figure 6: TEXTEDIT.INI showing the newly created .INI file and its saved contents.

MRUUpdate is called when a file is opened or saved. According to de facto standards for MRU lists, new files are not added to the list until they are assigned a filename and saved. Likewise, closed files are not added to the list unless they are saved.

To perform this, the *MRUUpdate* must be called from the application's *Open* and *Save* procedures. In the sample TEXTEDIT application, these procedures are *TEditForm.Open* and *TEditForm.Save1Click*. The *Open* procedure receives a filename from the *OpenDialog* procedure called in *TFrameForm.OpenChild*. By adding the line:

```
FrameForm.MRUUpdate(Self,Filename);
```

before the procedure's **end** keyword, we can tell the procedure to update the MRU list by using the open file's name as the new addition to the list.

```
procedure TFrameForm.MRUUpdate(Sender: TObject;
                              const AddFileName: String);
var
  Index: Integer;  { Declare index variable }
begin
  Index := 0;
  { Compare AddFileName to MRUList items }
  while Index < (MRUList.count - 1) do
    if AddFileName = MRUList[Index] then
      { If already there, delete each occurrence }
      MRUList.delete(Index)
    else
      Index := Index + 1;
  while MRUList.count > 3 do
    MRUList.delete(MRUList.Count - 1);
  while MRUList.count < 3 do
    MRUList.add('');
  { Add fourth item to the top }
  MRUList.Insert(0,AddFileName);
end;
```

Figure 7: The *MRUUpdate* procedure.

Adding the same statement at the same place in the *TEditForm.Save1Click* procedure ensures the MRU list is also updated when a file is saved. In TEXTEDIT, the *SaveAs* procedure calls the *Save* procedure. In applications structured differently, the *SaveAs* event handler must also reference the *MRUUpdate* procedure.

Displaying the List

The program now saves, loads, and updates the values in the MRU list, but it still does not display them. Displaying the MRU list menu items in an MDI application requires two identical procedures. In a Single Document Interface (SDI) application, only a single event handler is required. With multiple documents, however, both the parent and child forms have their own menus. When these menus are merged, the child form is opened and the parent menu becomes invisible. But when a child form is not open, accessing the form's menu produces an error event. Therefore, the procedure for displaying the MRU menu items must be precise.

The *MRUDisplay* procedure is quite simple. The caption of each menu item is set to the filename of the corresponding MRU list item. An ampersand, a number, and a space (e.g. `&1`) precede the filename, telling Delphi to underline the number so it can call the file. If a particular MRU menu item is blank, it will not display. If *MRU1* contains a filename, the separator line is visible.

The most efficient way to set the *Visible* property to *True* if the menu item is not blank, is to use the compound statement:

```
MRU1.Visible := (MRUList[0] <> '');
```

This sets the *Visible* property to *True* if the List item contains a filename (i.e. if `MRUList[x]` is not blank). Otherwise it sets the property to *False*, hiding the item.

The code for *TFrameForm.MRUDisplay* is shown in Figure 8. The *TEditForm.MRUDisplay* procedure is identical, except it's in the *EditForm* code, and references the MDI child menu items.

```
procedure TFrameForm.MRUDisplay(Sender: TObject);
begin
  MRU1.Caption := '&1 ' + MRUList[0];
  MRU1.Visible := (MRUList[0] <> '');
  MRUSeparator.Visible := MRU1.Visible;
  MRU2.Caption := '&2 ' + MRUList[1];
  MRU2.Visible := (MRUList[1] <> '');
  MRU3.Caption := '&3 ' + MRUList[2];
  MRU3.Visible := (MRUList[2] <> '');
  MRU4.Caption := '&4 ' + MRUList[3];
  MRU4.Visible := (MRUList[3] <> '');
end;
```

Figure 8: The *MRUDisplay* procedure.

The procedure would be called:

```
procedure TEditForm.MRUDisplay(Sender: TObject)
```

Normally, two identical procedures would not be necessary; one could call the other to reduce code size. However, this case is an exception. Because the MDI parent form cannot "see" the MDI child forms (except when creating them) it cannot update menu items on these forms. Hence, the duplicate procedures.

Although the routines are straightforward, they must be called at the correct time to display the current MRU list and avoid possible errors. The *EditForm* version is the busiest, as the *EditForm's* menu is displayed whenever files are open. The *FrameForm* version of the menu is only called when the application is first activated, and when an MDI child form is closed. This updates the parent form's menu in case the child form being closed is the last one and the parent form's menu will be displayed again. To enable these functions, add:

```
MRUDisplay(Sender);
```

to the last line of the *FrameForm.FormCreate* procedure, and add:

```
FrameForm.MRUDisplay(Sender);
```

to the last line of the *EditForm.FormClose* procedure.

The *EditForm.MRUDisplay* procedure is called by the *FrameForm.OpenChild* procedure by adding the statement:

```
EditForm.MRUDisplay(Sender);
```

at the last line of the procedure. This procedure defines and creates the child form. Since *EditForm* is declared as a local variable in this *FrameForm* procedure, the procedure can call *EditForm.MRUDisplay*, even though it is invisible to the rest of the program. The complete *OpenChild* procedure is shown in Figure 9. (Note that all lines but:

```
EditForm.MRUDisplay(Sender);
```

already exist in the TEXTEDIT sample application.)

The same statement:

```
EditForm.MRUDisplay(Sender);
```

must also be added to the end of the *EditForm.Save1Click* procedure as shown in Figure 10. This allows the program to display the current MRU information when a file has been saved and the MRU list is updated.

```
procedure TFrameForm.OpenChild(Sender: TObject);
var
  EditForm: TEditForm;
begin
  if OpenFileDialog.Execute then
    begin
      EditForm := TEditForm.Create(Self);
      EditForm.Open(OpenFileDialog.Filename);
      EditForm.Show;
      EditForm.MRUDisplay(Sender); { Update MRU menu items }
    end;
end;
```

```
procedure TEditForm.Save1Click(Sender: TObject);
  { ... listing partially omitted }
begin
  if (Filename = '') or IsReadOnly(Filename) then
    SaveAs1Click(Sender)
  else
    begin
      CreateBackup(Filename);
      Memo1.Lines.SaveToFile(Filename);
      Memo1.Modified := False;
      FrameForm.MRUUpdate(Sender,Filename);
      MRUDisplay(Sender);  { Display MRU List (EditForm) }
    end;
end;
```

Figure 9 (Top): The *OpenChild* procedure.
Figure 10 (Bottom): The *EditForm.Save1Click* procedure.

Finally, to update the MRU menu items in MDI child windows, the application must be instructed to call *EditForm.MRUDisplay* whenever the focus switches between forms. Otherwise, each child window displays the MRU list items as they were when the form was last updated, and not necessarily the current information. To enable this display update, set the *EditForm's* *OnActivate* event to *MRUDisplay*, using the Events Page of the Object Inspector.

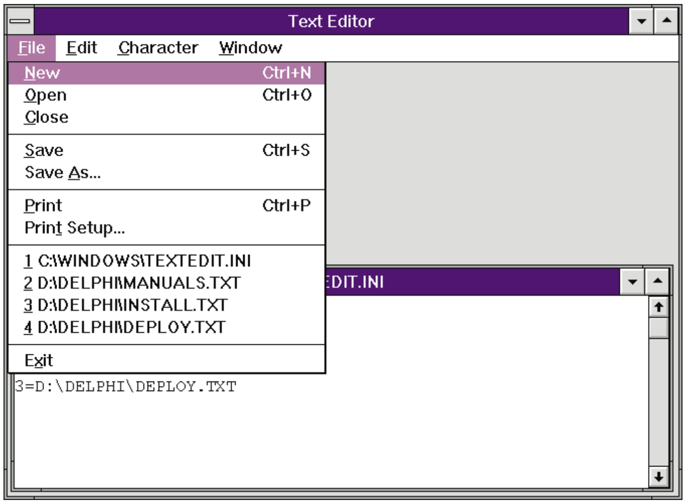
Opening Files from the MRU List

The last task in adding an MRU list to an application is actually opening the files with the click of a button. The TEXTEDIT application can already open a child window using the *FrameForm.OpenChild* procedure. This procedure obtains the name of the file to open from an Open File common dialog box. Since the MRU list already contains the name of the desired file, the dialog box is unnecessary.

The *FrameForm.MRUOpenChild* procedure (see Figure 11) must only reference the MRU list and obtain the filename associated with the selected MRU menu item. To do this, the procedure uses the Sender's *Tag* property (set when the MRU menu items were created in step one), and assumes the Sender is a *TMenuItem*.

Once this event handler is written, it should be set as the *OnClick* event handler for all four MRU menu items (i.e. *MRU1*, *MRU2*, *MRU3*, and *MRU4*) on the *FrameForm* main menu. The corresponding event handler for the *EditForm* MRU menu items should

```
procedure TFrameForm.MRUOpenChild(Sender: TObject);
var
  EditForm: TEditForm;
  Index: Integer;
begin
  { Set Index using Sender.Tag }
  Index := TMenuItem(Sender).Tag;
  if MRUList[Index] <> '' then
    begin
      EditForm := TEditForm.Create(Self);
      EditForm.open(MRUList[Index]);
      EditForm.show;
      EditForm.MRUDisplay(Sender);
    end;
end;
```



```
3=D:\DELPHI\DEPLOY.TXT
```

Figure 11 (Left): The *FrameForm.MRUOpenChild* procedure. **Figure 12 (Right):** The TEXTEDIT application with its new MRU list displayed.

be called *MRUClick*. The procedure *EditForm.MRUClick* references *FrameForm.MRUOpenChild*:

```
procedure TEditForm.MRUClick(Sender: TObject);
begin
  FrameForm.MRUOpenChild(Sender);
end;
```

Ready to Run

The MRU list in TEXTEDIT.DPR is now ready to run (see Figure 12). It will keep track of the files and allow those used last to be instantly opened.

It should now be easy to add an MRU list to any new or existing Delphi application by following these steps. First, add the menu objects and string list to the application. Next, add simple procedures for moving data back and forth between the string list and an .INI file. Then, write an event handler to update the string list as files are opened and saved. Add procedures to display the contents of the string list to the menu items on both parent and child forms. Finally, include a simple event handler to open files from the string list when the corresponding menu item is clicked.

So there you have it! Five steps to a more professional, user-friendly application. ∆

The demonstration project referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\DEC\DI9512DH.

Douglas Horn is a free-lance writer and computer consultant in Seattle, WA. He specializes in multilingual applications, particularly those using Japanese and other Asian languages. He can be reached via e-mail at internet:horn@halcyon.com.

table of CONTENTS

USELESS STUFF

DELPHI / OBJECT PASCAL

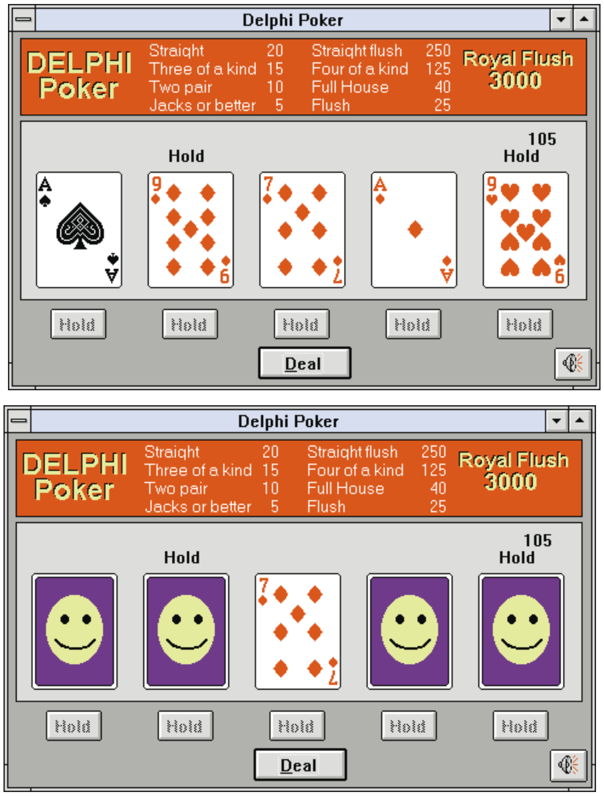
By *David Faulkner*

Upping the Ante

Building a Poker Game with the *TCardDeck* Component

Oh sure, *Delphi Informant* is full of useful, business-like, get-the-job-done-better-and-faster components. Boring stuff. Now for some fun.

This article presents a playing card component, *TCardDeck*, and a video poker game (see Figure 1) easily made from that component. We'll start by learning how to use the component, and then deal with some technical details of its implementation. (And if you download this component, you could try cutting in on the lucrative Solitaire market for Windows 95.)



Delphi Poker

DELPHI Poker					Royal Flush 3000
	Straight	20	Straight flush	250	
	Three of a kind	15	Four of a kind	125	
	Two pair	10	Full House	40	
	Jacks or better	5	Flush	25	

Figure 1: This video poker game was built using the *TCardDeck* component. The first screen shows the player held onto a pair of nines and then pressed the **Deal** button. When a pair of aces is added to the held cards, Delphi flashes smiles.

Installing the CardDeck Component

To use the CardDeck component, download the source code and place it in a directory (e.g. \DELPHI\CARDDECK). The necessary files are shown in Figure 2.

With the source code downloaded, you can install the component. To do this, start Delphi, open the **Options** menu, and select **Install Components** to display the Install Components dialog box. Click the **Add** button and supply the path and name of CARDDECK.PAS. If you don't have a component named *TCardDeck*, Delphi adds `CardDeck` to the **Installed Units** list. Click the **OK** button. Delphi compiles the new component and places it on the DI (*Delphi Informant*) tab of the Component Palette (see Figure 3).

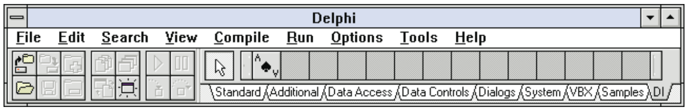
File	Function
CARDDECK.PAS	The source code.
CARDDECK.RES	A resource file with the cardface bitmaps.
CARDDECK.DCR	A Delphi resource file with the CardDeck's Component Palette icon.

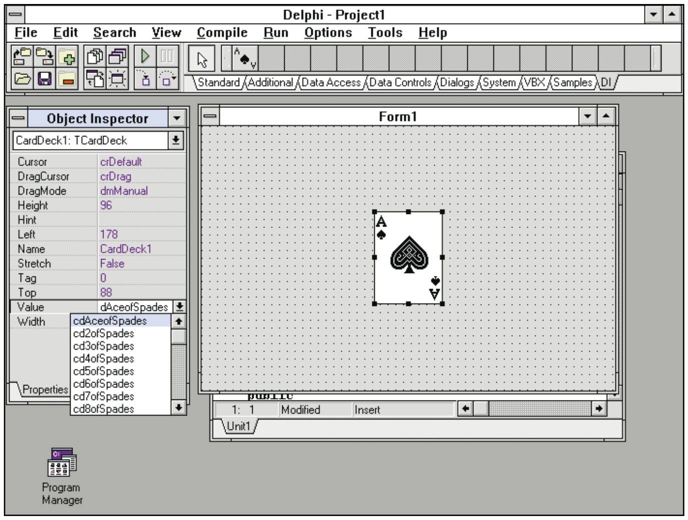
Figure 2: The Delphi files necessary to get you onto the playing table.

Using the *TCardDeck*

Start a new project and drop a CardDeck component on it. The visual representation of *TCardDeck* is an Ace of Spades, 71 pixels wide and 96 pixels tall (see Figure 4).

TCardDeck is a descendant of *TGraphicControl* (as is *TLabel* and *TShape*) so you are probably familiar with its properties. Two new properties introduced by this component are *Stretch* and *Value*. *Stretch* is a Boolean type, and when it's set to *True* the card face bitmap is stretched to *TCardDeck*'s height and width. When *False*, the card face bitmap is always 71 x 96 pixels, regardless of the component's height and width properties.


Delphi
File Edit Search View Compile Run Options Tools Help
Standard Additional Data Access Data Controls Dialogs System VBX Samples DI


Delphi - Project1
File Edit Search View Compile Run Options Tools Help
Standard Additional Data Access Data Controls Dialogs System VBX Samples DI
Object Inspector
Form1
Properties
1: 1 Modified Insert
Unit1
Program Manager

CardDeck1: TCardDeck

Cursor	crDefault
DragCursor	crDrag
DragMode	dmManual
Height	96
Hint	
Left	178
Name	CardDeck1
Stretch	False
Tag	0
Top	88
Value	dAceofSpades
Width	

cdAceofSpades
cd2ofSpades
cd3ofSpades
cd4ofSpades
cd5ofSpades
cd6ofSpades
cd7ofSpades
cd8ofSpades

Figure 3 (Top): The *TCardDeck* component on the DI page of the Component Palette. **Figure 4 (Bottom):** The component on a form.

The *Value* property is an integer that represents the card face displayed. 1 is the Ace of Spades, 2 is the 2 of Spades, 14 is the Ace of Hearts, 27 is the Ace of Clubs, 40 is the Ace of Diamonds, and 52 is the King of Diamonds. You can work with numbers or a set of constants that adds readability to programs using *TCardDeck* components. Figure 4, for example, shows the constant *cdAceOfSpades* in the Object Inspector.

The last 10 "cards" are Card 53 (*cdJoker*), the Joker, and cards 54 to 62 (*cdCardBack1..cdCardBack9*) with bitmaps representing the back of each card.

TCardDeck Utility Functions

The ability to display card faces on a form makes it easy to write card games, but *TCardDeck* goes further with some utility functions. When a *TCardDeck* component is placed on a form, Delphi automatically includes the CARDDECK.PAS unit in your form's unit. By using *TCardDeck*, you automatically have access to the functions shown in Figure 5.

There is no built-in help for these functions or the extra properties defined by *TCardDeck*. Therefore, you may want to keep the source code open on an editor tab. To do this, select **File | Open File** from Delphi's menu. Then, select CARDDECK.PAS from the appropriate directory. You'll find a component's source code is as good as any help system.

We often have several tabs open in the editor to Delphi's units (buy the VCL source if you don't have it). Note that although the File Open dialog box starts with a *.PAS filter, the editor can be used to view any text file (such as those written for an import routine, or HTML source code that may be interfaced).

TCardDeck Informant

When writing a component, you must first decide which existing component to descend from. In *TCardDeck*'s case, descending from *TGraphicControl* seems logical. It already has some essential properties, such as *Position* and *Size*, and has a canvas that makes drawing easy. Additionally, *TGraphicControl* has events such as *Paint* and drag-and-drop support that are necessary in the *TCardDeck* component.

TCardDeck was created with the Component Expert (see Figure 6). To open this dialog box, select **File | New Component**. When you press the **OK** button, Delphi builds a unit with the necessary class declaration and register procedure. Now, complete the code to make the component functional.

Constants

CARDDECK.PAS begins by declaring many constants. Each card name, color, and card are assigned numeric values. A number of subrange types are created to simplify programming with *TCardDeck*. For example, the constants:

```
cdSpades     = [1..13];
cdHearts     = [14..26];
cdClubs      = [27..39];
cdDiamonds   = [40..52];
cdBlackCards = [1..13,27..39];
cdRedCards   = [14..26,40..52];
```

make it easy to determine a card's color or suit.

A set of *TCardEntries* is also configured as a constant. It's declared as:

```
type
  TCardEntry = record
    Value: cdCardDeck;
    Name: PChar;
end;
const
  Cards: array[1..62] of TCardEntry =
    ((Value:  cdAceofSpades;
      Name: 'cdAceofSpades'),
      Value: cd2ofSpades;
      Name: 'cd2ofSpades'),
```

Using the constant *Cards* array, it's simple to map the string version of a card's value to its integer value. This is necessary to set up *TCardDeck's Value* property which accepts a string or an integer.

TCardDeck.Paint

TCardDeck's most important feature is its ability to display bitmaps that represent card faces. To do this, *TCardDeck* overrides *TGraphicComponent*'s *Paint* procedure:

```
type
  TCardDeck = class(TGraphicControl)

protected
  { Protected declarations }
  procedure Paint; override;
```

Function	Description
function IsCardRed(Value: cdCardDeck) : Boolean;	Accepts a card value. Returns *True* if card is red, otherwise *False*.
function IsCardBlack(Value: cdCardDeck) : Boolean;	Accepts a card value. Returns *True* if card is black, otherwise *False*.
function IsFaceCard(Value: cdCardDeck) : Boolean;	Accepts a card value. Returns *True* if card is a face card, othewise *False*.

**function AreCardsSameColor(Value1,
Value2: cdCardDeck) :
Boolean;**
	Accepts two card values. Returns *True* if cards are the same color, otherwise *False*.

**function AreCardsSameSuit(Value1,
Value2: cdCardDeck) :
Boolean;**
	Accepts two card values. Returns *True* if cards are the same suit, otherwise *False*.

**function AreCardsSameValue(Value1,
Value2: cdCardDeck) :
Boolean;**
	Accepts two card values. Returns *True* if the cards have the same face value and the same color, otherwise *False*. For example, if the 2 of spades and 2 of diamonds are passed, *True* is returned.

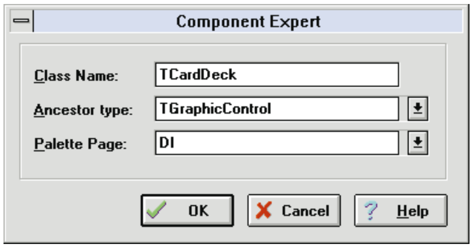
function CardColor(Value: cdCardDeck) : integer;	Returns an integer representing the color of the card passed to it. CARDDECK.PAS defines the following constants for card colors:
	`cdBlack = 1;`
	`cdRed   = 2;`

function CardSuit(Value: cdCardDeck) : integer;	Returns an integer representing the suit of the card passed to it. CARDDECK.PAS defines the following constants for card suits:
	`cdSpade   = 1;`
	`cdHeart   = 2;`
	`cdClub    = 3;`
	`cdDiamond = 4;`

function CardValue(Value: cdCardDeck) : integer;	Returns an integer representing the face value of the card passed to it. The number 1 represents an Ace, 2 represents a 2, and 13 represents a King.

Figure 5: Each function used by *TCardDeck*.

Simply stated, whenever Windows requests *TCardDeck* to paint itself, the *TCardDeck.Paint* procedure is called. See Figure 7 for an abbreviated version. The code not shown paints the corners of the cards the same color as the background. The entire code listing for the project begins on page 27.



Class Name: TCardDeck
Ancestor type: TGraphicControl
Palette Page: DI

Figure 6: The Component Expert dialog box.

Some important information should be noted within this procedure. Once the variable *BitMap* is created, the code uses a **try..finally** block. This protects the component from leaking memory if a GPF occurs. The *BitMap.Create* command (or any *Create* command) allocates memory. As a component designer, you must free this memory, even if the component causes an exception during processing. In this case, if something goes wrong while painting the component, the **finally** clause calls *BitMap.Free* to release the allocated memory.

The bitmaps for card faces are stored in CARDDECK.RES, a standard Windows resource file. With the Borland Resource Editor or Delphi's Image Editor, you can view and modify bitmaps outside Delphi. To make resources in an .RES file available at run-time, a compiler directive is necessary. It instructs Delphi to include the .RES file in the .EXE file. For example, *TCardDeck* uses the:

```
{$R CARDDECK.RES}
```

compiler directive. Although it resembles one, it's not a comment. To learn more about the many compiler directives, type the dollar symbol ($) on a blank line in the code editor and press F1. You can now surf the help system for information. With CARDDECK.RES included in the .EXE, this code is used to extract a bitmap from the resource file:

```
BitMap.handle := LoadBitMap(HInstance,FacePChar);
```

```
procedure TCardDeck.Paint;
var
  BitMap:TBitMap;
  FacePChar: array [0..25] of char;
begin
  strpcopy(FacePChar,CardToString(Value));
  BitMap := TBitMap.create;
  try
    BitMap.handle := LoadBitMap(HInstance,FacePChar);
    if not BitMap.empty then
     begin
        if Stretch then
          canvas.stretchdraw(clientrect,BitMap)
        else
          Canvas.draw(0,0,BitMap);
      end;
  finally
    BitMap.free;
  end;
end;
```

Figure 7: An abbreviated version of the *TCardDeck.Paint* procedure.

The *LoadBitMap* procedure is actually a Windows API call. Since the WINPROCS.PAS unit wraps many Windows API calls, including this one, calling *LoadBitMap* is relatively simple. The *HInstance* variable is a global variable available in Delphi programs. In the call above, *HInstance* tells Windows which "instance" of a program is running, and to find the bitmap in the correct .EXE. The *FacePChar* is a PChar version of a *CardDeck.Value*. The bitmaps in the resource files are identified by these values, i.e. *cdAceOfSpades*, *cd2OfSpades*, etc.

With the bitmap loaded, this code paints the screen:

```
if Stretch then
  canvas.stretchdraw(clientrect,BitMap)
else
  canvas.draw(0,0,BitMap);
```

The *cdCardDeck* Custom Property Editor

The next fun part of the CardDeck component is the custom property editor for *TCardDeck*'s *Value* property. When *Value* is displayed in the Object Inspector, a card name or number can be entered, or a card can be selected from a list. Although it's beneficial to the component user, it's work for the component writer.

First, *TCardValueProperty* is declared by descending from *TIntegerProperty*:

```
type TCardValueProperty = class(TIntegerProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    function GetValue: string; override;
    procedure GetValues(Proc: TGetStrProc); override;
    procedure SetValue(const Value: string); override;
end;
```

GetAttributes: While *TCardValueProperty* is similar to *TIntegerProperty*, it's different because we want a drop-down list of values. To tell the Object Inspector that *TCardValueProperty* has a list of values, the code cited above overrides the *GetAttributes* function with a single line of code:

```
function TCardValueProperty.GetAttributes:
  TPropertyAttributes;
begin
  Result := [paValueList,paMultiSelect];
end;
```

The *paValueList* constant tells the Object Inspector that a drop-down list is available. The *paMultiSelect* constant instructs the Object Inspector to continue displaying this property if more than one *TCardDeck* component is selected. This allows you to select multiple instances of a *TCardDeck* object and set their values simultaneously in the Object Inspector.

GetValue: The Object Inspector can only display properties as strings. Since the *TCardDeck Value* property is an integer, the *GetValue* procedure is overridden to convert the integer into a string:

```
function TCardValueProperty.GetValue: string;
begin
  Result := CardToString(GetOrdValue);
end;
```

The *CardToString* function is part of the code in the *TCardDeck* unit and returns a user-readable string such as *cdAceOfSpades*.

SetValue: *TCardValueProperty* must limit the property's expected values to integers between 1 and 62, as well as identifiers from *cdAceOfSpades* to *cdCardBack9*. When the user inputs a value into this property, the Object Inspector calls the *SetValue* procedure:

```
procedure TCardValueProperty.SetValue(const Value: string);
var
  NewValue: cdCardDeck;
begin
  if IdentToCard(Value, NewValue) then
    SetOrdValue(NewValue)
  else inherited SetValue(Value);
end;
```

This code checks the new value. If the check fails, i.e. *IdentToCard* returns *False*, the code calls the *TIntegerProperty.SetValue* procedure to trigger an error.

GetValues: Since *TCardValueProperty* allows a card name to be selected from a drop-down list, *TCardValueProperty* must supply that list to the Object Inspector. This is accomplished with the *GetValues* procedure:

```
procedure TCardValueProperty.GetValues(Proc: TGetStrProc);
var
  I: Integer;
begin
  for I := Low(Cards) to High(Cards) do
    Proc(Cards[I].Name);
end;
```

This is an odd procedure. It's passed a pointer to another procedure as a parameter, and this passed procedure calls for each card name in the drop-down list. When *GetValues* is finished, each item in the drop-down list is registered with the Object Inspector.

Registering the *TCardValueProperty*

With *TCardValueProperty* declared and its code written, Delphi must be told that the new property editor exists. This is done in *TCardDeck*'s *Register* procedure:

```
procedure Register;
begin
  RegisterComponents('DI',[TCardDeck]);
  RegisterPropertyEditor(TypeInfo(cdCardDeck),TCardDeck,
                         'Value',TCardValueProperty);
end;
```

RegisterPropertyEditor informs Delphi of the new property editor type. The first parameter is information about the property's *Type*. The second parameter indicates that the *TCardValueProperty* can only be applied to *TCardDeck* components, and the third tells Delphi the *TCardValueProperty* can only be applied to properties named *Value*. This parameter can remain blank to use *TCardValueProperty* with properties other than *Value*. The final parameter is the class of property editor.

Conclusion

Several years ago I received a little hand-held US$20 poker game as a Christmas gift. It seemed to me a reasonable experiment to recreate this game on a US$3000 90MHz Pentium. This exercise is made easy with the help of Delphi and the *TCardDeck* component. Download the program and have some fun. We'll explore the game's inner workings in a future article. If you've never written a component before, give it a try. To see your custom component on the Component Palette, and set its custom properties in the Object Inspector — it's a real feeling of accomplishment. ∆

The CardDeck component and sample poker game referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\DEC\DI9512DF.

David Faulkner is a developer with Silver Software in Kula, HI. He is also Contributing Editor to *Paradox Informant*, and co-author of *Using Delphi: Special Edition* (Que, 1995). Mr Faulkner can be reached at (808) 878-2714, or on CompuServe at 76116,3513.

Begin Listing One — Cardsp.DPR

```
program Cardsp;

uses
  Forms,
  Cardu in 'CARDU.PAS' { Form1 };

{$R *.RES}

begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

End Listing One

Begin Listing Two — CardDeck.PAS

```
{$R CARDDECK.RES}

unit CardDeck;

interface

uses SysUtils, WinProcs, Classes, Graphics,
  Controls, DsgnIntf;

const
  cdAceofSpades     =  1;
  cd2ofSpades       =  2;
  cd3ofSpades       =  3;
  cd4ofSpades       =  4;
  cd5ofSpades       =  5;
  cd6ofSpades       =  6;
  cd7ofSpades       =  7;
  cd8ofSpades       =  8;
  cd9ofSpades       =  9;
  cd10ofSpades      = 10;
  cdJackofSpades    = 11;
  cdQueenofSpades   = 12;
  cdKingofSpades    = 13;

  cdAceofHearts     = 14;
  cd2ofHearts       = 15;
  cd3ofHearts       = 16;
  cd4ofHearts       = 17;
  cd5ofHearts       = 18;
  cd6ofHearts       = 19;
  cd7ofHearts       = 20;
  cd8ofHearts       = 21;
  cd9ofHearts       = 22;
  cd10ofHearts      = 23;
  cdJackofHearts    = 24;
  cdQueenofHearts   = 25;
  cdKingofHearts    = 26;
  cdAceofClubs      = 27;
  cd2ofClubs        = 28;
  cd3ofClubs        = 29;
  cd4ofClubs        = 30;
  cd5ofClubs        = 31;
  cd6ofClubs        = 32;
  cd7ofClubs        = 33;
  cd8ofClubs        = 34;
  cd9ofClubs        = 35;
  cd10ofClubs       = 36;
  cdJackofClubs     = 37;
  cdQueenofClubs    = 38;
  cdKingofClubs     = 39;

  cdAceofDiamonds   = 40;
  cd2ofDiamonds     = 41;
  cd3ofDiamonds     = 42;
  cd4ofDiamonds     = 43;
  cd5ofDiamonds     = 44;

  cd6ofDiamonds     = 45;
  cd7ofDiamonds     = 46;
  cd8ofDiamonds     = 47;
  cd9ofDiamonds     = 48;
  cd10ofDiamonds    = 49;
  cdJackofDiamonds  = 50;
  cdQueenofDiamonds = 51;
  cdKingofDiamonds  = 52;

  CdJoker           = 53;
  cdCardBack1       = 54;
  cdCardBack2       = 55;
  cdCardBack3       = 56;
  cdCardBack4       = 57;
  cdCardBack5       = 58;
  cdCardBack6       = 59;
  cdCardBack7       = 60;
  cdCardBack8       = 61;
  cdCardBack9       = 62;

  cdFaceCards = [1,11..14,24..27,37..40,50..52];

  cdSpades      =  [1..13];
  cdHearts      =  [14..26];
  cdClubs       =  [27..39];
  cdDiamonds    =  [40..52];

  cdBlackCards  =  [1..13,27..39];
  cdRedCards    =  [14..26,40..52];

  cdSpade    = 1;
  cdHeart    = 2;
  cdClub     = 3;
  cdDiamond  = 4;
  cdBlack    = 1;
  cdRed      = 2;

type
  cdCardDeck = 1..62;  { 52 cards + 1 joker + 9 backs }

{ Create custom property editor that accepts both
  card names or card numbers more easily }
type
  TCardEntry = record
    Value: cdCardDeck;
    Name: string[18];
  end;

{ This array maps card numbers to card names }
const
  Cards: array[1..62] of TCardEntry = (
    (Value: cdAceofSpades;      Name:  'cdAceofSpades'),
    (Value: cd2ofSpades;       Name: 'cd2ofSpades'),
    (Value: cd3ofSpades;       Name: 'cd3ofSpades'),
    (Value: cd4ofSpades;       Name: 'cd4ofSpades'),
    (Value: cd5ofSpades;       Name: 'cd5ofSpades'),
    (Value: cd6ofSpades;       Name: 'cd6ofSpades'),
    (Value: cd7ofSpades;       Name: 'cd7ofSpades'),
    (Value: cd8ofSpades;       Name: 'cd8ofSpades'),
    (Value: cd9ofSpades;       Name: 'cd9ofSpades'),
    (Value: cd10ofSpades;      Name: 'cd10ofSpades'),
    (Value: cdJackofSpades;    Name: 'cdJackofSpades'),
    (Value: cdQueenofSpades;   Name: 'cdQueenofSpades'),
    (Value: cdKingofSpades;    Name: 'cdKingofSpades'),

    (Value: cdAceofHearts;     Name: 'cdAceofHearts'),
    (Value: cd2ofHearts;       Name: 'cd2ofHearts'),
    (Value: cd3ofHearts;       Name: 'cd3ofHearts'),
    (Value: cd4ofHearts;       Name: 'cd4ofHearts'),
    (Value: cd5ofHearts;       Name: 'cd5ofHearts'),

    (Value: cd6ofHearts;       Name: 'cd6ofHearts'),
    (Value: cd7ofHearts;       Name: 'cd7ofHearts'),
    (Value: cd8ofHearts;       Name: 'cd8ofHearts'),
    (Value: cd9ofHearts;       Name: 'cd9ofHearts'),
    (Value: cd10ofHearts;      Name: 'cd10ofHearts'),
    (Value: cdJackofHearts;    Name: 'cdJackofHearts'),
    (Value: cdQueenofHearts;   Name: 'cdQueenofHearts'),
    (Value: cdKingofHearts;    Name: 'cdKingofHearts'),

    (Value: cdAceofClubs;      Name: 'cdAceofClubs'),
    (Value: cd2ofClubs  ;      Name: 'cd2ofClubs'),
    (Value: cd3ofClubs  ;      Name: 'cd3ofClubs'),
    (Value: cd4ofClubs  ;      Name: 'cd4ofClubs'),
    (Value: cd5ofClubs  ;      Name: 'cd5ofClubs'),
    (Value: cd6ofClubs  ;      Name: 'cd6ofClubs'),
    (Value: cd7ofClubs  ;      Name: 'cd7ofClubs'),
    (Value: cd8ofClubs  ;      Name: 'cd8ofClubs'),
    (Value: cd9ofClubs  ;      Name: 'cd9ofClubs'),
    (Value: cd10ofClubs;       Name: 'cd10ofClubs'),
    (Value: cdJackofClubs;     Name: 'cdJackofClubs'),
    (Value: cdQueenofClubs;    Name: 'cdQueenofClubs'),
    (Value: cdKingofClubs;     Name: 'cdKingofClubs'),

    (Value: cdAceofDiamonds;   Name: 'cdAceofDiamonds'),
    (Value: cd2ofDiamonds;     Name: 'cd2ofDiamonds'),
    (Value: cd3ofDiamonds;     Name: 'cd3ofDiamonds'),
    (Value: cd4ofDiamonds;     Name: 'cd4ofDiamonds'),
    (Value: cd5ofDiamonds;     Name: 'cd5ofDiamonds'),
    (Value: cd6ofDiamonds;     Name: 'cd6ofDiamonds'),
    (Value: cd7ofDiamonds;     Name: 'cd7ofDiamonds'),
    (Value: cd8ofDiamonds;     Name: 'cd8ofDiamonds'),
    (Value: cd9ofDiamonds;     Name: 'cd9ofDiamonds'),
    (Value: cd10ofDiamonds;    Name: 'cd10ofDiamonds'),
    (Value: cdJackofDiamonds;  Name: 'cdJackofDiamonds'),
    (Value: cdQueenofDiamonds; Name: 'cdQueenofDiamonds'),
    (Value: cdKingofDiamonds;  Name: 'cdKingofDiamonds'),

    (Value: CdJoker;           Name: 'CdJoker'),
    (Value: cdCardBack1;       Name: 'cdCardBack1'),
    (Value: cdCardBack2;       Name: 'cdCardBack2'),
    (Value: cdCardBack3;       Name: 'cdCardBack3'),
    (Value: cdCardBack4;       Name: 'cdCardBack4'),
    (Value: cdCardBack5;       Name: 'cdCardBack5'),
    (Value: cdCardBack6;       Name: 'cdCardBack6'),
    (Value: cdCardBack7;       Name: 'cdCardBack7'),
    (Value: cdCardBack8;       Name: 'cdCardBack8'),
    (Value: cdCardBack9;       Name: 'cdCardBack9') );

type
   { Create a custom property editor for cdCardDeck --
     allows user to type in name of card, type in number
     of card, or select card from list }
  TCardValueProperty = class(TIntegerProperty)
```

```
  public
    function GetAttributes: TPropertyAttributes; override;
    function GetValue: string; override;
    procedure GetValues(Proc: TGetStrProc); override;
    procedure SetValue(const Value: string); override;
  end;

{ Now create TCardDeck }
type
  TCardDeck = class(TGraphicControl)

  private
    FValue: cdCardDeck;  { value of card being diplayed }
    FStretch: Boolean;   { stretch bitmap to fit client? }
    procedure SetValue(Value: cdCardDeck);
    procedure SetStretch(Value: Boolean);

  protected
    procedure Paint; override;

  public
    constructor Create(AOwner: TComponent); override;

  published
    property Value: cdCardDeck read FValue
      write SetValue default cdAceofSpades;
    property Stretch: Boolean read FStretch
      write SetStretch default False;
   { surface drag and drop events so programmers can
     make cards that drag }
    property DragCursor;
    property DragMode;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnClick;
    property OnDblClick;

  end;

function CardToIdent(Card: cdCardDeck;
                     var Ident: string): Boolean;
function IdentToCard(const Ident: string;
                     var Card: cdCardDeck): Boolean;
function CardToString(Card: cdCardDeck): string;
procedure Register;

{ Utility routines for the programmer }
function IsCardRed(Value:cdCardDeck): Boolean;
function IsCardBlack(Value:cdCardDeck): Boolean;
function IsFaceCard(Value:cdCardDeck): Boolean;

function AreCardsSameColor(Value1,
                           Value2: cdCardDeck) : Boolean;
function AreCardsSameSuit(Value1,
                          Value2: cdCardDeck) : Boolean;
function AreCardsSameValue(Value1,
                           Value2: cdCardDeck) : Boolean;

function CardColor(Value: cdCardDeck): integer;
function CardSuit(Value: cdCardDeck): integer;
function CardValue(Value: cdCardDeck): integer;

implementation

{ Put cdCardDeck on palette, register custom property editor }
procedure Register;
begin
  RegisterComponents('DI', [TCardDeck]);
  RegisterPropertyEditor(TypeInfo(cdCardDeck),TCardDeck,
    'Value',TCardValueProperty);
end;

{ paint the card face bitmap on the screen }
procedure TCardDeck.Paint;
var
  BitMap:TBitMap;
  BackGroundColor:TColor;
  FacePChar: array [0..25] of char;
begin
 { convert value to Pchar so can call Windows API }

  strpcopy(FacePChar,CardToString(Value));
  BitMap := TBitMap.create;
  try
    BitMap.handle := LoadBitMap(HInstance,FacePChar);
    if not BitMap.empty then begin
      if Stretch then
        canvas.stretchdraw(clientrect,BitMap)

      else begin
       { draw clear rectangle to see color of background }
        canvas.brush.style:=bsclear;
        canvas.pen.style:=psclear;
        canvas.rectangle(0,0,0,0);
        BackGroundColor:=canvas.pixels[0,0];
        { now draw the bitmap }
        Canvas.draw(0,0,BitMap);
        { set corners of bitmap to background color }
        canvas.pixels[0,0]:=BackGroundColor;
        canvas.pixels[1,0]:=BackGroundColor;
        canvas.pixels[0,1]:=BackGroundColor;

        canvas.pixels[bitmap.width-1,0]  := BackGroundColor;
        canvas.pixels[bitmap.width-2,0]  := BackGroundColor;
        canvas.pixels[bitmap.width-1,1]  := BackGroundColor;

        canvas.pixels[bitmap.width-1,
                      bitmap.height-1]  := BackGroundColor;
        canvas.pixels[bitmap.width-2,
                      bitmap.height-1]  := BackGroundColor;
        canvas.pixels[bitmap.width-1,
                      bitmap.height-2]  := BackGroundColor;

        canvas.pixels[0,bitmap.height-1]:= BackGroundColor;
        canvas.pixels[1,bitmap.height-1]:= BackGroundColor;
        canvas.pixels[0,bitmap.height-2]:= BackGroundColor;
      end;
    end;
  finally
    BitMap.free;
  end;
end;

constructor TCardDeck.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Height := 96;
  Width  := 71;
  Value  := cdAceofSpades;
end;

procedure TCardDeck.SetValue(Value: cdCardDeck);
begin
  if Value<>FValue then
  begin
    FValue := Value;
    repaint;
  end;
end;
```

```
procedure TCardDeck.SetStretch(Value: Boolean);
begin
  if Value<>FStretch then
  begin
    if not Value then begin
      Height  := 96;
      Width   := 71;
    end;
    FStretch := Value;
    Invalidate;
  end;
end;

{ ** TCardValueProperty custom property editor routines ** }
function TCardValueProperty.GetAttributes:
  TPropertyAttributes;
begin
  Result := [paValueList, paMultiSelect];
end;

function TCardValueProperty.GetValue: string;

begin
  Result := CardToString(GetOrdValue);
end;

procedure TCardValueProperty.GetValues(Proc: TGetStrProc);
var
  I: Integer;
begin
  for I := Low(Cards) to High(Cards) do Proc(Cards[I].Name);
end;

procedure TCardValueProperty.SetValue(const Value: string);
var
  NewValue: cdCardDeck;
begin
  if IdentToCard(Value, NewValue) then
    SetOrdValue(NewValue)
  else inherited SetValue(Value);
end;
{ end of TCardValueProperty custom property editor routines }

function CardToString(Card: cdCardDeck): string;
begin
  if not CardToIdent(Card, Result) then
    FmtStr(Result, '$%.8x', [Card]);
end;

function CardToIdent(Card: cdCardDeck;
                     var Ident: string): Boolean;
var
  I: Integer;
begin
  Result := False;
  for I := Low(Cards) to High(Cards) do

    if Cards[I].Value = Card then
    begin
      Result := True;
      Ident := Cards[I].Name;
      Exit;
    end;
end;

function IdentToCard(const Ident: string;
                     var Card: cdCardDeck): Boolean;
var
  I: Integer;
begin
  Result := False;
  for I := Low(Cards) to High(Cards) do
```

```
    if CompareText(Cards[I].Name, Ident) = 0 then
    begin
      Result := True;
      Card   := Cards[I].Value;
      Exit;
    end;
end;

{ ** utility routines for the programmer ** }
function IsCardRed(value: cdCardDeck): Boolean;
begin
  Result := value in cdRedCards;
end;

function IsCardBlack(value: cdCardDeck): Boolean;
begin
  Result := value in cdBlackCards;
end;



function AreCardsSameColor(Value1,
                           Value2: cdCardDeck): Boolean;
begin
  Result := CardColor(Value1) = CardColor(Value2);
end;

function AreCardsSameSuit(Value1,
                          Value2: cdCardDeck) : Boolean;
begin
  Result := CardSuit(Value1) = CardSuit(Value2);
end;

function AreCardsSameValue(Value1,
                           Value2: cdCardDeck) : Boolean;
begin
  Result := CardValue(Value1) = CardValue(Value2);
end;

function IsFaceCard(Value: cdCardDeck):Boolean;
begin
  Result := value in cdFaceCards;
end;

function CardColor(Value: cdCardDeck):integer;
begin
  result:=0;
  if value in cdRedCards then Result := cdRed;
  if value in cdBlackCards then Result := cdBlack;
end;

function CardSuit(Value: cdCardDeck):integer;
begin
  result:=0;
  if value in cdSpades    then Result := cdSpade;
  if value in cdHearts    then Result := cdHeart;
  if value in cdClubs     then Result := cdClub;
  if value in cdDiamonds  then Result := cdDiamond;
end;

function CardValue(Value: cdCardDeck): integer;
var
  Suit: integer;
begin
  result := 0;
  Suit   := CardSuit(Value);
  if Suit<>0 then result := Value-(Suit-1)*13
end;

end.
```

End Listing Two

Begin Listing Three — Cardu.PAS

```
unit Cardu;

interface

uses WinTypes, WinProcs, Classes, Graphics, Forms,
Controls, StdCtrls, SysUtils, Dialogs, mmSystem, ExtCtrls,
Carddeck, Buttons;

type
  TForm1 = class(TForm)
    DealOrDraw: TButton;
    Hold1: TLabel;
    Hold2: TLabel;
    Hold3: TLabel;
    Hold4: TLabel;
    Hold5: TLabel;
    HoldButton1: TButton;
    HoldButton2: TButton;
    HoldButton3: TButton;
    HoldButton4: TButton;
    HoldButton5: TButton;
    Timer1: TTimer;
    HoldOrDraw: TLabel;
    Shape1: TShape;
    Pot: TLabel;
    CardDeck1: TCardDeck;
    CardDeck2: TCardDeck;
    CardDeck3: TCardDeck;
    CardDeck4: TCardDeck;
    CardDeck5: TCardDeck;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    Label9: TLabel;
    Label10: TLabel;
    Label11: TLabel;
    Label12: TLabel;
    Label13: TLabel;
    Label14: TLabel;
    Label15: TLabel;
    Label16: TLabel;
    SoundButton: TBitBtn;

    procedure DealOrDrawClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure HoldButton1Click(Sender: TObject);
    procedure SoundButtonClick(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);

end;

type
  TCardHand = array[1..5] of byte;

var
  Form1: TForm1;
  aUsedCards: array[1..52] of Boolean;{ card dealt yet? }
  aUsedTag: array[1..5] of Boolean;   { card scored yet? }
  bFirstDeal: Boolean;                { opening deal? }
  aHand: TCardHand;                   { cards in hand }
  iWinnings: Longint;                 { loot from last win }
  bBlinker: Boolean;                  { blink winning cards }
  bNextSong: Boolean;                 { song to play on win }
  bSound: Boolean;                    { is sound turned on? }
  bJustWon: Boolean;                  { did user just win? }
```

```
implementation

{$R *.DFM}

procedure TForm1.DealOrDrawClick(Sender: TObject);
var
  x,y,z: byte;           { general purpose }
  iCurrentCount: byte;   { # of matching cards in hand }
  iCardCount: byte;      { used to step through 52 cards }
  bTwoOfAKind,           { two of a kind }
  bTwoPair,              { two pair }
  bThreeOfAKind,         { three of a kind }
  bFourOfAKind,          { four of a kind }
  bJacksOrBetter,        { a pair of jacks or better }
  bFlush,                { a flush }
  bStraight,             { a straight }
  bStraightFlush,        { a straight flush }
  bFullHouse,            { a full house }
  bRoyalFlush: Boolean;  { is this a really lucky player }
  aFaceCopy: TCardHand;  { a copy of the current hand }

begin
  if bSound then
    sndPlaySound('Deal.wav',3);

  Timer1.enabled := False;
  if iWinnings<>0 then begin
    Pot.caption:=IntToStr(StrToInt(Pot.caption)+iWinnings);
    iWinnings:=0;
  end;

  if bFirstDeal and (StrToInt(Pot.caption)=0) then begin
    MessageDlg('Busted, You have no money left to bet!',
               mtWarning,[mbOK], 0);
    exit;
  end;

  if bFirstDeal then begin
    for x:=1 to 52 do          { initialize deck }
      aUsedCards[x] := False;
    DealOrDraw.Caption := '&Draw';
    HoldOrDraw.Caption := 'Hold or Draw';
    Pot.caption := IntToStr(StrToInt(Pot.caption)-5);
    { Pick out 5 new cards }
    for iCardCount:=1 to 5 do begin
      TButton(FindComponent('HoldButton' +
              inttostr(iCardCount))).enabled:=True;
      TLabel(FindComponent('Hold' +
             inttostr(iCardCount))).visible:=False;
      repeat { Pick a random card }
        x := random(51)+1;
      until not aUsedCards[x];
      TCardDeck(FindComponent('CardDeck' +
                inttostr(iCardCount))).value := x;
      if bSound then sndPlaySound('CARD.WAV',2);
      aUsedCards[x] := True;
      aHand[iCardCount] := x;
      end;
    end
  else
    begin
      HoldOrDraw.caption := '';
      for iCardCount := 1 to 5 do
        begin  { deal new cards for non held cards }
          TButton(FindComponent('HoldButton'+
                  inttostr(iCardCount))).enabled := False;
          if Tlabel(FindComponent('Hold'+
                    inttostr(iCardCount))).visible then
            continue;
          repeat
            x := random(51)+1;
          until not aUsedCards[x];
```

```
     TCardDeck(FindComponent('CardDeck'+
               inttostr(iCardCount))).value:=x;
     if bSound then sndPlaySound('CARD.WAV',2);
     aUsedCards[x]:=True;
     aHand[iCardCount]:=x;
     Tlabel(FindComponent('Hold'
            inttostr(iCardCount))).visible := False;
   end;
 DealOrDraw.Caption:='&Deal';

 { Now score the whole thing }
 bTwoOfAKind     := False;
 bTwoPair        := False;
 bThreeOfAKind   := False;
 bFourOfAKind    := False;
 bJacksOrBetter  := False;
 bFlush          := True;
 bStraight       := True;
 bFullHouse      := False;
 for x:=1 to 5 do
   aUsedTag[x] := False;

 { compare face value of each card to other
   cards counting matches }
 for x:=1 to 5 do begin
   if aUsedtag[x] then continue;
   iCurrentCount:=0;
   for y:=x+1 to 5 do begin
     if (CardValue(aHand[x])=CardValue(aHand[y])) and
        (not aUsedTag[y]) then begin
       aUsedTag[y] := True;
       aUsedTag[x] := True;
       inc(iCurrentCount);
       if isFaceCard(aHand[x]) then
         bJacksOrBetter := True;
     end;
   end;  { end of inner for loop }

   case iCurrentCount of
     3: bFourOfAKind  := True;
     2: bThreeOfAKind := True;
     1: if bTwoOfAKind then
          bTwoPair := True
        else
          bTwoOfAKind := True;
   end;  { End of CurrentCount case }
 end;  { End of outer loop }

{ it's a flush if CardSuit of all five cards is the same }
for x:=2 to 5 do
  bFlush := (CardSuit(aHand[1])=
             CardSuit(aHand[x])) and bFlush;

{ now detect a straight, start by creating a sorted
 copy of hand using copy so can 'blink' winning cards }
for x:=1 to 5 do
  afacecopy[x] := CardValue(ahand[x]);

{ just a little bubble sort }
for x:= 1 to 4 do begin
  for y:=x to 5 do begin
    if afacecopy[y] > afacecopy[x] then begin
      z := afacecopy[x];
      afacecopy[x] := afacecopy[y];
      afacecopy[y] := z;
    end;
  end;
end;

{ if it’s a straight then each consecutive card will
 have face value of one more than previous card }
for x:=2 to 5 do
```

```
      if afacecopy[x]<>afacecopy[x-1]-1 then
         bStraight:=False;

       bFullHouse := bThreeOfAKind and bTwoOfAKind;
       bStraightFlush := bStraight and bFlush;
       bRoyalFlush :=
         bFlush and bStraight and (CardValue(ahand[1])=14);

       if bRoyalFlush then iWinnings:=3000 else
       if bStraightFlush then iWinnings:=250 else
       if bFourOfAKind then iWinnings:=125 else
       if bFullHouse then iWinnings:=40 else
       if bFlush then iWinnings:=25 else
       if bStraight then iWinnings:=20 else
       if bThreeOfAKind then iWinnings:=15 else
       if bTwoPair then iWinnings:=10 else
       if bTwoOfAKind and bJacksOrBetter then
         iWinnings:=5 else iWinnings:=0;

      { if winnings are >20 then all five cards are
        involved, otherwise aUsedTag is already set }


       if iWinnings >= 20 then
         for x:=1 to 5 do
           aUsedTag[x] := True;

      { if user won then set up timer to give the user
        the loot and play sounds }
       if Boolean(iWinnings) then begin
         if bSound then
           sndPlaySound('Win.wav',2);
         bJustWon := True;
         Timer1.Interval := 200;
         Timer1Timer(Tobject(TForm1));
         Timer1.enabled := True;

       end;
     end;  { End of if second hand }

  bFirstDeal := not bFirstDeal;
end;

procedure TForm1.FormCreate(Sender: TObject);

begin
  bSound       := True;
  bNextSong    := False;
  bBlinker     := False;
  Pot.caption  := '100';
  Randomize;
  bFirstDeal   := True;
end;

procedure TForm1.HoldButton1Click(Sender: TObject);
var
  s: string[1];
  i: byte;
begin
  if bFirstDeal then exit;
   { get the last character of component that called this
     routine, the last character is either 1,2,3,4, or 5 }
  s := (sender as Tcomponent).name[length(
       (sender as Tcomponent).name)];

   { Each card has a label with the word 'Hold' as its
     caption above the card, toggle the visibility of that
     label when user clicks Hold button or card }
  TLabel(FindComponent('Hold'+s)).visible :=
    not TLabel(FindComponent('Hold'+s)).visible;

  if bSound then begin

    if TLabel(FindComponent('Hold'+s)).visible then
      sndPlaySound('HOLD.WAV',3)
    else
      sndPlaySound('UNHOLD.WAV',3);
  end;

  ActiveControl:=DealOrDraw;

end;

procedure TForm1.SoundButtonClick(Sender: TObject);
begin
  bSound := Not bSound;
  ActiveControl := DealOrDraw;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var
  x: byte;
begin
  if bBlinker then begin
    for x:=1 to 5 do
      if aUsedTag[x] then
      TCardDeck(FindComponent('CardDeck'+
                inttostr(x))).value:=cdCardBack5;
    end
  else begin
    for x:=1 to 5 do
      if aUsedTag[x] then
        TCardDeck(FindComponent('CardDeck'+
          inttostr(x))).value:=aHand[x];
    end;
    bBlinker := not bBlinker;

  if iWinnings>0 then begin
  if bSound then
    sndPlaySound('Coin.wav',3);
    Pot.caption := IntToStr(StrToInt(Pot.caption)+1);
    iWinnings   := iWinnings-1;
  end
else
  if bJustWon then begin
    Timer1.Interval := 300;
    bJustWon := False;
    if bSound then begin
      bNextSong:= not bNextSong;
      if bNextSong then
        sndPlaySound('Money.wav',3)
      else
        sndPlaySound('Happy.wav',3);
    end;
  end;
end;

end.
```

End Listing Three

SIGHTS & SOUNDS

DELPHI / OBJECT PASCAL

By *Sedge Simons, Ph.D.*

Hotspots

Creating Mouse-Sensitive Areas on Your Forms

Creating a full-featured graphical user interface for applications involving maps, drawings, or charts introduces a challenge for the Delphi developer. Such applications usually require *hotspots*, i.e. on-screen areas or objects that are sensitive to mouse events. This article introduces regions, a Windows API data structure that enables you to create hotspots of any size or shape.

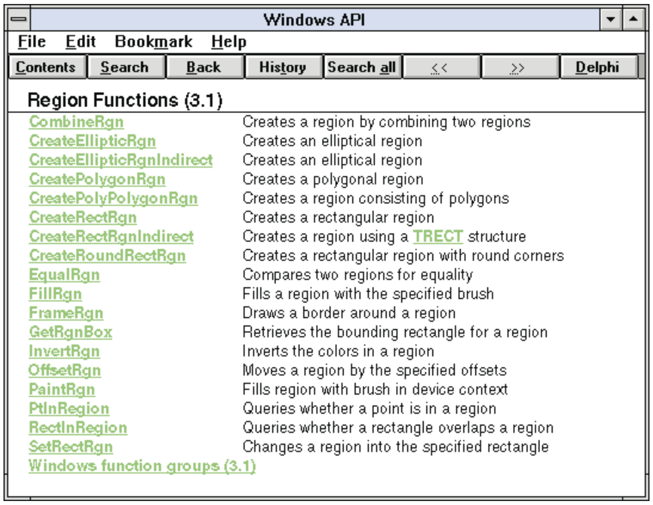
Interacting with an Image

Most of us have used drawing, charting, or mapping software that allows interaction with an image. For instance, a drawing tool may be used to create a circle that can then be selected and its properties modified. In a geographical information system (GIS), we can click on a map's feature to view information about it stored in a database.

These examples illustrate hotspots — areas of an image that are sensitive to mouse events. These hotspots usually coincide with what Delphi developers consider objects: shapes on a drawing or features on a map.

The simplest way to create hotspots in a Delphi application is to place components on the image. [For tips on hotspots, see David Rippy's "At Your Fingertips" in the August 1995 *Delphi Informant*.] These components may be visible objects or transparent ones that overlap part of another image. This technique has two significant drawbacks:

- Although components can appear to be any shape, they always cover a rectangular area, and are sensitive to mouse events anywhere in that rectangle.
- Components that are part of the visible image can require special programming when printing.



Region Functions (3.1)

CombineRgn	Creates a region by combining two regions
CreateEllipticRgn	Creates an elliptical region
CreateEllipticRgnIndirect	Creates an elliptical region
CreatePolygonRgn	Creates a polygonal region
CreatePolyPolygonRgn	Creates a region consisting of polygons
CreateRectRgn	Creates a rectangular region
CreateRectRgnIndirect	Creates a region using a TRECT structure
CreateRoundRectRgn	Creates a rectangular region with round corners
EqualRgn	Compares two regions for equality
FillRgn	Fills a region with the specified brush
FrameRgn	Draws a border around a region
GetRgnBox	Retrieves the bounding rectangle for a region
InvertRgn	Inverts the colors in a region
OffsetRgn	Moves a region by the specified offsets
PaintRgn	Fills region with brush in device context
PtInRegion	Queries whether a point is in a region
RectInRegion	Queries whether a rectangle overlaps a region
SetRectRgn	Changes a region into the specified rectangle

Windows function groups (3.1)

Figure 1: Region functions as defined in Delphi's Windows API on-line help.

Regions

Regions are a flexible means of creating hotspots. They are independent of any on-screen visual image and are stored as data structures. The relationship between regions and an image is decided by the programmer. The Windows API includes a full suite of functions for creating and manipulating regions of practically any size or shape. Under Windows 3.1, the only restriction is that each region is limited to a size of 32,767 by 32,767 units, or 64KB of memory, whichever is smaller.

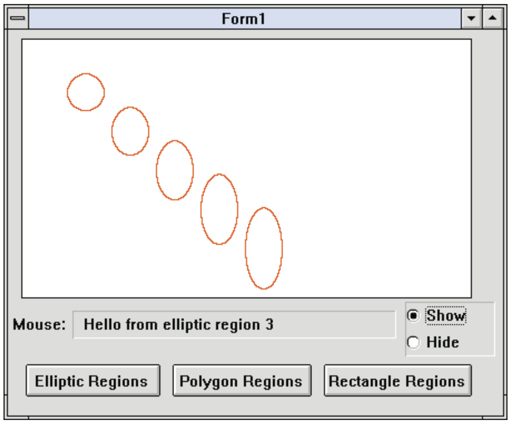
A region can be viewed as a set of points. Normally, these points will coincide with pixels in an image. A function in the Windows API tests a point to determine if it's within a specified region. This function can also map mouse events from an on-screen image to a set of previously defined regions. Figure 1 shows a list of Windows API region functions.

Using Regions

The typical steps for using regions are to:

- allocate memory
- create and store regions
- respond to events
- free memory

To illustrate these steps, the example application defines regions that correspond to various shapes shown in a *TImage* object. It then responds to *MouseDown* events in the image (see Figure 2).



Mouse: Hello from elliptic region 3

● Show
○ Hide

Figure 2: Our sample application. The shapes shown in the *TImage* object correspond to regions defined by the code in Listing Five.

Allocate memory. When a region is created, the Windows API allocates memory for the data structure, then returns a handle. It's your responsibility to track these handles in the Delphi application, and to free corresponding objects when they become unnecessary.

Create and store regions. Typically, information about the region is stored with the region handle. For example, if each region represents a record in a table, storing that record's key with the region handle allows information to be displayed from the table in response to a mouse click. One way to do this is to define a record object that contains the region handle and any other useful fields.

In this example, the record is defined as:

```
arec = record
          rgn: HRgn;
          id: string;
       end;
```

In this record `rgn` is the region handle, and `id` is a simple string we create and store as a short description of the region.

The records of each region can be stored in an array or in a *TList*. An array may be more convenient, but it requires space in the 64KB of memory used for the data and stack segments (a Windows 3.1 limitation). In this example, the records are stored in a *TList*.

```
procedure TForm1.ButtonEllipseClick(Sender: TObject);
var
  x1,x2,y1,y2: Integer;
begin
  FreeRegions;
  ImageHandle := Image1.Canvas.Handle;
  Nregions := 5;
  for i := 1 to Nregions do
    begin
      x1 := i*40;
      x2 := x1 + 35;
      y1 := i*30;
      y2 := x1 + 25;
      New(RegionRecord);
      RegionRecord^.rgn := CreateEllipticRgn(x1,y1,x2,y2);
      RegionRecord^.id  := ' Hello from elliptic region ' +
                           IntToStr(i);
      RegionList.Add(RegionRecord);
    end;
  ShowOrHide(Sender);
  Label1.Caption := 'Elliptic regions created.'
end;
```

Figure 3: The custom *ButtonEllipseClick* procedure.

The Object Pascal code shown in Figure 3 creates five regions when the user clicks the button labeled **Elliptic Regions**. A record is stored for each region. Similar code, shown in Listing Five on page 35, creates regions of different shapes in response to clicks on the other buttons.

It's important to remember that the regions are completely independent of what is shown on screen. In this example, the *ShowOrHide* procedure is called after the regions are created. If the **Show** radio button is checked, *ShowOrHide* will draw shapes on the *TImage* representing the regions. Otherwise the image remains blank. Regardless of what the image displays, the regions exist and are ready to use.

Respond to events. From a practical standpoint, something on screen will almost always coincide with a region you create. This is done by executing independent drawing commands when creating a region, or creating it first and then drawing a frame around it in a specified device context. The latter method represents the most versatile code and is implemented in the *ShowOrHide* procedure (see Figure 4).

Interacting with regions means responding to events. In Figure 4, we want to retrieve data from the region record in response to mouse clicks in the *TImage* control. Therefore, the code is attached to the *TImage MouseDown* event that will attempt to map the coordinates of the mouse event to one of the regions we have defined.

The Windows API function, **PtInRegion**, maps events to regions. It returns *True* if the specified point is inside the specified region. In response to a *MouseDown* event on *Image1*, we simply iterate through the list of regions, testing each to see if the event occurred inside it (see Figure 5).

```
procedure TForm1.ShowOrHide(Sender: TObject);
begin
  Bitmap         := Tbitmap.Create;
  Bitmap.Width   := Image1.Width;
  BitMap.Height  := Image1.Height;
  Image1.Picture.Graphic := Bitmap;
  if RadioButtonShow.Checked then
    begin
      ImageHandle := Image1.Canvas.Handle;
        for i := 0 to RegionList.Count - 1 do
          begin
            RegionRecord := RegionList.Items[i];
            FrameRgn(ImageHandle,RegionRecord^.Rgn,
                     RedBrush,1,1);
          end;
      Image1.Invalidate;
      Label1.Caption := '';
    end;
  Bitmap.Free;
end;
```

```
procedure TForm1.Image1MouseDown (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  i: Integer;
begin
  Label1.Caption := '';
  for i := 0 to RegionList.Count - 1 do
    begin
      RegionRecord := RegionList.Items[i];
        if PtInRegion(RegionRecord^.Rgn, x, y) then
          Label1.Caption := RegionRecord^.id;
    end;
end;
```

Figure 4 (Top): The custom *ShowOrHide* procedure. **Figure 5 (Bottom):** The custom *Image1MouseDown* procedure.

Free memory. As mentioned earlier, it's the programmer's responsibility to free the memory occupied by region objects. In this example, we must also free the *TList* object that stores the records for each region. Each region is freed with a call to *DeleteObject*. Then we can clear the *TList*. On the *FormClose* event (see Figure 6), the *TList* is freed in the usual Delphi manner. And don't free the *TList* first! You need those handles to free the regions.

Using the Application

To create regions when you first run the application, select the **Show** radio button and click one of the three region buttons. Outlines corresponding to the regions are then drawn on the image. Remember, the regions are *not* part of the image. As you click on different parts of the image, the program searches the regions to determine if the *MouseDown* event occurred inside a region. If so, it will display the string that was saved with the region record.

Select the **Hide** radio button and continue clicking on the image. The regions remain in place, and the mouse response is the same. However, nothing has been drawn on the image.

Conclusion

Regions let you create just about any type of hotspot you can imagine. Although hotspots are probably the most common application, regions and the powerful set of Windows API region functions are problem solvers in other situations, too. Just try writing a function like *PtInRegion* yourself! ∆

```
procedure TForm1.FormClose(Sender: TObject;
                           var Action: TCloseAction);
begin
  FreeRegions;
  DeleteObject(RedBrush);
  RegionList.Free;
end;

procedure TForm1.FreeRegions;
var
  i: Integer;
begin
  for i := 0 to RegionList.Count - 1 do
    begin
      RegionRecord := RegionList.Items[i];
      DeleteObject(RegionRecord^.Rgn);
    end;
  RegionList.Clear;
end;
```

Figure 6: The *FormClose* and custom *FreeRegions* procedures are responsible for releasing resources.

The demonstration project referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\DEC\DI9512SS.

Dr. Simons is a senior systems analyst at Jensen Data Systems, Inc., a Texas-based provider of database training, consulting, and application development. He writes applications and does consulting in Delphi and Paradox. You can reach him through CompuServe at 70771,75 or by calling Jensen Data Systems, Inc. at (713) 359-3311.

Begin Listing Four — Region.DPR

```
program Region;

uses
  Forms,
  Region1 in 'REGION1.PAS' {Form1};

{$R *.RES}

begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

End Listing Four

Begin Listing Five — Region1.PAS

```
unit Region1;

interface

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

type
  Prec = ^arec;
  arec = record
           rgn: HRgn;
           id:  string;
         end;
  TForm1 = class(TForm)
    ScrollBox1: TScrollBox;
    Image1: TImage;
    ButtonEllipse: TButton;
    Bevel1: TBevel;
    Label1: TLabel;
    ButtonPolygon: TButton;
    ButtonRectangle: TButton;
    Label2: TLabel;
    Bevel2: TBevel;
    RadioButtonShow: TRadioButton;
    RadioButtonHide: TRadioButton;
    procedure Image1MouseDown(Sender: TObject; Button:
      TMouseButton; Shift: TShiftState; X, Y: Integer);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
    procedure FreeRegions;
    procedure ButtonPolygonClick(Sender: TObject);
    procedure ButtonEllipseClick(Sender: TObject);
    procedure ButtonRectangleClick(Sender: TObject);
    procedure ShowOrHide(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1:         TForm1;
  RedBrush:      HBrush;
  ImageHandle:   HDC;
  Bitmap:        Tbitmap;

  Nregions:      Integer;
  RegionList:    Tlist;
  RegionRecord:  Prec;
  i:             Integer;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  RedBrush    := CreateSolidBrush(clRed);
  Nregions    := 0;
  RegionList  := Tlist.Create;
end;

procedure TForm1.FormClose(Sender: TObject;
                           var Action: TCloseAction);
begin
  FreeRegions;
  DeleteObject(RedBrush);
  RegionList.Free;
end;

procedure TForm1.FreeRegions;
var
  i: Integer;
begin
  for i := 0 to RegionList.Count - 1 do
  begin
    RegionRecord := RegionList.Items[i];
    DeleteObject(RegionRecord^.Rgn);
  end;

  RegionList.Clear;
end;
```

```
{ The next three procedures create regions in resonse to
  the buttons on the form. }
procedure TForm1.ButtonEllipseClick(Sender: TObject);
var
  x1,x2,y1,y2:  Integer;
begin
  FreeRegions;
  ImageHandle  := Image1.Canvas.Handle;
  Nregions := 5;
  for i := 1 to Nregions do
  begin
    x1 := i*40;
    x2 := x1 + 35;
    y1 := i*30;
    y2 := x1 + 25;
    New(RegionRecord);
    RegionRecord^.rgn := CreateEllipticRgn(x1,y1, x2,y2);
    RegionRecord^.id  := ' Hello from elliptic region ' +
                         IntToStr(i);
    RegionList.Add(RegionRecord);
  end;
  ShowOrHide(Sender);
  Label1.Caption := 'Elliptic regions created.'
end;

procedure TForm1.ButtonPolygonClick(Sender: TObject);
var
  PointArray: array[1..10] of TPoint;
  npoints: integer;
begin
  FreeRegions;
  ImageHandle  := Image1.Canvas.Handle;
  Nregions := 3;
  PointArray[1]  := Point(5, 5);
  PointArray[2]  := Point(100, 5);
  PointArray[3]  := Point(110, 70);
  New(RegionRecord);
  RegionRecord^.rgn := CreatePolygonRgn(PointArray, 3,
                                        WINDING);
  RegionRecord^.id  := ' Hello from polygon region 1';
  RegionList.Add(RegionRecord);
  PointArray[1]  := Point(15, 80);
  PointArray[2]  := Point(120, 100);
  PointArray[3]  := Point(20, 120);
  PointArray[4]  := Point(15, 160);
  New(RegionRecord);
  RegionRecord^.rgn := CreatePolygonRgn(PointArray, 4,
                                        WINDING);
  RegionRecord^.id  := ' Hello from polygon region 2';
  RegionList.Add(RegionRecord);
  PointArray[1]  := Point(200, 5);
  PointArray[2]  := Point(300, 10);
  PointArray[3]  := Point(280, 100);
  PointArray[4]  := Point(250, 160);
  PointArray[5]  := Point(200, 90);
  New(RegionRecord);
  RegionRecord^.rgn := CreatePolygonRgn(PointArray, 5,
                                        WINDING);
  RegionRecord^.id  := ' Hello from polygon region 3';
  RegionList.Add(RegionRecord);
  ShowOrHide(Sender);
  Label1.Caption := 'Polygon regions created.'
end;

procedure TForm1.ButtonRectangleClick(Sender: TObject);
var
  x1,x2,y1,y2,y0,i0: Integer;


begin
  FreeRegions;
  ImageHandle := Image1.Canvas.Handle;
  for i0 := 0 to 5 do
  begin
    y0 := i0*40;
    for i := 1 to 50 do
    begin
      x1 := i*7;
      x2 := x1 + 6;
      y1 := 1 + y0;
      y2 := y1 + 30;
      New(RegionRecord);
      RegionRecord^.rgn:= CreateRectRgn(x1,y1, x2,y2);
      RegionRecord^.id  := ' Hello from rectangle region ' +
                           IntToStr(i0*100+i);
      RegionList.Add(RegionRecord);
    end;
  end;
  ShowOrHide(Sender);
  Label1.Caption := 'Rectangle regions created.'
end;

{ The next procedure will iterate through the regions
  checking if a MouseDown event occured inside a region.
  If so, the caption of Label1 is changed to show the id
  of the region. }
procedure TForm1.Image1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X,Y: Integer);
var
  i: Integer;
begin
  Label1.Caption := '';
  for i := 0 to RegionList.Count - 1 do
  begin
    RegionRecord := RegionList.Items[i];
    if ptInRegion(RegionRecord^.Rgn, x, y) then
      Label1.Caption := RegionRecord^.id;
  end;
end;

{ When the Show button is selected, this procedure prepares
  a bitmap and draws outlines on it that correspond to the
  regions that have been created. }
procedure TForm1.ShowOrHide(Sender: TObject);
begin
  Bitmap                 := Tbitmap.Create;
  Bitmap.Width           := Image1.Width;
  BitMap.Height          := Image1.Height;
  Image1.Picture.Graphic := Bitmap;
  if RadioButtonShow.Checked then
    begin
      ImageHandle := Image1.Canvas.Handle;
        for i := 0 to RegionList.Count - 1 do
          begin
            RegionRecord := RegionList.Items[i];
            FrameRgn(ImageHandle, RegionRecord^.Rgn,
                      RedBrush, 1, 1);
          end;
      Image1.Invalidate;
      Label1.Caption := '';
    end;
  Bitmap.Free;
end;

end.
```

End Listing Five

DBNAVIGATOR

DELPHI / OBJECT PASCAL

By Cary Jensen, Ph.D.

Finding Your Keys

Locating Records in Paradox Tables

Most database applications provide a technique to select a particular record. For example, in a customer database, it's necessary to locate a specific record based on a customer's name, account number, street address, or some other unique piece of information.

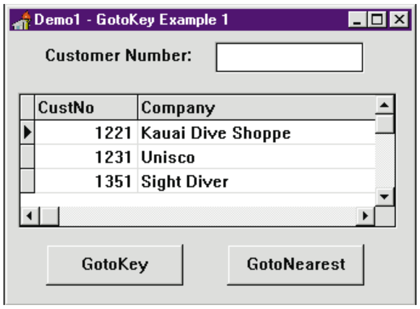
There are several techniques you can use to find a particular record in a table using a Table component. However, most of these only permit you to locate a record based on the fields of a table's index. This month's DBNavigator looks at the basics of locating records, including the use of the *TTable* methods *SetKey*, *FindKey*, and *FindNearest*, and sequential record searches.

SetKey and *GotoKey*

The first technique involves the *dsSetKey* state. A Table component's state identifies what is happening to it. For instance, if a table is in the *dsInsert* state, a record has been inserted, but has not been posted. Likewise, if a table is in the *dsEdit* state, the current record is being modified.

The table state, *dsSetKey*, is a special state that permits you to identify the index field values of the record you want to move to. You can enter this state in one of two ways, using the *SetKey* or *EditKey* method.

The Table method *SetKey* is used most commonly. When you call this method, the table is placed into a state where assignments to its individual fields are not applied to it, but are stored in the *search key buffer*. This buffer contains values only for index fields.



CustNo	Company
1221	Kauai Dive Shoppe
1231	Unisco
1351	Sight Diver

Figure 1: The DEMO1 project demonstrates using the *GotoKey* and *GotoNearest* methods on the NEWCUST.DB table.

Once values are assigned to one or more indexed fields within the *dsSetKey* state, you can call one of four methods. The method *GotoKey* is used to move to the record whose indexed field values match those in the search key buffer. *GotoNearest* is used to move to the record that most closely matches the buffer values. The last two methods, *FindKey* and *FindNearest*, do not require the *dsSetKey* state and will be discussed later in this article.

The use of *SetKey*, *GotoKey*, and *GotoNearest* is demonstrated in the project named DEMO1 shown in Figure 1. This project uses the NEWCUST.DB table and is pointed to by the alias DBDEMOS. (DBDEMOS is created by default by the Delphi installation program. NEWCUST.DB is based on the CUSTOMER.DB table that ships with Delphi.) NEWCUST.DB has been modified to include two additional indexes:

Company, a secondary, case-sensitive index similar to the ByCompany index defined by CUSTOMER.DB, and ByCityState, a two-field, case-insensitive index.

DEMO1 contains an Edit component that allows the user to enter a search value. The code attached to the button labeled **GotoKey** places the Table component associated with NEWCUST.DB, *Table1*, into the *dsSetKey* state. In this form, *Table1* is using the primary index of NEWCUST.DB, which is based on the CustNo field. If a value entered into the Edit component exactly matches one of the values in the CustNo field, clicking **GotoKey** moves you to that record. Here's the *OnClick* event handler for *Button1*:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.SetKey;
  Table1.FieldByName('CustNo').AsString := Edit1.Text;
  if Table1.GotoKey then
    DBGrid1.SetFocus;
end;
```

The first statement

```
Table1.SetKey;
```

places the table pointed to by *Table1* into the *dsSetKey* state. Next, the value from the *Text* property of the Edit component, *Edit1*, is assigned to the CustNo field. This assignment looks as though data is being written to the table. However, because of the *dsSetKey* state, it's written to the search key buffer. Next, the *GotoKey* method of *Table1* is called. This is a function that returns a Boolean value of *True* if an exact match is found, and *False* otherwise.

If the *GotoKey* method returns *True*, a call to the *SetFocus* method for the DBEdit component is executed. When you click on the button, it receives focus. In addition, if the *GotoKey* method returns *True* (meaning a match is located), the statement

```
DBGrid1.SetFocus;
```

moves the focus onto the DBGrid and places the cursor on the located record.

Button2 on DEMO1, labeled **GotoNearest**, is associated with an event handler similar to that assigned to *Button1*. The primary difference is that the *GotoNearest* method (a procedure) is used rather than *GotoKey* (a function). *GotoNearest* doesn't return a value because it's always successful in moving to the record that most closely matches the values stored in the search key buffer. Here's the *OnClick* event handler for *Button2*:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Table1.SetKey;
  Table1.FieldByName('CustNo').AsString := Edit1.Text;
  Table1.GotoNearest;
  DBGrid1.SetFocus;
end;
```

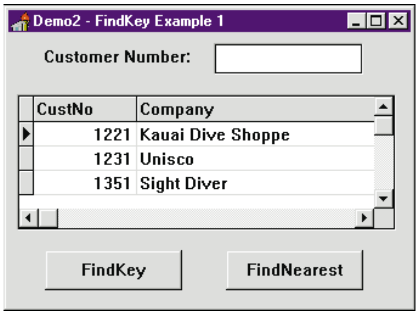
Using *FindKey*

The methods, *FindKey* and *FindNearest*, provide essentially the same capabilities as *GotoKey* and *GotoNearest*, respectively. However, there are two major differences. First, it's not necessary to place a table into the *dsSetKey* state before calling *FindKey* or *FindNearest*. And second, you do not explicitly assign values to the search key buffer — this is done by *FindKey* and *FindNearest*.

Both the *FindKey* and *FindNearest* methods require a single argument, which consists of a comma-delimited array of values to search. The first element of this array is associated with the search value for the first field of the index, the second array element with the second field, and so on.

The DEMO2 project, shown in Figure 2, performs the same searches as DEMO1, except DEMO2 uses *FindKey* and *FindNearest* instead of *GotoKey* and *GotoNearest*. For example, the following code is associated with the *OnClick* event handler for the button labeled **FindKey**:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Table1.FindKey([Edit1.Text]) then
    DBGrid1.SetFocus;
end;
```



Customer Number:

CustNo	Company
1221	Kauai Dive Shoppe
1231	Unisco
1351	Sight Diver

Figure 2: The DEMO2 project demonstrates the use of *FindKey* and *FindNearest* on the NEWCUST.DB table.

This code segment is associated with the OnClick event handler for the button labeled **FindNearest**:

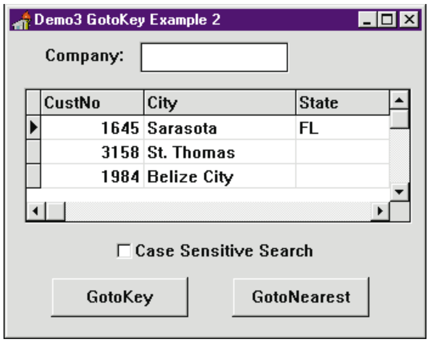
```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Table1.FindNearest([Edit1.Text]);
  DBGrid1.SetFocus;
end;
```

Case-Insensitive Searches

Whether you use *GotoKey*, *GotoNearest*, *FindKey*, or *FindNearest*, the table searched for must use an index. In the preceding examples, the index consisted of a single field, based on the field CustNo. This is the primary index, and because NEWCUST.DB is a Paradox table, it's case-sensitive and unique. (A *unique index* cannot have two records with the same combination of values on the indexed fields.)

Whether your searches are case-sensitive or not depends on the index. Searching on a case-sensitive index always produces a case-sensitive search. Similarly, if your Table component has a case-insensitive index, a case-insensitive search is performed.

This is demonstrated in DEMO3, shown in Figure 3. This project, which uses the *dsSetKey* state, permits selection of a case-sensitive index for NEWCUST.DB (using an index named Company) or a case-insensitive search (using an index named ByCompany). Each time you click on the checkbox labeled **Case Sensitive Search**, you modify the *IndexName* property of the Table component, as shown in this event handler:



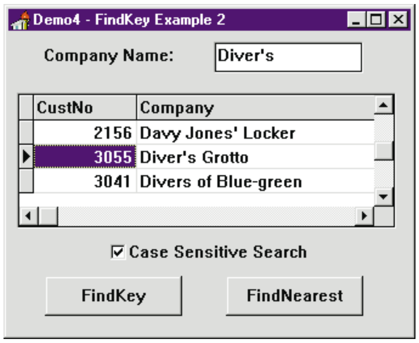
CustNo	City	State
1645	Sarasota	FL
3158	St. Thomas	
1984	Belize City	

☐ Case Sensitive Search

Figure 3: Searches are case-sensitive when the Table component being searched uses a case-sensitive index. Searches are case-insensitive when a case-insensitive index is used, as demonstrated by the DEMO3 project.

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked then
    Table1.IndexName := 'Company'
  else
    Table1.IndexName := 'ByCompany';
end;
```

The event handlers associated with the two search buttons contain the same code as DEMO1.



Company Name: Diver's

CustNo	Company
2156	Davy Jones' Locker
3055	Diver's Grotto
3041	Divers of Blue-green

☑ Case Sensitive Search

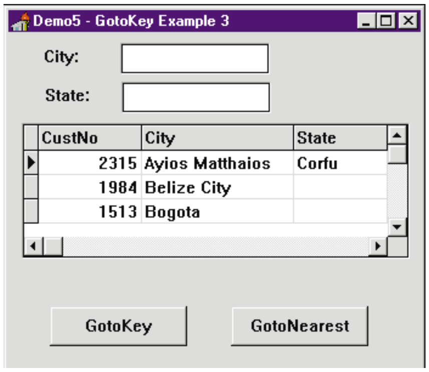
Figure 4: The DEMO4 project demonstrates using *FindKey* and *FindNearest* in case-sensitive and case-insensitive searches.

The DEMO4 project, shown in Figure 4, demonstrates case-sensitive and case-insensitive searches with *FindKey* and *FindNearest*.

Multi-Field Indexes

Searching on multi-field indexes is slightly more involved. Using *GotoKey* and *GotoNearest*, values can be assigned to one or more fields of the index. However, if values are assigned to less than all of the index fields in the search key buffer, you must follow this rule: A field in a later position in the index cannot be assigned a value unless all earlier position index fields are assigned values. For example, with a four-field index, a value can be assigned to the third field only if the first two fields have also been assigned values. Likewise, if you assign a value to only one field of the index, it must be the first field.

The DEMO5 project is shown in Figure 5. The Table component's *IndexName* property has been assigned `ByCityState`, a two-field index. Since City is the first field in this index, a search is only successful if a value is assigned to this field of the search key buffer. Searches that include only a value for the City field locate the first record that matches that city. To search for a particular city-state combination, both the City and State fields must be assigned a value.



CustNo	City	State
2315	Ayios Matthaios	Corfu
1984	Belize City	
1513	Bogota	

Figure 5: The DEMO5 project demonstrates using *GotoKey* on a multi-field index.

Here's the *OnClick* event handler for the button labeled **GotoKey** on the DEMO5 project:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.SetKey;
  Table1.FieldByName('City').AsString  := Edit1.Text;
  Table1.FieldByName('State').AsString := Edit2.Text;
  if Table1.GotoKey then
    DBGrid1.SetFocus;
end;
```

This code assigns values to both the City and State fields of the search key buffer. In this instance, if the user does not enter a city name, *Edit1* contains a blank string, and the search is only successful if there's a record with a blank city value and the matching state value.

FindKey and *FindNearest* can also be used with multi-field indexes. This is demonstrated by the DEMO6 project. This code is associated with the **FindKey** button on DEMO6:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.FindKey([Edit1.Text,Edit2.Text]);
  DBGrid1.SetFocus;
end;
```

EditKey Versus *SetKey*

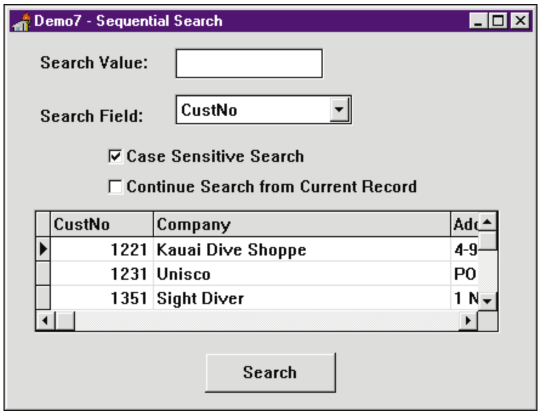
Closely related to the *SetKey* method is *EditKey*. When you call *SetKey*, the search key buffer is cleared of the previous search contents. In contrast, calling *EditKey* leaves the search key buffer intact. This permits you to modify some search values. If you perform a second search, and some of the index field values for this search are identical to those in the first search, you can call *EditKey* to enter the *dsSetKey* state, and then assign search values to the modified search key buffer. If you called *SetKey* prior to a second search, all search key buffer fields would need an assigned value. Therefore, the difference between *SetKey* and *EditKey* is only important when working with multi-field indexes.

Sequential Searches

It's not always possible, or practical, to search a table based on an index. For example, you may want to provide the user with the ability to search any field, or combination of fields, and it's very unlikely that an index for such a search will always be available.

Although there are several techniques used to look for a particular record in a table, the most straightforward is the sequential search. To use it, move to the first record (if you are not required to begin from the current one) and access each record to test for the desired field values. If a match is found, the search is successful and can be terminated.

This technique is demonstrated in the DEMO7 project, shown in Figure 6, where the user selects the value to search for and the field to search for it in.



Demo7 - Sequential Search

Search Value:

Search Field: CustNo

☑ Case Sensitive Search

□ Continue Search from Current Record

CustNo	Company	Add
1221	Kauai Dive Shoppe	4-9
1231	Unisco	PO
1351	Sight Diver	1 N

Figure 6: The DEMO7 project demonstrates one technique for scanning a table sequentially.

Figure 7 shows the Object Pascal code associated with the *OnClick* event handler for the **Search** button. This code begins by declaring a variable of the type *TBookMark* that defines which record to move to at the end of a search. The first statement in the body of the procedure is a test of the Table component's state. If the current record has not been posted, the user is asked to confirm the posting of the record before continuing.

Next, a bookmark is assigned to the current record, the form's cursor is changed to an hourglass, and the Table component's *DisableControls* method is called. Calling this method prevents the user from seeing the accessed records while locating the specified value, and speeds the search by avoiding a repaint of any data-aware controls.

The search begins after disabling the Table component's controls. If the checkbox labeled **Continue Search From Current Record** has been checked, a call to the Table's *First* method moves to the first record in the table. Next, a **while** loop is entered that repeats until all records have been processed based on the *EOF* (end-of-file) property of the table.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  GotoRecord: TBookMark;
begin
  if Table1.State in [dsEdit,dsInsert] then
    if MessageDlg('Post record?',mtConfirmation,
                  [mbOK,mbCancel],0) <> mrOK then
      Exit
    else
      Table1.Post;

  GotoRecord := Table1.GetBookMark;
  Self.Cursor := crHourGlass;
  Table1.DisableControls;

  try
    if not CheckBox2.Checked then
      Table1.First;
    while not Table1.EOF do
      begin
        if Table1.FieldByName(ComboBox1.Text).AsString =
           Edit1.Text then
          begin
            GotoRecord := Table1.GetBookMark;
            Break;
          end
        else
          Table1.Next;
      end;
  finally
    Table1.GotoBookMark(GotoRecord);
    Table1.FreeBookMark(GotoRecord);
    Table1.EnableControls;
    Self.Cursor := crDefault;
    DBGrid1.SetFocus;
  end;

end;
```

Figure 7: The *OnClick* event handler for the **Search** button.

Within the **while** loop, each record is tested for the search value. If a match is found, the bookmark is assigned to the record and the **while** loop is terminated using a Break statement.

The final part of this event handler appears in the **finally** clause of the **try..finally** block. The statements in the **finally** clause are executed even if an exception is raised within the **try** clause. The last steps are performed in the search: moving to the record pointed to by the bookmark, freeing the bookmark, enabling the Table's data-aware controls, restoring the form's cursor, and setting focus to the DBGrid.

This sequential-search technique is simple but limited. First, it's not advisable to use it with SQL tables. These tables are set-oriented, not record-oriented. Therefore, it's better to use SQL statements to locate records in SQL tables. Second, this technique is affected directly by the number of records in the table being searched, and the location of a matching record within that table. For example, searching a 200,000 record table for a value found in the last record requires just over one minute to complete on a 100MHz Pentium with 24MB of RAM. By comparison, a search on the same table using an index and

GotoNearest was practically instantaneous. However, in a small table of less than 5,000 records or so, the speed of a sequential search is acceptable.

Conclusion

Whether your tables are indexed or not, you have several options for locating specific records. Whenever possible, you should try to use indexes. In fact, one of the main reasons for creating indexes is to provide fast searches based on commonly searched fields. However, if indexes are not practical, a sequential search can always be used, even though it is far less efficient. ∆

The demonstration projects referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\DEC\DI9512CJ.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is a developer, trainer, and author of numerous books on database software. You can reach Jensen Data Systems at (713) 359-3311, or through CompuServe at 76307,1533.

Begin Listing Six — Demo1 Project

```
program Demo1;
uses
  Forms, Demo1u in 'DEMO1U.PAS' {Form1};
{$R *.RES}
begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.

unit Demo1u;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls, Grids,
  DBGrids, DB, DBTables;
type
  TForm1 = class(TForm)
    DataSource1: TDataSource;
    Table1: TTable;
    DBGrid1: TDBGrid;
    Button1: TButton;
    Label1: TLabel;
    Edit1: TEdit;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.SetKey;
  Table1.FieldByName('CustNo').AsString := Edit1.Text;
  if Table1.GotoKey then DBGrid1.SetFocus;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  Table1.Open;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  Table1.SetKey;
  Table1.FieldByName('CustNo').AsString := Edit1.Text;
  Table1.GotoNearest;
  DBGrid1.SetFocus;
end;
end.
```

End Listing Six

Begin Listing Seven — Demo2 Project

```
program Demo2;
uses
  Forms, Demo2u in 'DEMO2U.PAS' {Form1};
{$R *.RES}
begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.

unit Demo2u;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls, Grids,
  DBGrids, DB, DBTables;
type
  TForm1 = class(TForm)
    DataSource1: TDataSource;
    Table1: TTable;
    DBGrid1: TDBGrid;
    Button1: TButton;
    Label1: TLabel;
    Edit1: TEdit;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Table1.FindKey([Edit1.Text]) then DBGrid1.SetFocus;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  Table1.Open;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  Table1.FindNearest([Edit1.Text]);
  DBGrid1.SetFocus;
end;
end.
```

End Listing Seven

Begin Listing Eight — Demo3 Project

```
program Demo3;
uses
  Forms, Demo3u in 'DEMO3U.PAS' {Form1};
{$R *.RES}
begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.

unit Demo3u;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls, Grids,
  DBGrids, DB, DBTables;
type
  TForm1 = class(TForm)
```

```
    DataSource1: TDataSource;
    Table1: TTable;
    DBGrid1: TDBGrid;
    Button1, Button2: TButton;
    Label1: TLabel;
    Edit1: TEdit;
    Table1CustNo: TFloatField;
    Table1Company: TStringField;
    Table1Addr1: TStringField;
    Table1Addr2: TStringField;
    Table1City: TStringField;
    Table1State: TStringField;
    Table1Zip: TStringField;
    Table1Country: TStringField;
    Table1Phone: TStringField;
    Table1FAX: TStringField;
    Table1TaxRate: TFloatField;
    Table1Contact: TStringField;
    Table1LastInvoiceDate: TDateTimeField;
    CheckBox1: TCheckBox;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure CheckBox1Click(Sender: TObject);
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.SetKey;
  Table1.FieldByName('Company').AsString := Edit1.Text;
  if Table1.GotoKey then DBGrid1.SetFocus;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  Table1.Open;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  Table1.SetKey;
  Table1.FieldByName('Company').AsString := Edit1.Text;
  Table1.GotoNearest;
  DBGrid1.SetFocus;
end;
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked then Table1.IndexName := 'Company'
  else Table1.IndexName := 'ByCompany';
end;
end.
```

End Listing Eight

At Your Fingertips

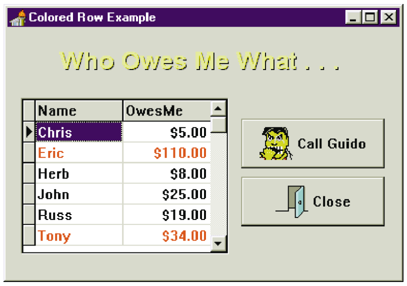
By David Rippy

Delphi / Object Pascal

We must either find a way or make one.
— Hannibal

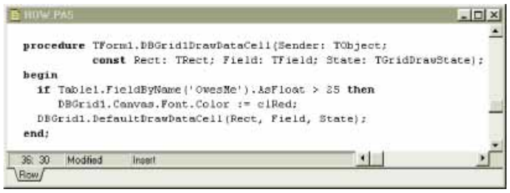
How can I conditionally change the font color of a row in a DBGrid?

This is one of the most frequently asked questions I receive. By changing the color of selected rows in a DBGrid, entries can be found quickly without having to study the values in the columns. An example of this feature is shown in Figure 1. This form contains a DBGrid displaying the names of several friends and the amount of money they owe me. If the amount owed is greater than US$25, the font color for that row becomes red. Conversely, if the amount owed is less than or equal to US$25, the font color remains black.



Name	OwesMe
Chris	$5.00
Eric	$110.00
Herb	$8.00
John	$25.00
Russ	$19.00
Tony	$34.00

Figure 1: Values in the **OwesMe** column greater than US$25 are displayed in red.



```
procedure TForm1.DBGrid1DrawDataCell(Sender: TObject;
          const Rect: TRect; Field: TField; State: TGridDrawState);
begin
  if Table1.FieldByName('OwesMe').AsFloat > 25 then
    DBGrid1.Canvas.Font.Color := clRed;
  DBGrid1.DefaultDrawDataCell(Rect, Field, State);
end;
```

Figure 2: This code is attached to the DBGrid's *OnDraw-DataCell* event handler.

The code that adds this functionality to the DBGrid is attached to the grid's *OnDrawDataCell* event (see Figure 2). First, the **OwesMe** column from *Table1* is examined. If its value is greater than 25, the font color for the row is changed to the color constant *clRed*. The cell is then drawn on the screen with the *DefaultDrawDataCell* command. Not bad for two Object Pascal statements! — *D.R.*

How can I run another Windows application within my program?

Users will often ask for the ability to execute other programs within your Delphi application. For example, they may want to access the Windows Calculator without returning to Windows itself (see Figure 3).

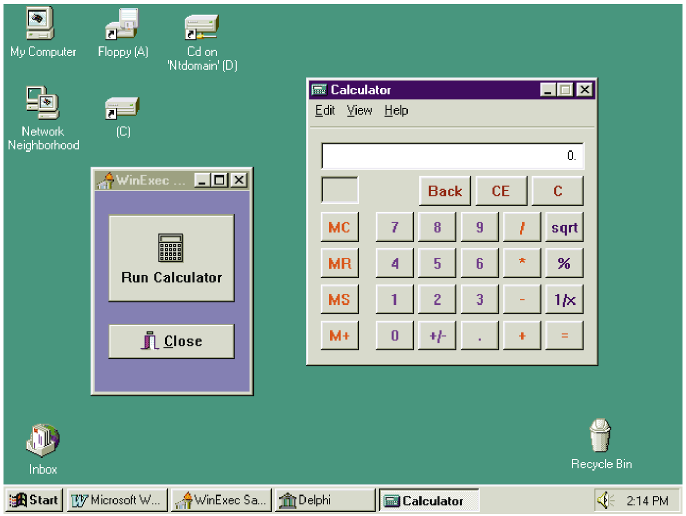


Figure 3: Pressing the **Run Calculator** button executes the Windows Calculator.

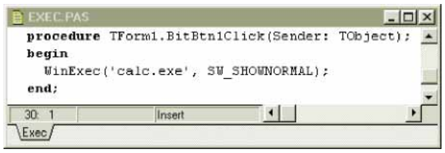
This is an easy feature to implement with Delphi. Simply call the Windows API function **WinExec**. The syntax for using **WinExec** is:

WinExec(*CmdLine*: PChar; *CmdShow*: Word) : Word;

WinExec accepts two parameters: *CmdLine*, a PChar containing the name of the program to execute; and *CmdShow*, a Word specifying how the program should run (minimized, maximized, etc.). For a listing of the values for *CmdShow*, search for "ShowWindow" in Delphi's Windows API on-line help. **WinExec** returns a Word that indicates the result of the function call. A return value of less than 32 indicates the function call was unsuccessful.

If *CmdLine* does not include a path, **WinExec** will search for the executable in the following order: the current directory, the Windows directory, the Windows system directory, the directory containing the executable of the current application, all directories in the path, and the directories mapped on a network.

Figure 4 shows the *OnClick* event handler for the BitButton labeled **Run Calculator**. CALC.EXE was specified as the *CmdLine* parameter, and SW_SHOWNORMAL as the *CmdShow* parameter. Because CALC.EXE is contained in the Windows directory, no path was specified. SW_SHOWNORMAL will display the calculator in its normal state (i.e. not minimized, hidden, etc.). — *Russ Acker, Ensemble Corporation*



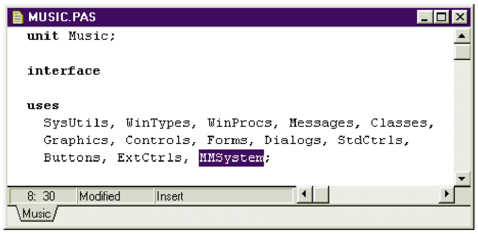
```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  WinExec('calc.exe', SW_SHOWNORMAL);
end;
```

Figure 4: This code is attached to the BitButton's *OnClick* event handler.

How can I play a .WAV file in my application?

You might be tempted to use the MediaPlayer component to play a .WAV file, but it's like driving a nail with a sledgehammer. With the ability to play MIDI, .AVI, .DAT, and other files, MediaPlayer is probably my favorite component in Delphi. However, this functionality adds a lot of unnecessary overhead to your application if you simply want to play a .WAV file. A more efficient way to play a .WAV is to call the Windows API function **sndPlaySound**.

sndPlaySound is part of the Windows Multimedia extensions and is contained in the MMSYSTEM.DLL. Delphi has already encapsulated this function for us in the MMSYSTEM unit. To gain access to this function, simply append MMSystem to the end of your unit's **uses** statement, shown in Figure 5.

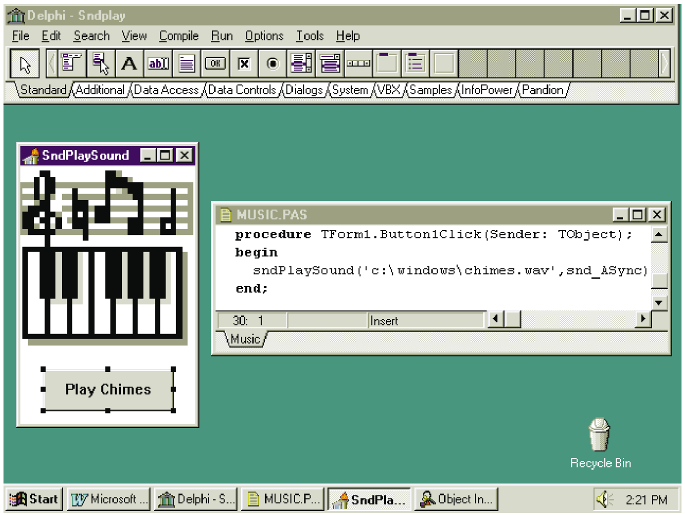


```
unit Music;

interface

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

Figure 5: Append MMSystem to the unit's **uses** clause to access the **sndPlay-Sound** function.

Figure 6 shows a form with a button labeled **Play Chimes**. When the button is pressed, the familiar Windows chimes .WAV file is played. Examine the code attached to the button's *OnClick* event. **sndPlaySound** accepts two parameters: a PChar for the name of the .WAV file, and a Word for the mode to play the .WAV. In this example, CHIMES.WAV is specified as the .WAV file, and the constant **snd_Async** is used for the play mode. The most commonly used values for the play mode parameter are **snd_Sync** and **snd_Async**. Additional parameters can be found in the MMSYSTEM unit located in the \DELPHI\SOURCE\RTL\WIN directory. Specifying **snd_Sync** will suspend your application until the .WAV has completed playing. Specifying **snd_Async** will play the sound in the background while your application continues running.

Note that the constants mentioned in MMSYSTEM.PAS for the **sndPlaySound** function are flags and can be combined to produce effects different from just these two. Note also that the MMSYSTEM.HLP help is available in the \DELPHI\BIN directory for reference. However, it isn't "hooked up" with the rest of the Delphi help system so you'll have to call it directly. — *D.R.*


Delphi - Sndplay
File Edit Search View Compile Run Options Tools Help
Standard Additional Data Access Data Controls Dialogs System VBX Samples InfoPower Pandion
SndPlaySound
Play Chimes
MUSIC.PAS
30: 1
Insert
Music
Recycle Bin
Start Microsoft ... Delphi - S... MUSIC.P... SndPla... Object In...
2:21 PM

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  sndPlaySound('c:\windows\chimes.wav',snd_ASync)
end;
```

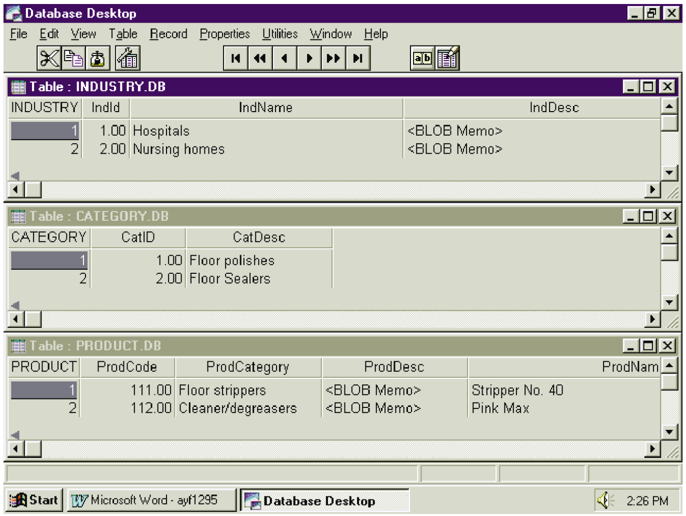

Database Desktop
File Edit View Table Record Properties Utilities Window Help
Start Microsoft Word - ayf1295 Database Desktop
2:26 PM

Table : INDUSTRY.DB

INDUSTRY	IndId	IndName	IndDesc
1	1.00	Hospitals	<BLOB Memo>
2	2.00	Nursing homes	<BLOB Memo>

Table : CATEGORY.DB

CATEGORY	CatID	CatDesc
1	1.00	Floor polishes
2	2.00	Floor Sealers

Table : PRODUCT.DB

PRODUCT	ProdCode	ProdCategory	ProdDesc	ProdNam
1	111.00	Floor strippers	<BLOB Memo>	Stripper No. 40
2	112.00	Cleaner/degreasers	<BLOB Memo>	Pink Max

Figure 6 (Top): This code is attached to the button's *OnClick* event handler. **Figure 7 (Bottom):** To tile windows vertically down the screen in the Database Desktop, hold down Shift while selecting **Window | Tile**.

Quick Tip

In the Database Desktop, you've probably used the **Window | Tile** command to arrange your desktop windows horizontally across the screen. Did you know that holding Shift down while selecting **Window | Tile** will arrange your windows vertically down the screen? Check out Figure 7. Δ — *Russ Acker, Ensemble Corporation*

The demonstration projects referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\DEC\DI9512DR.

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 in the *Paradox Informant*. David can be reached on CompuServe at 74444,415.

VISUAL PROGRAMMING

DELPHI / OBJECT PASCAL

By Blake Versiga

Breed and Rank

Using the Outline Component to Represent a Family Tree

Many developers consider Delphi's Outline control the de-facto standard for representing hierarchical relationships. Outline controls are ideal for visually displaying document components, family trees, organizational relationships, object models, etc. Delphi itself uses the Outline control to implement its ObjectBrowser. [For an in-depth discussion of this tool, see Douglas Horn's article "The ObjectBrowser" in the November 1995 *Delphi Informant*.]

This article will reveal some of the mysteries behind Delphi's Outline control, and discuss various ways to work with it. For instance, it can be completely configured at design-time or run-time (or variations of the two) by querying a database. In addition, owner-draw capabilities will be explored. The example program provides an efficient user interface for displaying the family tree of Great Pyraneese dogs.

Distinct Markings

The Outline control (*TOutline* class) is on the Additional page of the Component Palette. This control is an array of OutlineNodes (*TOutlineNode* class) with indexes from 1 to the number of items in the list. For this reason, it's important to distinguish between the Outline control and each item (or OutlineNode).

Both Outline and OutlineNode have individual sets of properties, methods, and events. The attributes of Outline affect it as a whole. On the other hand, attributes of the OutlineNode affect only the specific nodes of the Outline. Each item in the Outline can have 0 to 16368 sub-items.

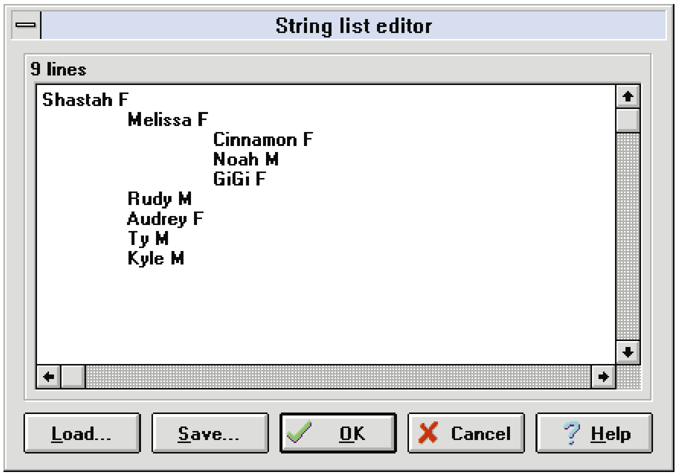
Its large number of properties and methods make Outline a flexible component. For example, the *PictureClosed*, *PictureLeaf*, *PictureMinus*, *PictureOpen*, and *PicturePlus* properties contain standard File Manager-like graphics that can be customized for a particular look. If five graphics will not meet the challenge, however, the Outline component comes to the rescue with its owner-draw capabilities.

Opening Eyes

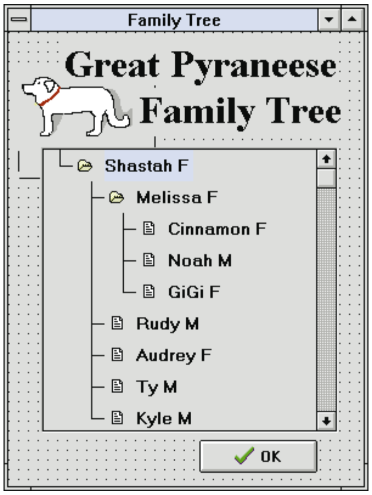
Let's begin with a small example program to demonstrate some important Outline component concepts. First, create a new project. Place an Outline component on the form and then drag its handles to enlarge it.

Next, double-click on the Outline's *Lines* property in the Object Inspector to display the String list editor. Each line represents a node in the Outline. Press Ctrl Tab or Spacebar to add one level of indention to the node. If spaces are used, the String list editor will convert them to tabs before saving.

The sample application initializes each node to include the name of the dog followed by a character denoting its gender (see Figure 1). This is used to determine which bitmap should be associated with the node when the *Style* property is set to *otOwnerDraw*. After adding entries in the String list editor, run the application by pressing F9. The Outline will expand and collapse without any additional code (see Figure 2).



```
9 lines
Shastah F
        Melissa F
                Cinnamon F
                Noah M
                GiGi F
        Rudy M
        Audrey F
        Ty M
        Kyle M
```



```
Shastah F
    Melissa F
        Cinnamon F
        Noah M
        GiGi F
    Rudy M
    Audrey F
    Ty M
    Kyle M
```

Figure 1 (Top): The String list editor is used to define the genealogy of Great Pyraneese dogs.
Figure 2 (Left): The sample project in action. Clicking on the yellow folders causes this "object browser" to expand the genealogy of the dogs. The name and gender of each dog in this canine family are shown.

The *OutlineStyle* and *Picture* properties can be used to modify the appearance of the Outline control and allow for customization without code. It's also important to remember that the bitmap's size is fixed. The default bitmaps are 14 pixels high and 14 pixels wide, and reside in the \DELPHI\IMAGES\DEFAULT directory (if images are installed).

The *ItemHeight* property works in conjunction with owner-draw controls, for example:

```
Style := otOwnerDraw
```

Therefore, changing this property does not affect the controls when the *Style* property is set to *otStandard*. *Style* can be set to *otOwnerDraw* to activate the *ItemHeight* property. If there is no code in the *OnDrawItem* event, Delphi uses the default drawing method. This allows the standard pictures to be enlarged without the additional complexity of a full owner-draw control.

Barking at the Moon

Now that we have a functioning application, it's essential to capture events and respond to the user's interaction with the control. Typically, the developer must perform some tasks when the user clicks on a node of the Outline. The following example modifies the form's caption with the Outline index and the full text path of the selected item. The index is a one-based long integer indicating the ordinal position of the item with respect to the entire list.

The full text path represents the text and separators of the nodes the user must navigate to arrive at the selected node. The *ItemSeparator* property determines the character string used to delimit items in the *FullPath* property. To illustrate these properties, simply add this line of code to the *Click* event of the Outline control:

```
Form1.Caption :='Outline Index ' +
  IntToStr(Outline1.SelectedItem) + '. ' +
  Outline1.Items[Outline1.SelectedItem].FullPath;
```

Howlin' with the Hounds

If adventure runs in your blood, the owner-draw style is for you. The owner-draw Outline style offers the most control and is the ultimate in flexibility. (Again, Delphi's ObjectBrowser is an excellent example.) The concept behind owner-draw controls is that the programmer controls *all* aspects of a control's appearance, including tree lines, pictures, text, and indention. Although this sounds daunting, it's quite simple if you avoid a few pitfalls.

To receive events essential for owner-draw operations, set the *Style* property to *otOwnerDraw*. This instructs the Outline control to call the *OnDrawItem* event whenever it needs to draw a node of the Outline. Note: In early versions of the VCL, the Outline component contained a bug that suppressed the *OnDrawItem* message if the *ScrollBars* property was set to *ssHorizontal* or *ssBoth*. To avoid this, set the *ScrollBars* property to *ssVertical* or *ssNone*. Any version of Delphi subsequent to version 1.0 (i.e. from the first patch onward) fixed this problem.

First, add StdCtrls to the form's **uses** clause. This module adds the needed definitions for the *TOwnerDrawState*. Next, add any variables required of type *TBitmap* to the **implementation** section of the unit. This will declare bitmaps that are accessible throughout the module.

Next, code must be added to create and free the bitmaps. Select the Events page in the Object Inspector and double-click in the Values column of the *OnCreate* event. Now enter a statement to allocate storage space on the stack for each bitmap, for example:

```
BitmapVariable := TBitmap.Create;
```

Next, initialize each bitmap. If the bitmaps reside on disk, the *LoadFromFile* method can be used:

```
BitmapVariable.LoadFromFile('Bitmap.bmp');
```

The bitmap is now allocated on the stack, so the memory must be freed when it's no longer needed. To accomplish this, use this statement in the *OnClose* event:

```
BitmapVariable.Free;
```

Finally, the Outline node must be drawn when indicated by the Outline control. As mentioned earlier, the Outline control calls the *OnDrawItem* procedure whenever a cell requires repainting. This routine should clear the canvas and paint any pictures, graphics, or text. The *OnDraw* procedure is also responsible for showing hierarchical indention, if necessary. Here's the prototype for the *DrawItem* procedure:

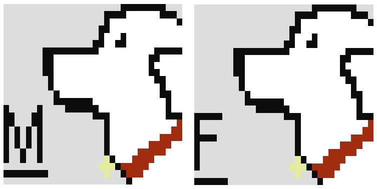
```
procedure TForm1.Outline1DrawItem(Control: TWinControl;
  Index: Integer; Rect: TRect; State: TOwnerDrawState);
```

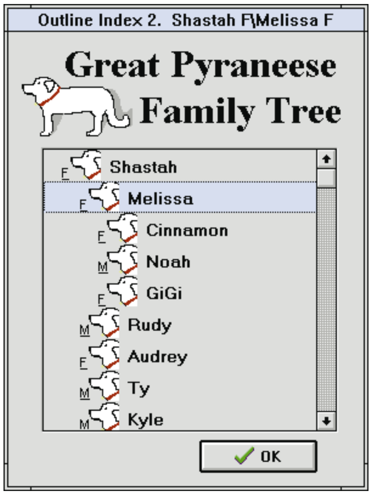
The *Control* parameter contains a reference to the Outline control. *Index* is the position of the item in the Outline control. Note that this does *not* correspond to the node's index, but represents the ordinal index of visible items within the Outline. *Rect* is the area available to the node. The *State* is defined as:

```
TOwnerDrawState = set of (odSelected,odGrayed,odDisabled,
                          odChecked,odFocused);
```

The *State* parameter will contain information required to determine how a node will be drawn. Within the *DrawItem* procedure, the code should paint the node. A typical scenario involves clearing the area that will be painted, drawing the graphic, and finally, painting the text. As mentioned earlier, the index passed to the *OnDraw* event does not represent the node *Index* that will be drawn. To obtain the actual *Index*, the *GetItem* method is used. It translates any X,Y point to a node *Index*. Given the node *Index*, all the pieces have come together.

Information about the node is used to determine exactly how it will be painted. In our example, the current node is drawn highlighted and a different bitmap is used depending on the dog's gender. A bitmap with an underlined **M** in the lower left corner represents male dogs, and females are denoted with an underlined **F** (see Figure 3). The example project at runtime is shown in Figure 4, and the project source is shown in Listing 9 on page 49.





Outline Index 2. Shastah F\Melissa F

Great Pyraneese Family Tree

F Shastah
F Melissa
F Cinnamon
M Noah
F GiGi
M Rudy
F Audrey
M Ty
M Kyle

Figure 3 (Top): The male and female Pyraneese bitmaps. Not just pretty faces, these bitmaps demonstrate how the *DrawItem* procedure and *GetItem* method are used to properly paint the appropriate node. **Figure 4 (Bottom):** The sample project at run-time.

Conclusion

Now that we have mastered the fundamentals of owner-draw outlines, we can apply these techniques to other owner-draw controls that are handled in a similar manner. Additional owner-draw controls include *TComboBox*, *TDBComboBox*, *TDBListBox*, *TListBox*, and *TOutline*.

Owner-draw controls offer tremendous flexibility to the programmer. These techniques can be used to display anything from hierarchical data relationships to sections of text (e.g. the Microsoft Word for Windows Outline feature). The Outline control is a master of distilling tremendous amounts of information into manageable chunks of data. The key is creativity. Now it's your turn to bark up a storm. ∆

The Delphi project referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\DEC\DI9512BV.

Blake Versiga is a Systems Engineer at Computer Language Research. He has been involved in producing corporate solutions ranging from Pen-based computing to large client/server applications. He can be reached on CompuServe at 73123,2737 or via Internet mail at 73123.2737@compuserve.com.

Begin Listing Nine — Outline Demo Program

```
program Project1;

uses
  Forms,
  Unit1 in 'UNIT1.PAS' {Form1};

{$R *.RES}

begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.


unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Outline1: TOutline;
    Image1: TImage;
    Label1: TLabel;
    Image2: TImage;
    BitBtn1: TBitBtn;
    procedure Outline1Click(Sender: TObject);
    procedure Outline1DrawItem(Control: TWinControl;
                               Index: Integer;
                               Rect: TRect;
                               State: TOwnerDrawState);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
    procedure BitBtn1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

var
  bmpMalePyr,
  bmpFemalePyr: TBitMap;

{$R *.DFM}

procedure TForm1.Outline1Click(Sender: TObject);
begin
  Form1.Caption := 'Outline Index ' +
           IntToStr(Outline1.SelectedItem) +
           '.   ' +
           Outline1.Items[Outline1.SelectedItem].FullPath
end;

procedure TForm1.Outline1DrawItem(Control: TWinControl;
                                  Index: Integer;
                                  Rect: TRect;
                                  State: TOwnerDrawState);

var
  s:             string;
  x,y,
  Offset:        Integer;
  ItemIndex:     LongInt;
  Node:          TOutlineNode;

begin
  ItemIndex := Outline1.GetItem(Rect.Left, Rect.Top);
  Node := Outline1.Items[ItemIndex];

  with Outline1.Canvas do
    begin
      if State = [odSelected,odFocused] then
        begin
          Brush.Color  := clHighlight;
          Pen.Color    := clHighlightText;
        end
      else
        begin
          Brush.Color  := Outline1.Color;
          Pen.Color    := clWindowText;
        end;

      FillRect(Rect);
      Rect.Left := Rect.Left + (Node.Level * 15);

      { First draw the icon }
      if Copy(Node.Text,length(Node.Text),1) = 'F' then
        BrushCopy(Bounds(Rect.Left,Rect.Top,35,25),
                  bmpFemalePyr,Bounds(0,0,35,25),clLtGray)
      else
        BrushCopy(Bounds(Rect.Left,Rect.Top,35,25),
                  bmpMalePyr,Bounds(0,0,35,25),clLtGray);

      Font := Outline1.Font; { Set the font }
      { Then draw the string }
      s := copy(Node.Text, 1, Length(Node.Text) - 2);
      { Indent and center Text }
      x := Rect.left + bmpMalePyr.Width + 7;
      y := Rect.Top + (((Rect.Bottom-Rect.Top) –
                          abs(Font.Height)) div 2);

      TextOut(x, y, s);
    end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  { Load the bitmap from the current directory }
  bmpMalePyr   := TBitMap.Create;
  bmpMalePyr.LoadFromFile('MalePyr.Bmp');
  bmpFemalePyr := TBitMap.Create;
  bmpFemalePyr.LoadFromFile('FemPyr.Bmp');
end;

procedure TForm1.FormClose(Sender: TObject;
                           var Action: TCloseAction);
begin
  bmpMalePyr.Free;
  bmpFemalePyr.Free;
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  Form1.Close;
end;

end.
```

End Listing Nine

NEW & USED

BY BILL TODD

ReportPrinter

Nevrona Designs' New Report Writer

Every now and then a whole new category of product comes along that makes you say, "Wow. Why hasn't somebody done this before?" ReportPrinter from Nevrona Designs is just such a product. What can be so different about a report writer? The answer is control, integration, and size.

Reports are created in ReportPrinter by writing Object Pascal code, not by using a visual designer such as those in ReportSmith and Crystal Reports. At first this may sound like a big step backward, but it's really a step to the side — and a step into control over report design that you've never had before.

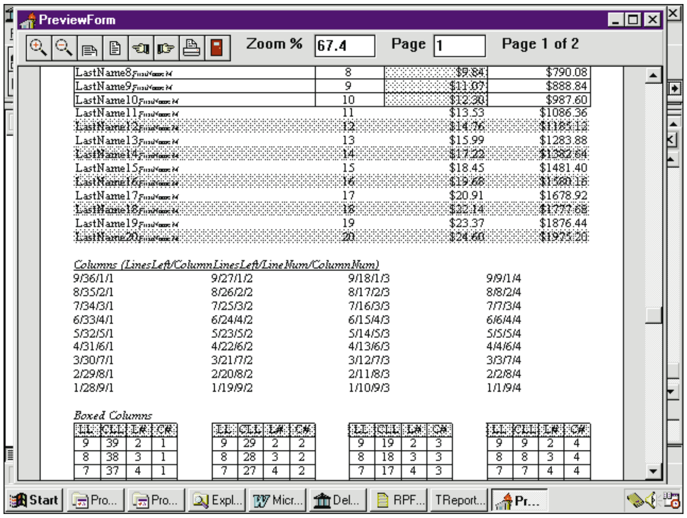
To understand the control and flexibility of ReportPrinter, look at the two-page report illustrated in Figures 1 and 2. This is one of the reports produced by the demonstration program that comes with ReportPrinter. The two pages in this report are completely different. Figure 1 contains multiple tables in varying formats and locations, while Figure 2 contains a collection of graphic shapes intermixed with text.

You simply cannot create a report like this in a traditional Windows report writer.

Let's Compare

Is ReportPrinter a replacement for traditional report writers like Crystal Reports and ReportSmith? Definitely not. Each has its place defined by its strengths and weaknesses. With ReportPrinter you get:

- **Integration**. ReportPrinter is a set of native Delphi components to add to your application. Everything is compiled into the .EXE file. There are no separate .DLLs or report files to distribute. And since the report is created entirely by compiled Pascal code, you get the output fast.



LastName8 FirstName M	8	$9.84	$790.08
LastName9 FirstName M	9	$11.07	$888.84
LastName10 FirstName M	10	$12.30	$987.60
LastName11 FirstName M	11	$13.53	$1086.36
LastName12 FirstName M	12	$14.76	$1185.12
LastName13 FirstName M	13	$15.99	$1283.88
LastName14 FirstName M	14	$17.22	$1382.64
LastName15 FirstName M	15	$18.45	$1481.40
LastName16 FirstName M	16	$19.68	$1580.16
LastName17 FirstName M	17	$20.91	$1678.92
LastName18 FirstName M	18	$22.14	$1777.68
LastName19 FirstName M	19	$23.37	$1876.44
LastName20 FirstName M	20	$24.60	$1975.20

Columns (LinesLeft/ColumnLinesLeft/LineNum/ColumnNum)

```
9/36/1/1    9/27/1/2    9/18/1/3    9/9/1/4
8/35/2/1    8/26/2/2    8/17/2/3    8/8/2/4
7/34/3/1    7/25/3/2    7/16/3/3    7/7/3/4
6/33/4/1    6/24/4/2    6/15/4/3    6/6/4/4
5/32/5/1    5/23/5/2    5/14/5/3    5/5/5/4
4/31/6/1    4/22/6/2    4/13/6/3    4/4/6/4
3/30/7/1    3/21/7/2    3/12/7/3    3/3/7/4
2/29/8/1    2/20/8/2    2/11/8/3    2/2/8/4
1/28/9/1    1/19/9/2    1/10/9/3    1/1/9/4
```

Boxed Columns

LL	CLL	L#	C#
9	39	2	1
8	38	3	1
7	37	4	1

LL	CLL	L#	C#
9	29	2	2
8	28	3	2
7	27	4	2

LL	CLL	L#	C#
9	19	2	3
8	18	3	3
7	17	4	3

LL	CLL	L#	C#
9	9	2	4
8	8	3	4
7	7	4	4

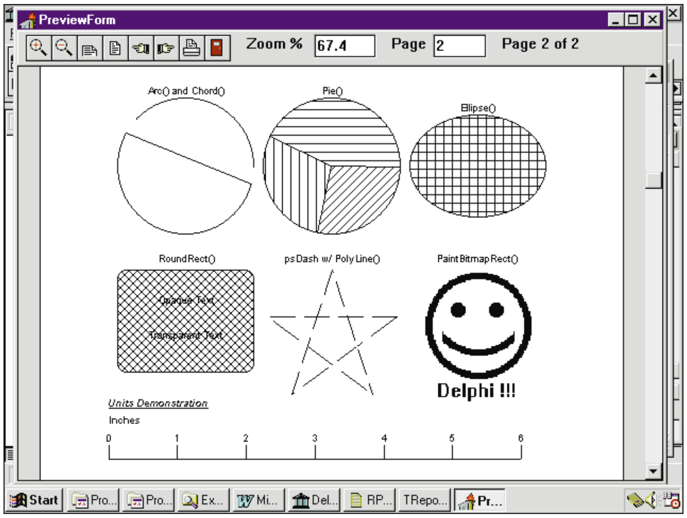


Figure 1 (Top): A ReportPrinter report containing multiple tables in various formats. **Figure 2 (Bottom):** The second page of the report shown in Figure 1 containing mixed graphics and text.

Crystal Reports Delphi Control

Now available, Crystal Reports version 4.5 is a welcome addition for Delphi developers. It ships with a native Delphi component that provides an easy interface to the Crystal print engine through a host of properties and methods.

Basic report printing and viewing is very easy. After adding the Crystal component to the Delphi Component Palette you will find it located on the Data Access page. Just drop a *TCcrpe* component on your form and set the following properties to print your report:

CopiesToPrinter. This property defaults to 1 so you will only need to change it if you need more than one copy of your report.

Destination. This property determines where your report will go. You can select from any of the following constants:

- *toWindow*
- *toPrinter*
- *toFile*
- *toEmailViaMAPI*
- *toEmailViaVIM*

ReportName. The name of the file that contains the Crystal report to print.

WindowState. You need only be concerned with this property if the *Destination* property is set to *toWindow*. To control the size of the report preview window, *WindowState* can be set to *wsMaximized*, *wsMinimized*, or *wsDefault*.

WindowTitle. This property is optional, but if you are sending the report to a window, it allows you to set the title for the report preview window.

Once you have set the required properties, run the report by setting the *Action* property to *1*. While these are the properties you will use most, the Crystal component includes over three pages of properties you can set via the Property Inspector, and even more that you can set at run-time in your code. This provides access to just about all the flexibility for which the Crystal print engine is renowned.

Unfortunately this component is seriously flawed by bad design and poor documentation. The component is supposed to provide a native Delphi alternative to the Crystal VBX that shipped with earlier versions of Crystal Reports (which you may have been using in your Delphi applications). However, the designers changed the names of some of the properties and many of the constants used to set properties. For example, the VBX property that stores the name of the report to run is *ReportFileName*, while in the Delphi component it is *ReportName*. Another example is that the *PrintReport* method from the VBX is not supported in the Delphi component. This means that converting your existing programs from the VBX to the Delphi component is more work than it should be.

Learning to use the component is a challenge too. The only documentation is a Windows help file and it is woefully incomplete. While it does include the component properties, it does not include any of the methods. The "readme" file is no help either. It is simply a list of changes and bug fixes to prior versions of the component. You may expect this in the readme for a beta version of a shareware product, but not in the shipping version of a commercial product. The best advice I can offer is to look at the documentation for the VBX in the *Crystal Reports Developer's Reference* and use that as a guide for getting started with the Delphi component.

The version of the Delphi component that ships with Crystal Reports 4.5 is 1.06. By the time you read this review, version 1.08 should be available on the Crystal bulletin board and on their CompuServe forum. The pre-release copy of version 1.08 that I saw includes a much more complete help file and a demonstration program that shows you how to use many of the component's features. This new version is worth the download time just for the help file and the demonstration program.

The Crystal Reports component is a welcome tool that makes it easy to use this powerful report writer in your Delphi programs, while giving you access to all the features of the Crystal print engine. Although it will take some time, it is worth your while to convert your programs from the VBX to the Delphi component. You will no longer need to distribute the VBX file, and you'll be ready to compile your program with Delphi32 where VBXes are not supported.

— Bill Todd

- **Small size.** Adding a report to an application increases the .EXE file size by about 30KB for the first report, and less for subsequent reports. Contrast that with the megabytes of additional files distributed with a conventional report writer.
- **Control**. You can put anything, anywhere, on any page, in any order to get the report you want. Since the position is specified in your code in inches or centimeters, each object on the report prints exactly where you want it.
- **Direct access to data**. Since a report is part of an application, you access data using Delphi's Table, Query, and StoredProc components. There is no separate database connection for your reports, which is especially convenient for printing them based on the current record.
- **Slower development time**. This is the drawback to using ReportPrinter. Because you create each report by writing code, it takes longer to design most reports using ReportPrinter than with a traditional report writer. This is particularly true if the report requires numerous groups and subtotals.

With a traditional report writer you get:

- **Fast, easy, WYSIWYG development**. You can see what the finished report will look like as it's created. This saves time.
- **Instant calculations.** Traditional report writers provide fast, easy sorting, grouping, and subtotaling. Most offer other functions such as counts and averages. You must write code to perform these calculations in ReportPrinter.
- **A large run-time environment**. This is the drawback to traditional report writers. To create reports with a traditional report writer, you must distribute its run-time environment with your application. This means 2 to 4MB of DLLs and supporting files, plus at least one file for each report.

Consider the Options

When you need to include reports with custom applications, consider which tool is best suited for the job, as well as the time you are willing to invest creating the reports.

Do not feel limited because you have to create ReportPrinter reports by writing Object Pascal code. ReportPrinter supports tabular reports, snaked columns, memos, graphics, line and shape drawing, and fonts. To create business graphs in your reports, pass the ReportPrinter canvas handle to the ChartFX component that ships with Delphi.

ReportPrinter consists of four Delphi components that afford the functionality you require.

- The **ReportPrinter** component lets you send a report to the current printer.
- **ReportFiler** is virtually identical to the ReportPrinter component except it sends a report to either a disk file or memory stream.
- ReportFiler lets you create the report; then send it to the printer using the **FilePrinter** component, or to the screen using the FilePreview component.
- **FilePreview** supports zooming, panning, and printing from the preview window. You can either create your own preview form, or use the one included in ReportPrinter's demonstration program.

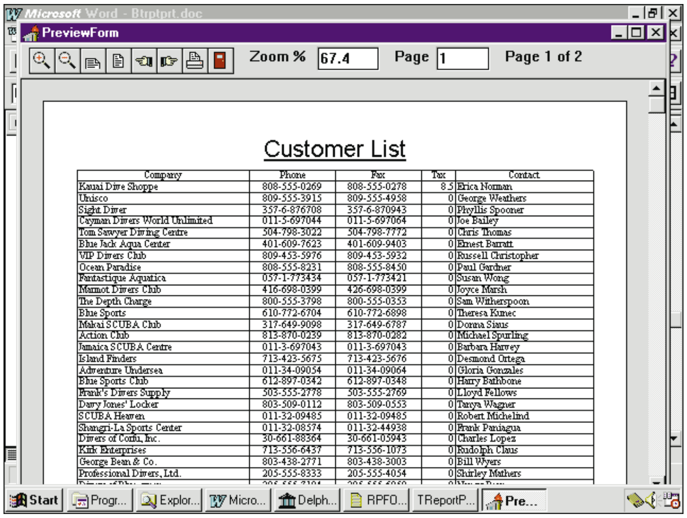
The ReportPrinter and ReportFiler components make creating a report easy with a rich suite of events. You can attach the reporting code to:

- *OnBeforePrint*
- *OnAfterPrint*
- *OnNewColumn*
- *OnNewPage*
- *OnPrint*
- *OnPrintPage*
- *OnPrintHeader*
- *OnPrintFooter*

The *OnPrintPage* event is particularly well suited to reports where each page has the same layout, while *OnPrint* is the event to use for reports where the layout varies from page to page. Figure 3 shows one of the sample reports based on a table.

The Documentation

If the current version of ReportPrinter has a shortcoming, it's the documentation. The manual is only 56 pages, which is mostly a reference for the component methods and properties. Only six pages are devoted to the tutorial, and that is not enough introduction to ReportPrinter's features.



Customer List

Company	Phone	Fax	Tax	Contact
Kauai Dive Shoppe	808-555-0269	808-555-0278	8.5	Erica Norman
Unisco	809-555-3915	809-555-4958	0	George Weathers
Sight Diver	357-6-876708	357-6-870943	0	Phyllis Spooner
Cayman Divers World Unlimited	011-5-697044	011-5-697064	0	Joe Bailey
Tom Sawyer Diving Centre	504-798-3022	504-798-7772	0	Chris Thomas
Blue Jack Aqua Center	401-609-7623	401-609-9403	0	Ernest Barratt
VIP Divers Club	809-453-5976	809-453-5932	0	Russell Christopher
Ocean Paradise	808-555-8231	808-555-8450	0	Paul Gardner
Fantastique Aquatica	057-1-773434	057-1-773421	0	Susan Wong
Marmot Divers Club	416-698-0399	426-698-0399	0	Joyce Marsh
The Depth Charge	800-555-3798	800-555-0353	0	Sam Witherspoon
Blue Sports	610-772-6704	610-772-6898	0	Theresa Kunec
Makai SCUBA Club	317-649-9098	317-649-6787	0	Donna Siaus
Action Club	813-870-0239	813-870-0282	0	Michael Spurling
Jamaica SCUBA Centre	011-3-697043	011-3-697043	0	Barbara Harvey
Island Finders	713-423-5675	713-423-5676	0	Desmond Ortega
Adventure Undersea	011-34-09054	011-34-09064	0	Gloria Gonzales
Blue Sports Club	612-897-0342	612-897-0348	0	Harry Bathbone
Frank's Divers Supply	503-555-2778	503-555-2769	0	Lloyd Fellows
Davy Jones' Locker	803-509-0112	803-509-0553	0	Tanya Wagner
SCUBA Heaven	011-32-09485	011-32-09485	0	Robert Michelind
Shangri-La Sports Center	011-32-08574	011-32-44938	0	Frank Paniagua
Divers of Corfu, Inc.	30-661-88364	30-661-05943	0	Charles Lopez
Kirk Enterprises	713-556-6437	713-556-1073	0	Rudolph Claus
George Bean & Co.	803-438-2771	803-438-3003	0	Bill Wyers
Professional Divers, Ltd.	205-555-8333	205-555-4054	0	Shirley Mathers

Figure 3: A sample ReportPrinter report based on a table using the *OnPrintPage* event.

On the other hand, the demonstration program is excellent. It includes five reports and an excellent general purpose preview form. To use this powerful tool effectively, be prepared to spend time working through the code in the demonstration program. ReportPrinter also ships with full source code.

Conclusion

ReportPrinter is a must-have tool for the serious Delphi developer. It lets you do things you cannot do with any other report writer. Whether you need reports compiled into a program for easy distribution, flexible page layout that varies from page to page, or precise positioning for pre-printed forms, ReportPrinter is the tool that handles almost any reporting task. ∆

INFORMANT FACT FILE

ReportPrinter version 1.1 by Nevrona Designs is a report writer for Delphi. Consisting of four Delphi components, it enables developers to use Object Pascal to create custom reports. Delphi applications that integrate a report created with ReportPrinter are more size-efficient than applications distributed with conventional report writers. ReportPrinter also gives developers more control over reports, and direct access to data using the Table, Query, and StoredProc components.

Nevrona Designs
2581 East Commonwealth Circle
Chandler, AZ 85225
Voice: (602) 899-0794
Fax: (602) 530-4823
E-Mail: CIS: 70711,2020 or
Internet: info@nevrona.com
Price: US$99

Bill Todd is President of The Database Group, Inc., a consulting firm based near Phoenix. Bill is co-author of *Delphi: A Developer's Guide* (M&T Books, 1995). He is also a member of Team Borland and a speaker at all Borland database conferences. Bill can be reached at (602) 802-0178, or on CompuServe at 71333,2146.

table of CONTENTS