March 1998, Volume 4, Number 3

Cover Art by: *Tom McKeith*

DCOM Streaming

Passing Delphi Objects to a DCOM Server

ON THE COVER

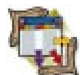
6 DCOM Streaming — David Body
Automatic marshaling is terrific, but what if you need to pass data that isn't automation-compatible? Mr Body shares his slick technique for using Delphi's component streaming mechanism to pass entire objects along the DCOM trail — and still not have to play marshal yourself.

FEATURES

11 Informant Spotlight
Deploying ActiveX Controls — Dan Miser
Swell! You've created an ActiveX control. Getting it "out there" to users, however, is another story. Mr Miser lays out the issues and their solutions when it comes to Web deployment.

15 In Development
Sounds Gud to Me — Rod Stephens
Do your users need to search for names without knowing the exact spelling? Check out Mr Stephens' Delphi implementation of proven Soundex algorithms. You'll like what you hear.

19 DBNavigator
More Code Editor Tricks — Cary Jensen, Ph.D.
You read Dr Jensen's article regarding Code Insight last month, so you're hip to all of the Code Editor's tricks. Think again! There's still plenty you don't know about the tool you use every day.



23 Columns & Rows
Developing Object Databases — Chu Moy
Things can get confusing when OOP meets a relational database. Mr Moy says the object database, POET, is the answer, and puts one to work with Delphi 3. Is there an ODBMS in your future?

29 Sights & Sounds
Multimedia Buttons — Christopher Coppola
Is the standard Button component ill suited to your way-cool interface? Mr Coppola shares his techniques for creating buttons that blend with the environment — and make a little noise.

36 The API Calls
Restoring Animation — John Ayres
Ever notice that Delphi-created applications don't exhibit that cute animated effect when they're minimized or restored? Well so has Mr Ayres, and he knows what to do about it.

REVIEWS

38 Crystal Reports Professional 6.0
Product Review by Warren Rachele

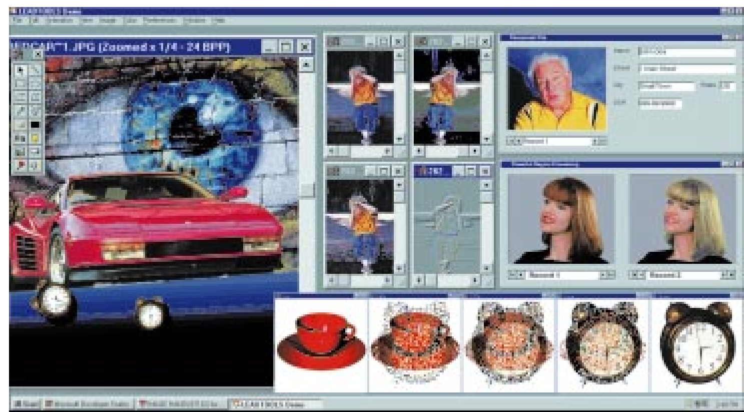
42 Rubicon for Delphi
Product Review by Peter Hyde

45 *Delphi Developer's Handbook*
Book Review by Alan Moore, Ph.D.

DEPARTMENTS

2 Delphi Tools
5 Newsline
46 File | New by Alan Moore, Ph.D.

LEAD Technologies Announces LEADTOOLS 9.0



LEAD Technologies announced *LEADTOOLS 9.0*, the company's imaging toolkit. This version offers more than 500 functions, properties, and methods to integrate black and white, grayscale, and color imaging into applications. Among the new features are common dialog boxes to speed development, support for over 50 image file formats, and optional FlashPix, video, and OCR modules.

The common dialog boxes extend Windows common functions to provide image-specific capabilities, such as image processing and conversion. The dialog boxes also offer thumbnail previewing of image changes.

LEADTOOLS 9.0 offers additional import/export capabilities, including increased DICOM multi-page support with 9- to 16-bit grayscale and window leveling, and 16-bit-per-pixel grayscale TIF support.

Version 9 also offers ISIS high-speed scanning, annotation support to include read/write support for the Wang annotation file format, multi-level password access to annotation, hyperlinks for all annotation objects, increased control over free-hand drawing, and angle and line size support.

Included with the toolkit are source code examples for Delphi, Borland C/C++, Microsoft Visual C/C++, Visual Basic, Visual FoxPro, Access, and Java Script.

LEAD Technologies
Price: From US$295 for the VBX Pro package, to US$1,995 for the Pro Express package.
Phone: (704) 332-5532
Web Site: http://www.leadtools.com

Snowbound Announces RasterMaster 7.0

Snowbound Software Corp. introduced *RasterMaster 7.0*, a set of imaging tools for Windows 95/NT, Windows 3.x, and Macintosh 68K and PPC. This version includes enhanced support for the pre-press, medical imaging, banking, and defense industries.

Raster image support includes TIFF/CMYK, JPEG/CMYK, JEDMICS, Flashpix, Winfax, DICOM, Alpha Channel, Group 4, Group 3, CALS, PNG, PhotoCD, BMP, PCX, PICT, Targa, and others. Functions include CMYK 4 plane support, RGB to CMYK and CMYK to RGB conversion features, anti-aliasing (in the Silver toolkit), fit to Window, fit to width/height, image encryption, reading TIFF tags, animated GIF, Winfax, display callbacks, erase rectangle/black border removal, 10- to 16-bit grayscale support, and tiled image support.

RasterMaster 7.0 is available as a Windows 95/NT DLL, Windows 95/NT ActiveX/OCX, or Windows 3.x DLL.

Snowbound Software Corp.
Price: US$1,350 for Silver toolkit; US$1,995 for Platinum toolkit.
Phone: (617) 630-9495
Web Site: http://www.snowbnd.com

IDEAL Introduces Virtual Print Engine 2.2

IDEAL Software announced *Virtual Print Engine 2.2*, which offers developers enhanced control over printed output from their Windows applications. Developers using Delphi, Visual Basic, C/C++, FoxPro, and other languages can dynamically create, preview, and print reports, rich documents, drawings, etc. by calling functions.

Objects such as text, lines, polygons, bitmaps (JPEG, PNG, TIFF, GIF, BMP, PCX, WMF, EMF, DXF), and 21 different barcodes can be positioned, rotated, and scaled with 0.1 mm precision on any number of pages. The vector graphics offer a free, scalable WYSIWYG preview and printer-independent output.

The package includes the OCX, VCL, DLL, and over 600KB of sample-sources for all common programming languages, and is available in 16- and 32-bit versions. There are no run-time fees or royalties.

IDEAL Software
Price: US$548
Phone: 49 2131 9800 23
Web Site: http://www.idealsoftware.com

Delphi TOOLS

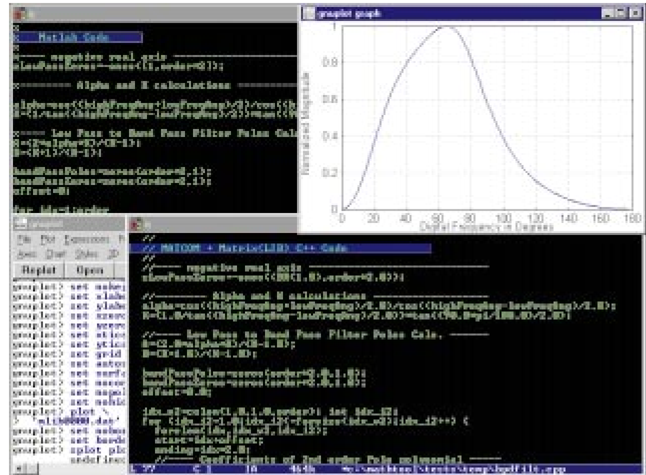
New Products and Solutions

MathTools Launches MATCOM

MathTools Ltd. announced *MATCOM*, a MATLAB compiling and integration solution for Delphi, Microsoft Excel, and Visual Basic.

MATCOM allows MATLAB users to incorporate MATLAB's numerical and computational strength into GUI and spreadsheet tools. Scientists and engineers can develop algorithms in MATLAB and automatically incorporate them in applications created with these tools, saving a manual translation phase. The integration is achieved by compiling the algorithms to independent, royalty-free DLLs using MATCOM and incorporating the DLLs into an application. Compilation is automatic, using a smart project manager.

MathTools Ltd.
Price: US$99 for single student license, US$249 for single academic license, and US$499 for single commercial license; additional US$24, US$49, and US$99 for Delphi Integration add-in for the student, academic, and commercial licenses, respectively; site licenses and other add-ins are also available.
Phone: (888) 628-4866 or (212) 208-4476
Web Site: http://www.mathtools.com



Ashley Godfrey Releases Delphi Voyager 2

Ashley Godfrey has released *Delphi Voyager 2*, the updated version of LinkWizard for Delphi. Delphi Voyager, a resource explorer and resource management facility, offers a rebuilt user interface. Combining Delphi, ActiveX, the BDE, and the Internet, Delphi Voyager enables Delphi programmers to explore and manage their other resources.

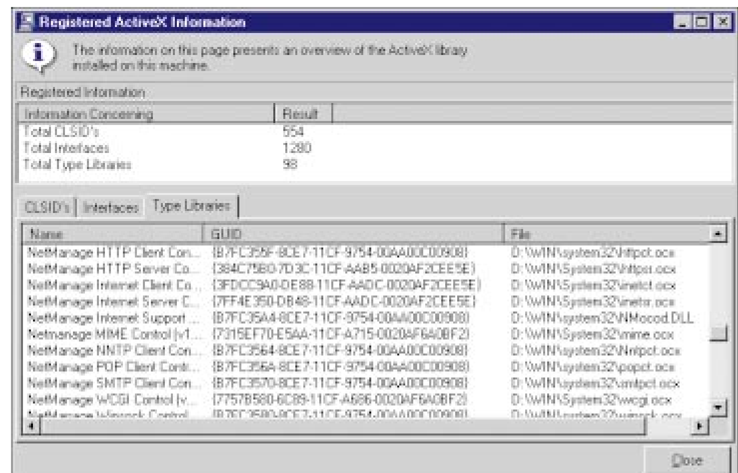
Delphi Voyager enables the viewing of the structure and contents of any registered ActiveX type library. Delphi Voyager lists all registered type libraries without the need to specify an OCX or DLL for every ActiveX library to be viewed. With Delphi 3 installed, Delphi Voyager can decompile that library.

Delphi Voyager also browses ActiveX interfaces. CLSIDs are displayed and, along with ActiveX interfaces, can be linked to their containing type library.

Delphi Voyager explores BDE installations, enumerates databases and their aliases, lists the tables of each alias, and displays their associated parameters. Also included are links to Borland's BDE tech site.

Delphi Voyager also contains a view of the Internet sites most likely to be used by a Delphi programmer. In addition, Delphi Voyager contains a Delphi Super Page offline viewer, which allows a programmer to search for and obtain a component by clicking on a download button.

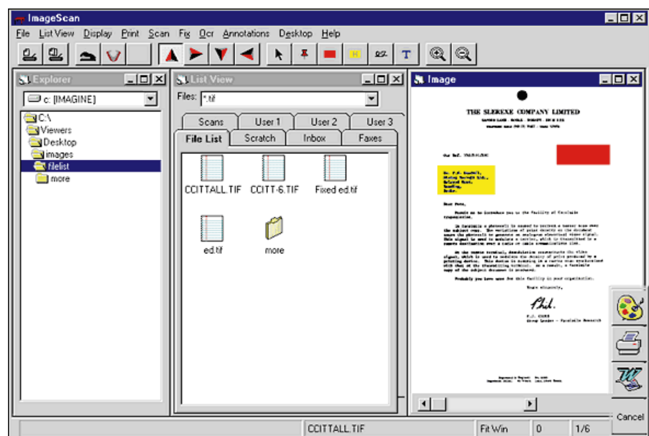
Ashley Godfrey
Price: US$60 for e-mailed copy; US$65 for diskette.
E-Mail: godfa@hotmail.com
Web Site: http://members.tripod.com/~nfi/index.html



Delphi TOOLS

New Products and Solutions

Imagination Introduces IMAGinE ActiveX Control



Imagination Software, Inc. introduced the *IMAGinE* ActiveX control, a thin-client, intranet imaging solution designed for use by the government, Fortune 500 companies, integrators, independent software vendors, and value-added resellers. The ActiveX control allows paper management capabilities to be added to an intranet. Its features include displaying, printing, annotating, fiximage, scanning, OCR, ICR, OMR, reading barcode, and processing forms. IMAGinE also comes with pre-built imaging application prototypes and their source codes. This ActiveX control can be used with Delphi, Visual Basic, Power Builder, Visual C++, or any developer language that supports ActiveX.

IMAGinE ActiveX control runs under Windows 95/NT, and is compatible with all OLE automation applications.

Imagination Software, Inc.
Price: From US$799 for IMAGinE Desktop, to US$4,999 for the IMAGinE Suite Plus.
Phone: (800) IMAGE10
Web Site: http://www.imagination-software.com

Indigo Rose Announces AutoPlay Menu Studio 1.1

Indigo Rose Corp. announced the release of *AutoPlay Menu Studio 1.1* for Windows 95 and Windows NT 4.0. This tool is designed for multimedia software developers, CD-ROM content providers, network administrators, or anyone using CD-ROM as a distribution medium.

This WYSIWYG development tool is designed for creating CD-ROM menu systems that automatically load and execute upon insertion of the CD-ROM. The drag-and-drop environment allows developers to create a customized menu system. Multimedia Action Buttons can be used to accomplish a variety of tasks without programming, including executing programs, jumping to Internet/intranet sites, opening documents, browsing local, CD-ROM, and network drives, sending e-mail, starting an installation program, printing files, displaying Help files, and downloading via FTP.

AutoPlay Menu Studio includes full support for UNC filenames, integrated palette management, alignment tools, undo/redo, background gradients, bitmap tiling, and a graphics and sound library.

Indigo Rose Corp.
Price: US$195
Phone: (800) 665-9668 or (204) 946-0263
Web Site: http://www.indigorose.com

Seagate Launches Crystal Info 6

Seagate Software announced *Seagate Crystal Info 6*, a suite of products that simultaneously incorporate Online Analytical Processing (OLAP) and push technology.

Crystal Info 6 integrates OLAP functionality and Crystal Reports into Crystal Info's enterprise-wide reporting infrastructure. The Info Cube Designer extracts information from relational databases and consolidates the data into multi-dimensional OLAP cubes. The Info Worksheet then allows users to view the contents from various perspectives. The interface allows users to pivot rows and columns, drill down, drill across, and insert graphs and calculations. The underlying data-structures of Crystal Info 6 are compatible with Seagate Holos.

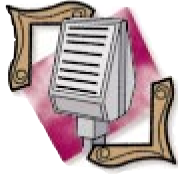
Integrating Crystal Reports Designer 6 allows users to create presentation-quality reports, and distribute them throughout the enterprise. Page-On-Demand technology downloads only the required pages of a report.

Crystal Info 6 offers report-viewing and distribution and OLAP analysis via the Web, using HTML, ActiveX, or Java. It also integrates push technology as a way to access information via the Web, allowing users to flag reports and OLAP cubes and objects.

Seagate Software
Price: US$749 per user for client license and Report/Query and OLAP add-in modules; license and modules are also sold separately.
Phone: (800) 877-2340 or (604) 681-3435
Web Site: http://www.-seagatesoftware.com

News LINE

March 1998



Borland Buys Visigenic

Borland has signed an agreement to acquire Visigenic Software, a developer of object request broker (ORB) software. Visigenic shareholders will receive .81988 shares of Borland common stock for each outstanding share of Visigenic stock. Approximately 12.5 million shares of Borland stock will be issued at the close of the transaction, slated for the first quarter of 1998.

The companies also plan to build application server software that will combine Visigenic's VisiBroker ORB with Borland's JBuilder Java development tool. Borland's chief technology officer, Rick LeFaivre, will head research and development for the combined companies.

Visigenic will gain a broader distribution channel and the tools to make its ORB more competitive. The acquisition will not affect the company's existing licensing deals with Oracle, Netscape Communications, or other software vendors.

Borland Expands in Eastern Europe

Frankfurt, Germany — Borland announced it has expanded its operations in Eastern Europe by establishing Borland Magyarorszag. Borland has transformed Delphi Szoft (Delphi Software), the company's master distributor in Hungary since January 1995, into Borland Magyarorszag. This venture reinforces Borland's presence in Hungary, providing direct contact and focus for the company's Hungarian customer base.

Borland is a major provider of development tools in Hungary and has established a strong market share as a result of its focus on the market. Borland is also a supplier of development tools to the Hungarian Ministry of Education.

For more information on Borland Magyarorszag, visit its Web site at http://www.borland.hu/ or call (49) 6103 979-281.

Apogee Awarded Premier Partner Status by Borland

Marlboro, MA — Apogee Information Systems, Inc. announced that Borland has given the firm Premier Partner status, providing increased access to strategic business and technical information.

Borland established the Premier Partner program in 1996 to shift the focus of the company to larger clients and enterprise-wide solutions. Over half of their revenues come from development tools specifically designed for those markets. Apogee's success delivering services has led to rapid, sustained growth and the upgraded partnership with Borland.

Apogee Information Systems is a custom application consulting and development firm specializing in multi-tier Information Network solutions. The company assists clients worldwide in integrating data and business processes across legacy, client/server, Internet, and intranet environments. For more information, call (508) 481-1400 or e-mail cpatton@apogeeis.com.

Keshet Broadcasting to Use Delphi in Windows-based Traffic System

Jerusalem, Israel — El-On Software Systems Ltd. has signed an agreement with IBM Israel to build a Windows-based broadcasting traffic system for Keshet Ltd., one of Israel's three commercial television concessionaires. The 32-bit application is being developed using Delphi 3. Approximately 50 client stations will be linked to an IBM RS/6000 server running Oracle version 7.3.

The application will be object-oriented and will include a visual component to handle program scheduling. The component supports Windows 95 functions. The application will enable users to display various objects along a time-ruler, as well as place an object within a container object, while modifying the container's properties. It will also include a Win32 IPC component that will synchronize data across the network. All the components will support Hebrew and English.

The application will use the Borland Database Engine and Oracle's SQL/NET client for direct SQL access to the database. The system will manage all aspects of controlling the station, including advertising spot management, tape library management, and program scheduling.

Borland Ships JBuilder Client/Server Suite

Las Vegas, NV — Borland announced JBuilder Client/Server Suite, an addition to their line of visual Java development tools. JBuilder Client/Server Suite supports development of large-scale, multi-platform Java applications by integrating Visigenic Software's VisiBroker ORB for seamless CORBA integration; Borland DataGateway Enterprise for high-performance database connectivity; InterSolv PVCS software for team management and version control; Borland's SQL tools for robust client/server development; and BeansExpress for JavaBeans creation. JBuilder also includes the DataExpress database architecture and a fully functional CORBA e-commerce reference application (with source code).

JBuilder's scalable, component-based environment is designed for all levels of information network development projects, ranging from applets and applications that require networked database connectivity, to client/server and enterprise-wide, distributed multi-tier systems. It supports 100% Pure Java, JDK 1.1, JavaBeans, JFC, CORBA, RMI, JDBC, and all major corporate database servers.

JBuilder Client/Server Suite is available for US$2,495. Owners of other Borland tools can purchase JBuilder Client/Server Suite for US$2,000. For more information, call (800) 233-2444, or visit the JBuilder Web site at http://www.borland.com/jbuilder.

table of CONTENTS

By David W. Body

DCOM Streaming

Using Delphi's Component-Streaming Mechanism to Pass Objects through DCOM

Developers increasingly find it useful to create *distributed* applications, so called because their functionality is distributed across multiple processes running on the same or different machines on a network. These processes communicate and cooperate with each other to perform the functions required of the application as a whole.

Delphi 3 provides excellent support for a variety of distributed-application techniques. For example, developers can use the Remote Data Broker technology in the Delphi 3 Client/Server Suite to create and deploy multi-tiered database applications. In these, the client process typically communicates with a middle-tier process that implements and enforces business rules. In turn, the middle-tier process typically communicates with a database-server process. These three processes can be deployed in a variety of configurations, but often each is run on a separate computer on the network.

COM Automation

Delphi 3 makes it easy to create other kinds of distributed applications without using the Remote Data Broker technology. The easiest way to do this is with COM/DCOM "automation." Automation clients can call procedures and functions in automation servers running in other processes. Through DCOM, automation clients can call procedures and functions in automation servers running on other machines on the network. For a good introduction to creating automation servers and clients with Delphi 3, see Jeremy Rule's article, "Delphi 3 DCOM," in the September 1997 *Delphi Informant*.

The nice thing about COM/DCOM automation is that it automatically takes care of all the details of passing parameters and return values across process and machine boundaries (a process known as *marshaling*). Part of the price you pay for this automatic marshaling is that you can use only automation-compatible data types for your parameters and function-return values. Figure 1 contains a list of automation-compatible data types.

At first this seems a rather serious limitation. What if you want to pass a Pascal object as a procedure parameter, or return a Pascal object from a function call? If you're limited to the data types listed in Figure 1, you have no choice but to break your object into its separate fields, and pass these one-by-one. This requires writing a separate procedure or function to pass each field, resulting in less-readable and less-maintainable code. Worse, because of the overhead associated with marshaling variables through COM/DCOM, your application's performance will suffer dramatically if you must pass many fields in this manner.

Pascal Type	OLE Variant Type	Description
Smallint	VT_I2	2-byte signed integer
Integer	VT_I4	4-byte signed integer
Single	VT_R4	4-byte real
Double	VT_R8	8-byte real
Currency	VT_CY	currency
TDateTime	VT_DATE	date
WideString	VT_BSTR	binary string
IDispatch	VT_DISPATCH	pointer to IDispatch interface
SCODE	VT_ERROR	OLE error code
WordBool	VT_BOOL	True = -1, False = 0
OleVariant	VT_VARIANT	OLE Variant
IUnknown	VT_UNKNOWN	pointer to IUnknown interface
Byte	VT_UI1	1-byte signed integer

Figure 1: A list of automation-compatible data types.

Streaming Objects

If only there were an easy way to convert complex Pascal objects to and from one or more of the types listed in Figure 1, i.e. those automatically marshaled by COM. Well there is, and it's an integral part of Delphi. Most Delphi developers are aware that you can right-click a Delphi form at design time, and choose **View as Text**. Doing so will generate text listing the form's attributes — including all of the form's components and their attributes (see Figure 2). When you right-click the text listing of your form's attributes and choose **View as Form**, Delphi converts the text back into a graphical representation of your form.

Delphi uses a similar streaming technique to save your form as a .DFM file. In this case, Delphi uses a more efficient binary representation of your form and its components. This takes less space, and makes saving and retrieving forms a little faster than using a text representation. It turns out that Delphi can automatically convert any component derived from *TComponent* to and from the text and binary representations used with forms and their components. All of the component's published (and streamable) read/write properties are automatically included in this conversion process. (Note: A property needs to be published *and* streamable. Some properties use the **stored** directive to determine if the property writes itself out to a stream. If **stored** is False, the property will not be sent over — even if it's published.)

```
object Form1: TForm1
  Left = 200
  Top = 108
  Width = 696
  Height = 480
  Caption = 'Form1'
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 306
    Top = 214
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
  end
end
```

Figure 2: A text listing of a form's attributes.

Through COM/DCOM automation we can use Delphi's built-in object-streaming capabilities to pass entire components as procedure parameters or function-return values. For example, we could simply convert a component to text, pass the text from one process to another as a WideString, then convert the text back into a component in the second process. Of course, it's slightly more efficient to convert a component to and from a binary stream of bytes (the way Delphi stores .DFM files) rather than using the text representation. Using this approach, we can pass components across process and machine boundaries with a construct known as a *variant array* of bytes.

Listing One (beginning on page 9) is a Pascal unit that contains two pairs of functions used to convert any object derived from *TComponent* to and from automation-compatible data

```
TSystemStatus = class(TComponent)
private
  FComputerName: string;
  FUserName: string;
  FMajorVersion: Integer;
  FMinorVersion: Integer;
  FBuildNumber: Integer;
  FPlatformID: Integer;
  FTotalPhys: Integer;
  FAvailPhys: Integer;
  FTotalPageFile: Integer;
  FAvailPageFile: Integer;
  FTotalVirtual: Integer;
  FAvailVirtual: Integer;
public
  procedure GetSystemStatus;
  procedure DisplaySystemStatus(Strings: TStrings);
published
  property ComputerName: string
    read FComputerName write FComputerName;
  property UserName: string
    read FUserName write FUserName;
  property MajorVersion: Integer
    read FMajorVersion write FMajorVersion;
  property MinorVersion: Integer
    read FMinorVersion write FMinorVersion;
  property BuildNumber: Integer
    read FBuildNumber write FBuildNumber;
  property PlatformID: Integer
    read FPlatformID write FPlatformID;
  property TotalPhys: Integer
    read FTotalPhys write FTotalPhys;
  property AvailPhys: Integer
    read FAvailPhys write FAvailPhys;
  property TotalPageFile: Integer
    read FTotalPageFile write FTotalPageFile;
  property AvailPageFile: Integer
    read FAvailPageFile write FAvailPageFile;
  property TotalVirtual: Integer
    read FTotalVirtual write FTotalVirtual;
  property AvailVirtual: Integer
    read FAvailVirtual write FAvailVirtual;
end;
```

Figure 3: The interface of a *TSystemStatus* component.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  SystemStatus: TSystemStatus;
begin
  SystemStatus := TSystemStatus.Create(Self);
  try
    SystemStatus.GetSystemStatus;
    SystemStatus.DisplaySystemStatus(ListBox1.Items);
  finally
    SystemStatus.Free;
  end;
end;
```

Figure 4: Putting the *TSystemStatus* component to work.

types. The *ComponentToString* and *StringToComponent* functions will convert components to and from text representations, respectively. The *ComponentToVariant* and *VariantToComponent* functions will convert components to and from binary variant-array representations.

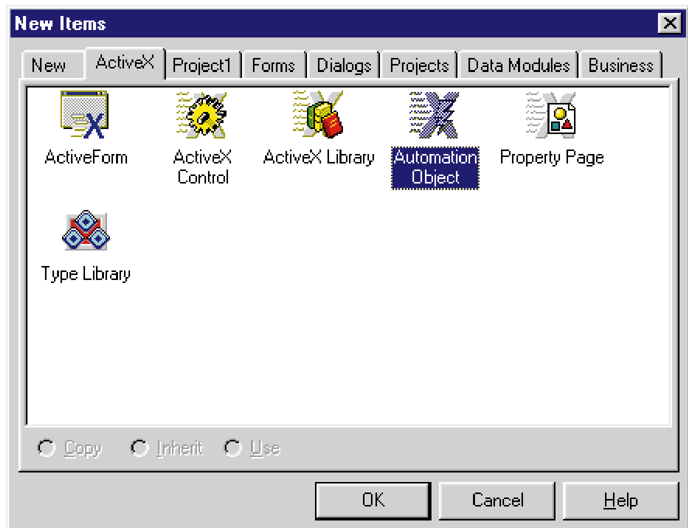


Figure 5: The ActiveX page of the New Items dialog box.

An Example

As an example of how to use these functions, let's suppose we want to monitor the system status of another machine on the network. Figure 3 depicts the interface of a *TSystemStatus* component that captures in its properties certain system attributes of interest. (Code for the implementation section is included in the source code for this article; download instructions appear at the end of this article.) This component is for illustration purposes only; you can modify it to suit your particular needs. The *TSystemStatus* component also implements a procedure called *DisplaySystemStatus*, which will display these attributes using a supplied *TStrings* parameter.

The *TSystemStatus* component is easy to use. Simply create an instance of the component by calling *TSystemStatus.Create*. A call to *TSystemStatus.GetSystemStatus* will record a snapshot of the current system status in the component's properties. To display the system status, simply call *TSystemStatus.DisplaySystemStatus*, passing it a *TStrings* component into which the *TSystemStatus* component can write its output. Figure 4 illustrates these steps in the context of an ordinary Delphi application.

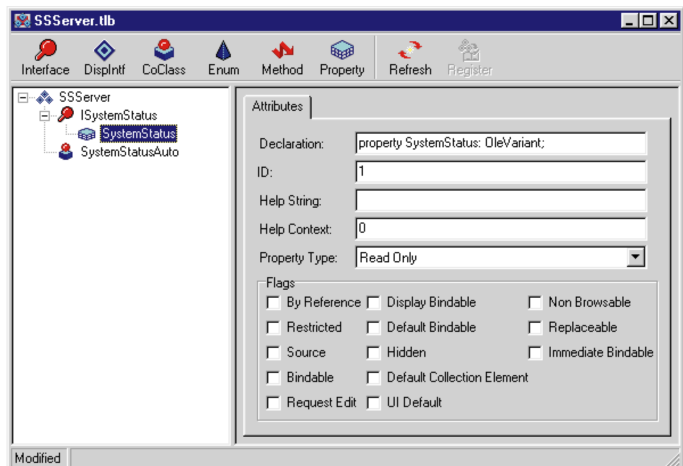


Figure 6: The Type Library Editor.

Server Application

It's almost as easy to use *TSystemStatus* through COM/DCOM automation. First, we need to create a server application to run on the computer whose system attributes we want to monitor. By adding an Automation Object we can give our server application the ability to report the system status of its host. From the **File** menu, choose **New**; then select the **Automation Object** icon from the ActiveX tab, as shown in Figure 5. Enter a class name of `SystemStatus`. Delphi will automatically create a type library for you, and display the Type Library Editor (see Figure 6). Note that Delphi has created a COM interface named *ISystemStatus*.

Click the **Property** icon to create a new property for the *ISystemStatus* interface. Name this property `SystemStatus`, change its data type to OleVariant, and change the **Property Type** to *Read Only*. (Our *TSystemStatus* component is for monitoring system status, not for modifying it.) Finally, click the **Refresh** icon. Delphi will automatically create a skeleton for a function named *Get_SystemStatus*, which you should complete as shown in Figure 7.

```
function TSystemStatusAuto.Get_SystemStatus: OleVariant;
var
  SS: TSystemStatus;
begin
  SS := TSystemStatus.Create(nil);
  try
    SS.GetSystemStatus;
    Result := ComponentToVariant(SS);
  finally
    SS.Free;
  end;
end;
```

Figure 7: The *Get_SystemStatus* function.

This function simply creates an instance of *TSystemStatus*, calls the *GetSystemStatus* method, then converts the *TSystemStatus* component into an OleVariant that can be returned as the function's result. You will need to add the CompStream unit (again, see Listing One) to your **uses** clause before compiling the server application. After the server application is complete, be sure to run it once with the /REGSERVER parameter to register it.

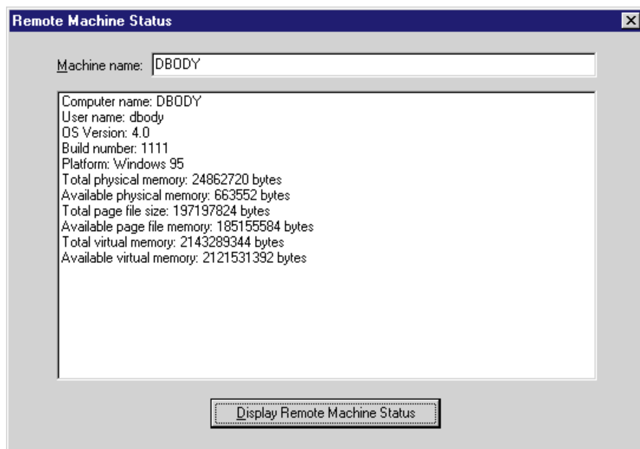


Figure 8: The client application at run time.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  SS: ISystemStatus;
  SystemStatus: TSystemStatus;
begin
  SS := CoSystemStatusAuto.CreateRemote(Edit1.Text);
  SystemStatus := _
    VariantToComponent(SS.SystemStatus) as TSystemStatus;
  try
    SystemStatus.DisplaySystemStatus(ListBox1.Items);
  finally
    SystemStatus.Free;
  end;
end;
```

Figure 9: The client application uses this code to create, use, and release the DCOM server.

Client Application

Now all we need is a client application that can retrieve and display the system status from our server application. Our client application will have a screen layout like that shown in Figure 8. Clicking the button will execute the code in Figure 9. You'll need to import the type library from our server application, and add the SystemStatus and CompStream units to your **uses** clause before compiling.

As shown in Figure 9, the client application first acquires a pointer to an *ISystemStatus* interface by calling *CoSystemStatusAuto.CreateRemote*, passing the machine name entered in the edit box. Delphi and COM/DCOM take care of establishing a connection to the server application on the specified machine. Our client then passes the OleVariant — returned by the *SystemStatus* property of *ISystemStatus* — to the *VariantToComponent* function, and casts the result to a *TSystemStatus* component using the **as** operator. This gives our client a local copy of the *TSystemStatus* component created on the server, complete with all the properties set by the server's call to *GetSystemStatus* (refer to Figure 7).

Finally, the client displays the system status of the remote machine by calling *DisplaySystemStatus*, and frees the *TSystemStatus* component. That's all there is to using the *TSystemStatus* component to monitor the system status on a remote machine.

Practical Application

I have successfully used this technique to allow multiple users to monitor and control complex calculation processes running on a server. For example, the Iowa Legislative Fiscal Bureau uses a Delphi application to project aggregate payroll costs for the State of Iowa, using assumptions supplied by users. Multiple users can simultaneously perform multiple projections. Each projection uses actual payroll data from a database containing approximately one million records. Depending on the size of the specified population and the number of years being projected, a single projection can take up to approximately 15 minutes to complete. (Note: The server is a dual-processor 166MHz Pentium Pro machine with 128MB of RAM running Windows NT 4.0 and Microsoft SQL Server 6.5.)

The calculations required for each projection are performed by a separate process running on the server, and are monitored and controlled by a "calculator manager" application also running on the server. The calculator manager communicates with client applications running on user workstations employing the technique described in this article. Performing the calculations on the server in this manner improves performance, decreases network traffic, and prevents sensitive payroll data from leaving the server during the calculations.

Conclusion

Using the technique discussed in this article, you can easily pass instances of any object derived from *TComponent* across process or machine boundaries. The technique uses Delphi's built-in component-streaming support to convert a component instance in one process into a data type that can be automatically marshaled by COM/DCOM automation. We pass the streamed component to the other process and convert it back into a component instance, again using Delphi's built-in streaming support. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\MAR\DI9803DB.

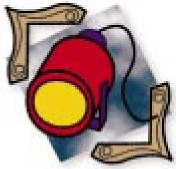
David W. Body is the owner of Big Creek Software, a custom software development and consulting firm specializing in business, financial, and legal applications (http://www.bigcreek.com). David is Borland Delphi 3 Client/Server certified, and president of the Central Iowa Delphi Users Group.

Begin Listing One — The CompStream Unit

```
unit CompStream;

interface

uses Classes;

function ComponentToString(Component: TComponent): string;
function StringToComponent(Value: string): TComponent;
function ComponentToVariant(Component: TComponent):
  Variant;
```

```
function VariantToComponent(Value: Variant): TComponent;

implementation

function ComponentToString(Component: TComponent): string;
var
  BinStream: TMemoryStream;
  StrStream: TStringStream;
  s: string;
begin
  BinStream := TMemoryStream.Create;
  try
    StrStream := TStringStream.Create(s);
    try
      BinStream.WriteComponent(Component);
      BinStream.Seek(0, soFromBeginning);
      ObjectBinaryToText(BinStream, StrStream);
      StrStream.Seek(0, soFromBeginning);
      Result := StrStream.DataString;
    finally
      StrStream.Free;
    end;
  finally
    BinStream.Free;
  end;
end;

function StringToComponent(Value: string): TComponent;
var
  StrStream: TStringStream;
  BinStream: TMemoryStream;
begin
  StrStream := TStringStream.Create(Value);
  try
    BinStream := TMemoryStream.Create;
    try
      ObjectTextToBinary(StrStream, BinStream);
      BinStream.Seek(0, soFromBeginning);
      Result := BinStream.ReadComponent(nil);
    finally
      BinStream.Free;
    end;
  finally
    StrStream.Free;
  end;
end;

function ComponentToVariant(Component: TComponent):
  Variant;
var
  BinStream: TMemoryStream;
  Data: Pointer;
begin
  BinStream := TMemoryStream.Create;
  try
    BinStream.WriteComponent(Component);
    Result := VarArrayCreate([0,BinStream.Size-1],varByte);
    Data := VarArrayLock(Result);
    try
      Move(BinStream.Memory^, Data^, BinStream.Size);
    finally
      VarArrayUnlock(Result);
    end;
  finally
    BinStream.Free;
  end;
end;

function VariantToComponent(Value: Variant): TComponent;
var
  BinStream: TMemoryStream;
  Data: Pointer;
begin
  BinStream := TMemoryStream.Create;
  try
    Data := VarArrayLock(Value);
    try
      BinStream.WriteBuffer(Data^,
                            VarArrayHighBound(Value,1)+1);
    finally
      VarArrayUnlock(Value);
    end;
    BinStream.Seek(0, soFromBeginning);
    Result := BinStream.ReadComponent(nil);
  finally
    BinStream.Free;
  end;
end;

end.
```

End Listing One



Delphi 3.01 / ActiveX

By Dan Miser

Deploying ActiveX Controls

Techniques for Testing, Deployment, Security, and More

Last month, we offered a handy introduction of different ways to create and use ActiveX controls within Delphi. This month, we'll demonstrate how to deploy those controls to a Web site, as well as how to debug your ActiveX controls.

Deploying ActiveX Controls

Deploying an ActiveX control to a Web server is straightforward, but one wrong move and people won't be able to use the ActiveX control from their browsers. Fortunately, everything you need to make your ActiveX deployment successful is wrapped up in one dialog box. If you have an ActiveX project open — for example, the CalendarX.DPR project we created last month — you can select the menu item **Project | Web Deployment Options** to display the dialog box shown in Figure 1. The options set here reflect a typical installation of Microsoft's Personal Web Server, which runs under Windows 95.

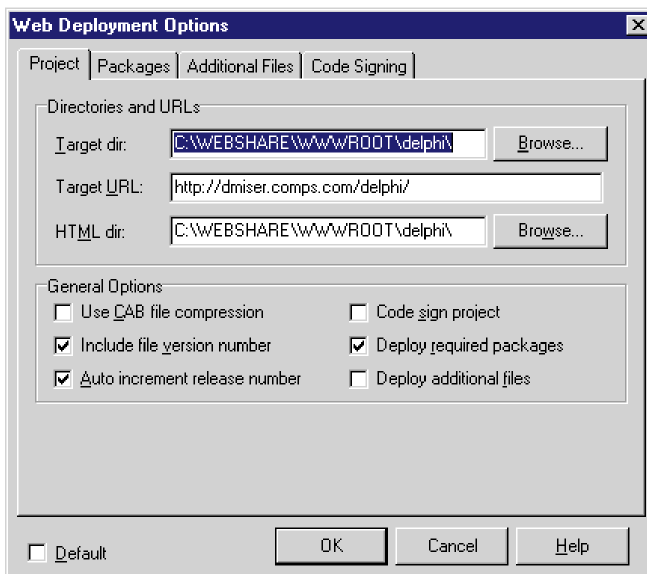


Figure 1: The Web Deployment Options dialog box.

Note: If you fully qualify your URL with the domain name, you must specify the protocol (i.e. http://) in the **Target URL** edit box, or the control will fail to download properly. Alternatively, you can designate that the control is in the same directory by placing "./" in **Target URL**.

Packages. The footprint required to create even a minimal ActiveX control is a couple of hundred kilobytes. We can reduce the size, however, by using run-time packages. This way, the user must only download the run-time packages once, and can gain substantial download savings each time they download any Delphi control compiled with run-time packages.

Using run-time packages for an ActiveX control is as simple as using run-time packages for an application. Select **Project | Options**, then select the **Build with runtime packages** check box on the Packages page of the Project Options dialog box. Once you've compiled your project with run-time packages, you can enable the deployment of those packages by checking the **Deploy required packages** check box on the Project tab of the Web Deployment Options dialog box.

Installing multiple files with your ActiveX control will require you to use some combination of an .INF file and a .CAB file. An .INF file is an installation script that points to the loca-

```
[Add.Code]
packages.ocx=packages.ocx
VCL30.dpl=VCL30.dpl

[packages.ocx]
file=http://dmiser.comps.com/delphi/packages.ocx
clsid={ 03B3F168-C13B-11D0-9BB9-00A024604D21 }
RegisterServer=yes
FileVersion=1,0,0,0

[VCL30.dpl]
file=http://www.borland.com/vcl30.cab
FileVersion=3,0,5,53
```

Figure 2: Sample .INF file for an ActiveX control built with run-time packages.

tions of all the files needed to make your ActiveX installation complete. .CAB files allow multiple files to be distributed over the Internet in compressed form, much like a .ZIP file. If you distribute many files, or if the file sizes are large, you may want to distribute the files in compressed mode via the .CAB file.

Borland has code-signed some of their run-time packages and placed them in .CAB files on their Web site (http://www.borland.com). Code-signing these packages proves that the run-time package files the end user receives are the same ones that Borland developed (see Figure 2).

Code-signing. ActiveX controls pose a potential security risk to clients who download them. For this reason, Microsoft has proposed a technology called *Authenticode*. Authenticode assures end users of the ActiveX control's origin, and that it hasn't been tampered with since the control has been signed. While this is not foolproof, it does provide some degree of accountability for the majority of cases.

To obtain certification for your ActiveX control, you must contact an authorized Certification Authority (one such company is VeriSign; http://www.verisign.com). You can apply for a certificate; upon acceptance, the authorized Certification Authority will return a certificate file you can use to sign a control.

For any serious ActiveX development, you must obtain the ActiveX SDK from Microsoft. In the ActiveX SDK you'll find documentation, online resources, and a suite of programs to help develop ActiveX content. For example, the program MAKECER generates test certificates you can use for local testing. For more information, visit http://www.microsoft.com/activex.

Deploy. When all the options have been properly set, select Project | Web Deploy to execute the deployment. This will copy the ActiveX control, an .INF file and a .CAB file if appropriate, and a skeleton HTML file to the directory specified. If your Web site isn't hosted locally, you can still deploy all the files to a local directory and use the same method of moving the files to your host directory after deployment.

One more note: Writing an ActiveX control or ActiveForm that accesses a database requires the client machine to have a copy of the BDE (Borland Database Engine) installed. Furthermore, the client machine must be able to physically connect to the database. For this reason, database applications are best suited to intranet deployment. If you do choose to deploy an ActiveX control that needs database access, Delphi 3.01 includes a .CAB file and an .INF file to take care of the BDE installation process. See the file BDEINST.TXT on the Delphi 3.01 CD for more information on how to do this. On the other hand, if you want a true thin-client solution over the Internet, you will need to use MIDAS.

Using ActiveX in HTML

HTML syntax. ActiveX controls are frequently used in HTML pages. The `<OBJECT>` tag is used to identify the control to the browser. In addition, several attributes are used for this tag to help the browser display the proper ActiveX control to the client:

- `CLASSID`: CLSID registered to the ActiveX control
- `CODEBASE`: URL used by the browser to download the control to the client

For more information on this tag, see the HTML working draft at http://www.w3.org/pub/WWW/TR/WD-object.html.

Version control. If you deploy your ActiveX control with version information enabled, the browser will automatically detect whether it needs to update the control. The syntax of the HTML to take advantage of version control is to append the version number to the `CODEBASE` attribute. For example, the following HTML code will only download the control if the version number of the control is later than 1.1:

```
<OBJECT
  CLASSID="clsid:7FD22F02-C0E1-11D0-9BB9-00A024604D21"
  CODEBASE=
    "http://dmiser.comps.com/calendarx.ocx#version=1,1,0,0"
>
```

Once the ActiveX control is deployed to a client, there will be two glaring features that need to be addressed:

1) How to use property names in HTML
2) Safety warnings when using the ActiveX control in HTML

Assigning property names. Delphi-created ActiveX controls implement the saving and restoring of property values through a standard persistence model. This provides a way for the ActiveX control to tell others about its properties. For example, the HTML listing in Figure 3 is the result of adding a Delphi-created ActiveX control to an HTML page using Microsoft ActiveX Control Pad.

```
<OBJECT ID="CalendarX1" WIDTH=320 HEIGHT=120
  CLASSID="CLSID:7FD22F05-C0E1-11D0-9BB9-00A024604D21"
DATA="DATA:application/x-oleobject;BASE64,BS/Sf+HA0BGbuQCgJ
GBNIVRQRjAJVENhbGVuZGFyAARMZWZ0AgADVG9wAgAFV2lkdGgDQAEGSGVp
Z2h0AngLU3RhcnRPZldlZWsCAAAASA=">
</OBJECT>
```

Figure 3: An <OBJECT> tag for an imported ActiveX control. (Note: There cannot be carriage returns in the actual DATA string; they were added here for formatting purposes.)

```
TCalendarX = class(TActiveXControl, ICalendarX,
                   IPersistPropertyBag)
  function IPersistPropertyBag.Load =
             PersistPropertyBagLoad;
  function IPersistPropertyBag.Save =
             PersistPropertyBagSave;
  function PersistPropertyBagLoad(
    const pPropBag: IPropertyBag;
    const pErrorLog: IErrorLog): HResult; stdcall;
  function PersistPropertyBagSave(
    const pPropBag: IPropertyBag; ClearDirty: BOOL;
    fSaveAllProperties: BOOL): HResult; stdcall;
end;
```

Figure 4: Implementation of *TCalendarX.IPersistPropertyBag*.

```
function TCalendarX.PersistPropertyBagSave(
  const pPropBag: IPropertyBag; fClearDirty: BOOL;
  fSaveAllProperties: BOOL): HResult; stdcall;
begin
  PutPropInBag(pPropBag, 'Color', FDelphiControl.Color);
  Result := S_OK;
end;
```

Figure 5: The *IPersistPropertyBag.Save* function.

```
<OBJECT ID="CalendarX1" WIDTH=320 HEIGHT=120
  CLASSID="CLSID:7FD22F05-C0E1-11D0-9BB9-00A024604D21">
  <PARAM NAME="BorderStyle" VALUE="1">
  <PARAM NAME="CalendarDate" VALUE="5/29/97">
  <PARAM NAME="Color" VALUE="-2147483643">
  <PARAM NAME="Ctl3d" VALUE="-1">
  <PARAM NAME="Cursor" VALUE="0">
  <PARAM NAME="Day" VALUE="29">
  <PARAM NAME="Enabled" VALUE="-1">
  <PARAM NAME="FontName" VALUE="0">
  <PARAM NAME="FontSize" VALUE="8">
  <PARAM NAME="GridLineWidth" VALUE="3">
  <PARAM NAME="Month" VALUE="5">
  <PARAM NAME="ParentColor" VALUE="0">
  <PARAM NAME="ReadOnly" VALUE="0">
  <PARAM NAME="StartOfWeek" VALUE="0">
  <PARAM NAME="UseCurrentDate" VALUE="-1">
  <PARAM NAME="Visible" VALUE="-1">
  <PARAM NAME="Year" VALUE="1997">
</OBJECT>
```

Figure 6: An <OBJECT> tag with *IPersistPropertyBag* implemented.

All the properties are set in the DATA element of this tag; however, there is no easy way to look at this data to determine what the value of a given property is. In addition, by using this method, the ActiveX control is more difficult to control in an HTML scripting environment, such as VBScript or JavaScript.

Microsoft defined the *IPersistPropertyBag* interface to turn the unreadable data into human-readable properties. To implement this interface, we simply need to add the two key methods, *Load* and *Save*, to our existing ActiveX implementation. However, because the *IPersistPropertyBag* interface already defines these methods, we must use Delphi's method resolution clauses to make our implementation work (see Figure 4). For more information on this topic, look up "Method resolution clauses" in Delphi's online Help.

The *IPersistPropertyBag* interface referenced in the *Load* and *Save* methods is the interface responsible for reading and writing individual properties. You can find the helper routines *ReadPropFromBag* and *PutPropInBag* in \Delphi 3\SOURCE\RTL\SYS\comobj.pas. These procedures make it easy to implement an *IPersistPropertyBag* interface for any ActiveX control (see Figure 5).

Each item you add to the property bag in this function will show up as a <PARAM> for the <OBJECT> tag. This provides another level of HTML interactivity by allowing the use of these properties in an HTML script (see Figure 6).

Automating persistence. We can borrow some logic to implement this interface in a more generic manner by looking at the demonstration application Pbag.dpr found in the \Delphi 3\Demos\Activex\Propbag directory. This demonstration was included in the Delphi 3.01 update. In it, the *IPersistPropertyBag* interface reads and writes the properties of the control using RTTI (Run-Time Type Information). Using this method ensures we won't forget any properties that need to be streamed.

It would be nice to use Borland's implementation of *IPersistPropertyBag* in all our controls. However, because COM only allows interface inheritance, we have but two options:
1) Place the common code in a separate unit and implement the interface for every ActiveX control that requires the *IPersistPropertyBag* interface.
2) Create a new component derived from *TActiveXControl* that implements the *IPersistPropertyBag* interface. By doing this, all that must be changed is the inheritance of your new ActiveX control from *TActiveXControl* to *TActiveXPropBag*. This will allow your control to automatically receive the *IPersistPropertyBag* implementation.

By descending from Delphi's object reference, as opposed to COM's interface reference, we can achieve implementation inheritance.

Marking the ActiveX control as safe. When using an ActiveX control in a script, Microsoft Internet Explorer (IE) will check if the control can be safely scripted. If the control is not deemed safe, a warning message will be displayed (see Figure 7). By default, any control in a script will generate this warning. According to the ActiveX SDK documenta-

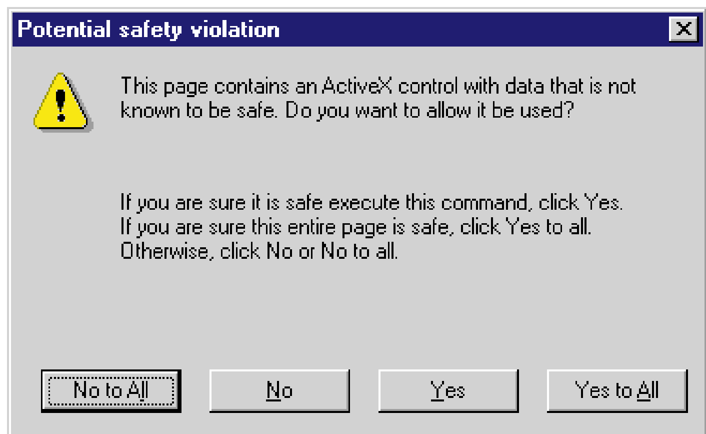


Figure 7: The error displayed when a control is not marked safe for scripting.

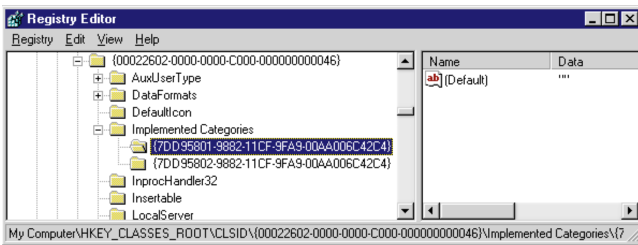


Figure 8: Regedit displaying a safe control.

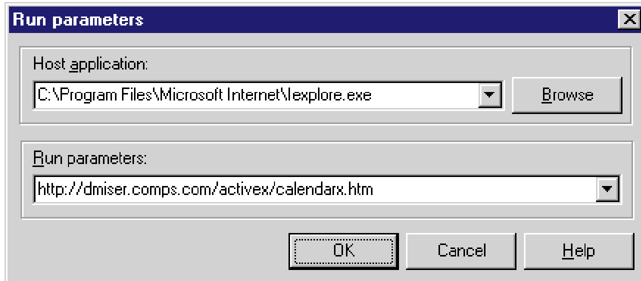


Figure 9: Setting run parameters to debug Personal Web Server.

tion, there are three ways to suppress this warning message:

1) Turn off the security checking option for all scripts in IE.
2) Register the control as "safe for scripting" in the Windows registry.
3) Implement the *IObjectSafety* interface.

Let's examine the implications of each scenario.

Marking all controls as "safe for scripting" in IE isn't a very secure way to stop these messages from occurring. Some controls might really be unsafe for scripting, and the user should be warned about those instances. Implementing the *IObjectSafety* interface is overkill for most ActiveX controls, as a control is usually either safe or unsafe in its entirety. Therefore, let's concentrate on the second option — marking the control as safe in the registry.

IE will check for the existence of two specific registry entries for every control that is placed in a script (see Figure 8). If these entries exist, IE knows the author has marked the control as safe, and will not display a warning. The two registry entries are CATID_SafeForScripting and CATID_SafeForInitializing. The entries are special because they are defined as Category IDs. A Category ID is created to group components into a particular category. To belong to these categories, the author of the control must believe the control will not harm your system — no matter how the ActiveX control is used in the script.

TActiveXControlFactory is the class factory responsible for creating and registering ActiveX controls. In this class factory, there is a method called *UpdateRegistry* that maintains registry settings for an ActiveX control. This seems like a logical place to add the registry entries that will mark this control as safe. Creating our class factory is as easy as descending the *TActiveXControlFactory* class and manipulating the registry settings in the overridden *UpdateRegistry* method:

```
TActiveXSafeFactory = class(TActiveXControlFactory)
  procedure UpdateRegistry(Register: Boolean); override;
end;
```

After implementing this class, change the *TActiveXControlFactory.Create* statement in the **initialization** section of the project to *TActiveXSafeFactory.Create*. Then, every time the control gets registered or unregistered, the registry settings that belong to the control will be automatically updated as well.

Testing ActiveX Controls

DLL debugging. Previous versions of Delphi required you to buy Borland's add-on package, Turbo Debugger for Windows (TDW), to debug DLLs. Delphi 3 allows you to debug the currently loaded DLL inside the Delphi IDE. To accomplish this, you need to open the DLL project file, then specify an .EXE that will use this DLL. This is known as the host application. Select **Run | Parameters** to bring up a dialog box where you can specify a host application and any command-line criteria for that application. Select **Run | Run** to start the host application. If you have a breakpoint set in the DLL source files, Delphi will stop execution of the host application and pass control to the Delphi IDE, where you can evaluate variables and step through code.

Remember that an ActiveX control is nothing more than a DLL. IE is an application that uses these special DLLs. Therefore, we can use the debugging technique previously described to debug our ActiveX controls.

Figure 9 shows the settings needed to debug the CalendarX project using IE as the host application. Place a breakpoint in the *InitializeControl* method in the unit CalImpl.pas. After modifying the run parameters for the project, execute the **Run | Run** command to launch IE. When the ActiveX control is loaded, the Delphi IDE will regain focus, and you can use the integrated debugger the same way you always have. To stop the debugging session, exit IE.

Conclusion

These two articles have covered a lot of ground in the ActiveX arena. However, even this lengthy treatment only scratches the surface of things that can be done with this impressive component technology. To further understand ActiveX, read the ActiveX SDK documentation; it's a price well worth paying as the line blurs between Delphi and ActiveX components. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\MAR\DI9803DM.

Dan Miser is a software developer residing in Southern California with his wife and daughter. He has been a Borland Certified Client/Server Developer since 1996, and is a frequent contributor to *Delphi Informant*. You can contact him at http://www.iinet.com/users/dmiser.

By Rod Stephens

Sounds Gud to Me

Soundex Encoding in Delphi

When you give your name at a restaurant, it doesn't matter whether the host spells it correctly. As long as the name can be pronounced to tell you when your table is ready, it can be spelled any number of ways. In a large database of names, however, spelling is critical. If you tell customer support that your name is MacCauslin, it matters a great deal whether they type "MacCauslun," "McCawslin," or "Mack Awzlin." If the database is large, finding the correct record by trial-and-error may be difficult. Even seemingly ordinary names like Smith can have many different spellings: Smith, Smithe, Smyth, Smythe, etc.

This problem is common in large name-database programs such as customer-support hotlines, company phone directories, and census-data files. Manually searching for a record slows the operator, and increases the length of the call. Eventually the operator may need to ask the customer for the correct spelling, further delaying the process — and possibly annoying the customer.

This article describes *soundex*, a method for encoding names based on how they sound, rather than how they are spelled. Adding soundex to a database application can make record retrieval faster and less frustrating for both the operator and the customer.

A Sound Beginning

Soundex methods come in several variations. The United States Census uses one of the simplest. The census represents a name using a letter followed by three numbers. For example, STEPHENS is represented by S315. The rules for creating a census-style soundex encoding are:

1) Remove vowels and H, W, and Y, except as the first letter.
2) Encode the character according to the key in Figure 1.
3) Remove adjacent duplicate digits. For example, change 552331 to 5231.
4) The encoding is the first letter followed by the second, third, and fourth numeric codes.

0	A, E, I, O, U, H, W, Y
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R

Figure 1: US Census Bureau soundex character encoding key.

For example, suppose you want to encode the name STEPHENS. In step 1 you remove the vowels, along with H, W, and Y, to get STPNS. In step 2 you encode the letters to get 23152. This value has no adjacent duplicates, so step 3 leaves the result unchanged. The final code is S315.

Compare this to the encoding of the other common spelling of this name: STEVENS. Step 1 changes the name to STVNS. Encoding the letters in step 2 yields 23152. This contains no adjacent duplicates, so step 3 has no effect. The final encoding is S315 — the same value as the encoding for STEPHENS. This is the important property of soundex encodings: Names that sound similar have similar encodings.

For another example, consider the name MacCauslin. Removing the vowels in step 1 leaves MCCSLN. Encoding the remaining letters in step 2 yields 522245. Removing duplicates in step 3 leaves 5245, so the final soundex encoding is M245. You can verify that this is the same as the encodings for MacCauslun, McCawslin, and Mack Awzlin.

Figure 2 shows a Delphi function that returns a census-style soundex encoding for a string. The Soundex program available on this month's Companion Disk uses this code to create census-style soundex encodings (see end of article for download

```
// Calculate a normal soundex encoding.
function Soundex(in_str: string) : string;
var
  no_vowels, coded, out_str: string;
  ch: Char;
  i : Integer;
begin
  // Make upper case; remove leading and trailing spaces.
  in_str := Trim(UpperCase(in_str));

  // Remove vowels, spaces, H, W, and Y, except as the
  // first character.
  no_vowels := in_str[1];
  for i := 2 to Length(in_str) do begin
    ch := in_str[i];
    case ch of
      'A', 'E', 'I', 'O', 'U', ' ', 'H', 'W', 'Y':
        ;  // Do nothing.
    else
      no_vowels := no_vowels + ch;
    end;
  end;

  // Encode the characters.
  for i := 1 to Length(no_vowels) do begin
    ch := no_vowels[i];
    case ch of
      'B', 'F', 'P', 'V':
        ch := '1';
      'C', 'G', 'J', 'K', 'Q', 'S', 'X', 'Z':
        ch := '2';
      'D', 'T':
        ch := '3';
      'L':
        ch := '4';
      'M', 'N':
        ch := '5';
      'R':
        ch := '6';
    else    // Vowels, H, W, and Y as the first letter.
      ch := '0';
    end;    // End case ch.
    coded := coded + ch;
  end;    // End for i := 1 to Length(no_vowels)

  // Use the first letter.
  out_str := no_vowels[1];

  // Find three non-repeating codes.
  for i := 2 to Length(no_vowels) do begin
    // Look for a non-repeating code.
    if (coded[i] <> coded[i - 1]) then
      begin
        // This one works.
        out_str := out_str + coded[i];
        if (Length(out_str) >= 4) then
          Break;
      end;
  end;

  Soundex := out_str;
end;
```

Figure 2: Converting text into census-style soundex.

details). The program also demonstrates other techniques described later in this article. Enter a string, then click the Encode button to make the program display the different kinds of encodings.

Now What?

Once you know how to create soundex encodings, you can use them in database applications. Start by computing soundex encodings for each record, and storing them in the database. If you use a relational database, make the soundex encoding a key, because you'll often search for the encoding.

```
// Calculate a numeric soundex encoding.
function NumericSoundex(in_str: string) : Smallint;
var
  value: Integer;
begin
  // Calculate the normal soundex encoding.
  in_str := Soundex(in_str);

  // Convert this to a numeric value.
  value := (Ord(in_str[1]) - Ord('A')) * 1000;
  if (Length(in_str) > 1) then
    value := value + StrToInt(Copy(in_str, 2,
                                   Length(in_str) - 1));
  NumericSoundex := value;
end;
```

Figure 3: Converting a string encoding into a numeric encoding.

To find a name, the program searches the database for the name's soundex code. If it finds more than one match, the program can present a list of names for the user to choose from. The program could even arrange the names so the most likely match comes first. For example, suppose the user enters STEVENS, ROD, and the best matches are STEVENS, MIKE and STEPHENS, ROD. If the program checks the soundex encoding of the first names as well as the last names, it can conclude that the second entry is probably the right one.

Soundex is useful for tasks other than name finding. For example, it's sometimes used in address-matching software. Given a street address that may have been misspelled, the program can use soundex to find possible correct street names. This type of software is usually customized, so it knows about the most common mistakes in street addressing. For example, MAIN AVE N and N MAIN AVE may be the same street.

Soundex is also used in some spelling checkers. If the user incorrectly types CRL, the program can ask if this should be CARL, CAROL, CURL, CORAL, or CHORAL, because these words have soundex codes similar to the code for CRL.

Optimization

Census-style soundex codes are four-character strings. You can make operations faster using integer codes instead of strings. You can convert a soundex encoding into an integer by using the code shown in Figure 3, which converts a soundex string encoding into a numeric code. The example program named Soundex uses this function to display numeric soundex encodings.

Soundex Variations

An advantage of the census-style soundex is that every name is represented by a four-character string that a program can easily translate into an integer. The small number of possible codes makes it likely that a word's soundex encoding will match the encoding of the correct spelling. In other words, when the user guesses how to spell a name, the program will probably find the correct record.

On the other hand, the program will also probably find a lot of incorrect records. A letter followed by three digits between 1

and 6 can generate only 26 * 6 * 6 * 6, or 5,616 possible codes. If you have a large database, many entries must map to the same code. For example, if your database has 600,000 records, each code will correspond to more than 100 records (on the average). Because names are not evenly distributed (e.g. more names start with S than with Q), most codes correspond to even more names. Finding the correct name in a list of more than 100 can be hard.

Short codes also mean the system cannot distinguish between long names that start the same way. For example, STEFFAN and STEVENOWSKI both have codes S513 — even though only their beginnings sound alike. The program will group these names together, while a human can easily tell they are not different spellings of the same name.

Even for shorter names, this scheme's simplistic method for giving similar letters the same code sometimes makes it group names that do not sound alike. For instance, MAGEE and MEESEK both have code M2, though they sound very different.

For similar reasons, this scheme sometimes assigns very different encodings for names that sound similar. For example, the code for PHISHMAN is P25, while the code for FISHMAN is F25. It would be difficult for a program to realize that P25 and F25 represented names with the same pronunciation.

Other versions of the soundex system address these issues. They replace common pairs of letters such as PH with shorter equivalents such as F. Some of these variations use more than four characters to encode a name. They also use the letters themselves, instead of numeric codes. The steps for generating one soundex variation are:

1) Convert CHR to CR, PH to F, and Z to S.
2) Remove adjacent duplicates.
3) Remove vowels, except the first letter.

(It's been suggested that these rules were used to name the UNIX commands. In many cases, the truth is much stranger.)

Following these rules, the code for STEFFAN is STFN, and the code for STEVENOWSKI is STVNWSK. These codes are still somewhat similar — because the words themselves are similar — but are different enough for a program to tell them apart. The new code for MAGEE is MG, and the new code for MEESEK is MSK. These new values correctly distinguish between the two. Finally, PHISHMAN and FISHMAN now both encode to FSHMN. The names sound alike, and now have the same code.

Figure 4 shows Delphi code that generates this new kind of soundex encoding. The Soundex example program uses this code to display extended encodings.

```
// Calculate an extended soundex encoding.
function ExtendedSoundex(in_str : string) : string;

  // Replace instances of fr_str with to_str in str.
  procedure ReplaceString(var str : string;
    fr_str, to_str : string);
  var
    fr_len, i : Integer;
  begin
    fr_len := Length(fr_str);
    i := Pos(fr_str, str);
    while (i > 0) do begin
      str := Copy(str, 1, i - 1) + to_str +
             Copy(str, i + fr_len,
                  Length(str) - i - fr_len + 1);
      i := Pos(fr_str, str);
    end;
  end;

var
  no_vowels : string;
  ch, last_ch : Char;
  i : Integer;
begin
  // Make upper case and remove
  // leading and trailing spaces.
  in_str := Trim(UpperCase(in_str));

  // Remove internal spaces.
  ReplaceString(in_str, ' ', '');

  // Convert CHR to CR.
  ReplaceString(in_str, 'CHR', 'CR');

  // Convert PH to F.
  ReplaceString(in_str, 'PH', 'F');

  // Convert Z to S.
  ReplaceString(in_str, 'Z', 'S');

  // Remove vowels and repeats.
  last_ch := in_str[1];  // The last character used.
  no_vowels := last_ch;
  for i := 2 to Length(in_str) do begin
    ch := in_str[i];
    case ch of
      'A', 'E', 'I', 'O', 'U':
        ;     // Do nothing.
    else
      // Skip it if it's a duplicate.
      if (ch <> last_ch) then
        begin
          no_vowels := no_vowels + ch;
          last_ch := ch;
        end;
    end;
  end;

  ExtendedSoundex := no_vowels;
end;
```

Figure 4: An extended soundex algorithm.

Other Enhancements

Other soundex algorithms use different rules to transform spellings into codes, based on the way they probably sound. Some change PF into F when it occurs at the beginning of a word, so PFEIFER, FIFER, and PHEIFER all have the same code. Others replace X with KS, so SOX and SOCKS have the same code: SKS.

The *metaphone* algorithm uses a series of rules to create phonetic codes for English words. The rules are quite complicated. For example, the letter C is converted into:

- Nothing (i.e. it's silent) if it occurs in SCI (conscience), SCE (ascent), or SCY (scythe).
- X, to represent the "sh" sound if it occurs in CIA (associate) or CH (luncheon).

- S if it occurs in CI (voicing), CE (forceful), or CY (cyclone).
- K otherwise (preschool).

Because metaphone is optimized for English spellings, it may not work well for all applications. In particular, it may have trouble with foreign names. You can find a more detailed description of metaphone, and Delphi source code for a metaphone library, at http://www.intellex.net/~wcs/delphi/-program.html.

Save Your Fingers

Adding soundex searches to an application can make finding records faster and less frustrating. Once users become comfortable with soundex, they will discover many shortcuts. They may start skipping vowels, typing only the beginning of a name, and even entering the beginnings of extended soundex codes. When a customer says "MALLACHIO," the user can enter "MALCH," and pick the correct entry from a short list. After a while, users may wonder how they avoided wearing their fingers to the bone typing all those extra characters. Of crs y shldnt cry ths id t fr. ("Of course, you shouldn't carry this idea too far.") ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\MAR\DI9803RS.

Rod Stephens is the author of several books, including *Visual Basic Algorithms* from John Wiley & Sons, Inc. He writes an algorithm column in *Visual Basic Developer*; some of the material presented here has appeared there in Visual Basic form. Reach him at RodStephens@compuserve.com or see what else he's doing at http://www.wiley.com/compbooks/stephens.

DBNAVIGATOR

Delphi 3

By Cary Jensen, Ph.D.

More Code Editor Tricks

Getting the Most from Delphi's Native Editor

Pop quiz: Does Delphi's Code Editor support keystroke recording and playback? If you answered "No," you're missing at least one feature that can save you a great deal of effort. The fact is, very few Delphi developers use the Code Editor to its full capacity.

Last month's "DBNavigator" contained an in-depth look at Code Insight, but there's a lot more to coding productivity than that offered by Code Insight. Delphi's editor has a wealth of features; so many, in fact, that I cannot fit a detailed discussion of them all into this article. Instead, I'm going to highlight some of my favorite techniques, leaving you to discover the remaining capabilities.

General Editor Issues

Before we start to explore some of its features, a general comment about Delphi's Code Editor is in order. The editor has four keystroke mappings. These mappings — Default, Classic, Brief, and Epsilon — define which features are available, and how you access them. The Default keystroke mapping provides keystroke mapping that is CUA-compliant. Most Windows applications use these keystrokes, making this setting a good choice if you don't already have a preference.

The other mappings are provided to permit you to customize the editor to emulate an editor you're already familiar with. For example, the Classic keystroke mapping emulates the editor-key combinations from Borland Pascal. Likewise, Brief and Epsilon modes emulate the keystrokes of those editors.

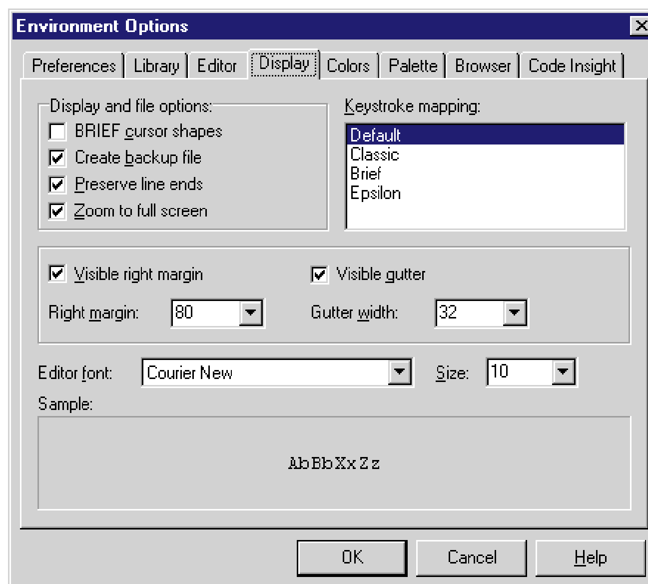


Figure 1: You control keystroke mapping using the Display page of the Environment Options dialog box.

You select which editor emulation you want from the **Keystroke mapping** list box on the Display page of the Environment Options dialog box (see Figure 1). You can display this dialog box by selecting **Tools | Environment Options**. You can further configure the editor using the **Editor SpeedSetting** combo box on the Editor page.

The techniques in this article refer to Default keystroke mapping key combinations, so if you aren't using the Default setting, the keystrokes may not work for you. In that case, use Delphi's online Help to find the appropriate keystroke combinations for a particular feature. In addition, a given keyboard mapping may provide several alternative keystroke combinations for accessing a particular feature. Again, refer to the online Help for this information.

Default keystroke mapping

The Default keystroke mapping scheme provides key bindings that match the CUA standard. For detailed information, choose one of the topics below for a list of keyboard shortcuts:

- Clipboard control
- Debugger
- Editor
  - Block commands
  - Bookmark operations
  - Cursor movement
  - Miscellaneous commands
- System

Figure 2: Use this hypertext window to display help for the category of keyboard mappings you're interested in.

Getting Help

There are several ways to access Delphi's Help pages regarding keyboard mappings. The easiest is to search on "keyboard shortcuts" in online Help. The Help system will display the Topics Found dialog box; select **About keyboard shortcuts** and press Enter↵. Then, from the displayed Help page, select the name of the keyboard emulation you've selected. This will result in the display of a hypertext window like the one shown in Figure 2, containing links to the Help pages for each category of editor operation. For example, if you want to view the keystrokes associated with cutting, copying, and pasting, select **Clipboard control**.

Key Macro Recording

A key macro is a series of one or more keystrokes you can record and play back repeatedly. The keystrokes that you record as part of your macro can contain simple characters from the keyboard, but can also include navigation keys, and even keystroke combinations. As a result, it's possible to record a sequence of keystrokes that can quickly and efficiently apply a wide range of changes to your code.

To begin recording a key macro, press Ctrl Shift R. The word **Recording** appears in the third panel of the Code Editor's status bar. At that point, each key press or key combination you enter is recorded. To conclude your recording, press Ctrl Shift R again.

You play back the keystrokes in exactly the same order in which they were recorded by pressing Ctrl Shift P. You can play back the same key macro as many times as you like. Obviously, since key macros contain only exact keystrokes and key combinations, your macros will be effective only when they contain sequences that can be repeated successfully.

Block Indent and Unindent

Indentation plays an important role in making your code easier to read and maintain. For example, most Delphi developers indent the **then** clause of an **if** statement:

```
if BooleanCondition then
  DoThisandThat;
```

Furthermore, when two or more statements are conditional, such as when a **begin..end** block is used to group a series of statements to be executed conditionally, the statements within the **begin..end** are traditionally indented to the same degree, providing additional visual cues of their grouping.

Block indenting and unindenting refers to the process of changing the indentation of two or more lines of code simultaneously. This feature is particularly valuable when you're introducing a new control structure to an existing section of code. For example, you might have originally written your code to execute a sequence of statements, but now you want to make that sequence conditional by introducing an **if** statement. Making the sequence of statements conditional generally means you will also want to further indent these statements (to maintain a consistent indentation). While this can be accomplished by indenting one line at a time (or by recording a key macro that indents one line and then moves to the next line, and then plays this macro back repeatedly), block indentation permits you to indent all of the conditional statements in a single step.

To perform block indentation, first highlight the lines that you want to indent. In the Default keystroke mapping, this can be accomplished by holding down Shift, while using the navigation (arrow) keys to highlight the rows to be indented. Dragging the mouse to highlight the desired rows also accomplishes this effect. Once the lines that you want to indent are highlighted, press Ctrl Shift I to indent. To block unindent, highlight the rows that you want to unindent, and press Ctrl Shift U. The number of columns that the selected text is moved depends on the **Block indent** setting on the Editor page of the Environment Options dialog box. I prefer to set **Block indent** to 1, to get the greatest control.

Using Bookmarks

A bookmark is a setting that permits you to return to a particular line of code in a unit (or any other file opened in the editor, for that matter). Bookmarks are ideal when you're working on a large unit, and you want to quickly move between two or more rows within that unit.

You can set up to 10 bookmarks in each file opened in the Code Editor. These bookmarks are labeled 0 through 9, and each bookmark can appear only once in a given unit. To set a bookmark, hold down Ctrl Shift and press a single-digit key, i.e. 0 to 9. The bookmark is identified in the left-hand gutter of the editor by a glyph that displays the specified digit. For example, the code shown in Figure 3 contains two bookmarks, one labeled 0 and the other labeled 5.

To move to a particular bookmark, press Ctrl plus the digit identifying the bookmark, e.g. Ctrl 5. To remove a bookmark, move to the line for that bookmark and press Ctrl Shift plus the digit identifying the bookmark. Alternatively, if you attempt to place the same bookmark in a new location within the same unit, the existing bookmark is removed before the new bookmark is placed.

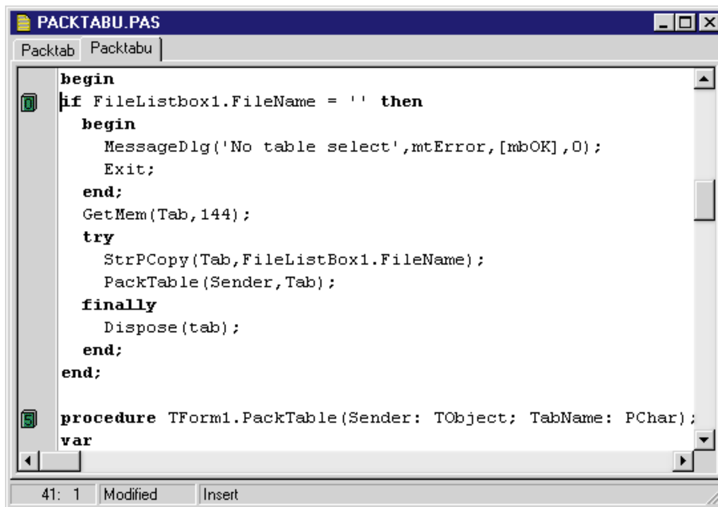


Figure 3: Bookmarks appear in the left-hand gutter of the editor. In this example, the bookmarks are in close proximity. However, in most cases bookmarks are in distant parts of a unit.

Bookmarks are temporary; they are lost when you close the file in which they are set.

Incremental Search

Delphi provides a wide variety of search options. Of these, the least used, though arguably the most interesting, is the incremental search. What makes the incremental search so nice is that it's quite easy to use.

When you initiate an incremental search, Delphi monitors your keystrokes, and positions your cursor at the first character sequence within the unit that matches the characters you have entered. You initiate an incremental search by pressing Ctrl E, or by selecting **Search | Incremental Search** from the menu. You conclude an incremental search by pressing Esc, or by navigating from the located string.

While you are performing an incremental search, Delphi displays **Searching for:** in the status bar. As you type the characters of your search string, these too are displayed in the status bar. If Delphi finds a match to your entered string, but not what you're looking for, you can press F3 to ask Delphi to search for the next occurrence of the entered string.

The use of an incremental search is easier to demonstrate than to describe. Start by creating a new project in Delphi. Display Unit1 in the editor, and press Ctrl E. Assuming that you want to find the string "Form1," begin typing `Form1`. When you type F, Delphi highlights the "f" in interface. Next, press O. Delphi now moves the highlighting to the first instance of "fo," which it finds in the string "Forms" that appears in the interface uses clause. Now enter the characters R M 1. At this point, the highlighting will be on the word "TForm1" in the class declaration for the form, since this is the first instance of the string "Form1." Now press F3; the highlighting will advance to the variable declaration "Form1," as shown in Figure 4. If you were to press F3 once more, highlighting will move to "TForm1" in the type portion of the variable declaration. Pressing F3 one last time will display a dialog box indicating that it finds no more instances of the string "Form1" within that unit.

An incremental search always begins from the current position of the cursor, and performs a forward search.

Find Matching Delimiters

Like most programming languages, Object Pascal makes extensive use of delimiters to organize code. For example, the parameters of a function call appear within parentheses, the contents of a multi-line comment appear within a pair of matching braces, and the members of a set are enclosed in a matching pair of brackets.

The Find matching delimiter feature of the Code Editor permits you to quickly locate the parenthesis, brace, or bracket that corresponds to one you have chosen. For example, if you place your cursor to the right of an open parenthesis, the Find matching delimiter command will move your cursor to the right of the corresponding close parenthesis. This feature is particularly valuable when working with nested matching delimiters, such as function calls that serve as parameters to other function calls. Using Find matching delimiter can help you verify that your parentheses are placed correctly.

To find the matching close delimiter, place your cursor to the right of the open delimiter and press Ctrl Q]. Likewise, to find the matching open delimiter, place your cursor to the right of the close delimiter and press Ctrl Q [.

Column Operations

Typically, block operations are based on rows. For example, earlier in this article you learned how to indent and unin-

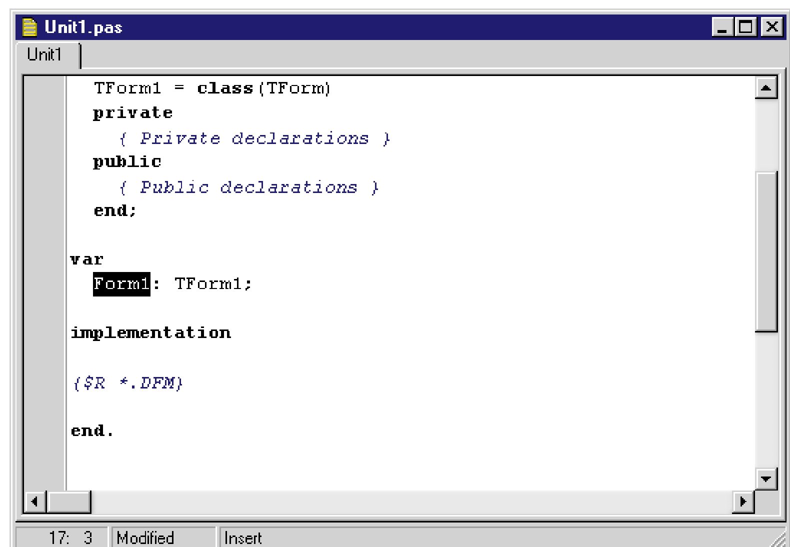


Figure 4: From Unit1 in a new project, you can quickly locate the *Form1* variable declaration by pressing Ctrl E, typing `Form1`, and then pressing F3.

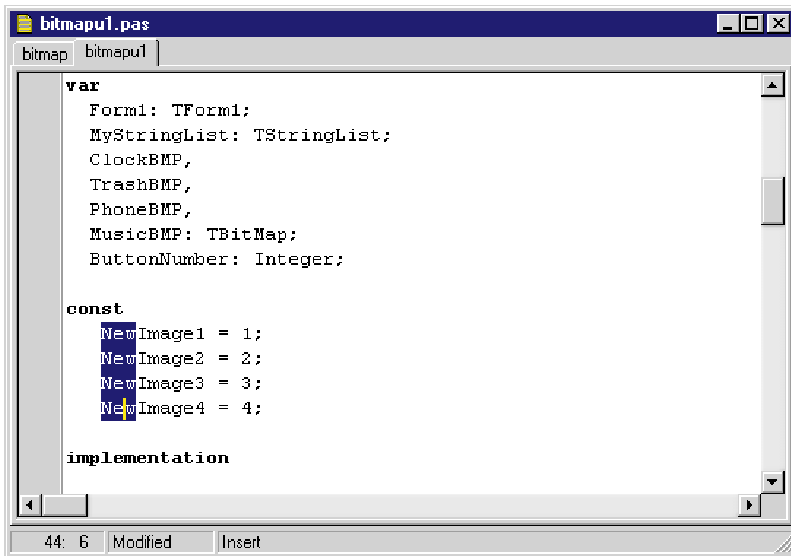


Figure 5: By selecting a column block, it's possible to delete the string "New" from multiple lines, quickly converting these constant declarations.

dent one or more rows with a single key combination. The highlighted rows in that example constituted a block. (It should be noted that there are a great many block operations, including deletion, converting the case of its characters to upper case or lower case, and so forth.)

In addition to permitting you to create blocks based on rows, Delphi's editor also provides for column-based blocks. These blocks, which are always rectangular in shape, permit you to perform block operations on one or more columns. For example, imagine that you have two or more lines where you want to delete only the character appearing in the third, fourth, and fifth rows. You can easily do this by creating a column block that includes only the third through fifth columns of the two or more rows. Once defined, these characters can be deleted by pressing Delete (or by cutting using Ctrl X, or whatever delete key combination is supported by your keyboard mapping).

You create column-based blocks by first marking the row and column in which you want the block to begin by pressing Ctrl O C. You then hold down Shift while you navigate to the row and column where you want the block to end. The column block appears as a highlighted rectangle. Once the block is highlighted, you can perform any block operation, such as copying the block to the Clipboard, or deleting it. Figure 5 depicts a code segment where a column block is being used to highlight "New" on multiple lines, and then delete them with a single delete operation.

Conclusion

The Delphi Code Editor provides a wealth of features that support your code-writing activities. This article has discussed only a few of these features. By taking a few minutes to read Delphi's online Help, you are sure to learn additional tricks that can improve your productivity. ∆

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including *Delphi in Depth* [Osborne McGraw-Hill, 1996]. He is also a Contributing Editor of *Delphi Informant*, and was a member of the Delphi Advisory Board for the 1997 Borland Developers Conference. For information concerning Jensen Data Systems' Delphi consulting and training services, visit the Jensen Data Systems Web site at http://idt.net/~jdsi. You can also reach Jensen Data Systems at (281) 359-3311, or via e-mail at cjensen@compuserve.com.

Delphi 3 / ODBMS / POET

By Chu Moy

Developing Object Databases

A RADical Approach to Business Applications

Recent trends in application development are highlighting the limitations of relational database technology. Business applications now routinely tap into complex data containing several levels of relationships and a multitude of data types, such as multimedia data. At the same time, developers are modeling applications using an object-based paradigm. Relational databases are not optimized for either of these situations. RDBMSes suffer significant performance degradation with complex data, and developers must spend considerable time writing code to map object data to relational tables. Object adapters to relational databases must still contend with the underlying relational foundation.

The emerging technology of pure object databases promises dramatic improvements in speed and development time. Some benchmarks have clocked ODBMSes performing up to 2,000 times faster than their relational counterparts. Once the exclusive province of C++ and Smalltalk programmers, ODBMSes have lacked connectivity options for the RAD market. ODBMS vendors have addressed these issues by releasing GUI-based development tools and ActiveX and ODBC interfaces to their products.

Delphi, with its fully object-oriented architecture and support for ActiveX type libraries (in Delphi 3), is an excellent language for RAD object database development. This article examines methods and tools for Delphi developers to design and access object databases. The code examples use POET ODBMS (http://www.poet.com). However, other ODBMSes, such as Object Design's ObjectStore (http://www.odi.com), and Computer Associates' Jasmine (http://www.cai.com), have similar connectivity options.

The Object-Database Schema

Whereas relational databases store data in fixed tabular formats of rows and columns, the structure of object databases is organized around the complexity of the data itself. An object schema can reflect real-world processes through inheritance, polymorphism, and encapsulation. Objects have unique characteristics — such as identity and behavior — that have no equivalents in traditional relational theory. Object-relational adapters add support for multimedia data types to RDBMSes, and may also provide an object-like programming interface; but the database engine still operates with tables. Object databases extend the OOP paradigm from the programming language to the database engine.

In an ODBMS, data and procedures are grouped together as object properties and methods. It's possible to view object databases in relational terms (see Figure 1), but only at a superficial level. Broadly speaking, the basic elements of an RDBMS have corresponding elements in an ODBMS. Thus, a table named Employee may contain records with columns such as Name, Address, Empl_id, and Dept_id. An *Employee* object is an instance of an *Employee* class, and may have attributes such as *Name*, *Address*, and *Empl_id*.

Object Term	Relational Term	Comment
Database	Database	*Object databases* store objects and their relationships, with support for encapsulation, inheritance, and polymorphism. Relational databases store record data in two-dimensional tables. Relationships between tables are constructed with queries.
Class/Extent	Table	*Classes* are templates for objects. A class can have definitions for attributes and methods. *Tables* use a row-and-column structure to hold data. An *extent* is the set of all objects of a particular class.
Object	Record	An *object* is an instance of a class. It encapsulates data and behavior. A *record* is a row of column data.
Object ID (OID)	Row ID (RID)	OIDs and RIDs are system-generated values that uniquely identify an object in an ODBMS, or a row in an RDBMS. However, traditional RDBMS theory does not require RIDs; some RDBMSes do not create them.
Attribute / Relationship	Column	Object-data *attributes* can map to table columns. Object relationships can be mapped as foreign-key values to columns.
Method	Stored Procedure	*Methods* are associated with objects, and have the capabilities of the programming language. Stored procedures are precompiled SQL code.
Event Management	Triggers	An ODBMS will usually have an object-management mechanism that can trigger the execution of methods based on the occurrence of object events. RDBMSes use *triggers* to execute SQL procedures on the occurrence of data events.
Index	Index	Object *indices* are conceptually similar to relational indices. The ODMG ODL includes syntax to specify multiple keys for a class.

Figure 1: Object-relational structure mappings.

Object schemas generally support two types of relationships: inheritance and associations. Inheritance simplifies schema design by reusing the design of the parent class. An association is represented as a pointer to another object. Thus, instead of a *Dept_id* attribute, an *Employee* object would have a pointer to a *Department* object. In a relational database, the relationship between the Employee and Department tables would be constructed in a query that uses *Dept_id* as a foreign key. An *Employee* object, however, can find its associated *Department* object by going to the referenced memory address. The improvement in response time compared to an RDBMS is especially dramatic with complex data.

There are two ways to create an object database:
- Specify the persistent classes in a schema script and compile the script into a database binary.
- Use a GUI-based schema design tool.

```
interface SalesRep: Employee
(     extent SalesReps)
{
      void RequestSalaryIncrease();
      relationship SalesManager Is_managed_by
        inverse SalesManager::Manages;
};

interface SalesManager: Employee
(     extent SalesManagers)
{
      void       DenyRaiseRequest();
      void       TerminateSalesRep();
      relationship Set<SalesRep> Manages
        inverse SalesRep::Is_managed_by;
};
```

Figure 2: Specifying inheritance and associations in an object schema with ODL.

The first may sometimes be more flexible. Those databases in compliance with the Object Database Management Group (ODMG) standard may support its Object Definition Language (ODL) for defining schemas. Other products may require the schema be defined according to programming language syntax such as C++.

An ODL class declaration has two basic sections: the heading and the body. The heading specifies the class inheritance, the name of the extent (the set of all objects of that class), and keys for indexing. The body lists the class attributes and methods. A script for defining an *Employee* class using ODL might appear as follows:

```
interface Employee
  (     extent employees
        key (name, empl_id)):persistent
  {
        attribute String Name;
        attribute String Address;
        attribute Short Empl_id;
  };
```

The ODL can specify two kinds of schema relationships: inheritance and associations called traversal paths. Inheritance simplifies schema design. A named traversal path is a designated relationship between classes. If the *SalesManager* and *SalesRep* classes descend from the *Employee* class and are also associated (e.g. SalesMangers supervise SalesReps), the ODL allows the relationship to be defined from either, or both, points of view (Figure 2). In the database, relationships are physically represented as pointers to associated objects. In a 1:1 relationship, the source object will have a single reference to its associated object. In a 1:*n* relationship, the source object has a set of references to its associated objects.

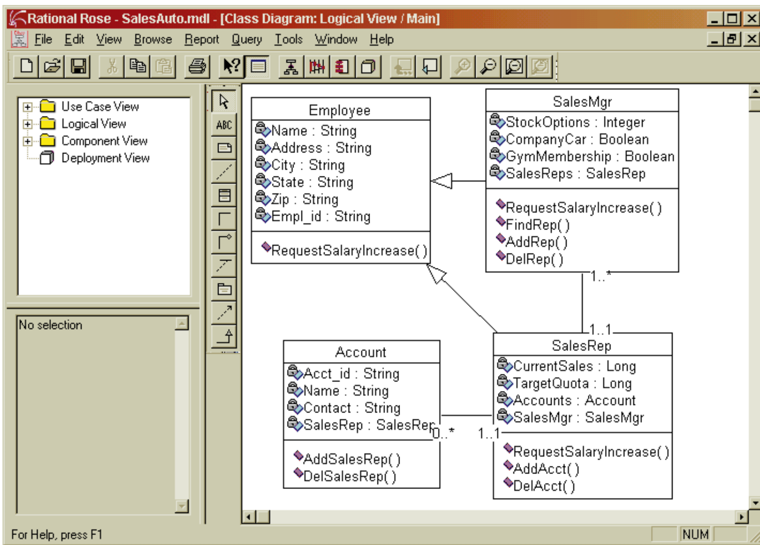


Figure 3: With the PtRose add-on, Rational Rose Modeler 4.0 can generate POET databases from class diagrams.

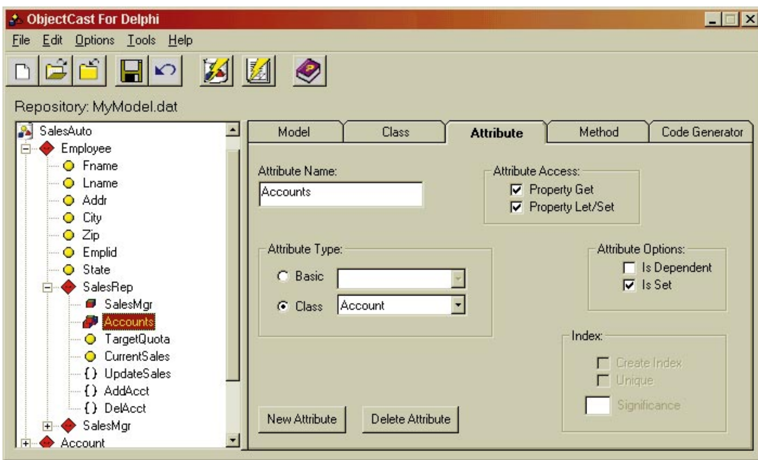


Figure 4: ObjectCast for Delphi, from CustomLink Software, is a RAD/ODBMS workbench that generates object databases and application frameworks.

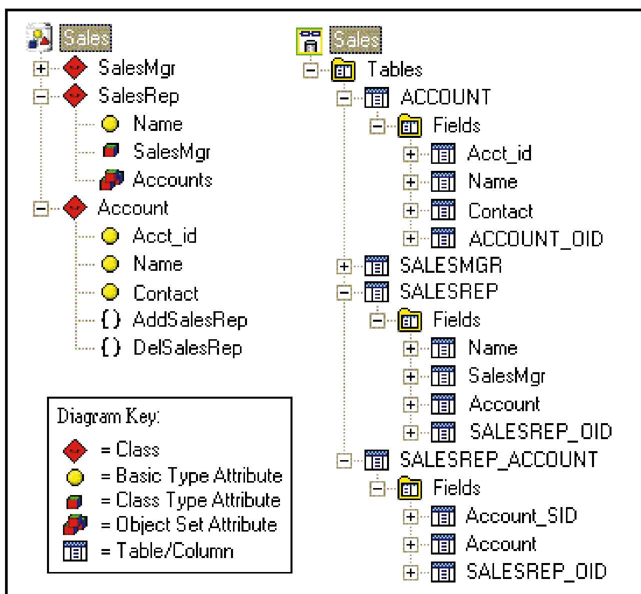


Figure 5: A sales ODBMS schema and its ODBC view.

Large schema scripts in ODL or C++ are difficult to maintain. GUI schema tools, such as PtRose from POET Software (see Figure 3) and Blueprint from Object Design, avoid the tedium and confusion of scripting schemas by generating databases from object models. Third-party tools, such as ObjectCast from CustomLink Software (see Figure 4), include code generation for RAD languages like Delphi.

Connecting Delphi Applications to Object Databases

An ODBMS vendor may supply a Delphi "tight binding" that integrates closely with the Delphi IDE, but ActiveX and ODBC appear to be the most popular interfaces for RAD development. ActiveX interfaces tend to offer better performance and greater functionality than ODBC, and applications can access and manipulate database objects through ActiveX. ODBC is a relational technology that's been adapted to map object schemas to virtual tables that can be queried with SQL. Usually, the developer must write code to map ODBC record sets to application objects.

However, the developer may choose ODBC because the application doesn't require high performance or access to all ODBMS features. Usually, these situations involve connecting an RDBMS application to ODBMS data, or the use of relational reporting tools on an object database. There are more efficient options for integrating ODBMS and RDBMS data, but ODBC is convenient and familiar. A discussion of ODBC is also a good way to introduce the topic of query languages for object databases.

Open Database Connectivity. ODBC drivers for object databases are just one option in a selection of object-relational mapping technologies, but ODBC is certainly Delphi-compatible, whereas the others may not be. As is true of ODBC drivers generally, ODBC drivers for object databases are not all the same; they may translate ODBMS structures differently. Some ODBC drivers use the SQL2 standard; others may have added object extensions to SQL, like those in the forthcoming SQL3 standard or the ODMG's OQL (Object Query Language). Although these are different standards, they have similarities because OQL is based on SQL and SQL3 borrows from OQL.

ODBC views object databases as a series of tables, and returns record sets which must be mapped to application objects. The relationships between objects are broken. Derived tables substitute a foreign-key column (OID or object IDs) for each relationship attribute. A SQL query can reconstruct the relationships with joins on the relationship keys.

The object schema of the simple sales database shown in Figure 5 differs slightly from its ODBC view. Although the

Query2 : Select Query

SALESREP.Na	Account_SID	ACCOUNT.Nan
SalesRep1	1	Account1
SalesRep1	0	Account2
SalesRep2	2	Account3
SalesRep2	1	Account4
SalesRep2	0	Account5

Record: 1 of 5

Figure 6: The Account_SID column in the result set of the ODBC query.

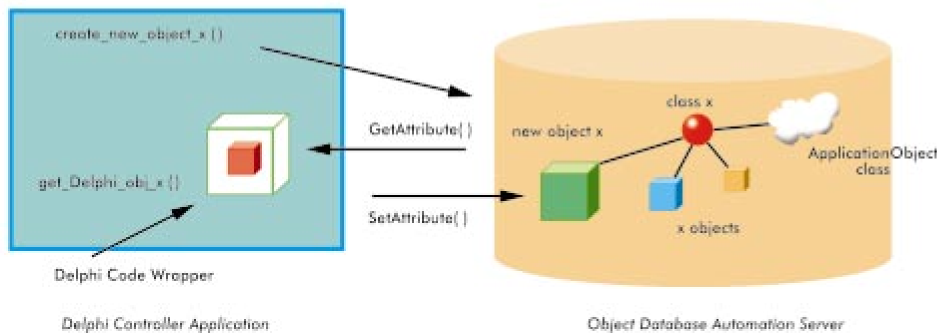


Figure 7: Accessing objects in an ODBMS, via ActiveX Automation.

object schema has only three classes (*SalesMgr*, *SalesRep*, and *Account*), the ODBC view shows four tables. One table, SalesRep_Account, represents the one-to-many (1:*n*) relationship between SalesReps and Accounts. The *SalesRep* object has an object set attribute named *Accounts*, which is the set of all assigned accounts. The Account_SID column preserves information about the order of the account objects in the set (see Figure 6).

From the ODBC table view, the following SQL2 query retrieves the name of the sales rep who has an account contact named "Henry Ford":

```
SELECT R.*
  FROM SalesRep R, SalesRep_Account RA, Account A
 WHERE R.SalesRep_OID = RA.SalesRep_OID
   AND RA.Account = A.ACCOUNT_OID
   AND A.Contact = "Henry Ford"
```

OQL and SQL3 queries navigate object trees by specifying the traversal path, and have more compact and efficient query syntax. An OQL query operates on object sets. Because the *SalesRep* class has two sets (a *SalesReps* extent and an *Accounts* set), an OQL query that traverses both sets must specify them. The following OQL query is functionally identical to the previous SQL query:

```
SELECT r
  FROM r IN SalesReps, a IN r.Accounts
 WHERE a.Contact = "Henry Ford"
```

Other object-relational mapping products create object views on relational schemas. Not all are Delphi-compatible. These high-performance interfaces, such as DBConnect from Object Design and SQL Object Factory from POET Software, automatically map relational data to application objects. They are designed to integrate ODBMS and RDBMS schemas into a single object model. Some relational adapters include an ODBC driver as part of the package.

ActiveX interfaces. A native Delphi binding to an object database would have the best performance and most seamless integration between application objects and database objects. ActiveX Automation interfaces are more commonly available. Since Delphi 3 supports type libraries, applications can use vtable binding to extract maximum performance from ActiveX interfaces. Some vendors also offer ActiveX controls for drag-and-drop visual programming.

Unlike a native binding, objects retrieved from an automation database are instantiated on the server side. The controller application receives COM pointers to server objects, and doesn't operate directly on the objects. If the application is only concerned with manipulating object attributes, the COM barrier isn't an issue. If the controller application must invoke object methods, the COM barrier is an issue because method code runs on the client, not on the sever. The solution is to encapsulate the COM object inside Delphi wrappers to simulate native Delphi objects, which are subject to inheritance and polymorphism (see Figure 7). The following class declaration defines an *Employee* class that contains a pointer to a database object:

```
PtEmployee = class
  // For early binding, use IPOETApplicationObject.
  PtObj: OleVariant;
  procedure SetName(value: string);
  function GetName: string;
  procedure SetAddress(value: string);
  function GetAddress: string;
  procedure RequestSalaryIncrease; virtual;
end;
```

The *Get/SetName* and *Get/SetAddress* methods use the COM object *PtObj* to access an *Employee* object in the database, while the *RequestSalaryIncrease* method contributes to the object behavior. When the application instantiates a *PtEmployee* object, it initializes the object by assigning the COM object to *PtObj*. *PtObj* already has its own accessor methods, but defining separate *Get/Set* meth-

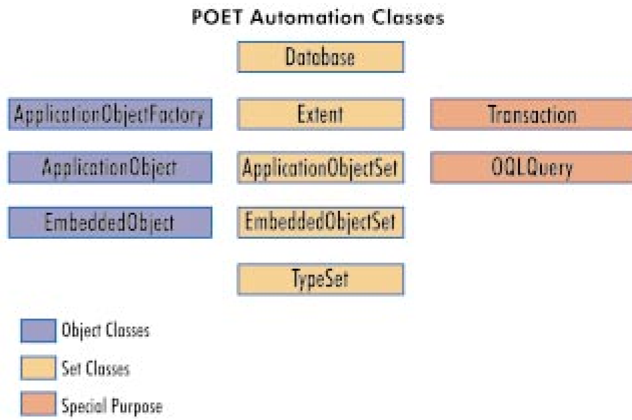


Figure 8: A simplified view of the POET Automation class hierarchy.

ods helps to further conceal the true nature of the object's COM identity.

An application could only require one instance of a *PtEmployee* object, because retrieving another object from the database is only a matter of reassigning the COM object. Since *PtEmployee* is a Delphi class, other Delphi classes can inherit from it and implement polymorphism. For example, a *PtSalesRep* class can inherit from *PtEmployee* and modify *PtEmployee*'s method implementation:

```
PtSalesRep = class(PtEmployee)
  function GetCurrentSales: Integer;
  procedure MakeSale (value: Integer);
  procedure RequestSalaryIncrease; override;
end;
```

Programming with an ActiveX Interface

The ODMG specification indicates that objects can be stored in a database as named objects and/or extents. A named object consists of a root object (which is assigned a name) and any associated objects. Once the root object is created, any objects referenced by the root are automatically stored in the database. To retrieve an object, an application first retrieves the root, then navigates the object tree to reach the desired object. However, large numbers of named objects can make programming difficult, and slow down the database. The alternative is to use extents.

Some ODBMSes automatically create and maintain extents each time an object is stored or removed from the database. For those that support named objects, extents can be simulated by creating named objects that are collections or sets, each containing objects from a single class. The extent programming model organizes objects to simplify access.

The POET Automation class hierarchy (see Figure 8) illustrates the basic classes of an ODBMS that uses extents. The *ApplicationObject* class endows each object with database functionality, such as the ability to store and delete itself. Set constructs such

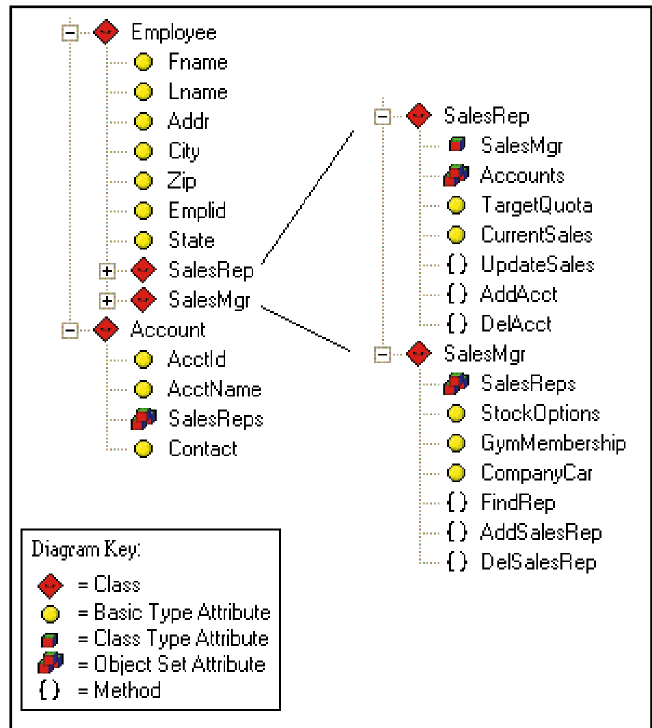


Figure 9: The sales-database schema for the sample application.

as *Extent* and *ApplicationObjectSet* represent groups of objects. Object sets are often used as object attributes for creating one-to-many relationships between classes. The *ApplicationObjectFactory* class is responsible for generating new database objects. ODBMSes that support named objects will often have even simpler class hierarchies.

The object schema of the sample application (see Figure 9) shows how extents and object sets work together. In this scenario, each SalesRep is assigned one SalesMgr, and is responsible for several accounts. Each account can be owned by several SalesReps. Each SalesMgr supervises several SalesReps. The *SalesRep* class has a 1:1 relationship with the *SalesMgr* class, and a 1:n relationship with the *Account* class. The *SalesMgr* class has a 1:n relationship with the *SalesRep* class. The relationship between SalesReps and Accounts is n:n, and is implemented as two 1:n relationships.

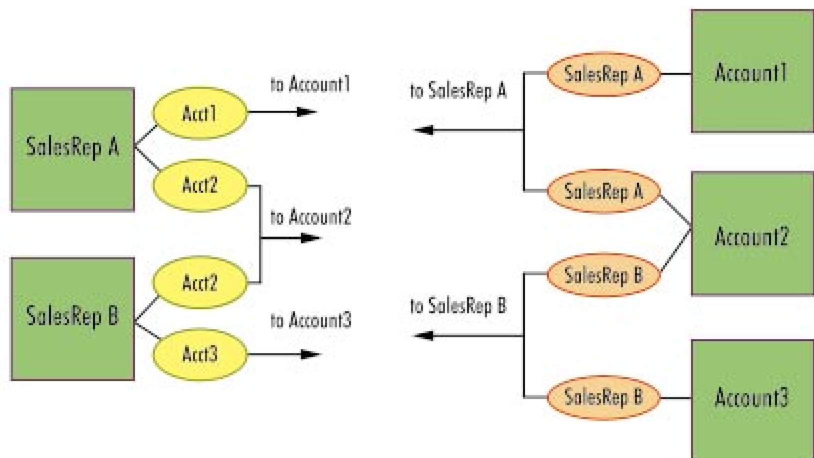


Figure 10: The many-to-many relationship between the *SalesRep* and *Account* classes in the sample database is implemented with object-set attributes.

Because the *SalesRep* and *Account* classes are in a *n*:*n* relationship, if a SalesRep is taken off an Account, the Account's SalesRep list must be updated as well (see Figure 10). Once the application has a pointer to the selected SalesRep, it can reference the selected Account, then locate the pointer to the same SalesRep inside the *Account* object's SalesRep set:

```
SalesRep.Account.SalesRep
```

The Object Pascal code to retrieve the second SalesRep pointer would take this form:

```
// Get SalesRep from SalesReps extent.
SalesRep := SalesReps.Get;
// Get SalesRep's Account set.
RepAccounts := SalesRep.GetAttribute("Accounts");

...locate desired Account pointer in RepAccount set...

// Delete current RepAccount pointer from set.
RepAccounts.Delete;
// Get Account's set of assigned SalesReps.
AcctSalesReps := RepAccount.GetAttribute("SalesReps");

...locate desired AcctSalesRep pointer in
     the AcctSalesReps set...

// Delete current AcctSalesRep pointer from set.
AcctSalesReps.Delete;
```

Of course, an OQL query might be easier. The sample application demonstrates a simple, embedded OQL query in the SalesMgr form. A query object (derived from the *OQLQuery* class) performs the query and returns a result set of objects. The principles of ODBMS sets are relevant to queries as well. An OQL query must specify the domain of extents and object sets over which the query will operate.

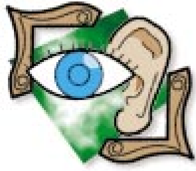
Conclusion

Object databases are a natural fit for Delphi's object-based development paradigm. This article has examined only a few object-database features available to Delphi, but has emphasized RAD connectivity techniques. From the simple beginnings discussed here, developers can build complex ODBMS applications that enjoy the same advantages of speed and flexibility as C++ or Smalltalk-based systems, yet can be designed and coded in a fraction of the time. ∆

The author thanks Prasad Jeevaniji, Kris Tanner, and Eric Vigna for helping prepare this article.

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\MAR\DI9803CM.

Chu Moy, formerly with POET Software, is a consultant with Thomson Technology Consulting Services in San Francisco. He has a Bachelor's degree in electrical engineering from Yale University. He is also a Microsoft Certified Solution Developer and a Master Certified Novell Engineer. He has had several years of programming experience with RAD tools, and has been working with Delphi for the last two years.



Delphi 1, 2, 3

By Christopher D. Coppola

Multimedia Buttons

Creating Special Buttons for Special Interfaces

One of the fundamental elements of GUIs is the button; the button metaphor is widely used and well understood as a component of interface design. In many applications, the interface calls for the functionality of a button, but demands a more integrated look than the standard button components are capable of producing (see Figure 1).

In this article, we'll develop a *TMMButton* component (see Listing Two, beginning on page 33) that has all the functionality of a standard button, but integrates seamlessly with a creative, high-resolution interface. The *TMMButton* component will also provide more feedback to the user than a standard button.

Standards and Intuitiveness

When we design a user interface, our primary goals should be to make the interface functional, make the user comfortable, and finally, to make the interface aesthetically pleasing. One of the fundamental challenges of creating an interface with these characteristics is to use the available interface components — buttons, text boxes, menus, etc. — in a way the user can intuitively understand.

Figure 1: Buttons seamlessly integrated with the background.

One key to designing an intuitive interface is to adhere to functionality standards. Interface metaphors, such as the button, have been employed for so long now that users intuitively know how to operate a button. Therefore, when developing a button component, carefully consider every aspect of how a button works.

The widely accepted standards for the functional operation of a button assert that the *OnClick* event is only fired if the mouse button is released when the mouse pointer is over the button. In other words, if I press a button and (with the mouse button still depressed) move the pointer away from the component, the component should visually return to its "normal" state, and shouldn't fire an event if I release the mouse button. A good example of button functionality is the X button that closes a window in Windows 95/NT. Figure 2 illustrates, in terms of mouse events, how a button component should function.

TMMButton vs. other Button Components

We're going to develop the *TMMButton* component so that it responds to mouse interaction similarly to the Delphi Button, SpeedButton, or BitBtn components.

Mouse Events	Standard Graphic Response
Left Button Down	Show the button in its "pressed" state.
Left Button Up	Show the button in its "highlighted" state or its "pressed" state based on these conditions: If the button is in an inactive "pressed" state, show the button in its "pressed" state; if the button is not in an inactive "pressed" state, show the button in its "highlighted" state.
Mouse Enter	If the pointer is over the button, show the button in its "highlighted" state, then fire the *OnClick* event. If the cursor isn't over the button, show the button in its "normal" state.
Mouse Leave	Show the button in its "normal" state. If the mouse button is pressed, set a flag to indicate it's in an inactive "pressed" state.

Figure 2: Button functionality in terms of mouse events.

Figure 3: The standard buttons just don't look right for this interface.

However, the *TMMButton* component will have three important differences. First, it will sport improved user feedback by displaying a "highlighted" state when the mouse is over the component. In many multimedia and game interfaces, this additional feedback is essential; buttons are integrated with the background so well that the highlight is necessary to identify them. Microsoft's new "Explorer style" button has popularized a similar feedback mechanism for business and productivity applications.

Another significant difference is how *TMMButton* applies graphics. All buttons in Windows use graphics, but most are simply rectangular regions with light and dark borders that simulate a three-dimensional look. Some button components feature bitmaps, but hardly integrate with a custom interface (see Figure 3). The *TMMButton* component, on the other hand, displays an entirely new bitmap for each of its four functional states. This gives the developer more flexibility in designing a high-resolution interface where the buttons are nicely integrated with the background.

Finally, *TMMButton* will support sound. The value of audio feedback is often overlooked. As we'll soon see, sound is simple to implement, and contributes greatly to the user experience.

Creating the Component

Beginning with the Component Expert, create the *TMMButton* class as an ancestor of *TGraphicControl*. The Component Expert graciously delivers a template from which we'll build the *TMMButton* component:

```
type
  TMMButton = class(TGraphicControl)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;
```

The first thing I do when defining a new class or subclass is think about the data the class must represent. The *TMMButton* class will clearly need to store bitmaps that represent the button in each of its visible states — "normal," "disabled," "highlighted," and "pushed." For these data elements we'll use Delphi's *TBitmap* object. In addition to storing the bitmaps, we'll need a couple of Boolean variables. These will assist the *TMMButton* component in determining which bitmap should be displayed when the component is painted. We'll also need four variables to deal with audio feedback. Two string variables will store the names of the mouse-over and button-push sounds. The other two variables tell the button how and where the sounds are stored. For greater flexibility, *TMMButton* will allow the developer to use sound files, sound resources embedded in the .EXE, or sound resources embedded in a DLL.

We'll add the data members we've just described to the **private** section of the class declaration:

```
private
  FBmpNormal, FBmpHiLight,
  FBmpPushed, FBmpDisabled: TBitmap;
  FSndOver, FSndPush: string;
  FSoundType: TSoundType;
  FDllInstance: Integer;
  FDown, FOver: Boolean;
```

After we've defined the class' data members, we need to consider the component's functionality. I described earlier how the component should respond to mouse interaction; now it's time to translate that description into Delphi code. We'll need the component to be aware of the following mouse actions:

- button press
- button release
- pointer entering the component boundaries
- pointer leaving the component boundaries

To react to the mouse button being pressed, the component implements a *MouseDown* method. We'll simply

override the *MouseDown* method already implemented in *TGraphicControl*, implement the specific functionality required in the *TMMButton* component, then call the inherited *MouseDown* method. The *OnMouseDown* event is fired only when the mouse button is pressed and the pointer is over the component. When the component receives a WM_MOUSEDOWN message, the *MouseDown* method sets *FDown* to True and calls the *Paint* method.

We'll implement the *MouseUp* method to detect when the mouse button has been released. *MouseUp* (just as *MouseDown*) keeps the value of *FDown* accurate. Each time *MouseUp* is called, *FDown* is set to False. *MouseUp* performs another vital function: It determines if an *OnClick* event is triggered. Within *MouseUp*, we perform a simple check to determine if the pointer is over the component. If it is, we paint the component and trigger its *OnClick* event. If the pointer isn't over the component, nothing more needs to be done.

To detect when the pointer enters and leaves the component's boundaries, we'll trap the CM_MOUSEENTER and CM_MOUSELEAVE messages. Do this by adding message-handling methods:

```
procedure CMMouseEnter(var Message: TMessage);
  message CM_MOUSEENTER;
```

The **message** directive tells Delphi that the declared method should be called whenever the component receives the message identified by the integer identifier following the **message** directive. In this case, we've told the component that *CMMouseEnter* should be fired whenever the component receives a CM_MOUSEENTER message, i.e. whenever the pointer enters the boundaries of the component.

When *CMMouseEnter* is executed, *FOver* is set to True. Conversely, when *CMMouseLeave* is executed, *FOver* is set to False. In addition, *CMMouseEnter* and *CMMouseLeave* will call the component's *Paint* method if the component has a "highlighted" state bitmap assigned, or *FDown* is set to True.

Painting the Component with the Correct Bitmap

We've seen several places where the component calls the *Paint* method to display the bitmap that correctly depicts the current state. Overriding the component's *OnPaint* event is advantageous because we can call the method from other component methods to explicitly cause the component to paint, but it has another distinct advantage. The component's *OnPaint* event is triggered whenever the component needs to be repainted because of other screen activity. Because we've overridden the event and placed all the logic for determining the correct bitmap within the *Paint* method, no additional logic or code is necessary to repaint the component when another screen event makes it necessary to repaint.

The component handles two conditions in the *Paint* method. Because this is a visual component, we must consider how the component will function at design time *and* run time. At design time the logic is simple: If the "normal" state bitmap has been assigned, then the component paints using the "normal" state bitmap; otherwise the component paints a line around the component to make it visible. Notice from the following code that I paint the outline using gray and the *pmXor* pen mode. This ensures that whatever the background color of the button's container, the button will be visible. Many components simply use black to paint the outline. The result is a component that's difficult to locate on a black form at design time.

```
with Canvas.Pen do begin
  Style := psSolid;
  Color := clGray;
  Mode := pmXor;
end;

Canvas.Brush.Style := bsClear;
Canvas.Rectangle(0, 0, Width, Height);
```

At run time, the *Paint* method first determines which bitmap should be displayed based on the evaluation of the enabled state of the button, as well as the *FDown* and *FOver* variables. If the button is not enabled, the "disabled" state bitmap is used. When the button is enabled, *FDown* and *FOver* are considered. If *FOver*, for example, is False, then the correct bitmap to display would be the "normal" state bitmap. After the appropriate bitmap has been determined, a final check is performed to ensure the correct bitmap has been assigned. It is, after all, conceivable that this component would be used without a "pushed" or "highlighted" state bitmap. If the appropriate bitmap is not assigned, then the component will simply use the "normal" state bitmap to repaint itself. The code that paints the component is trivial; I've used the *CopyRect* method of the component's *Canvas* to perform the paint.

Declaring Properties

Now that we've seen how the *TMMButton* component works, it's time to give the component the developer interface necessary to assign bitmaps and other properties unique to each instance of the component. The *TMMButton* component has the published interface shown in Figure 4.

```
published
  { Published declarations }
  property PicNormal: TBitMap
    read FBmpNormal write setNormal;
  property PicHiLight: TBitMap
    read FBmpHiLight write setHiLight;
  property PicPushed: TBitMap
    read FBmpPushed write setPushed;
  property PicDisabled: TBitmap
    read FBmpDisabled write setDisabled;
  property SndOver: string read FSndOver write FSndOver;
  property SndPush: string read FSndPush write FSndPush;
  property SoundType: TSoundType
    read FSoundType write FSoundType;
  property DLLInst: Integer
    read FDLLInstance write FDLLInstance;
  property Height default 30;
  property Width default 30;
  property Enabled;
  property Visible;
  property OnMouseDown;
  property OnMouseMove;
  property OnMouseUp;
  property OnClick;
```

Figure 4: The published interface of *TMMButton*.

The four bitmap properties simply return the value of the associated private variable. When one of the properties is set, however, it's done by a method. Setting the *PicNormal* property, for instance, would cause the component to execute the *setNormal* method. The other three bitmap properties, *PicHiLight*, *PicPushed*, and *PicDisabled*, use the *setHiLight*, *setPushed*, and *setDisabled* methods, respectively. Each of the four methods uses *TBitmap*'s *Assign* method to store the bitmap. The *setNormal* method, because it's associated with the most rudimentary of the three properties, also sets the component's *Width* and *Height*. The component user can change this, but it's reasonable to assume that in most cases the component should be the same dimensions as the "normal" state bitmap.

In addition to the bitmap properties, the *TMMButton* component has standard *Height*, *Width*, *Enabled*, and *Visible* properties, and it exposes the standard *OnMouseDown*, *OnMouseMove*, *OnMouseUp*, and *OnClick* events.

Figure 5: The interface of the sample application.

Audio Capabilities

Sound is the finishing touch to our new component. Using the properties declared earlier, the API function *PlaySound*, and a private function called *doSound*, this is a simple task. First, we must decide where sound is appropriate. Sounds are typically used in response to the pointer moving over the button, as when the button is pushed. This means we need to add a call to *doSound* to the *OnMouseDown* and *CMMouseEnter* events. In the *OnMouseDown* event, we call *doSound*, sending *FSndPush* as the parameter. Recall that *FSndPush* is the private variable that stores the name of the "push" sound.

The *doSound* procedure is the key to *TMMButton*'s audio features. Using the *FSoundType*, *FDLLInstance*, and *whichSound* parameters, *doSound* uses the *PlaySound* API function to play a sound file, a sound from an application resource, or a sound from a DLL resource.

Installing the *TMMButton* Component

As with any other component, installation is accomplished by selecting **Install Component** from the **Component** menu:

- Delphi 2: From the Install Components dialog box, click the **Add** button and locate MMButton.pas.
- Delphi 3: From the Install Component dialog box, use the **Browse** button to select MMButton.pas as the unit. Select the package you would like MMButton to be a part of, then press **OK**.

When Delphi is finished recompiling the component library, you'll find the MMButton icon on the component toolbar of the Additional page. To install the component on a different page, modify the component's *Register* method. For example:

```
procedure Register;
begin
  RegisterComponents('MyTab', [TMMButton]);
end;
```

Implementation Tips

One of the more important aspects of using *TMMButton* to create compelling interfaces is implementing it correctly. First, I'll show you how to prepare the bitmaps to make sure the buttons integrate seamlessly with the background. I'll also demonstrate how to embed sounds in your .EXE so you don't have to distribute .WAV files with the application. Then when the bitmaps are ready, we'll create a sample project in Delphi to demonstrate how the *TMMButton* component should be implemented.

I used Adobe Photoshop to prepare the bitmaps. If you're using another graphics package, some of the specific implementation examples might be different, but the concepts can be applied using any capable graphics software. Start with a good background design. My example is from my company's most recent demonstration CD (see Figure 5).

Next, design the buttons. Each button should have four independent graphics that are drawn in position over the background. Photoshop's layer transparency works great for this. When we're finished creating the bitmaps for each of the button states, we should be able to create a rectangular selection around each of the buttons. The selection should be the smallest rectangular area that contains each of the button's states. Jot down the *x*,*y* coordinates of the upper-left pixel of the selection. This will save having to manually position the buttons in Delphi. Using the selection, copy and paste each of the button states into new documents (see Figure 6).

Once the bitmaps are ready, start a new Delphi project and add one *TImage* component for the background and a *TMMButton* component for each button in the project. Our sample project only has one button, so you'll only need one *TMMButton* for now.

Set the *TImage* component's *Picture* property to the background bitmap. This will act as the application's background. You might also want to add some code to the form's *OnCreate* event to size the application correctly:

```
ClientWidth := imgBack.Width;
ClientHeight := imgBack.Height;
```

Normal

Highlight

Pushed

Disabled

Figure 6: The four bitmaps of the sample button.

Next, set the *PicNormal*, *PicHiLight*, *PicPushed*, and *PicDisabled* properties of the *TMMButton* component. The component will automatically size to the dimensions of the *PicNormal* bitmap, but you'll have to position the components on the form so they line up with the original design. If you wrote down the top-left coordinate from earlier, enter the values in the component's *Top* and *Left* properties. Otherwise, you must visually align the component with the background.

Finishing Up

Now let's add some audio. It would be simple enough to set the *SndPush* and *SndOver* properties to the names of .WAV files. In most cases, however, it's more efficient to embed the sounds in the application. This is simple to accomplish using a resource script and the command-line resource compiler that ships with Delphi (Brcc32.exe). Resource scripts are simple. The following sample script generates two WAVE resources that can be accessed by the names WAV_PUSH and WAV_OVER:

```
WAV_PUSH WAVE "push.wav"
WAV_OVER WAVE "over.wav"
```

Delphi's resource compiler generates a resource file (.RES) that can be linked with a Delphi executable by adding a line, such as the following, to the application source:

```
{$R sounds.res}
```

All that's left to do is to tell the button component which resources to use, and where they're located. In this case, the default *stAppResource* tells the component to find the resource specified in *SndPush* in the application's executable.

That's all there is to it. Run the test application (see Listing Three on page 35) and you'll see that the component handles everything related to the button display and audio. All you must do is add the desired code to the component's *OnClick* event. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\MAR\DI9803CC.

Chris Coppola is one of the founding principals of Advanced Creative Technologies III, Inc., a multimedia and Internet development company. Chris has written about multimedia development in *Delphi 2 Multimedia Adventure Set* [Coriolis Group Books, 1996], *Director 5 Wizardry* [Coriolis Group Books, 1996], and *Visual Basic 5 Multimedia & Web Adventure Set* [Coriolis Group Books, 1997]. You can reach him at coppola@act-3.com or http://www.act-3.com.

Begin Listing Two — *TMMButton* Component

```
unit MMButton;

interface

uses
  Windows, Messages, Classes, Graphics, Controls;

type
  TSoundType = (stAppResource, stDLLResource, stFile);
  TMMButton = class(TGraphicControl)
  private
    { Private declarations }
    FBmpNormal, FBmpHiLight,
    FBmpPushed, FBmpDisabled: TBitmap;
    FSndOver, FSndPush: string;
    FSoundType: TSoundType;
    FDllInstance: Integer;
    FDown, FOver: Boolean;
    procedure setNormal(Value: TBitMap);
    procedure setHiLight(Value: TBitMap);
    procedure setPushed(Value: TBitMap);
    procedure setDisabled(Value: TBitmap);
    procedure doSound(whichSound: string);
  protected
    { Protected declarations }
    procedure MouseDown(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
    procedure MouseUp(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
    procedure CMMouseEnter(var Message: TMessage);
      message CM_MOUSEENTER;
    procedure CMMouseLeave(var Message: TMessage);
      message CM_MOUSELEAVE;
    procedure Paint; override;
    procedure Click; override;
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    { Published declarations }
    property PicNormal: TBitMap
      read FBmpNormal write setNormal;
    property PicHiLight: TBitMap
      read FBmpHiLight write setHiLight;
    property PicPushed: TBitMap
      read FBmpPushed write setPushed;
    property PicDisabled: TBitmap
      read FBmpDisabled write setDisabled;
    property SndOver: string read FSndOver write FSndOver;
    property SndPush: string read FSndPush write FSndPush;
    property SoundType: TSoundType
      read FSoundType write FSoundType;
```

```
    property DLLInst: Integer
      read FDLLInstance write FDLLInstance;
    property Height default 30;
    property Width default 30;
    property Enabled;
    property Visible;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnClick;
  end;  //TMMButton

procedure Register;

implementation

uses
  MMSystem;

procedure Register;
begin
  RegisterComponents('Additional', [TMMButton]);
end;

constructor TMMButton.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Width  := 30;
  Height  := 30;
  FBmpNormal  := TBitMap.Create;
  FBmpHiLight  := TBitmap.Create;
  FBmpPushed  := TBitmap.Create;
  FBmpDisabled  := TBitmap.Create;
  FSoundType  := stAppResource;
  FDLLInstance  := 0;
end;

destructor TMMButton.Destroy;
begin
  FBmpNormal.Free;
  FBmpHiLight.Free;
  FBmpPushed.Free;
  FBmpDisabled.Free;
  inherited Destroy;
end;

procedure TMMButton.setNormal(Value: TBitMap);
begin
  FBmpNormal.Assign(Value);
  if not FBmpNormal.Empty then
    begin
      // Set the height and width of the component based
      // on the size of the Normal bitmap.
      Height  := FBmpNormal.Height;
      Width  := FBmpNormal.Width;
    end;
end;

procedure TMMButton.setHiLight(Value: TBitMap);
begin
  FBmpHiLight.Assign(Value);
end;

procedure TMMButton.setPushed(Value: TBitMap);
begin
  FBmpPushed.Assign(Value);
end;

procedure TMMButton.setDisabled(Value: TBitMap);
begin
  FBmpDisabled.Assign(Value);
end;

procedure TMMButton.doSound(whichSound: string);
begin
  case FSoundType of
    stAppResource:
      PlaySound(PChar(whichSound), hInstance,
        SND_RESOURCE or SND_ASYNC or SND_NODEFAULT);
    stDLLResource:
      PlaySound(PChar(whichSound), FDLLInstance,
        SND_RESOURCE or SND_ASYNC or SND_NODEFAULT);
    stFile:
      PlaySound(PChar(whichSound), 0,
        SND_FILENAME or SND_ASYNC or SND_NODEFAULT);
  end;
end;

procedure TMMButton.MouseDown(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  FDown := True;
  doSound(FSndPush);
  if not FBmpPushed.Empty then
    Paint;
  inherited MouseDown(Button, Shift, X, Y);
end;

procedure TMMButton.CMMouseEnter(var Message: TMessage);
begin
  FOver := True;
  doSound(FSndOver);
  if (not FBmpHiLight.Empty) or FDown then
    Paint;
end;

procedure TMMButton.CMMouseLeave(var Message: TMessage);
begin
  FOver := False;
  if (not FBmpHiLight.Empty) or FDown then
    Paint;
end;

procedure TMMButton.MouseUp(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  DoClick: Boolean;
begin
  FDown := False;
  DoClick := (X >= 0) and (X < ClientWidth) and
             (Y >= 0) and (Y <= ClientHeight);
  if DoClick then
    begin
      Paint;
      if Assigned(OnClick) then
        OnClick(Self);
    end;
  inherited MouseUp(Button, Shift, X, Y);
end;

procedure TMMButton.Click;
begin
end;

procedure TMMButton.Paint;
var
  ARect: TRect;
  Src: TBitMap;
  OldPal: HPalette;
begin
  OldPal  := SelectPalette(Canvas.Handle,
                           FBmpNormal.Palette,False);
  try
    RealizePalette(Canvas.Handle);
    ARect  := Rect(0,0,Width,Height);
    if (csDesigning in ComponentState) then
      // Design-time paint response.
      if FBmpNormal.Empty then
        begin
          // Add visibility when designing.
          with Canvas.Pen do begin
            Style  := psSolid;
            Color  := clGray;
            Mode  :=  pmXor;
          end;
          Canvas.Brush.Style  := bsClear;
          Canvas.Rectangle(0, 0, Width, Height);
        end
      else
        Canvas.CopyRect(ARect, FBmpNormal.Canvas, ARect)
```

```
    else
      begin  // Run-time paint response.
        // Check button state & assign appropriate bitmap.
        if not Enabled then
          Src := FBmpDisabled
        else if not FOver then
          Src := FBmpNormal
        else if FDown then
          Src := FBmpPushed
        else
          Src := FBmpHiLight;
        // Catch all if the Src bitmap is not valid at this
        // point, paint the normal bitmap.
        if Src.Empty and (not FBmpNormal.Empty) then
          Src := FBmpNormal;
        // Paint the component's canvas.
        if not Src.Empty then
          Canvas.CopyRect(ARect, Src.Canvas, ARect);
      end;
  finally
    if OldPal <> 0 then
      SelectPalette (Canvas.Handle,OldPal,False);
  end
end; // Paint

end.
```

End Listing Two

Begin Listing Three — Sample Application

```
program Test;

uses
  Forms,
  TestMain in 'TestMain.pas' {frmMain};

{$R *.RES}
{$R sounds.res}

begin
  Application.Initialize;
  Application.CreateForm(TfrmMain, frmMain);
  Application.Run;
end.

unit TestMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, MMButton, ExtCtrls;

type
  TfrmMain = class(TForm)
    imgBack: TImage;
    MMButton1: TMMButton;
    procedure FormCreate(Sender: TObject);
    procedure MMButton1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmMain: TfrmMain;

implementation

{$R *.DFM}

procedure TfrmMain.FormCreate(Sender: TObject);
begin
  ClientWidth := imgBack.Width;
  ClientHeight := imgBack.Height;
end;

procedure TfrmMain.MMButton1Click(Sender: TObject);
begin
  ShowMessage('Fired the cannon!');
end;

end.
```

End Listing Three

Delphi 1, 2, 3

By John Ayres

Restoring Animation

Delphi Apps *Can* Exhibit Standard Minimize and Restore

Delphi does a great job of encapsulating the Windows API, insulating developers from some nasty and mundane requirements of Windows programming. Due to this encapsulation, however, Delphi programs exhibit certain anomalies not present in Windows applications written in other development environments; for example, the lack of animation when a Delphi program is minimized or restored.

When a window is minimized, it displays a series of animated rectangles, decreasing in size and moving toward the task bar until the window is fully minimized. The reverse occurs when a window is restored from a minimized state. Unfortunately, the way Delphi encapsulates a Windows application prevents this from occurring with Delphi-created applications.

When a Delphi application is created, there are two windows present: the main window, and a hidden application window. It is the hidden application window that contains the entry and exit points for the main function of the Windows application. When the main window of an application is minimized, the "minimize" message sent (i.e. SC_MINIMIZE) is intercepted by this hidden application window. Instead of minimizing the main window, it's the hidden application window that minimizes; the main form and all other displayed forms are hidden. When an application is restored, it's the application window that's restored, and the main form and all other forms are returned to a visible state. Because the main window is never truly minimized, the minimizing animation never fires.

To correct this, we must re-route specific messages from the *Application* object to the main form, and provide additional processing to minimize and restore. The first step is to identify where code must be placed to intercept and handle the appropriate messages. We will be dealing with the WM_SYSCOMMAND message, which is sent when the user chooses Minimize or Restore from the system menu, or presses the appropriate button in the upper-right corner of a window.

The *OnMinimize* and *OnRestore* events of the *Application* object seem like a good place to start. However, both of these events fire after the *Application* object has been minimized or restored. If the *Application* object is minimized before the main form is minimized, the main form will simply vanish, and the minimizing animation will never appear. Therefore, the *OnMinimize* event will not work for our purposes. Conversely, the *Application* object must be restored before the main form is restored, so the main form will be visible and the restoring animation will occur. We can then use the *OnRestore* event to run code that will restore the main form. Another *Application* object event to consider is the *OnMessage* event. This event fires before the application is minimized, thus allowing us to send the SC_MINIMIZE message to the main form before it disappears.

Therefore, we will create event handlers for the *Application* object's *OnMessage* and *OnRestore* events to route the appropriate messages to the main form. We'll also need a message handler for the WM_SYSCOMMAND message. Thus, our *TForm* class declaration should appear as:

```
TForm1 = class(TForm)
  procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    procedure AppRestore(Sender: TObject);
    procedure AppOnMessage(var Msg: TMsg;
                           var Handled:
Boolean);
    procedure WMSysCommand(var Msg:
TWMSysCommand);
      message WM_SYSCOMMAND;
  end;
```

Now that the appropriate event handlers have been added to the class declaration of the main form, hook them up to the *Application* object in the form's *OnCreate* event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  { The Application.OnMinimize event fires after the appli-
    cation has been minimized and the main form is hidden.
    So we must provide a handler for the OnMessage event,
    which fires before the application is minimized. }
  Application.OnMessage := AppOnMessage;

  { The Application.OnRestore event fires after the appli-
    cation has been restored. This is fine for our purposes
    but the application doesn't send the Restore message to
    the form. Our main form will reappear, but it will be
    minimized in one of the lower corners behind the task
    bar. Thus, we must provide a handler for the OnRestore
    event so the Restore message can be sent to the form. }
  Application.OnRestore := AppRestore;
end;
```

When the *Application* object gets WM_SYSCOMMAND (indicating a minimize or restore action), the message is handled by the *Application* object, then destroyed (the main form never sees the message). The *OnMessage* event of the *Application* object will fire before the message has been handled and destroyed, allowing us to send the message to the main form before it's hidden. In the *OnMessage* event handler, determine if the message is WM_SYSCOMMAND, and if it indicates the application is being minimized. If so, call the *SendMessage* API function to pass this message to the main form:

```
procedure TForm1.AppOnMessage(var Msg: TMsg;
                              var Handled: Boolean);
begin
  { The OnMessage event fires before the OnMinimize event,
    so we must put code here to route the Minimize message
    to the form. If the message is coming from the
    system menu... }
  if Msg.Message = WM_SYSCOMMAND then
    { ...and it is specifically the Minimize command... }
    if (Msg.wParam and $FFF0) = SC_MINIMIZE then
      { ...send the Minimize message to the form. }
      SendMessage(Handle, WM_SYSCOMMAND, SC_MINIMIZE, 0);
end;
```

This message will be intercepted by our WM_SYSCOMMAND message handler. We must check the incoming message type to determine if it's SC_MINIMIZE. This handler will receive all system command messages; check for and handle only the appropriate command. All other commands must be sent to the inherited message handler so the standard functionality of a window won't be compromised. When checking the *CmdType* member of the *TWMSysCommand* structure passed to our handler, combine it with the value `$FFF0` using the Boolean **and** operator. The least significant four bits of this member are used internally by Windows, and we must mask them out to determine the appropriate message type. When we've received SC_MINIMIZE, pass the message information to the *DefWindowProc* API function, which performs the default behavior of the specified message on the window whose handle is passed to the function in its first parameter. This function causes the form to truly minimize, and the minimizing animation to appear. Then, call the *Minimize* method of the *Application* object, which minimizes the application and hides our minimized main form. Our WM_SYSCOMMAND message handler should look like:

```
procedure TForm1.WMSysCommand(var Msg: TWMSysCommand);
begin
  { If we are receiving the Minimize message... }
  if (Msg.CmdType and $FFF0) = SC_MINIMIZE then
    begin
      { ...pass it to the default window procedure. This
        causes the form to minimize with animation. }
      DefWindowProc(Handle, WM_SYSCOMMAND, SC_MINIMIZE, 0);

      { We must now minimize the application object. This
        actually minimizes the application, hiding the
        minimized form. }
      Application.Minimize;

      { Indicate that the message was handled. }
      Msg.Result := 0;
    end
  else
    { For any other message, we must send it to the
      inherited message handler. Because the form is
      already minimized, the Restore animation will work
      correctly. }
    inherited;
end;
```

This should cause animation to appear when minimizing the form. When the application is restored, the main form will reappear, but it will remain minimized unless it's specifically told to restore. To do this, we post the WM_SYSCOMMAND message, specifying the restore command (SC_RESTORE) to the main form from the application's *OnRestore* event handler. No other processing is required; Windows takes care of the rest:

```
procedure TForm1.AppRestore(Sender: TObject);
begin
  { When the minimized application is restored, the appli-
    cation object simply sets the Visible property of its
    forms to True; the forms don't actually receive the
    Restore message. Therefore, when the application object
    receives the Restore message, we must specifically send
    it to the main form. }
  PostMessage(Handle, WM_SYSCOMMAND, SC_RESTORE, 0);
end;
```

Your Delphi apps will now exhibit the minimizing and restoring animation common to Windows apps. But this method isn't perfect; the animation will fall to one side of the screen instead of animating toward the icon on the task bar. However, this technique does provide a more traditional Windows look and feel. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\MAR\DI9803JA.

John Ayres is a consultant for Ensemble Systems Consulting in Dallas, using Delphi to produce high-end client/server applications for various Fortune 500 companies. With over eight years of programming experience, he's worked for a variety of companies, producing a broad range of software, from third-party add-in utilities to games. He keeps himself busy by co-authoring *The Tomes of Delphi 3: Win32 Core API* (ISBN 1-55622-556-3) [WordWare, 1998] and other Windows programming books for Delphi. For more information, visit http://www.WordWare.com.

By Warren Rachele

Crystal Reports Professional 6.0

Seagate Software's Query and Report Writing Tool

The flagship product of the software development company where I work, The Hunter Group, was a case management tool that had a not-so-unique set of reporting specifications. In addition to a plethora of menu-chosen reports configured with user-modifiable parameters, the database was used to research trending not immediately apparent through the stock reports. For this reason, the user requirements specified the necessity of being able to create free-form reports and queries. While Borland has improved reporting functions with the addition of the QuickReport components, they remain internal, and can only be modified through a series of parameters passed from the user.

To satisfy this requirement, we turned our attention to external report-query tools, specifically report writers. The most powerful, non-intimidating tool available was Crystal Reports from Seagate Software, which we've included as a part of our product since its inception. In version 6.0, Crystal Reports works with data on a wide range of platforms, and comes with components that compile directly into all of the popular visual development environments in use today. The collection of tools included in the Crystal Reports package considerably expands the capabilities of, and market opportunities for, your software.

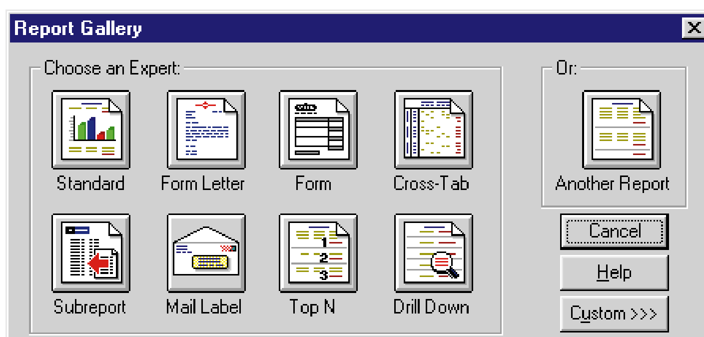


Figure 1: Selecting a report format from the Report Gallery.

Make Room!

A full installation of the 32-bit version of Crystal Reports Professional, including samples, documentation, the data access layers, and tools that ship with the product, takes a bit under 150MB. I recommend you install the sample files if you're not familiar with the product. There are numerous tutorials in the printed documentation that make use of these files, and each is helpful in shortening the learning curve. When no longer necessary, these files are easily deleted.

Performing a custom installation allows you to install only the components you're interested in utilizing in your development efforts. The installation goes quickly and without problems. Depending on the components and data-access layers you select, the Borland Delphi Engine (BDE) and ODBC (Open Database Connectivity) pieces may be installed. You will be reminded to configure them at the end of the installation process.

Installing the Crystal Reports Engine VCL interface component into Delphi 3 follows the standard process for installing any third-

party component. Choosing to install the new component adds the object to the Data Access tab of the VCL. Code for Delphi 3 is located in a separate subdirectory of the VCL directory, and the components and support code for earlier versions of Delphi are also provided.

Basic Component Usage

Building a report into your Delphi program is a two-step process. The report is designed in the Crystal Reports Designer environment. Integrating the report into your executable consists of adding *TCrpe* to your form and pointing the properties to the report. When the component is executed, it will call the Crystal Reports Engine to handle the build and printing of the report. New reports are formatted via a wizard that walks you through the design process, and ends with a completed report that needs very little modification. An included sample report demonstrates how easily a report can be assembled using the Biolife.DB, a sample Paradox table provided with Delphi and stored in the \Delphi 3\Demos\Data directory.

The first step in the process is to select a report format from the Report Gallery dialog box; the sample uses a Standard format (see Figure 1). Selecting a format takes you directly to the Create Report Expert dialog box in which the rest of your report selections are made (see Figure 2). Add the fields Species No, Category, Common_Name, Species Name, Length, and Length_In to the report (see Figure 3). Click on the Sort tab and select **BIOLIFE.Category** as the sort field (see Figure 4). Finally, click on the Style tab and select **Leading Break** from the list (see Figure 5). Your report is finished with one small annoyance to be corrected: Crystal Reports subtotals and totals all numeric fields unless told not to. Click on the Total tab and remove all the fields selected to be totaled. Click the **Preview Report** button, and your report is configured in a WYSIWYG window (see Figure 6).

This report is now usable in a Delphi program. As shown in Figure 7, two components are needed to test the report: *TCrpe* and a Button to execute the report. Add the following statement to *Button1Click*:

```
Report1.Execute;
```

TCrpe requires minimal modification to work. The only attribute that needs to be set for this example is the report name. This property is set to the report just created in Crystal Reports.

Run the sample program and you see the form window with a button. Clicking the **Report** button starts the report-generation process. The engine loads quickly (without a splash screen) and is printed to the screen for preview by default. The Crystal Reports Engine window automatically provides the user with full preview capabilities: zoom, page by page, top of report, and end of report. A button sends

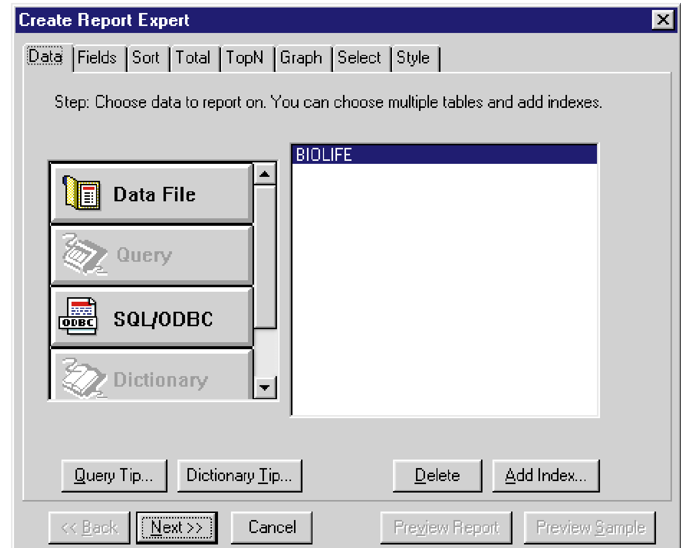


Figure 2: The Create Report Expert dialog box.

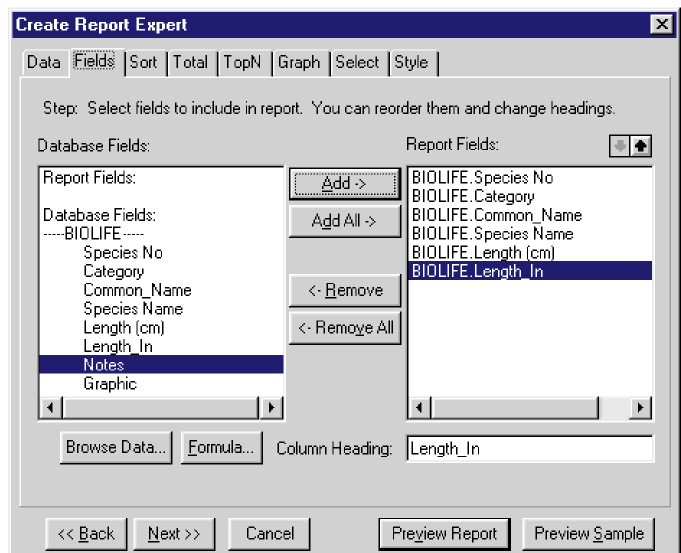


Figure 3: Adding fields in the Create Report Expert dialog box.

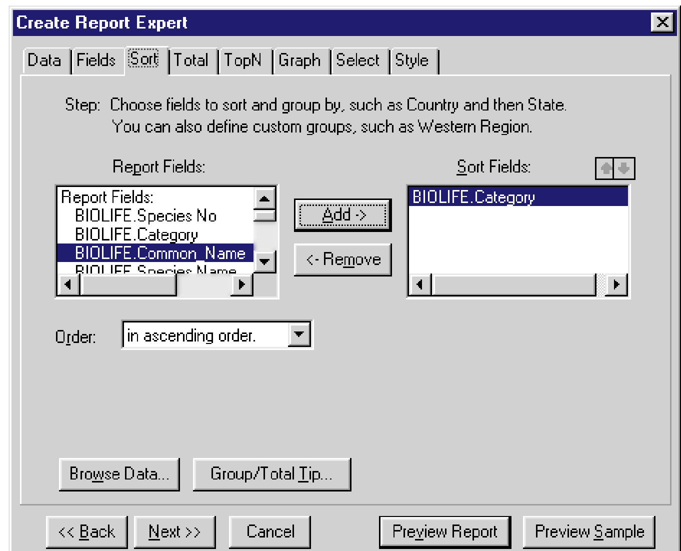


Figure 4: Selecting categories in the Create Report Expert dialog box.

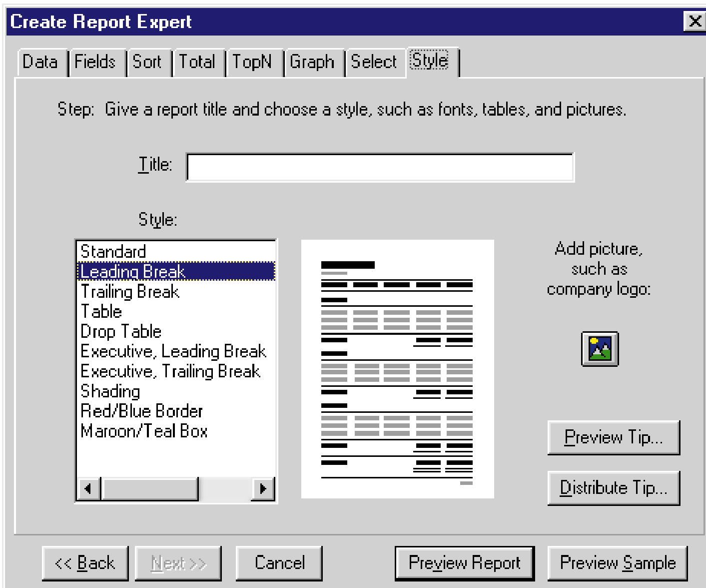


Figure 5: Selecting styles in the Create Report Expert dialog box.

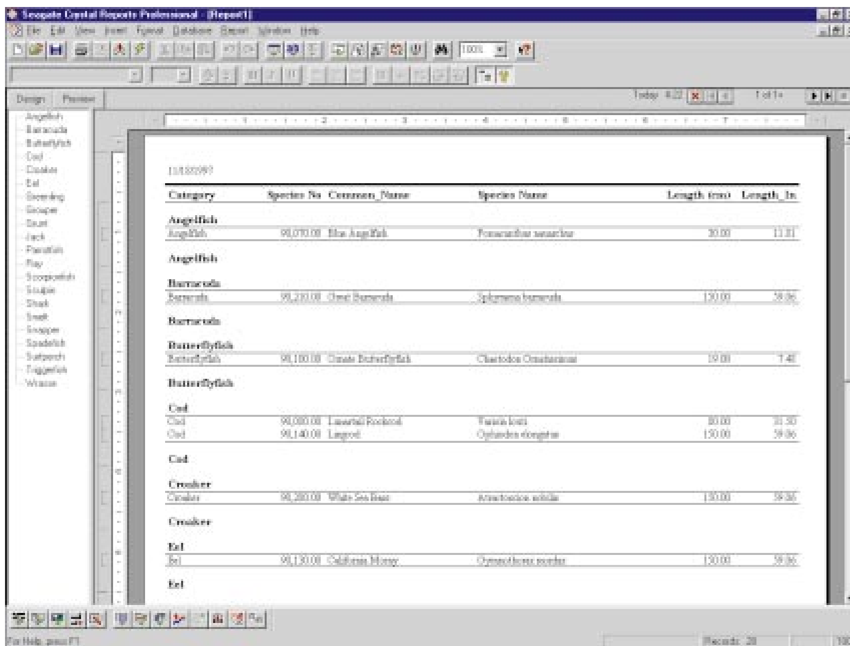


Figure 6: The report is configured in a WYSIWYG window.

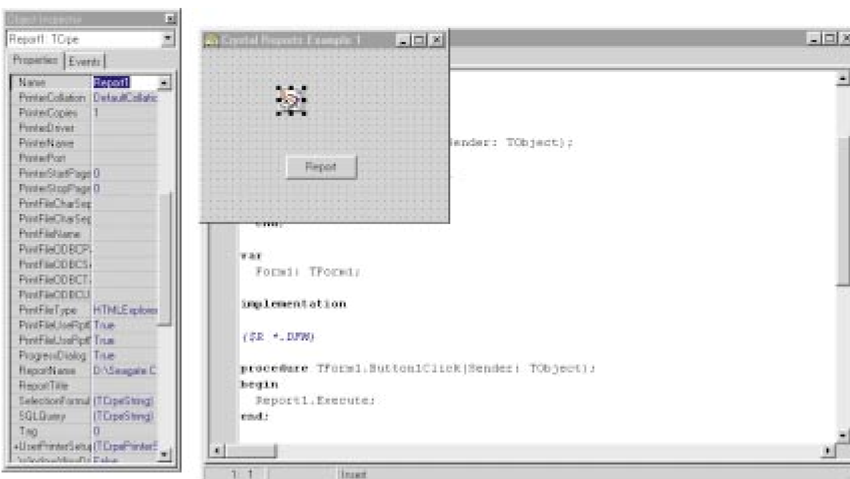


Figure 7: Putting Crystal Reports to work in Delphi.

the output to the printer and closes the window. Another button is provided that connects to the distribution expert for quickly sending the output to a destination other than the printer.

The properties of the VCL component provide an interface to nearly all the modifiable properties of the report and its display. Data-selection parameters and the selection formula itself can be passed directly from the component, allowing a broad range of user control over the data selection and window presentation. The extensive range of output destinations available through the report designer is also available through your executable. Crystal Reports provides a wide range of destinations and formats for the report, other than a printer, and facilitates this through an interface to the Destination Expert. The output can be directed to a disk file in various popular formats or sent directly to another person in the form of an attachment to an e-mail message. The e-mail formats that are supported include MAPI, VIM (cc:mail), and Microsoft Exchange. Crystal Reports can also directly produce HTML output for placing reports on a Web page or intranet.

Seagate Toolbox

Seagate provides tools to make end-user access to your data stores easier and more secure. The Crystal Dictionary system provides the database professional with the ability to make their data more accessible to users by formatting the data elements into more friendly terms while limiting access to the level needed by the individual. Users are often unable to access a database directly because of their lack of understanding of the relational model. A database made up of multiple interrelated tables is easily understood by the database designer, but the relationships aren't always clear to the end user.

A Crystal Dictionary creates a *view*, a way of presenting the data in a more user-friendly way that hides the complexity of the relationships. Field and table names can be changed through an alias making them more recognizable to the user. New column headings and contextual help can be attached to each field or table, making the data columns more meaningful. This friendlier interface adds to the productivity of the end-data users by making reporting and querying easier. The view lowers the number of support calls needed

Crystal Reports Professional 6.0 works with data on a wide range of platforms, and comes with components that compile directly into all of the popular visual development environments in use today. The collection of tools included in the Crystal Reports package considerably expands the capabilities of, and market opportunities for, your software. While the strength of the Report Designer is enough to recommend this product, the VCL provides a fast loading, highly modifiable interface to a powerful report engine.

Seagate Software
1095 West Pender St., 4th Floor
Vancouver, BC V6E 2M6
Canada
Phone: (800) 877-2340 or (604) 681-3435
Fax: (604) 681-2934
E-Mail: sales@img.seagatesoftware.com
Web Site: http://www.seagatesoftware.com
Price: Professional Edition, US$395, upgrade US$199; Standard Edition, US$149, upgrade, US$79.

when providing user access to the data stores, and has the additional benefit of limiting access to the specific data fields the designer deems appropriate to the user's security level.

The Crystal Dictionary has a secondary benefit: It makes Paradox and .DBF data immediately accessible to the Crystal Query Designer. The query tool is designed to create simple-to-complex SQL statements for data selection. The Crystal Query Designer works only with SQL databases or those accessible through ODBC. This tool is a visual query builder that steps the user through the process of assembling a workable SQL sentence. If the user is prepared for building more complex queries without the use of the wizard, the tool provides for direct entry of the sentence. Crystal Query Designer can make use of queries that the user has designed and used in other products by importing them into the tool. Executing the sentences is simply a matter of clicking the Run button and having the selected rows returned to the display window.

Documentation

The documentation supporting Crystal Reports is segmented by intended audience. The *User's Guide* offers a good basis for starting with the product, providing tutorials and numerous examples for those who will use the product interactively. Advanced users won't be satisfied with the level of this document; it leaves numerous important questions unanswered. Developer documentation is split among several online files. For the Delphi user there are two Windows Help files of interest: Ucrpe.hlp and Developr.hlp. The first documents the VCL component; the second offers general help for the developer accessing the Crystal Reports Engine. The quality of the online documentation is spotty, containing several errors, omissions, and poor examples. For more technical information regarding the report engine, the developer must switch formats completely. The *Technical Reference* is a .PDF file, readable through Adobe Acrobat.

Market Positioning

Seagate Software markets Crystal Reports as a desktop query and report writer tool within an application's segment they call Business Intelligence. The targets of this market, information workers, are substantially different from those of the software development population. Information workers, whose responsibilities include data mining and report production from corporate data stores, are seeking an intuitive interface and extensive selection and formatting capabilities. This group will be thrilled with this latest release. The learning curve for all but the fundamentals is rather steep because of the quality of the documentation, but the results are more than worth the effort.

Software developers have dual learning curves in their paths to success. The first task is to learn the Crystal Reports method of report creation. This will be easier for the developer than the information worker as the steps will be more intuitive. Creating a formula, for example, is a more comfortable process for the developer than the average information worker, just as the process of assembling a syntactically correct sentence is the programmer's domain. The programmer then must learn the interface specific to his or her development tools. Seagate doesn't make this easy. As mentioned earlier, the documentation is not set up to be read; it's intended for ad hoc use. Developers are expected to have an above-average knowledge of their platforms. And programmers working with the Delphi VCL component will be comfortable adding the object to their projects and setting their properties. The number of properties will require some study of the Help files in order to fully realize the power of the component, but the results are worth the effort.

Conclusion

Seagate Crystal Reports Professional 6.0 is an excellent product to consider on either of two levels. For the Delphi developer, the VCL component provides a fast loading, highly modifiable interface to a powerful report engine. Its formatting, selection, and output destination capabilities open numerous possibilities for your software development efforts.

If the user specification of your project calls for an external query and report writing tool, it would be difficult to find one this powerful, yet as easy to use. Properly trained users of Crystal Reports will discover ways of becoming more productive users of the data created by your application. The strength of the Report Designer alone is enough to recommend this product, but the value exceeds that offered by the component-only products. ∆

Warren Rachele is Chief Architect of The Hunter Group, an Evergreen, CO software development company specializing in database-management software. The company has served its customers since 1987. Warren also teaches programming, hardware architecture, and database management at the college level. He can be reached by e-mail at wrachele@earthlink.net, or by telephone at (303) 674-8095.

By Peter Hyde

Rubicon for Delphi

Tamarack Associates' Database Search Engine

Have you ever noticed the way users encounter a new way of doing things, grow to like it, and suddenly it has to be in every application you create? Of late, full-text searches, like those popularized by sites such as AltaVista (http://altavista.digital.com), have entered the ranks of the "must have."

This is no mere peccadillo on the part of the user. For many classes of applications, there are clear benefits to using a full-text database search, rather than (or as well as) a more structured and formal SQL-style one. Users find keyword-based searching easy to learn and master because they can dig into the content of BLOB fields containing entire documents and, above all, a well-implemented search can provide unsurpassed performance. Nor is there anything to prevent a savvy developer from implementing hybrid searches, i.e. those that use keywords to quickly narrow the target range, then apply more traditional comparisons on fields such as dates or prices to produce the final result set.

Figure 1: Rubicon search performance at the Delphi resource Web search engine.

There is a trade off, of course. Such an engine depends on generating, maintaining, and using indexed word tables for each database — a job that can be very resource hungry in memory, disk and network space, CPU terms, and time. Having gone down this track more than once, I've often wished for a component set that provides peak search performance and flexibility while keeping a close eye on resource usage.

Enter Rubicon from Tamarack Associates. I first saw it in action at a pre-launch demonstration of the Delphi resource site at http://developers.href.com (see Figure 1), where it was performing searches on 300MB InterBase tables in under half a second. (Now that the site has closer to a gigabyte of information, you can go and check its speed for yourself.) Tamarack claims speed improvements of up to 5,000 times compared to SQL queries — and the larger the database, the better it gets.

After a closer inspection, I wished I'd discovered it earlier, not least because of the creative and efficient way it minimizes the size of its index files. Consider how you'd go about preserving a list of the records in a 100,000-record database that might contain a given word. Using a BLOB field to store a Longint ID for each matching record seems like a reasonable approach. But in no time at all you have a table where each word's index BLOB might contain up to 100,000 Longints!

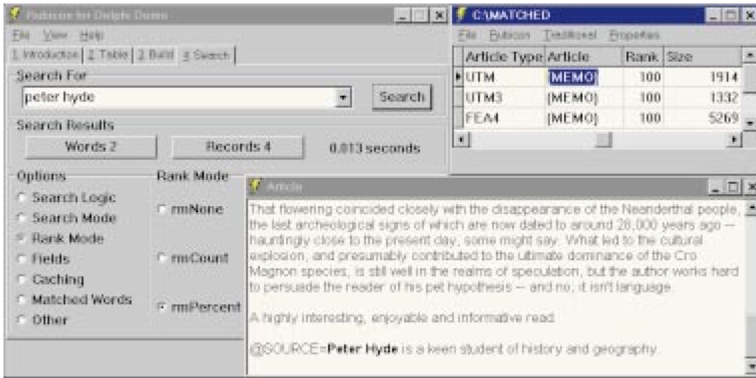


Figure 2: Rubicon located four of my articles — in a Paradox database — across a LAN in just over 1/100th of a second.

Economy *and* Speed

Rubicon does better than this: Instead of storing record numbers, it simply maintains a bitmapped array of record markers. If a word is present in a record, its bit is turned on; if not, it's turned off. Immediately, the maximum BLOB size in each word-table record is reduced from roughly 400KB to 12.5KB. Not only that, but smart bit-style operators are very fast and easy to use when performing "and" and "or" operations in multi-word searches. In effect, each Rubicon word ends up with a sparse array of bits, which leads to another optimization: The best thing to do when storing sparse arrays is to compress them. Rubicon will typically compress that 12.5KB to under 1KB, optionally in memory as well as on disk. The result is a search tool that's skimpy on resources without sacrificing speed. To test this, I ran its demonstration program against a 2,000-article database (approximately 6MB of data). Rubicon took a minute to create an index table of 2MB, then performed multi-word searches that located matching articles in under 1/50th of a second (see Figure 2). Only the optional creation of a result table of matching records took any noticeable time. This was natural because, at that point, each record must be physically copied across the network.

Flexibility

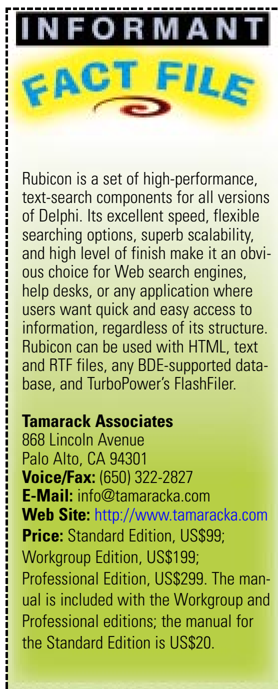
Rubicon gets full marks for speed and economy. What about flexibility? Well, power users won't be disappointed. In addition to words, phrases, and simple Boolean expressions, Rubicon also supports expressions such as "search near Delphi" or "like angle", and reasonably complex combinations such as "like deficit or like loss". Critical issues such as lookup fields have not been overlooked: Rubicon lets you define table links so words coming from the lookup tables can be indexed, not just words from the main table. Additional features such as ranking, word hints, search narrowing, and on-the-fly results mean that applications using Rubicon will have a friendly and finished feel. None of these features would be worth having if Rubicon were weak in the critical area of index creation and maintenance. No database or Web application would survive long in production if it had to be shut down for long periods so the index could be regenerated. Fortunately, again, Rubicon shines.

Highly Scalable

To begin with, Rubicon provides not one, but two index-creation components. *TMakeDictionary* is used for initial index-table creation and also (desirably) whenever extensive changes have been made to the database or external files being indexed. It is typically run in batch mode, perhaps overnight for very large jobs. For minor edits, updates, or deletes, the *TUpdateDictionary* component is much quicker as it works by amending the index rather than rebuilding it. (A list of key components is shown in Figure 3.)

Class	Description
TMakeDictionary	Used to scan the records of the source table, and create or recreate a dictionary of all the words used in the selected fields and their record locations in the table. Network, threaded, and huge table versions come with Workgroup and Professional editions.
TUpdateDictionary	Keeps a dictionary synchronized with changes in the source table. Network, threaded, and huge table versions come with Workgroup and Professional editions.
TSearchDictionary	Performs word searches using the dictionary. A huge table version comes with the Professional Edition.
TMakeProgress	A drop-in form that will automatically configure itself to display the progress of *TMakeDictionary*.
TUpdateStats	A drop-in form that will automatically configure and display itself when *TUpdateDictionary* is used. It's primarily designed for monitoring the update process during development.
TUpdateTable	A descendant of *TTable* that has several key *TUpdateDictionary* interface methods built in. *TUpdateDictionary* checks to see if its DataSource.DataSet is a *TUpdateTable*, and, if so, will automatically connect the appropriate methods.
TRubiconRichEdit	A read-only control that can display text or RTF files with matching words highlighted. The behavior of the control is similar to a *TDBMemo* or *TDBRichEdit*, but is only available on 32-bit platforms.
TSearchHints	A *TCustomGrid* descendant that displays words that match or nearly match the words the user has entered into an edit control. The words displayed may be updated as the user types, thus giving the user instant feedback.
TSearchController	Performs a search on multiple tables (Professional Edition only).

Figure 3: Key Rubicon components.



Any number of users can search using the Standard Edition of Rubicon, but only one process can do index builds or updates. The Workgroup Edition supports threading so updates can be carried out from multiple workstations, and also adds useful features, such as indexing of HTML and RTF files, and matching-word highlighting.

For high-end applications, the Professional Edition provides a version of the components that supports segmented indices for tables with more than 250,000 records, multiprocessor indexing, and multitable searching.

Help

Rubicon's documentation includes an excellent introduction to the components and the way they should be used, a "reference" section that alphabetically lists the components, properties, and methods with clear examples, and information on advanced topics, such as huge tables, linked lookup fields, working with *TQuery*, memory management, threading, and much more. In contrast, the online Help is a little disappointing. It's a straight facsimile of the manual, meaning the layout and hyperlinks are somewhat idiosyncratic. Most frustrating of all is that it's not context sensitive in Delphi 3. (The other compilers are fine.)

Apart from the demonstration program, a collection of tightly focused example projects is provided with Rubicon to highlight key features, components, and properties. Finally, not to be overlooked are utility components and programs that range from an indexing progress meter to index verification, optimization, and comparison programs. Together, they make this a refined product indeed.

Almost inevitably, a product as good as this will attract third-party add-ons. In the case of Rubicon, it's highly predictable that such support will be Web-oriented. In mid-1997, HREF Tools Corp. (http://www.href.com) released *TWebRubicon*, a component that integrated Rubicon operations into their WebHub application framework. If there are not similar add-ons out there already from other vendors, they are sure to follow.

Conclusion

Full-text searching is a very useful and oft-requested search strategy, yet not at all easy to implement efficiently. Tamarack Associates has tackled the job with intelligence, flair, and thoroughness. Search no further. Δ

Peter Hyde is the author of *TCompress* and *TCompLHA* component sets for Delphi and C++Builder, and is Development Director of South Pacific Information Services Ltd., which specializes in creating dynamic Web sites. He can be reached via e-mail at peter@spis.co.nz or http://www.spis.co.nz.

Delphi Developer's Handbook

While the early crop of Delphi books (circa 1995) tended to be general in nature and aimed at beginning- and intermediate-level developers, many of those from late 1996 through 1997 have included more specialized and advanced offerings. One of the last Delphi books released in 1997, *Delphi Developer's Handbook* [SYBEX, 1997], covers a wide range of topics, but is nevertheless advanced in its treatment of those topics. Written by Marco Cantù (of SYBEX's *Mastering Delphi* series) and Tim Gooch (former editor of the Cobb Group's *Delphi Developer's Journal*), this work will be a welcome addition to your library if you're looking for an advanced reference aimed more at application developers than tools (component/wizard) developers.

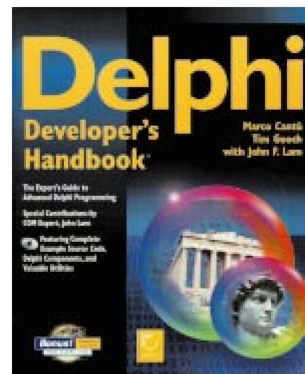
This book hits the ground running with advanced information on bit manipulation, long strings, classes, and levels of protection. While many of these topics are not unusual, the treatment certainly is: accessing the inner structure of long strings, manipulating classes in various useful ways, and circumventing Delphi's protection scheme. Similarly, the second chapter exposes some of the more esoteric aspects of Delphi's component structure, including working with component ownership, using the *FindComponent* method, and creating type-safe *TList* derivatives.

Chapter 3, on streaming and persistence, provides an excellent overview of this important topic with a thorough discussion of file and memory streams and techniques for copying data between them. New streaming classes are also developed. To my delight, the authors included a discussion of the *TWriter* and *TReader* support classes, and how to use them with streams.

Any comprehensive, advanced Delphi work should include a discussion of components and experts; *Delphi Developer's Handbook* certainly fulfills this requirement. There are chapters on building components throughout, from basic principles to compound components, from using *TCollection* classes as properties to data-aware controls. (In addition to the latter topic, two chapters cover advanced database and client/server programming.) Essential component-related topics such as component editors, properties, property editors, and packages are also covered.

There is also information regarding Experts/Wizards throughout, including a chapter on Run-Time Type Information (RTTI), a nearly 100-page chapter on Wizards, and a fascinating chapter on "Other Delphi Extensions." The latter includes a summary of the ToolsAPI, a discussion of how to handle Delphi notifications, and an overview of the Version Control System (VCS) interface. The section on hacking Delphi is most intriguing, with instructions regarding a simple means of changing Delphi's Component palette. Finally, many of the tools developed throughout *Handbook* will make useful additions to your programming arsenal. I'm particularly impressed with the Object Debugger, which clones the Object Inspector, makes it available at run time, and even adds editing capabilities to it. An entire chapter is devoted to building this useful tool. Several of these topics are rarely written about. There are others, including writing Windows applications without the VCL (Visual Component Library), using Windows messages and API function calls, and extending Delphi's *TApplication* and *TForm* classes.

Some of the newer technologies are dealt with in a substantial manner. There are two chapters on the



Component Object Model (COM) written by COM expert John F. Lam. There is also a good discussion of some of the Internet-related technologies, such as HTML, CGI, and ISAPI.

By no means have I mentioned every topic included in this fine volume. Suffice it to say, if you've been programming in Delphi for a while and are ready to explore its more advanced aspects, this book is for you. If you're new to Delphi, you should probably start with a more basic work such as Cantù's *Mastering Delphi 3* [SYBEX, 1997].

— *Alan C. Moore, Ph.D.*

Delphi Developer's Handbook by Marco Cantù, Tim Gooch, with John F. Lam, SYBEX, 1151 Marina Village Parkway, Alameda, CA 94501, (510) 523-2826.
ISBN: 0-7821-1987-5
Price: US$49.99
(1,134 pages, CD-ROM)

Delphi and the APIs

Getting to the Source

In an editorial in the December, 1997 issue of *Delphi Informant*, I discussed the need to make new technologies and their APIs available in Delphi and the Jedi movement that is attempting to do just that. But what about the standard APIs? Since Delphi has implemented a large part of the Windows operating system, do we need to know the gory details of Windows functions and messages? In everyday programming, probably not. However, if we're extending components or accessing functionality not supported by Delphi, we might need detailed information. Where can we find it?

In the bad old days BD (Before Delphi), if you were programming for Windows in C or Pascal, you needed to work at the API level to a greater extent. In those 16-bit days, a very popular reference work was *Windows API Bible* by James L. Conger [Waite Group Press, 1992]. This single volume contained the essential information of the time: the message system, the various standard controls, input/output, and even multimedia and communications. Today, the existing APIs have grown considerably, and new ones have been added to the standard package. To keep up, the Waite Group's *Bible*, written mainly by Richard J. Simon with help from Michael Gouker and Brian Barnes, jumped to three volumes between 1996 and 1997. Let's examine each.

Windows NT Win32 API SuperBible provides an excellent introduction to Windows 32-bit programming — both Windows 95 and NT 4.0 — and includes an explanation of how an application qualifies for the Windows 95 logo. This first volume includes a discussion of windows and dialog boxes, menus and other basic interface objects, message systems (messages themselves are in the second volume), memory management, input/output, graphics, and system information. Some of the new areas include threads and the registry. I found one of the last chapters particularly interesting: "File Decompression and Installation." If you're interested in building a set of installation components as part of a setup-generating system, you'll find much of what you need here: functions to check version information, functions to manipulate compressed files, and functions to determine what files (DLLs) are installed on a user's system.

Windows 95 Common Controls and Messages API Bible presents the controls inherited from Windows 3.x and the new ones added with Windows 95. It also includes a complete reference to Windows messages. If you want to learn more about the new controls, such as toolbars, status bars, tree views, and rich-text edit controls, and extend their functionality, you'll find this volume particularly helpful.

The final volume, *Windows 95 Multimedia and ODBC API Bible*, presents Windows' current support for Multimedia and ODBC. The latter topic is presented first and provides information on connecting to various DBMSes, executing SQL statements, converting data, and more. The multimedia section provides a detailed account of high- and low-level API function calls used when working with audio files (WAV and MIDI), AVI files, and the Media Control Interface (MCI). While not indicated in the title, this volume also discusses the Telephony Application Programming Interface (TAPI).

If you're planning to do a significant amount of low-level work at the API level, either in application programming or component creation, you should consider adding one or more of these volumes to your Delphi library. These references include numerous code examples to show how to access particular functions. While the code examples in this series are in C, you should be able to learn something from them. ∆

— Alan C. Moore, Ph.D.

Alan Moore is a Professor of Music at Kentucky State University, specializing in music composition and music theory. He has been developing education-related applications with the Borland languages for more than 10 years. He has published a number of articles in various technical journals. Using Delphi, he specializes in writing custom components and implementing multimedia capabilities in applications, particularly sound and music. You can reach Alan via e-mail at acmdoc@aol.com.