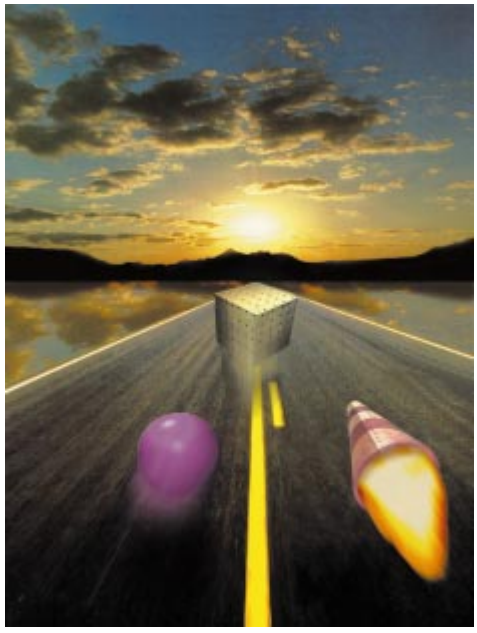
September 1997, Volume 3, Number 9

Delphi 3 DCOM

Building Multi-Tier Applications



Cover Art By: *Tom McKeith*

ON THE COVER

FEATURES

REVIEWS

DEPARTMENTS

Delphi TOOLS

New Products and Solutions

Speech Solutions Ships Speech Recognition for Delphi

Speech Solutions, Inc. is now shipping *ActiveX Voice Tools*, custom controls for the IBM VoiceType Dictation System version 3.0 speech recognition product.

This set of voice tools enables Delphi users to add speech recognition capabilities to their applications. Developers can drop the controls into their Windows applications, enabling free dictation, voice command, and voice control.

Voice Tools features Voice Notator, a playback and record control. Other controls include Press Panel, a 3D-voice command button and panel operated by voice, keyboard, or mouse; List Assist, a voice list box with standard list box properties and behavior; Voice Window, a replacement for the standard text box that accepts free dictation; and Sonar Tool, a voice navigator that controls voice-enabled objects, as well as standard Windows menus and sub menus.

The complete package includes Speech Solutions Voice Tools software, a developer run-time version of the IBM VoiceType Dictation System for Windows, a noise canceling microphone, documentation, and sample applications.

Speech Solutions, Inc.
Price: US$299
Phone: (800) SPEECH-7 or (215) 643-2100
Fax: (215) 643-9175
E-Mail: info@speechsolutions.com
Web Site: http://www.speech-solutions.com

RT Registration Control Available for Delphi 3

R&T Software has added Delphi 3 compatible modules to its *RT Registration Control*. The new features include packages, support for direct registration key validation from within GLBS Wise Installation System's setup scripts, and DES-based registration key generation.

RT Registration Control is a software protection library whose function is to provide a safe way to electronically distribute software. The product offers typical features such as time- and use-based evaluation periods with a system clock backup control, leased/rented software support, as well as more sophisticated features, such as PC-stamp copy protection or a backup security file.

RT Registration is available for Delphi as a native VCL.

R&T Software
Price: Starts at US$59
Phone: (49) 5132-836021
E-Mail: info@rtsoftware.com
Web Site: http://www.rtsoftware.com

Multi-Edit Offers Delphi 3 Integration

American Cybernetics has announced its Multi-Edit for Windows now includes integration with Delphi 3. The Delphi 3 integration is part of Multi-Edit's Borland IDE Integration Package, which also supports Delphi 1 and 2, as well as C++Builder. As with Multi-Edit's integration with Delphi 1 and 2, Delphi-specific syntax highlighting, code templates, and language support are provided. Multi-Edit features such as columnar blocking, collapsible editing, multi-file search and replace, and side-by-side synchronized file compare are available as hotkeys.

For information, contact American Cybernetics at (800) 899-0100 or (617) 449-1440, or visit http://www.multiedit.com.

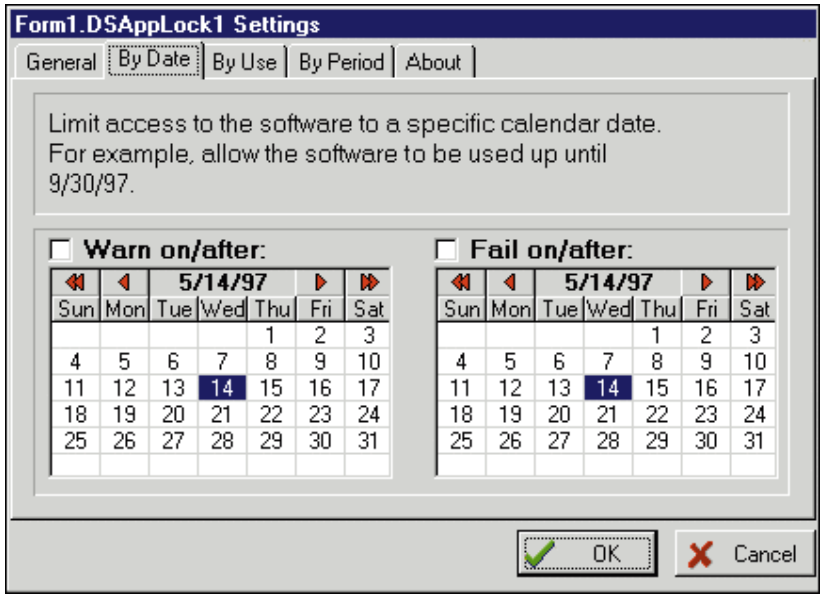
DesignSystems Announces DSAppLock 1.1 for Delphi

DesignSystems of East Sandwich, MA has released version 1.1 of *DSAppLock* for Delphi and C++Builder, providing component-based application protection in both environments.

DSAppLock enables developers to add application protection in less than a minute. It allows developers to create demonstration versions, as well as prevent time-critical software from being used beyond its proper life cycle.

DSAppLock lets developers customize an application's protection needs based on date, days since installation, number of times run, or a customized scheme. In addition, it can automatically generate unique codes to allow protected applications to be unlocked. DSAppLock ships with online Help, demonstrations, sample applications, unlimited support, and a 60-day money back guarantee.

DSAppLock supports Delphi 1, 2, and 3, as well as C++Builder. Trial and registered versions can be downloaded from DesignSystems' Web site.

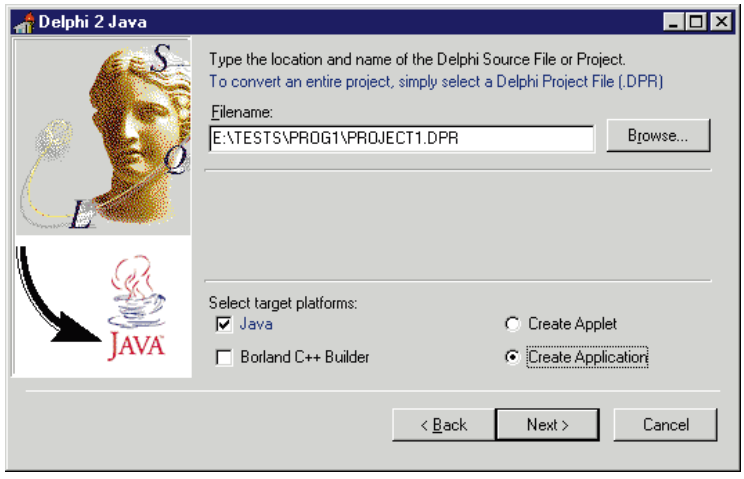


DesignSystems
Price: US$139 for Delphi or C++Builder editions; US$159 for both editions.
Phone: (508) 888-4964
E-Mail: support@dsgnsystms.com
Web Site: http://www.dsgnsystms.com

Delphi 2 Java Makes Delphi to Java Conversions Possible

PowerBBS Computing has launched ***Delphi 2 Java***, a software tool that converts Delphi applications to Java. Delphi 2 Java allows programmers to integrate Delphi Object Pascal code and visual forms to the Internet using Java. Delphi 2 Java does this by inputting Delphi projects, then converting the Object Pascal classes and visual forms to Java.

Delphi 2 Java can produce applications targeted to both languages, or simply to move applications entirely to Java. It also converts Delphi Object Pascal code to C++ for C++Builder.



PowerBBS Computing
Price: US$99; registered users will receive free upgrades.
Phone: (516) 938-0506
Fax: (516) 681-3226
E-Mail: support@java-delphi.com
Web Site: http://www.java-delphi.com

Tamarack Supports Delphi 3 with Rubicon 1.4

Tamarack Associates has updated ***Rubicon 1.4***, adding support for Delphi 3, C++Builder, and TurboPower Software Co.'s FlashFiler.

Rubicon technology performs database searches by indexing the words in a database or files on a disk. The end-user is then able to perform searches by entering words or phrases. Rubicon supports And, Or, Near, Not, and Like search logic and wildcards; searches may also be iteratively narrowed or widened. Search results may be used to filter the search table, navigate to matching records, or create a match or answer table in natural or rank order.

Version 1.4 introduces two new visual controls: Search and Rich Edit. Search displays a list of words and the number of occurrences of the word in the database as the user enters a query. This may be combined with on-the-fly search execution to provide real-time feedback. The Rich Edit control highlights the matching words in a field or document.

Searches can be performed across multiple tables with the new search controller component. The tables being searched need not have the same field structures.

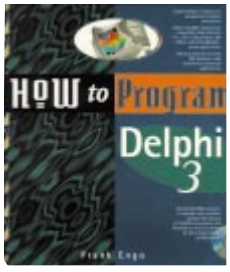
All Rubicon components are thread safe. This enables indexing and searches to be performed in the background. The Professional Edition allows indexes to be processed on multiple processors and/or computers.

All the components include full source code, and are compatible with Delphi 1, 2, and 3, as well as C++Builder.

Three editions of Rubicon are available. The Standard Edition is designed for small, medium, and large tables, and supports single-user applications, as well as multiple simultaneous searches. The Workgroup Edition supports multiple simultaneous updates to the search table, visual controls, thread support, and HTML and RTF handling. The Professional Edition supports large tables, indexing on multiple computers or processors, and multi-table searches.

Tamarack Associates
Price: Standard Edition, US$99; Workgroup Edition, US$199; and Professional Edition, US$299. All products include free 1.xx updates and support via e-mail.
Phone: (415) 322-2827
Fax: (415) 322-2827
E-Mail: info@tamaracka.com
Web Site: http://www.tamaracka.com

How to Program Delphi 3
Frank Engo
Ziff-Davis Press



ISBN: 1-56276-526-4
Price: US$39.99
(390 pages, CD-ROM)
Phone: (800) 428-5331 or (317) 581-3500

News LINE

September 1997

Free Delphi 3 Books and CD Giveaway

Nutshell Software has launched http://www.Delphi3.com, offering a drawing to win a copy of Delphi 3, Client/Server. In addition, publishers such as Coriolis Group are giving away free books and magazine subscriptions.
The www.Delphi3.com Web site features Delphi resources, including a Delphi Web Index (a Delphi-specific search engine) and Delphi programming articles with source code.
The site also offers a Delphi utility archive and other contests.

Borland Completes Equity Financing; Raises US$25 Million

Scotts Valley, CA — Borland reported the closing of the first round of a privately placed equity financing arranged by the Promethean Investment Group, LLC of New York City. Borland raised approximately US$25 million, net of issuance costs, through the sale of a newly created class of Series B Convertible Preferred Stock ("Series B Shares") and warrants. Borland is subject to certain conditions, and may call for a second round of financing for up to an additional US$25 million.

In the initial closing of US$25 million, Borland issued 495 Series B Shares and warrants to purchase up to 198,000 shares of the company's common stock. Borland is obligated to issue an additional 55 Series B Shares and warrants for the purchase of an additional 22,000 shares of common stock. Subject to certain conditions, including registration of the underlying shares of the common stock, Borland will receive an additional US$2.5 million from such additional issuance.

New IntraBuilder Client/Server 1.5 Ships

San Jose, CA — Borland announced IntraBuilder Client/Server 1.5. English, French, and German versions of IntraBuilder Client/Server 1.5 are available for visually building scalable, data-driven Web applications that integrate with existing database servers and legacy information systems, including Oracle, Sybase, Microsoft SQL Server, Informix, IBM DB2, and InterBase.

IntraBuilder Client/Server 1.5 combines visual development tools with an application server, allowing developers to build Web applications for enterprise-wide data sharing over intranets, extranets, and the Internet. The product supports the creation of thin-client applications that can run on a PC, Macintosh, or UNIX Web browser.

IntraBuilder employs a server-side implementation of JavaScript. It also supports most Internet standards, including HTML, HTTP, CGI, NSAPI, ISAPI, ActiveX components, and Java applets, and is compatible with Microsoft and Netscape Web browsers and servers.

Key enhancements to version 1.5 include enhanced manageability. IntraBuilder runs as an NT service for higher performance and enhanced security; customizable error messages give developers greater control for directing users; and the new BDE Administration tool makes it easier to create and maintain database connections.

Version 1.5 features HTTP transactions that are up to 25 times faster, as well as a shared database connection. It offers enhanced Netscape ONE support via LiveConnect for extending applications with Java; OLEnterprise for building distributed applications and integrating legacy systems; MIDAS Business ObjectBroker Development Server for 24x7 failover safety and integration with Borland's Multitier Distributed Application Services Suite; and a Data Migration Wizard for application scaling and rapidly moving data between database formats and servers.

IntraBuilder users can update native SQL Links drivers for Oracle, Sybase, Informix, IBM DB2, MS SQL Server and InterBase. This version also offers new native drivers for Microsoft Access and FoxPro.

A 30-day trial version of IntraBuilder Client/Server can be downloaded free-of-charge for evaluation from Borland Online at http://www.borland.com.

Client/Server edition is US$1,995. For more information or to place orders, call Borland at (800) 233-2444.

Borland Announces Borland C++Builder for IBM AS/400

Scotts Valley, CA — Borland has announced its Borland C++Builder/400 Client/Server Suite for IBM AS/400.

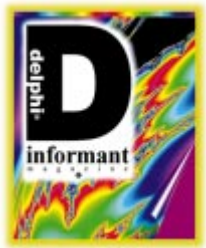
The C++Builder/400 Client/Server Suite is based on Borland's C++Builder Client/Server Suite for Windows 95 and Windows NT and ClientObjects/400. (The AS/400-compatible connectivity and development technology is licensed by Borland from TCIS of Paris, France.)

C++Builder/400 combines native connectivity to the AS/400, the C++ compiler, a reusable object-oriented component library, and visual design tools.

Delphi/400 Client/Server Suite is available now and C++Builder/400 Client/Server Suite is scheduled to be available later in 1997.

For more information, visit Borland at http://www.-borland.com/borland400/ or call (800) 233-2444.

table of CONTENTS



By Jeremy Rule

Delphi 3 DCOM

Comparing a Miata, a Hummer, and a Mack Truck

One of the more significant features in Delphi 3 is its native support for ActiveX technologies. Now Delphi programmers can write ActiveX controls, servers, and documents that are compatible at a binary level with servers written in compilers such as Visual Basic 5.

ActiveX, based on Microsoft's COM (Component Object Model) architecture, breaks applications into smaller, reusable components. Because COM provides internal support for version information, lifetime information, and standard data types, building a component can be easy. Then, with the components built, programmers (and sometimes users) can simply assemble applications from a stock of available components.

Taken one step further, DCOM (Distributed COM) involves writing an application using objects that could be running on another machine on the network. To understand *how* we can do this, it's important to understand *why* we would do this.

Multi-tier Architecture

Distributed multi-tier architecture presents an attractive solution to developers who need to maintain large-scale systems or divide time-consuming, specialized tasks among many powerful computers. Creating a multi-tiered system first involves defining standard interfaces for each logical level, or *tier*. The functions of the tier are then written as components that can be maintained and upgraded independently. One of the most popular uses of this architecture is with databases. Delphi includes ClientDataSet, RemoteServer, and Provider components specifically for multi-tiered database development. The database application can be broken into three or more logical tiers that handle the user interface, business rules, and persistent storage.

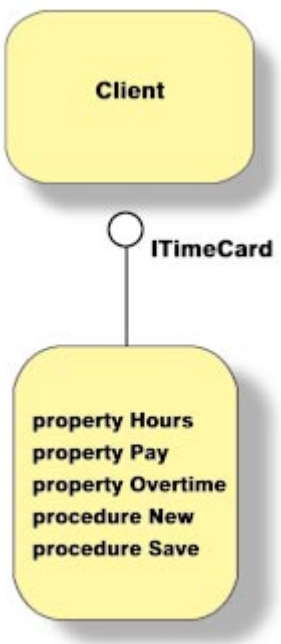


Figure 1: A *TimeCard* object implements an interface with *New* and *Save* procedures, and *Hours*, *Pay*, and *Overtime* properties. As long as the interface *ITimeCard* remains the same, clients can communicate with the TimeCard server.

In Figure 1, a simple *TimeCard* object implements an interface that contains the procedures *New* and *Save*, and the properties *Hours*, *Pay*, and *Overtime*. The "I" prefacing "TimeCard" is a naming convention that denotes an interface. As long as the interface *ITimeCard* remains the same, clients can communicate with the TimeCard server. The benefits of this model are realized when the business logic changes.

Let's say a *Set_Hours* method is invoked whenever the *Hours* property is changed. In the *Set_Hours* method, *Overtime* is calculated to be 1.5 times the *Hours* over 40. Administration decides *Overtime* should be calculated as 2.0 times the hours over 40. Instead of re-deploying client applications to the users, or changing a stored procedure in the database, the TimeCard server could be altered. In an over-simplified example such as this, changing a stored procedure might be more reasonable, but when the business rules become exceedingly complex and when the servers need to communicate with other servers, the multi-tiered approach begins to make sense.

Another benefit of DCOM is the ability to write distributed systems. COM objects can be deployed on the same machine as the client, or on any machine on the network. No changes need to be made to the server or client. This means that powerful workstations running COM servers can perform the complex

calculations in parallel. In Figure 2 for example, a client application uses four workstations to render a ray-traced scene. The workstations are all exposing the *IRender* interface with a *Render* function, which takes the scene description and a block of lines to render, then returns a partial image that can later be concatenated by the client to form a complete image. Render time will be cut significantly on a large and complex scene, because all four computers are doing the work.

To make this system more robust, one could add a RenderManager server whose only function is to divide the image into equal parts, and distribute the task among the least-busy machines. The client would only have to communicate with the RenderManager, and not care which machines were available for work. Of course, rendering an image is just one example of what distributed computing with DCOM could be used for.

To further elaborate using DCOM, we'll create several example ActiveX servers. We'll create an *ICar* interface and implement it in three ActiveX servers. The first server will be a .DLL, the second an .EXE, and the third an .EXE that will be put on a remote machine. Last, we'll write a test program to benchmark calls to all three servers.

The Car Type Library

The first step will be to create a Car type library that defines the *ICar* interface. Type libraries have no implementation details, only interface definitions. From Delphi 3's menu, select **File | New | ActiveX | Type Library**. Fill in the *Name* property with `Car`; then note the *GUID* property that's been completed for you. The GUID, a 128-bit Globally Unique Identifier, is generated by an algorithm that guarantees uniqueness (i.e. world-wide past, present, and future). Pressing the **Register** button will register *ICar* under the unique GUID, and you can have peace of mind that no other software installed will clobber your server.

Press the **Interface** button to create a new interface. Name it `ICar`. Press the **Property** button and add a "Name" property to *ICar*. In the declaration attribute of *Name*, change *Integer* to *WideString*. Press the **Method** button to add a method to *ICar*. Call it `Drive`.

When you're done, the type library should look like that in Figure 3. Save the type library as Car.tlb, and press the **Register** button to register the class.

Miata: An In-Process Server

An in-process server is an ActiveX server that runs in the same process space as the client application. Function calls execute quickly, because function parameters don't have to be packed, sent, and unpacked over process boundaries, or even machine boundaries.

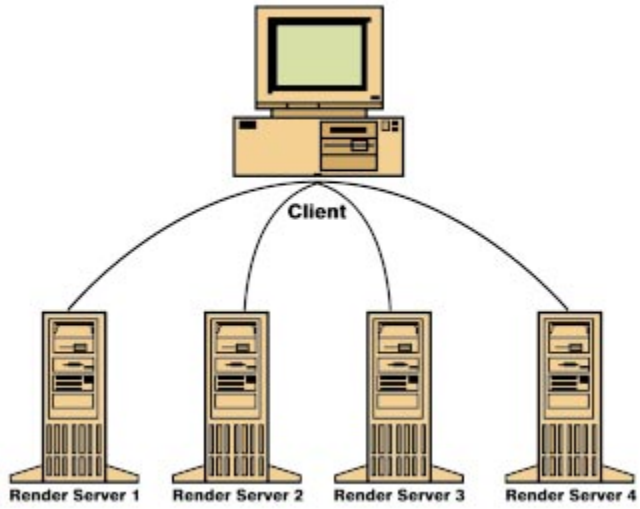


Figure 2: One benefit of DCOM is the ability to write distributed systems, allowing multiple PCs to perform functions in parallel. In this example, four machines are rendering a ray-traced scene.

To create the Miata server, select **File | New | ActiveX | ActiveX Library**. Save the project as MiataLibrary.dpr. Now that you have a library to host the server, select **File | New | ActiveX | Automation Object**. When Delphi prompts you for a class name, use `Miata`. Leave the instancing set to *Multiple Instance*. Close the Type Library editor that will display automatically. Save this unit as Miata.pas.

From the **Project** menu, click **Import Type Library** and choose **Car Library (Version 1.0)**. Complete the *TMiata* declaration:

```
type
  TMiata = class(TAutoObject, IMiata, ICar)
  private
    FName : WideString;
  protected
    function Get_Name: WideString; safecall;
    procedure Set_Name(const Value: WideString); safecall;
    procedure Drive; safecall;
    property Name: WideString read Get_Name write Set_Name;
  end;
```

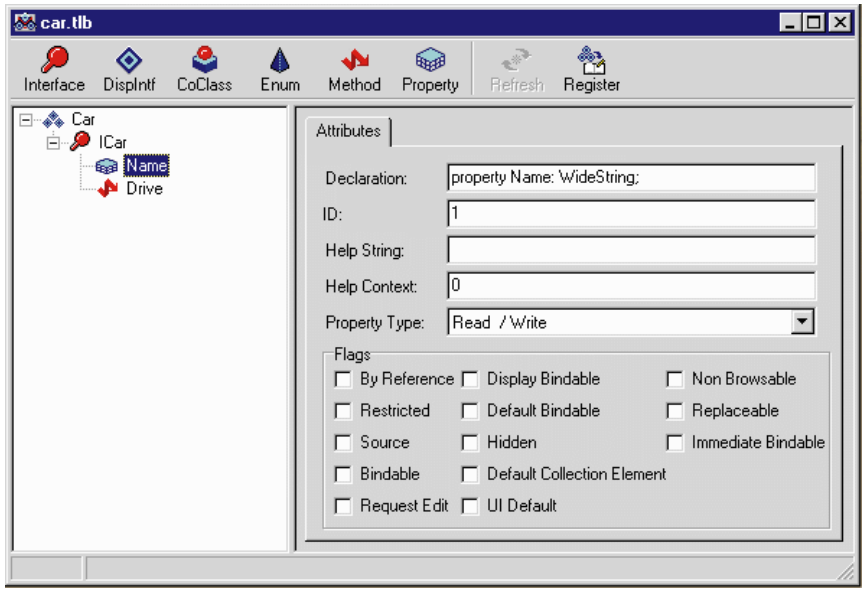


Figure 3: The Car.tlb type library.

By adding *ICar* to the class declaration, we are obligated to implement the whole *ICar* interface. *ICar* should be thought of as a contract with a *Name* property and a *Drive* method. By providing functionality for the whole interface, we have fulfilled the contract. *strName* is a private variable to keep track of the *Name* property.

Now complete the body of each function:

```
function TMiata.Get_Name: WideString;
begin
  Result := FName;
end;

procedure TMiata.Set_Name(const Value: WideString);
begin
  FName := Value;
end;

procedure TMiata.Drive;
begin
  ShowMessage('Driving a Miata');
end;
```

Normally we wouldn't want to put user-interface code in a server; if the server were ever moved to another machine, no one would be available to respond to the dialog boxes. However, for illustration and debugging purposes, a user interface is sometimes needed.

Make sure Dialogs and Car_TLB have been added to the **uses** statement, and compile the project (see Listing One on page 9). Finally, select **Register ActiveX Server** from the **Run** menu to make the Miata server available to other applications.

Hummer: An Out-of-Process Server

The second server, Hummer, will be an out-of-process server. Out-of-process servers (.EXEs) have the advantage of running in their own security context and will not crash if their clients go down. Clients and out-of-process servers can still communicate even if they are of different "bit-ness;" 16-bit, 32-bit, or (theoretically) 64-bit clients and servers should be able to communicate. The major disadvantage of an out-of-process server, as we'll see later, is that it must communicate with the client over a process boundary that makes method calls considerably slower.

The first step in making the Hummer server is to create a new application and hide the main form. Open the project source and add:

```
Application.ShowMainForm := False;
```

before the *Application.Initialize* statement. Now we have a formless project. Save the project as HummerServer.dpr. Add automation support with **File | New | ActiveX | Automation Object**. Delphi will prompt you for a class name (choose `Hummer`) and automatically display the Type Library editor. Close the Type Library editor and import the Car type library the same way we did with the Miata server.

Implement the *ICar* interface in the *THummer* class:

```
type
  THummer = class(TAutoObject, IHummer, ICar)
    FName : WideString;
  protected
    function Get_Name: WideString; safecall;
    procedure Set_Name(const Value: WideString); safecall;
    procedure Drive; safecall;
    property Name: WideString read Get_Name write Set_Name;
  end;
```

and each function:

```
function THummer.Get_Name: WideString;
begin
  Result := FName;
end;

procedure THummer.Set_Name(const Value: WideString);
begin
  FName := Value;
end;

procedure THummer.Drive;
begin
  ShowMessage('Driving a hummer');
end;
```

Save the unit as Hummer.pas and compile. Again, don't forget to add Dialogs and Car_TLB to the **uses** statement (see Listing Two on page 9). To register the Hummer server for use with other projects, run the project with a /REGSERVER parameter. All out-of-process servers implement /REGSERVER and /UNREGSERVER parameters behind the scenes.

You now have an Automation server that will dispense Hummer objects.

MackTruck: A Remote Server

The final server, MackTruck, will be an out-of-process server that runs on another machine. Developing the remote server will be no different than developing a local server. This is important! Later, when you're tweaking the system for performance, you can move servers to different locations without redesign. The physical layout should have no bearing on the logical design.

Remote servers have similar advantages and disadvantages to out-of-process servers. Method parameters must be packed and sent from a proxy on the local machine to a stub on the remote machine. Network constraints can also hurt the performance of a remote server. However, remote servers add a physical layer of abstraction between client and server. A new server needs only to be distributed to one spot, rather than to each client machine. Remote servers also let you distribute the workload over many powerful computers (as described earlier).

Writing the MackTruck server is similar to writing the Hummer server. Start a new project, hide Form1, and import the Car type library. Add automation support with **File | New | ActiveX | Automation Object**, and call the server `MackTruck`.

Close the Type Library editor and have the *TMackTruck* class implement the *ICar* interface:

```
type
  TMackTruck = class(TAutoObject, IMackTruck, ICar)
  private
    FName : WideString;
  protected
    function Get_Name: WideString; safecall;
    procedure Set_Name(const Value: WideString); safecall;
    procedure Drive; safecall;
    property Name: WideString read Get_Name write Set_Name;
  end;
```

Complete the methods:

```
function TMackTruck.Get_Name: WideString;
begin
  Result := FName;
end;

procedure TMackTruck.Set_Name(const Value: WideString);
begin
  FName := Value;
end;

procedure TMackTruck.Drive;
begin
  ShowMessage('Driving a Mack Truck');
end;
```

Save the project as MackServer.dpr, and the unit as MackTruck.pas (see Listing Three on page 10). Compile and run with the /REGSERVER switch. Now we have a MackTruck server set up to run locally.

There are two ways to make the server remote. The first is to use Delphi's *CreateRemoteComObject* function. The second, a more visual way, is to use Dcomcnfg.exe, which ships with Windows NT, or can be downloaded with the DCOM for Windows 95 SDK available at http://www.microsoft.com/oledev.

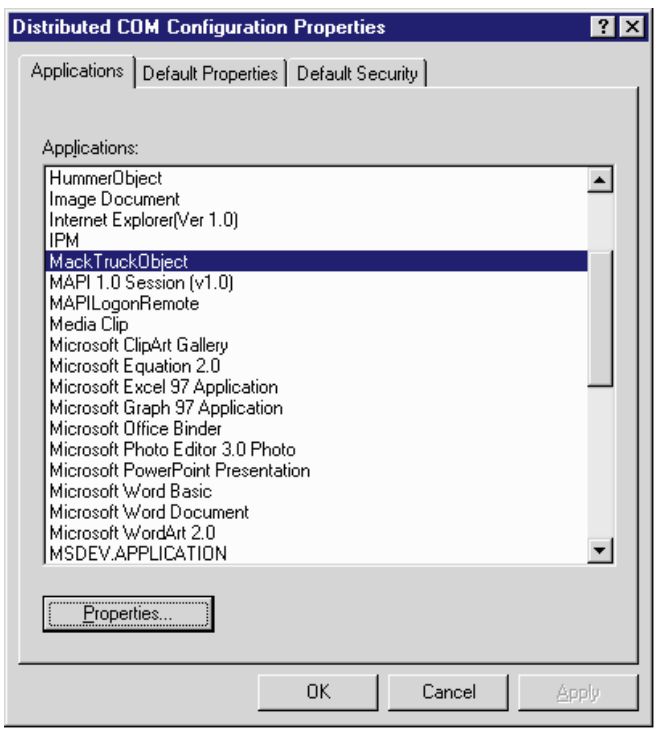


Figure 4: The **MackTruckObject** and **HummerObject** applications now appear in the list.

For the purposes of this example, we'll use Dcomcnfg.exe to make the MackTruck server remote. Start Dcomcnfg from the command line, or with **Start | Run**. In the list of applications, you should see the **MackTruckObject** and **HummerObject** applications available (see Figure 4). Select **MackTruckObject** and use the Properties dialog box to specify on which computer you would like the object to be created. You can either specify a UNC name, an IP address, or a host name.

Next, copy the MackServer.exe file to the remote machine and run "MackServer.exe /REGSERVER" on the remote machine. This registers the MackTruckObject as available for automation. Run Dcomcnfg.exe on the remote machine and verify that MackTruckObject is present. Within the properties pages for MackTruckObject you can specify security settings, including who can create objects. The Identity tab lets you specify which account runs the server. Make sure the local user has an account on the remote machine if you have **The launching user** chosen. Copy car.tlb to the remote machine and use the Type Library editor to register the type library. If the Type Library editor is not available on the remote machine, you can build the Car type information into a DLL and move it to the remote machine. Then, use REGSVR32.EXE (which ships with Windows NT and Windows 95) to register the type library.

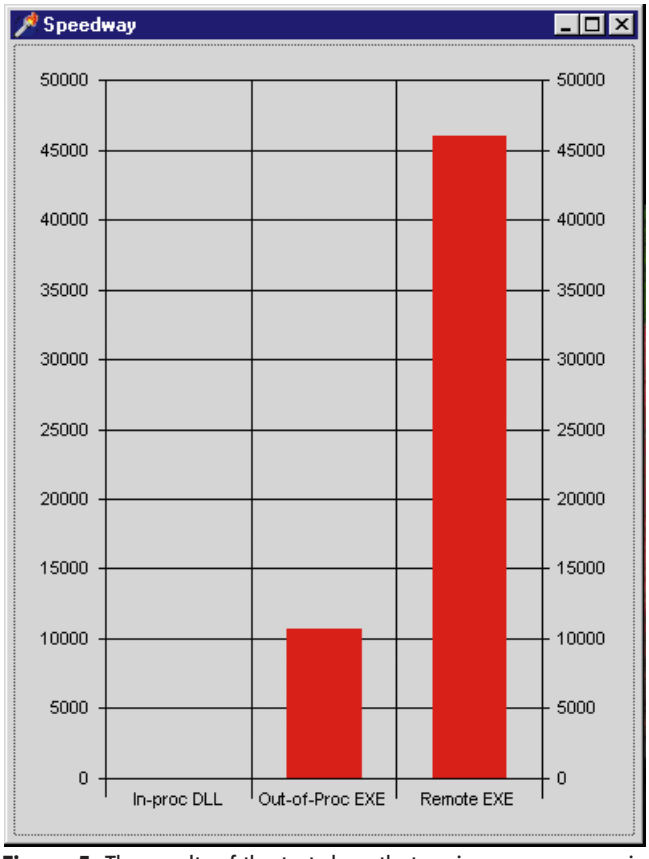


Figure 5: The results of the test show that an in-process server is much faster than an out-of-process server, which is, in turn, quicker than the remote server.

Moving back to the local machine, the final step in this project is to create a test program that will use all three car servers. The test application will simply create a car of each type, set its name property, and read the name property 2,000 times. By doing this we can see the performance differences between in-process, out-of-process, and remote servers, then graph the results.

Speedway: Testing Each Car

Create a new application, save it as SpeedWay.dpr, and save the main form as Main.pas. From the **Component** menu, use **Import ActiveX Control** to import a Microsoft Chart Control. (Note: *TChart* will work similarly, but in the spirit of ActiveX, this example uses the Microsoft Chart Control). Drop an MSChart component on the main form and name it `Chart`. To import the types, add Car_TLB, Miatalibrary_TLB, Hummerserver_TLB, and Mackserver_TLB to the **uses** statement of Main.pas.

Write a *TestCar* function, as shown in Listing Four on page10, to time the speeds of each car and graph the results. Notice that *TestCar* shows evidence of COM's polymorphism because it receives *ICar* instead of a specific car type. If the car's *Drive* procedure were called, the car object would show the appropriate dialog box.

The results of the test (see Figure 5) show that an in-process server is much faster than an out-of-process server, which is, in turn, quicker than the remote server. The bottom line: The Miata is fast and lean, but if you're going to be in an accident, you might want to pick a larger, heavier car. This is similar to a .DLL, which offers the quickest server, but will crash along with any clients it's serving. The Hummer is secure and robust, but considerably slower — much like an .EXE. The remote .EXE is like a Mack truck; it can do a lot of work over a long distance.

Conclusion

Although the example shown here is trivial, there are many situations where Delphi and DCOM will prove useful. Distributed and multi-tiered systems are becoming more popular, and Delphi 3 offers excellent support for DCOM. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\SEP\DI9709JR.

Jeremy Rule lives in Houston, TX with his wife and two parrots. He works for Enron Capital and Trade writing multi-tiered systems. Jeremy can be reached at (713) 853-3501, or via e-mail at jrule@ect.enron.com.

Begin Listing One — Miata.pas

```
unit Miata;

interface

uses
  ComObj, MiataLibrary_TLB, Car_TLB, Dialogs;

type
  TMiata = class(TAutoObject, IMiata, ICar)
  private
    FName: WideString;

 protected
    function Get_Name: WideString; safecall;
    procedure Set_Name(const Value: WideString); safecall;
    procedure Drive; safecall;
    property Name: WideString read Get_Name write Set_Name;
  end;

implementation

uses ComServ;

function TMiata.Get_Name: WideString;
begin
  Result := FName;
end;

procedure TMiata.Set_Name(const Value: WideString);
begin
  FName := Value;
end;

procedure TMiata.Drive;
begin
  ShowMessage('Driving a Miata');
end;

initialization
  TAutoObjectFactory.Create(ComServer, TMiata,
                            Class_Miata, ciMultiInstance);
end.
```

End Listing One

Begin Listing Two — Hummer.pas

```
unit Hummer;

interface

uses
  ComObj, HummerServer_TLB, Car_TLB, Dialogs;

type
  THummer = class(TAutoObject, IHummer, ICar)
    FName: WideString;
  protected
    function Get_Name: WideString; safecall;
    procedure Set_Name(const Value: WideString); safecall;
    procedure Drive; safecall;
    property Name: WideString read Get_Name write Set_Name;
  end;

implementation

uses ComServ;

function THummer.Get_Name: WideString;
begin
  Result := FName;
end;

procedure THummer.Set_Name(const Value: WideString);
begin
  FName := Value;
end;

procedure THummer.Drive;
begin
  ShowMessage('Driving a hummer');
end;

initialization
  TAutoObjectFactory.Create(ComServer, THummer,
                            Class_Hummer, ciMultiInstance);
end.
```

End Listing Two

Begin Listing Three — MackTruck.pas

```
unit MackTruck;


interface

uses
  ComObj, Dialogs, MackServer_TLB, Car_TLB;
type
  TMackTruck = class(TAutoObject, IMackTruck, ICar)
  private
    FName: WideString;
  protected
    function Get_Name: WideString; safecall;
    procedure Set_Name(const Value: WideString); safecall;
    procedure Drive; safecall;
    property Name: WideString read Get_Name write Set_Name;
  end;

implementation

uses ComServ;

function TMackTruck.Get_Name: WideString;
begin
  Result := FName;
end;

procedure TMackTruck.Set_Name(const Value: WideString);
begin
  FName := Value;
end;

procedure TMackTruck.Drive;
begin
  ShowMessage('Driving a Mack Truck');
end;

initialization
  TAutoObjectFactory.Create(
    ComServer,TMackTruck,Class_MackTruck,ciMultiInstance);
end.
```

End Listing Three

Begin Listing Four — Main.pas

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, OleCtrls, Car_TLB, Miatalibrary_TLB,
  Hummerserver_TLB, Mackserver_TLB, MSChartLib_TLB;

const
  ROUNDTRIPS = 3000;

type
  TfrmMain = class(TForm)
    Chart: TMSChart;
    procedure FormShow(Sender: TObject);
  private
    FMiata: Miata;
    FHummer: Hummer;
    FMackTruck: MackTruck;
    function TestCar(car: ICar; iTrips: Integer): Real;
  end;

var
  frmMain: TfrmMain;

implementation

{$R *.DFM}

procedure TfrmMain.FormShow(Sender: TObject);
var
  MiataTime,
  HummerTime,
  MackTime: Real;
begin

  // Create the three cars.
  FMiata := coMiata.Create;
  FHummer := coHummer.Create;
  FMackTruck := coMackTruck.Create;

  // Test each one.
  MiataTime := TestCar(FMiata as ICar, ROUNDTRIPS);
  HummerTime := TestCar(FHummer as ICar, ROUNDTRIPS);
  MackTime := TestCar(FMackTruck as ICar, ROUNDTRIPS);

  // Chart the results.
  with Chart do begin
    Row := 1;
    Data := '1';
    Row := 2;
    Data := FloatToStr(HummerTime / MiataTime);
    Row := 3;
    Data := FloatToStr(MackTime / MiataTime);
  end;

end;

function TfrmMain.TestCar(car: ICar; iTrips: Integer): Real;
var
  i: Integer;
  nStart: Real;
  sName: string;
begin
  nStart := Now;
  for i := 1 to iTrips do
    sName := car.Name;
  Result := Now - nStart;
end;

end.
```

End Listing Four

By Ray Lischner

The Expert Tool Kit

A Collection of Components and Experts to Help You Create Experts and Wizards

The Open Tools API lets you add your own extensions to Delphi, e.g. insert new menu items in Delphi's menu bar, define form and project wizards for creating new modules and applications, or write add-in experts to do just about anything. The Open Tools classes can be tricky to use, however. You must define all your classes correctly, create and free objects at the appropriate times, and clean up properly when Delphi shuts down or unloads an expert.

Make a mistake, and you can crash Delphi.

Writing an Open Tools expert would be easier if you could take advantage of Delphi's component model, i.e. create components that help you write your own experts. The Expert Tool Kit does just this — it's a collection of components and experts to help you write new experts and wizards. This article outlines how the Expert Tool Kit works, and how you can use it for writing experts.

Introducing Experts

Before jumping into the Expert Tool Kit, let's take a minute to review the Open Tools API. Open Tools is a set of classes for extending Delphi's integrated development environment (IDE). You can add items to Delphi's menu bar, define experts to create new forms, units, and projects, edit the files in a project, edit component properties, or modify a project's resources. Because the Open Tools API is a set of Delphi classes, you can write your extensions in Delphi — you don't need to learn a separate scripting language (as is the case with many other development products).

One drawback of the Open Tools API is its complete lack of documentation — all that ships with Delphi is a set of source files that declare interface classes. These are found in Delphi's \Source\Toolsapi directory. The comments are mostly correct, and can tell you much about the Open Tools API. Start with ExptIntf.pas, which declares the expert interface class. ToolIntf.pas declares *TIToolServices*, which is the primary link between an expert and Delphi's IDE. EditIntf.pas declares several interface classes for accessing information about source files, forms, and resources.

Every Open Tools expert must declare an expert interface class, inheriting from *TIExpert*. This class tells Delphi what kind of expert you are creating, the expert's name, an icon to use in the Object Repository, and other similar information. When you derive your class from *TIExpert*, make sure you override every method.

When the user invokes an expert, Delphi usually calls the expert's *Execute* method, which can create a new form or project; open, close, and save files; edit a source file; create components; modify a component's properties; and so on. The Open Tools API defines several interface classes that give your expert access to a project and its forms, components, source files, and resources.

Writing an expert at the level of *TIExpert* is roughly akin to writing a Windows application by using the Win32 API. Delphi's component model makes your job easier when you create Windows applications, just as the Expert Tool Kit makes it easier to write experts and wizards.

For example, to create a new unit or form, you must use a module creator or the undocumented Proxies unit. To use a module creator, you must derive a class from *TIModuleCreator* and override all its methods. Most module creators have several one-line methods, often returning an empty string. This can be tedious, and is error-prone if you forget to override one method.

The Expert Tool Kit simplifies module creation with its *TEtkModuleCreator* component. Instead of overriding abstract methods, you set the component's properties and event handlers. The component supplies suitable default values for its properties, and you can easily set new property values. This component is just one way the Expert Tool Kit helps you write experts and wizards. Let's take a look at the whole package to understand how the Expert Tool Kit works.

Using the Expert Tool Kit

To install the Expert Tool Kit, run ETK10.EXE (see the end of this article for download details). This set-up program installs the packages and source files into a directory of your choice. It automatically installs the wizards and components and adjusts Delphi's search path to include the Expert Tool Kit files. The next time you run Delphi, you'll be able to use the Expert Tool Kit.

The first step for using the Expert Tool Kit is simple. From Delphi's menu bar, choose **File | New**, select the Projects page, then double-click the Expert DLL Wizard icon. The wizard inquires what style of expert you want to create: Add-in, Form, Project, or Standard. Then it creates a new DLL project, with an expert .DPR file and an expert module. An expert module represents a single expert, and its properties correspond to the methods of *TIExpert*. When you use the Expert Tool Kit, you no longer need to concern yourself with the *TIExpert* class. Simply create an expert module and set its properties. The Expert DLL creates the expert modules and automatically registers the experts for you.

Expert DLL

The Expert DLL Wizard creates a library project (.DPR) file. This library exports a library initialization function, which Delphi requires of every expert library. The main body of the library file automatically creates the expert modules, so the DLL can register the experts in its initialization function.

Ordinarily, Delphi reserves the automatic creation of forms and data modules for applications (.EXE projects). By adding the statement:

```
Application.Run;
```

```
library LibWiz;

uses
  ShareMem,
  ExptIntf,
  EtkInit,
  NewLib in 'NewLib.pas' { NewLibraryWizard: TEtkModule },
  NewUnit in 'NewUnit.pas' { NewUnitWizard: TEtkModule };

{$R *.RES}

exports InitializeExperts name ExpertEntryPoint;

begin
  Application.CreateForm(TNewLibraryWizard,
                         NewLibraryWizard);
  Application.CreateForm(TNewUnitWizard, NewUnitWizard);
  Application.Run;
end.
```

Figure 1: Source code for the Expert DLL.

to the main library file, however, the Expert DLL Wizard tells Delphi that that library will create forms and data modules automatically. When you add a new form or data module (such as an expert module) to the project, Delphi's IDE adds them to the library's main body of code the same way it does for applications. Figure 1 shows what the Expert DLL's source code looks like.

The Expert DLL project treats the *Application* object somewhat differently from a real application. In an expert, the *Application* object monitors the expert modules that your project automatically creates. When Delphi loads and registers the DLL, it calls a library initialization function, which iterates through the list of experts in the *Application* object and registers all the expert modules' experts. To prevent this use of *Application* from interfering with your experts, the EtkInit unit declares *Application* this way. Your expert never needs to use EtkInit, so you can use *Application* normally in the rest of your expert.

Note that only a library can create expert modules automatically. If you use a package for your expert, you will need to create a *Register* procedure that calls each expert module's *CreateAndRegister* method. This class method creates the expert module, assigns the reference to a variable, and registers the expert. Following is an example of calling *CreateAndRegister*:

```
procedure Register;
begin
  TEtkModule1.CreateAndRegister(EtkModule1);
end;
```

Expert Module

The Expert DLL Wizard automatically creates an expert module, and you can add any number of additional expert modules. Each expert module is a single expert. By using expert modules, you do not need to concern yourself with the *TIExpert* class. Instead, simply set properties of the expert module, and the Expert Tool Kit takes care of the rest.

To create a new expert module, choose **File | New**; then on the Data Modules page, double-click **Expert Module**. The Expert Module Wizard asks what style of expert you want to create, then creates a new expert module of that style, with suitable property values.

The expert style stipulates how the user invokes the expert: The Object Repository lists form and project experts in the New Items dialog box. A project expert typically creates a new application or library project; a form expert creates a form, unit, or other file. Delphi lists standard experts in its **Help** menu. An add-in expert has no predefined user interface. Instead, it is up to you to decide how the user interacts with an add-in expert.

After you have chosen an expert style, the Expert Tool Kit creates a blank expert module. You can set any of the expert module's properties in the Object Inspector. For example, if you want a standard expert, Delphi needs to know the menu caption for the menu item it will add to its **Help** menu. You can also set the state of this menu item to be enabled or disabled, and to have an optional check mark. A form or project expert needs an icon, comment, and author name to list in the Object Repository. The Expert Module Wizard chooses default values for these properties, such as a default icon for project and form experts, but you will want to change some of them to suit your specific needs.

Every expert needs a name, which is the name the user sees for the expert, and an ID string, which is a unique identifier. The user never sees the ID string, so you can sacrifice readability for guaranteeing uniqueness. By convention, an ID string has the form "author.name". The expert module chooses defaults for all of these properties, so change only those you feel you must.

The expert module automatically takes care of defining the expert interface and registering the expert with Delphi. When Delphi unloads the expert, it automatically frees the expert module, which frees all of its components. The automatic creation of the expert module works only in an expert project, that is, a DLL that you create with the Expert DLL Wizard.

The expert module is a custom data module, which is a new feature in Delphi 3. A custom form or data module can publish properties that are visible in the Object Inspector. Use the expert module the same way you would a data module. For example, you can drop any non-visual component on the expert module. The most useful components are those in the Expert Tool Kit, which you can find on the Experts tab of the Component palette.

Components

The Expert Tool Kit comes with three components: *TEtkMenuItem*, *TEtkModuleCreator*, and *TEtkProjectCreator*. The first adds a menu item to Delphi's menu bar. The next two create new modules and new projects. Let's take a closer look at each of these components.

The expert menu-item component, *TEtkMenuItem*, creates a menu item in Delphi's menu bar. Its properties closely match the properties of an ordinary menu item. The difference is that an expert adds a menu item to Delphi's menu bar, and you must tell it where the new menu item belongs. Set the *InsertName* property to the name of an item that is already in the menu bar, and set *InsertAction* to say whether the new item goes before or after the target item, or whether to add the new item in a submenu (*iaChild*).

The Expert Tool Kit defines a property editor for the *InsertName* property. The property editor defines a dialog box where you can easily choose a menu item from a facsimile of Delphi's menu bar. Set any of the other properties to set the menu item's flags. Set the component's *OnClick* event handler to perform your expert's work. You can add menu items to any expert, as well as any number of menu items.

The Expert Tool Kit includes two other components: a module creator and a project creator. You can use these components to create a new unit or form, or to create a new project. [For more information about module and project creators, refer to Lischner's article "What's New with Experts?" in the July 1997 issue of *Delphi Informant*.] Using these components is much easier than writing your own creator classes.

The *TEtkProjectCreator* component creates a new Delphi project. You can create a default application or library, i.e. the same project that Delphi creates when you choose **Application** or **DLL** from the New Items dialog box. To create a default application, set the project creator's *Flags* property so that *cpApplication* is *True*, set *FileName* and *FileSystem* to empty strings, and set *Existing* to *False*. For the default library project, set *cpLibrary* to *True*. You'll probably want to set *cpCanShowSource* to *True* so Delphi will show the project's source file.

If the project files exist, set *Existing* to *True*. Otherwise, the value of *NewProjectSource* is the source code for the new .DPR file. The project creator uses this string to call *Format*, passing the project name as the sole argument. Thus, you will usually start the source code string with `program %s;` or `library %s;`. The default value for this property is the source code for a default application.

When Delphi creates the new project, it calls back to the project creator. The Expert Tool Kit invokes corresponding event handlers in the creator component. After creating the project's source file, the Expert Tool Kit calls the *OnDefaultModule* handler, which you can set to create new modules (units or forms) for your project. In the *OnProjectResource* event handler, you can add resources to your project. If you do not set this event handler, Delphi will supply its default MAINICON resource. Finally, the Expert Tool Kit calls the *OnModuleCreated* event handler. If you want to do anything special with the new project, you can use the module interface. See Delphi's EditIntf.pas file for more information about module interfaces.

The *TEtkModuleCreator* component is similar to *TEtkProjectCreator*, except that it creates a single file, possibly adding it to the current project. The module creator has many more flags than the project creator. Read Delphi's ToolIntf.pas file for a description of the *TCreateModuleFlags* choices.

The default value for the *NewModuleSource* property is the source code for a new form. If you want the same source code that Delphi uses when you choose **File | New Form**, you can leave this property alone. If you want to create a unit without a form, remove the line `{$R *.DFM}` from the file.

As with the project creator, the module creator passes the *NewModuleSource* string to *Format*. The three arguments passed to *Format* are the unit name, form name, and ancestor name. In other words, `%0:s` is the unit name, `%1:s` is the form name, and `%2:s` is the ancestor name. To obtain the form type, use `T%1:s`. The ancestor of an ordinary form is `Form`.

Set the other properties of the module creator according to your needs. The default property values usually work quite well. If you want to create a data module, set the *Ancestor* property to *DataModule*. To use form inheritance, set *Ancestor* to the name (not the type) of the ancestor form. When your expert creates the new module, the ancestor form must be open. (See ToolIntf.pas to learn how to open a file in the IDE.) To create a module, call the module creator's *ModuleCreate* method. If the source code contains a .DFM directive, Delphi will call the *OnFormCreated* event handler. You can add components to the form, using the form and component interfaces (which are declared in EditIntf.pas). After Delphi has created the new unit and form, the module creator calls its *OnModuleCreated* event handler, which you can use to save the new module (or do anything else with it).

Road Test

The best way to learn how to use the Expert Tool Kit is to try it. Create an expert DLL, and choose the Standard style. Set the *MenuText* property to something suitable, such as `Testing...`. Set the *OnExecute* event handler to show a simple message:

```
procedure TEtkModule1.EtkModule1Execute(Sender: TObject);
begin
  ShowMessage('This expert works!')
end;
```

Now save and compile your new project. Exit Delphi and run REGEDIT to register your expert. Create a new string entry under HKEY_CURRENT_USER\Software\Borland\-Delphi\3.0\Experts. Use any unique identifier, such as your expert's ID string, for the new entry name. Set the entry value to the complete path to your DLL. Run Delphi again, and you'll see a new menu item under Delphi's **Help** menu. Choose this menu item and Delphi will run your expert, showing you the message.

Now it's time to take a look under the hood and see how the Expert Tool Kit works.

```
unit EtkProject;
{ Expert tool kit. Expert Project expert. This expert
  module creates a library project for a Delphi 3
  expert. It primes the expert with an expert module. }

interface

uses Windows, SysUtils, Classes, Graphics, Dialogs, Forms,
  ExptIntf, ToolIntf, EditIntf, Etk;

type
  TEtkProjectWizard = class(TEtkModule)
    ProjectCreator: TEtkProjectCreator;
    ModuleCreator: TEtkModuleCreator;
    procedure EtkProjectWizardExecute(Sender: TObject);
    procedure ProjectCreatorDefaultModule(Sender: TObject);
    procedure ModuleCreatorFormCreated(Sender: TObject;
      Form: TIFormInterface);
  private
    { private declarations }
    Style: TExpertStyle;
  public
    { public declarations }
  end;

var
  EtkProjectWizard: TEtkProjectWizard;

implementation

uses EtkModule, EtkStyleChooser;

{$R *.DFM}

procedure TEtkProjectWizard.EtkProjectWizardExecute(
  Sender: TObject);
begin
  if ChooseExpertStyle(Style) then
    ProjectCreator.ProjectCreate
end;

procedure TEtkProjectWizard.ProjectCreatorDefaultModule(
  Sender: TObject);
begin
  ModuleCreator.ModuleCreate
end;

procedure TEtkProjectWizard.ModuleCreatorFormCreated(
  Sender: TObject; Form: TIFormInterface);
begin
  InitializeForm(Form, Style);
end;

end.
```

Figure 2: Creating an expert is this easy.

Inside the Expert Tool Kit

The Expert Tool Kit is the easiest way to create experts in Delphi 3. Therefore, you should not be surprised to learn that I used the Expert Tool Kit to create itself. (Strictly speaking, the Expert Tool Kit evolved; for each iteration, I used the earlier version to write the next generation.)

The expert module is a custom data module. The data module publishes properties that correspond to the methods of *TIExpert*. The expert module works with a helper class, *THelperExpert*, which inherits from *TIExpert* and communicates with an expert module to obtain the information it needs. Each expert module creates and registers a separate instance of *THelperExpert*. Each *THelperExpert* object has an *Owner* prop-

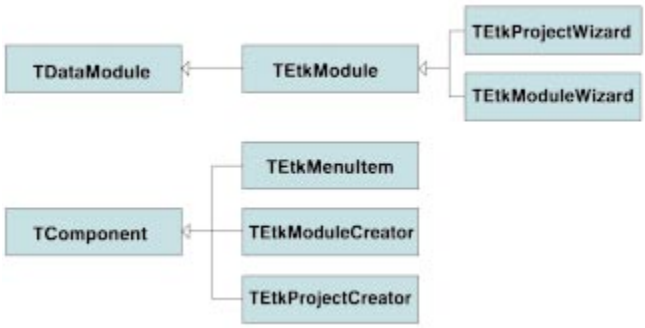


Figure 3: The organization of experts and components in the Expert Tool Kit.

erty that refers back to the expert module that created it. When Delphi unloads the expert, the helper expert releases the expert module. When Delphi shuts down, it frees the module first, so the expert module clears the *Owner* property in the helper expert (so the expert doesn't try to free the module twice).

A form expert (Expert Module Wizard) creates the custom data module. Of course, this wizard uses an expert module to create its output, which is a new expert module. The wizard uses a module creator component to create the new expert module and its source code. The EtkModule.pas file defines the Expert Module Wizard. Most of the work in this file is setting the default properties of the new module according to the expert style.

The Expert DLL Wizard also uses the Expert Tool Kit. The Expert DLL Wizard is a project expert that creates the main .DPR file (using a project creator component) and creates an expert module (using a module creator component). EtkProject.pas declares the Expert DLL Wizard. This was an easy expert to write: I created an expert module with style *esProject*. I then added a project creator and a module creator to the expert module, defined the *NewProjectSource* property, set a couple event handlers, and *voilà* — instant expert! Using an expert module is similar to using a form; most of your work is simply filling in the bodies of event handlers, as you can see in Figure 2.

As simple as that was, it's easy to get lost in the maze of expert modules creating expert modules, so Figure 3 illustrates the organization of the experts and components in the Expert Tool Kit.

The Etk.pas file declares the components in the Expert Tool Kit, and the expert module class, TEtkModule. The EtkRun10.dpk package contains this unit, so you can easily write other packages that use these components. EtkCmp10.dpk is the main design-time package for the Expert Tool Kit. It contains EtkReg.pas, which is the main registration unit for the Expert Tool Kit. This unit registers the expert module as a custom module, registers some property editors, and creates and registers the expert modules.

The simplest way to create your own experts is to use the Expert DLL Wizard. If you prefer to use packages, you can create a registration unit, similar to EtkReg.pas, and create a package similar to EtkCmp10.dpk. Make sure your package has EtkRun10 in its requires list. Then simply compile and install your package, and you're ready to start using your new expert.

Conclusion

The Open Tools API makes it possible to write your own extensions to Delphi, but the Open Tools classes are not always easy to use. The Expert Tool Kit is a collection of wizards and components that make it easier to use the Open Tools API. The Expert DLL Wizard creates a new project for writing an expert. You can add any number of expert modules to the DLL, where each expert module defines a Delphi expert or wizard. The Expert Tool Kit also contains several components that help you add menu items to Delphi's menu bar, create new units or forms, and create new projects.

For more information about experts, wizards, and the Open Tools API, refer to *Hidden Paths of Delphi 3* [Informant Press, 1997]. This book provides in-depth coverage of the Open Tools API, so you can learn about module, form, and component interfaces, and more. Also visit the Open Tools Web site at http://www.tempest-sw.com/opentools/, where you can download the latest version of the Expert Tool Kit and get information about the Open Tools API. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\SEP\DI9709RL.

Ray Lischner is the author of *Hidden Paths of Delphi 3* [Informant Press, 1997], a recently published book that explores the Open Tools API in depth. You can use the code and components from this book to further enhance the functionality of the Expert Tool Kit. He also wrote *Secrets of Delphi 2* [Waite Group Press, 1996], a book that reveals undocumented features of Delphi 1 and Delphi 2. He is a contributor to several Delphi periodicals, and is a familiar figure on the Delphi Usenet newsgroups. Mr Lischner is the founder and president of Tempest Software, which specializes in consulting and training for object-oriented languages, components, and tools. He also teaches Computer Science at Oregon State University, and serves on the board of directors for the Pacific Northwest Software Quality Conference. You can contact Ray via e-mail at lisch@tempest-sw.com.

By Gregory Lee

MIME's the Word

Internet Delphi: Part III

In the first two articles of this series, we developed an application to send e-mail over the Internet with the Simple Mail Transfer Protocol (SMTP), and another to retrieve e-mail using the Post Office Protocol (POP). While these applications are certainly functional, they lack one of the most useful features of present-day e-mail programs: the ability to include file attachments along with the message text. In this installment, we'll focus on the set of protocol extensions that make this possible.

Although the sample programs that accompany this article contain quite a bit of code (to implement SMTP, POP3, and all the low-level socket calls), we won't review that information here. What we're concerned with now is something called Multipurpose Internet Mail Extensions, or MIME.

The Rules of MIME

The blueprint for MIME is laid out in RFC 1521 and RFC 1522, "MIME (Multipurpose Internet Mail Extensions)," by Borenstein, Bellcore, Freed, Innosoft, and Moore. RFC stands for Request For Comment. Virtually every Internet standard is documented somewhere in an RFC file. You can find this and other RFC documents at ftp://ds.internic.net/rfc. In short, these documents detail the addition of five new mail headers and the e-mail contents they describe:

- MIME-Version
- Content-Type
- Content-Transfer-Encoding
- Content-ID
- Content-Description

Content-Type	Description
text/plain	Unformatted text (CR/LF pairs only).
text/richtext	Text with simple formatting.
text/enriched	An updated version of rich text.
multipart/mixed	Multiple parts to be processed sequentially.
multipart/parallel	Multiple parts to be processed in parallel.
multipart/digest	A series of RFC 822 messages.
multipart/alternative	Multiple parts with similar content.
message/rfc822	A message following the RFC 822 standard.
message/partial	A message fragment.
message/external-body	A pointer to an external message.
application/octet-stream	Binary data.
application/postscript	Postscript data.
image/jpeg	A JPEG graphic.
image/gif	A GIF graphic.
audio/basic	Encoded audio data.
video/mpeg	MPEG encoded video data.

Figure 1: MIME Content-Types.

The MIME-Version header simply indicates which version of MIME the sending e-mail program supports. For our purposes, this is not particularly significant. We'll simply use version 1.0 on all messages generated by the SMTP implementation, and assume we can handle whatever version is received on the POP3 side. The second header, however, is more important.

Content-Type

The Content-Type header indicates the kind of information contained in the message. Possible values for this field are described in Figure 1. Because we want to add binary files as attachments to e-mail messages — as opposed to sending the files alone — we'll start by defining the Content-Type in all messages as `multipart/mixed`.

To mark the spots where the message text ends and a file attachment begins, we must also define something called an *encapsulation boundary*. That sounds a little scary, but it's just a string of text that we pre-define, so that the mail program can use it to identify the start and end of each message part. If you defined the boundary text with the string of characters `abc`, each message part would be preceded by a line of text containing two hyphens followed by the string `abc`.

Obviously, you want to use something a little more unique for the boundary string. That way, the odds of the boundary text popping up somewhere in the middle of a message text or a file attachment is extremely low. Because the nature of the boundary string is similar to that of the Message-ID, we can re-use the information from the Message-ID string in the boundary text, with the following code:

```
DateTimeToString(S,'mmmddyyhhnn',Now);
S := S + EmailFrom.Text;
BuildAndSend('Message-ID: <',S,'>',Unchanged);
BoundaryString := '= Multipart Boundary ' + S;
```

Content-Transfer-Encoding

Following each section boundary is a separate Content-Type header that indicates the kind of data its section contains. In addition, each section should contain a Content-Transfer-Encoding header to indicate how the data has been encoded. Possible values for this field are described in Figure 2.

Because we've already decided that our mail program will work only with standard English text and binary file attachments, we're really interested in just two of these types: 7bit for the text, and base64 for the attachments.

At this point, you may be wondering why we would even bother with base64 encoding. Clearly, there are two Content-Transfer-Encoding types for 8-bit data. While it's true there are two type indicators for raw binary data, there are currently no standardized e-mail transports to handle 8-bit information. So we could build a message that technically meets the standards laid out in RFC 1521 and RFC 1522, but we couldn't send it anywhere — at least not over the Internet — and we couldn't use SMTP to send it, or POP3 to retrieve it. So that leaves us with base64. The following code will generate these new headers:

```
case HeaderLinesSent of
  0: BuildAndSend('' , '', '', Unchanged);
  1: BuildAndSend('--' , BoundaryString, '', Unchanged);
  2: BuildAndSend('Content-Type: application/octet-stream',
                  '', '', Unchanged);
  3: BuildAndSend('Content-Transfer-Encoding: base64',
                  '', '', Unchanged);
  4: BuildAndSend('' , '', '', Unchanged);
end;
```

Content-ID

To link different message parts, the Content-ID header lets you assign a unique ID to each part. Typically, this header is used in situations where not all the message components are contained in one file. As a result, not only does the Content-ID have to be different for each part of a single message, it also needs to distinguish itself from every other part of every other message. This is the same way the Message-ID header is used to uniquely identify a piece of e-mail. We'll be including message text and any file attachments in a single file, so we won't bother generating this optional header.

Content-Description

To give the receiver some hint as to what encoded message parts actually contain, the Content-Description header allows you to tie a string of descriptive text to each message part. This could be a file name, a description for an embedded sound or graphics file, a copyright notice — just about anything you want. Like the Content-ID header, this is an optional item, and to keep things simple, we won't bother with it in the sample program.

Encoding base64 Message Parts

Now that we've got all the new headers taken care of, we really have just one task left: encoding the file attachments as ASCII characters. The algorithm used to accomplish this is called base64. In short, this method takes three bytes of data — 24 bits, in all — and breaks them into four pieces that are each six bits in length. Each six-bit piece is then used as an index into a table containing 64 ASCII characters. The resulting characters are then dropped into the message, one after another, with line breaks thrown in every so often to create short, manageable lines. That's a fairly succinct explanation of how base64 encoding works; but if you're not accustomed to dealing with bit-shifting operators or lookup tables, the code that makes this happen (see Figure 3) may not be entirely clear.

We'll take a little sample data and walk through the encoding mechanism to make the process more obvious. Let's say you just ran across a handy e-mail program in *Delphi Informant*. You downloaded the source code and now you want to pass the .ZIP file along to a co-worker, because you know that's the sort of thing he or she would enjoy.

The first three bytes in the .ZIP file are `#80 #75 #03`. If you take that as a string of bits, you get something like this:

```
010100000100101100000011
```

Break that into four pieces of six bits each, and the result looks like this:

```
010100 000100 101100 000011
```

Content-Transfer-Encoding	Description
7bit	Short lines of ASCII text.
quoted-printable	Short lines of ASCII text with quoted escape characters when needed.
base64	Binary data encoded as short lines of text.
8bit	Short lines of raw binary data.
binary	Raw binary data.

Figure 2: MIME Content-Transfer-Encoding types.

```
while not Eof(fhAttachment) do begin

  BlockRead(fhAttachment, DataIn, 3, ByteCount);
  DataOut[Count] := (DataIn[0] and $FC) shr 2;
  DataOut[Count+1] := (DataIn[0] and $03) shl 4;

  if ByteCount > 1 then
    begin
      DataOut[Count+1] :=
        DataOut[Count+1] + (DataIn[1] and $F0) shr 4;
      DataOut[Count+2] := (DataIn[1] and $0F) shl 2;
      if ByteCount > 2 then
        begin
          DataOut[Count+2] := DataOut[Count+2] +
                              (DataIn[2] and $C0) shr 6;
          DataOut[Count+3] := (DataIn[2] and $3F);
        end
      else
        begin
          DataOut[Count+3]:=$40;
        end;
    end
  else
    begin
      DataOut[Count+2] := $40;
      DataOut[Count+3] := $40;
    end;

  for I := 0 to 3 do
    DataOut[Count+I] := Byte(Base64Out[DataOut[Count+I]]);

  Count := Count+4;
  if Count>59 then
    Break;

end;
```

Figure 3: This encodes a line of base64 data.

which is the same as:

```
#20 #04 #44 #03.
```

In Figure 4, that translates into the string `UEsD`.

All the bytes in the file are translated in this manner, until the last few remaining bytes are reached. If the file size is evenly divisible by 3, the translation will end perfectly with the last three bytes in the .ZIP file being translated into the final four characters in the attachment. If the file is not evenly divisible by 3, one or two bytes will remain. These can't be grouped according to the original plan.

With these last few bytes, the normal process is repeated — except that any missing six-bit groups in the translation are replaced with the pad character =, and any partially formed six-bit group is padded on the right, using zeros.

So if one odd byte remains at the end of the file, the first six bits will be translated normally, while the last two will be padded with zeros on the right to form a full six-bit group; the two missing six-bit groups will automatically be replaced by the pad character.

The same procedure is used when two odd bytes remain, except that the first two six-bit groups will translate normally, the third group will use the remaining four bits plus two zero-bits on the right, and the final, missing six-bit group will be replaced by the pad character.

Recognizing and Decoding

On the other side of the transfer, the receiving e-mail program needs a process for identifying and decoding binary file attachments.

Because a message's previously defined MIME headers positively identify an encoded message part, all that's required is to scan the incoming text for Content-Transfer-Encoding headers of type base64.

From there, the decoding process is simply the opposite of the encoding algorithm: Take four bytes of data, translate them using the lookup table (or by some other means that achieves the same result), strip off the two highest bits from each byte, combine the remaining 24 bits, and break them into three groups of eight bits each.

Let's see how the decoding process reconstitutes the previous example's original binary information.

As we saw earlier, the first three bytes in our .ZIP file were encoded as the four-character string `UEsD`. Referring back in the table allows us to translate this back into this sequence:

```
#20 #04 #44 #03
```

In binary, the sequence looks like this:

```
00010100 00000100 00101100 00000011
```

Value	Encoded	Value	Encoded	Value	Encoded	Value	Encoded
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

Figure 4: The base64 encoding table.

By stripping off the two leading zero-bits in each byte and putting it all together, we get the following string of bits:

```
010100000100101100000011
```

Divide that into three groups of eight bits each, and we have the original sequence:

```
#80 #75 #03
```

The code to accomplish this is shown in Figure 5.

```
while (szWork[Count]<>#0) and (szWork[Count]<>' ') do begin

  for I := 0 to 3 do
    DataIn[I] := Base64In[Byte(szWork[Count+I])];

  ByteCount := 4;
  DataOut[0] := (DataIn[0] and $3F) shl 2 +
                (DataIn[1] and $30) shr 4;

  if DataIn[2]<>$40 then
    begin
      DataOut[1] := (DataIn[1] and $0F) shl 4 +
                    (DataIn[2] and $3C) shr 2;
      if DataIn[3] <> $40 then
        begin
          DataOut[2] := (DataIn[2] and $03) shl 6 +
                        (DataIn[3] and $3F);
          ByteCount := 3;
        end
      else
        begin
          ByteCount := 2;
        end;
    end

  else
    begin
      ByteCount := 1;
    end;

  BlockWrite(fhAttachment, DataOut, ByteCount);
  Inc(Count,4);

end;
```

Figure 5: This code reconstitutes a line of base64-encoded data.

Because the base64 encoded characters do not provide a convenient index into the original table, I've created a second lookup table (see Figure 6) to speed the decoding process. This new table is set up according to the ASCII values of the encoded characters. That way, the encoded character itself can be used as the index into this new lookup table.

A by-product of this arrangement is that several "holes" in the table indicate where certain ASCII values do not correspond to valid base64 translations.

Obviously, wasting these few bytes of space is a bargain when you consider the time that we would otherwise spend doing character-by-character searches in the original lookup table, or evaluating a set of **if** conditions to arrive at the same result.

Bringing It All Together

Our updated sample programs process the MIME headers, and incorporate the encoding and decoding methods we just reviewed, to make binary file attachments work within the existing SMTP and POP3 programs. These programs still don't allow us to store e-mail messages effectively, or to do simple things like forward and reply to messages we've received.

Next month, we'll combine the SMTP and POP3 programs into a single project, and add some of these basic functions. We'll also look at some simple additions that extend the usefulness of the program beyond that of a typical e-mail interface. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\SEP\DI9709GL.

Gregory Lee is a programmer with over 15 years of experience writing applications and development tools. He is currently the president of Software Avenue, Inc., which has just released a C++Builder edition of its Delphi development tool, Internet Developer's Kit. Greg can be reached by e-mail at 76455.3236@compuserve.com.

Value	Decoded	Value	Decoded	Value	Decoded	Value	Decoded
0...42	*invalid*	B	1	T	19	k	36
+	62	C	2	U	20	l	37
44...46	*invalid*	D	3	V	21	m	38
/	63	E	4	W	22	n	39
0	52	F	5	X	23	o	40
1	53	G	6	Y	24	p	41
2	54	H	7	Z	25	q	42
3	55	I	8	91...96	*invalid*	r	43
4	56	J	9	a	26	s	44
5	57	K	10	b	27	t	45
6	58	L	11	c	28	u	46
7	59	M	12	d	29	v	47
8	60	N	13	e	30	w	48
9	61	O	14	f	31	x	49
58...60	*invalid*	P	15	g	32	y	50
=	64	Q	16	h	33	z	51
62...64	*invalid*	R	17	i	34		
A	0	S	18	j	35		

Figure 6: The base64 decoding table.

By Cary Jensen, Ph.D.

Creating Mailing Labels

QuickReport 2: Part II

Last month's "DBNavigator" began an overview of QuickReport version 2, the new version of this powerful VCL-based reporting tool. This month we conclude the series with additional QuickReport techniques, including how to create mailing labels, master-detail reports, and custom report previewers, as well as how to generate reports on-the-fly.

All techniques demonstrated in this article are associated with the project named QUICKREP. The main form for this project is shown in Figure 1.

The key to creating mailing labels in QuickReport is the use of more than one column (assuming that you're printing on sheets of labels that have more than one column). In fact, the easiest way to create mailing labels is to use the QuickReport Label template. To use this template, select **File | New**. Then from the Object Repository, select the Forms page, then double-click the **QuickReport Labels** template (see Figure 2).

This creates a report based on a QuickRep component, but not a form. If you want your QuickRep to appear on a form, select **File | New Form**, then place a QuickRep component on this form.

Whichever technique you use to create your mailing-label report, the *Columns* subproperty of the QuickRep *Page* property allows you to print more than one label across each page.

If the database you are printing from is simple enough (i.e. it has one name field, one address field, and fields for city, state/province, country, and postal code), the placement of QRDBText fields to display this data is straightforward. However, most databases don't conform to this description. For example, most mailing-label databases include optional fields, such as a title, or a field for a second address line. Creating mailing labels for these databases requires some coding.

Actually, there are several approaches for creating mailing labels from a more complex database. If your data is stored on a database server, you can write a stored procedure — on the server — that formats the data correctly. Alternatively, you can create a number of calculated fields, and perform the formatting from the DataSet's *OnCalcFields* event handler.

The final technique, and the one used in the QUICKREP example project, is to place

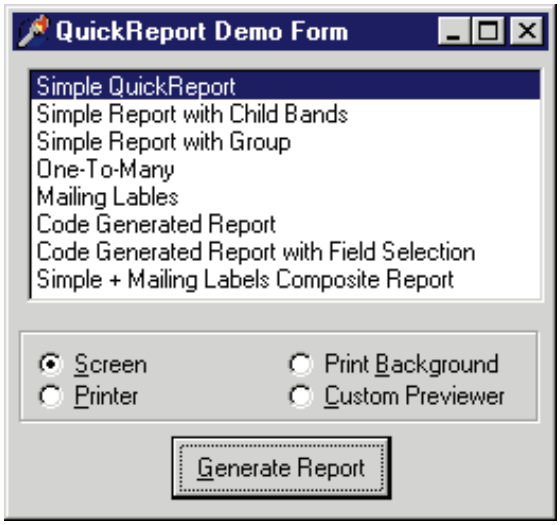


Figure 1: The main form of the QUICKREP project.

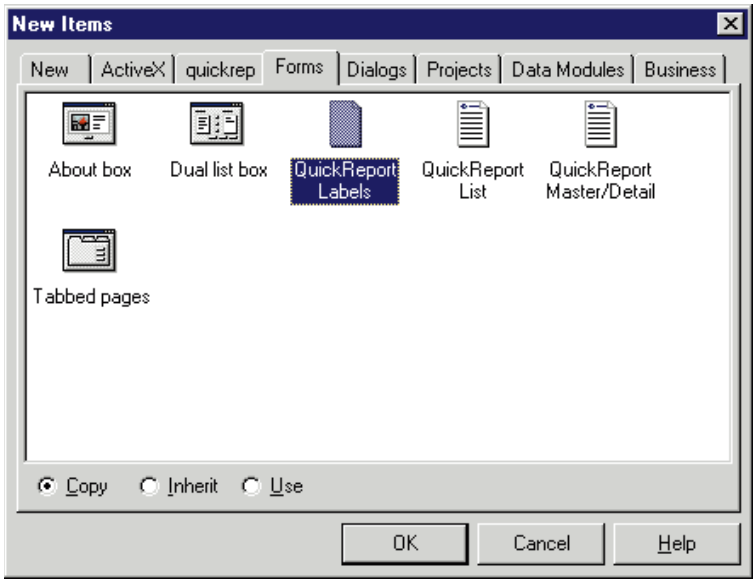


Figure 2: The **QuickReport Labels** template in the Object Repository.

simple QRLabels in the Detail band, and perform all necessary formatting from within the *BeforePrint* event handler for the Detail band. The code in Figure 3 is attached to this event handler, and produces the report output shown in Figure 4.

Creating Master-Detail Reports

Master-detail reports display records from two or more tables. For example, a master-detail report may display individual sales records by customer. In this case, the CUSTOMER table is the master table, and the SALES table is the detail table. This is also referred to as one-to-many reporting.

When creating master-detail reports in QuickReport version 1, it was commonplace to use DataSource components. These permitted you to define a link between the detail and the master table. While you can also use this technique in QuickReport version 2, you should avoid doing so unless you're certain that you will never want to print the report in a background thread. For the same reason, you shouldn't use the **QuickReport Master/Detail** template, which also uses DataSource components.

The form named One2ManyFrm, shown in Figure 5, demonstrates a report that includes three tables in a one-to-many-to-many relationship. The report was initially created by adding a QuickRep component to a form. The Title, PageHeader, Detail, and PageFooter bands were enabled. The *DataSet* property of this QuickRep was assigned to *Table1*, which points to the CUSTOMER.DB table in the DBDEMOS database.

To produce the detail section, you then place a QRSubDetail component. This component appears as a SubDetail band. As with the QuickRep component, you set the QRSubDetail's *Bands* subproperties if you want a header or footer for the detail band. In this report, the QRSubDetail's *DataSet* property points to *Table2*, which is associated with the SALES.DB table in the DBDEMOS database.

For the third-level detail, another QRSubDetail was placed. This QRSubDetail's *DataSet* property points to *Table3*, which is associated with the ITEMS.DB table in the DBDEMOS database.

If you were using DataSource components, you would link *Table3* to *Table2* using a DataSource that points to *Table2* and the *MasterSource* and *MasterFields* properties of *Table3*. Likewise, you would link *Table2* to *Table1* using a DataSource pointing to *Table1*, and the *MasterSource* and *MasterFields* properties of *Table2*. However, because this report may be used in a background thread, the use of DataSource components is ruled out. Instead, event handlers are used.

For each master-detail relationship, you attach code to the *BeforePrint* event handler of the band in which the master table record appears. From within this event handler, you restrict the records displayed in the detail table using code. This code can either include a call to *SetRange* (or define a Filter, although *SetRange* is far more efficient), or execute a Query (if the detail DataSet is a query). For example, the following code is attached to the *BeforePrint* event handler associated with the Detail band of the QuickRep component:

```
Table2.SetRange([Table1.FieldByName('CustNo').Value],
                [Table1.FieldByName('CustNo').Value]);
```

Just before each new CUSTOMER table record is printed, the SALES table records are restricted to display only those records for the particular customer. Likewise, before each SALES detail record is printed, an event handler is executed that restricts the ITEMS table to only those items associated with a particular order. This is accomplished with the following code, which is attached to the *SubDetail1 BeforePrint* event handler:

```
Table3.SetRange([Table2.FieldByName('OrderNo').Value],
                [Table2.FieldByName('OrderNo').Value]);
```

```
procedure TMailLabels.DetailBand1BeforePrint(
  Sender: TQRCustomBand; var PrintBand: Boolean);
var
  i, j: Integer;
  WhichLabel: TQRLabel;
begin

  i := 2;  // Initialize LabelNum.
  QRLabel1.Caption :=
    Table1.FieldbyName('Contact').AsString;
  WhichLabel :=
    TQRLabel(Self.FindComponent('QRLabel' + IntToStr(i)));

  if Table1.FieldByName('Company').AsString <> '' then
    begin
      WhichLabel.Caption :=
        Table1.FieldByName('Company').AsString;
      Inc(i);
    end;

  WhichLabel :=
    TQRLabel(Self.FindComponent('QRLabel' + IntToStr(i)));
  WhichLabel.Caption :=
    Table1.FieldByName('Addr1').AsString;
  Inc(i);
  WhichLabel :=
    TQRLabel(Self.FindComponent('QRLabel' + IntToStr(i)));

  if Table1.FieldByName('Addr2').AsString <> '' then
    begin
      WhichLabel.Caption :=
        Table1.FieldByName('Addr2').AsString;
      Inc(i);
    end;

  WhichLabel :=
    TQRLabel(Self.FindComponent('QRLabel' + IntToStr(i)));
  WhichLabel.Caption :=
    Table1.FieldByName('City').AsString + ', '+
    Table1.FieldByName('State').AsString + '  ' +
    Table1.FieldByName('Country').AsString + '  ' +
    Table1.FieldByName('Zip').AsString;

  Inc(i);
  for j := i to 5 do begin
    // Blank out remaining QRLabels from last record.
    WhichLabel :=
      TQRLabel(Self.FindComponent('QRLabel'+IntToStr(i)));
    WhichLabel.Caption := '';
  end;

end;
```

Figure 3: One technique for formatting mailing labels.

The output created by the One2ManyFrm is shown in Figure 6.

Recall that *SetRange* applies to the currently selected index. Consequently, it was necessary to specifically select the appropriate index for each of the detail tables in this report.

Creating Composite Reports

A composite report is a single report that combines information from two or more reports. Once combined, these reports share consecutive page numbers, and avoid unwanted page breaks between reports. You create a composite report using a QRCompositeReport component with two or more QuickRep components.

The key to using a composite report is the *OnAddReports* event handler. This event handler is called by the QRCompositeReport component each time it's previewed or printed. From within this

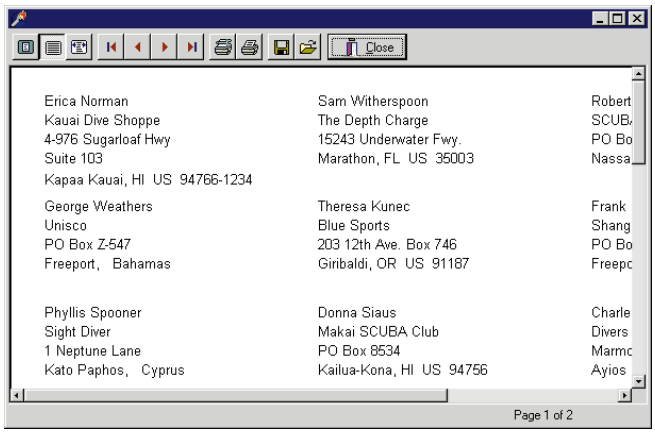


Figure 4: The Mailing Labels report in the default QuickReport previewer.

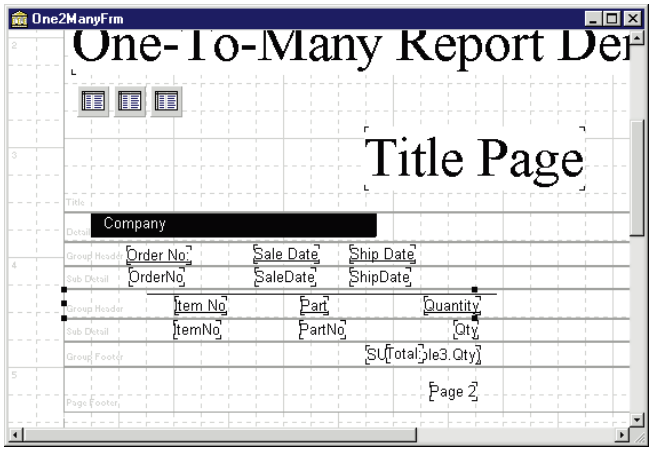


Figure 5: The One2ManyFrm in the QUICKREP project.

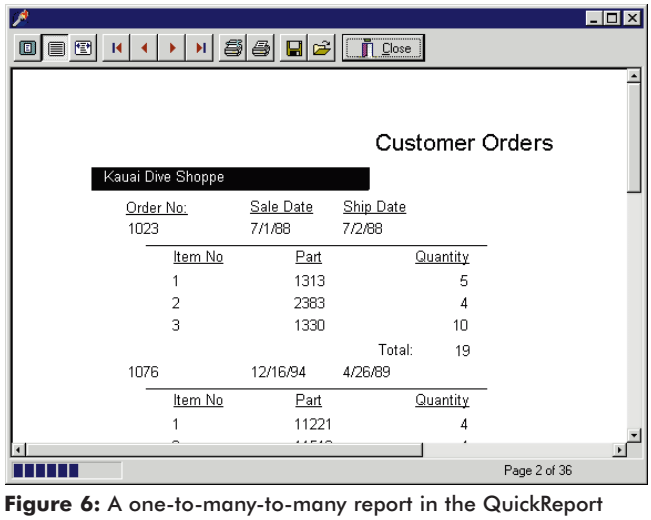


Figure 6: A one-to-many-to-many report in the QuickReport default previewer.

event handler, you add each of the reports that you want included in the composite report to the *Reports* property of the QRCompositeReport component. *Reports* is a *TList* property, meaning it can hold a list of pointers to other objects. You add the individual objects to this list using the *Add* method.

The following is the *OnAddReport* event handler for the *QRCompositeReport1* object on the QUICKREP project main form:

```
procedure TMainForm.QRCompositeReport1AddReports(
  Sender: TObject);
begin
  with QRCompositeReport1.Reports do begin
    Clear;
    Add(SimpleFrm.QuickRep1);
    Add(MailLabels.QuickRep1);
  end;
end;
```

This code defines the composite report as consisting of the contents of the *SimpleFrm* report combined with the *MailLabels* report. Unfortunately, there are some limitations when using composite reports. For example, you cannot use a custom previewer with a composite report. Similarly, the QRCompositeReport component doesn't support the *PrintBackground* method. Finally, if the last report in the Report list has the *PageHeader* option set to *False*, the QRCompositeReport component will not print the page header on the first page of the first report.

Creating a Custom Report Previewer

The default report previewer available to all QuickReports allows users to easily view and navigate a report. There may be times, however, when you want to have complete control over the look and behavior of the report previewer. In those cases, you will create a custom previewer, and instruct the QuickRep component to use that previewer when its *Preview* method is called.

There are two steps to creating a custom report previewer. The first is to place a QRPreview component on a form. The QRPreview component is a *TScrollBox* descendant that QuickReports can use to display the contents of a report. Figure 7 shows the CustPreview form from the QUICKREP project. This form contains a QRPreview with its *Align* property set to *alClient*. If the QRPreview is the only object on the form, the user can only preview the first page of the report, and nothing else. Therefore, you need to add an interface that permits the user to navigate the report, as well as zoom in and out, select printer options, and so on. The form shown in Figure 7 also contains a menu and a toolbar from which these features are provided.

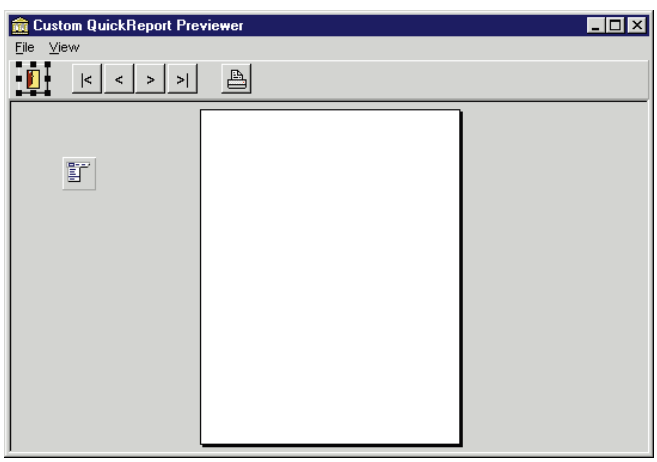


Figure 7: The CustPreview form of the QUICKREP project.

To control the previewing of a QuickReport from within a custom Previewer, you use the properties and methods of two objects. The first, and most obvious, is the QRPreview component itself. This component provides properties that enable you to set the zoom proportion of the display, as well as navigate the report. It also provides methods to fit the display to the width or height of the report.

The following code example demonstrates how to display the second page of a multi-page report in the previewer named *QRPreview1*:

```
QRPreview1.PageNumber := 2;
```

The second object used to control the custom previewer is the QRPrinter component. Each QRPreview has a *QRPrinter* property that points to an instance of a QRPrinter.

Using this property, you can issue a form-feed, load a previously saved QuickReport, save a QuickReport, or print the current report, among many other tasks. For example, you can execute the following statement from a custom preview form to permit the user to change the current printer settings:

```
QRPreview1.QRPrinter.PrinterSetup;
```

The following is the event handler associated with the Next Page button on the toolbar from the custom previewer:

```
procedure TCustPreview.NextPage1Click(Sender: TObject);
begin
  if QRPreview1.PageNumber =
       QRPreview1.QRPrinter.PageCount then
     MessageBeep(MB_OK)
  else
    QRPreview1.PageNumber := QRPreview1.PageNumber + 1;
end;
```

For more examples of calling the methods of QRPreview and QRPrinter, inspect the CUSTPREV.PAS unit of the QUICKREP project (see the end of the article for download information).

After you've created a form to be used as the custom previewer, you must instruct the QuickRep component to use this previewer instead of its default. Assign an event handler to the *OnPreview* event property of the QuickRep object. From within this event handler, assign the QuickRep's *QRPrinter* property to the *QRPrinter* property of the QRPreview component, then display the form on which the QRPreview component appears. This form must be displayed non-modally, using the form's *Show* method.

The *OnPreview* event handler is similar to a *TNotifyEvent* method pointer. The one interesting element about it is that *Sender*, the sole parameter of a *TNotifyEvent* method, is not the object from which the event handler was called, but instead is the QRPrinter associated with the QuickRep component being previewed.

From within the *OnPreview* event handler, you must cast *Sender* as a *TQRPrinter* object, then assign this object to your preview-

er's *QRPrinter* property. To perform this type-casting, you must include the QRPrntr unit in a **uses** clause for the unit that includes this event handler. This is necessary because *TQRPrinter* is defined in the QRPrntr unit, not the QuickRep unit.

To create your *OnPreview* event handler, you must also add an *OnPreview* method to the form from which the preview method will be called. This requires you to add the following line (or one similar to it) to the **published** (default) or **public** section of your form's **type** declaration.

```
procedure CustomPreview(Sender: TObject);
```

You must also implement this method. Assuming it's being defined for a form class named *TMainForm*, the following is an example of this implementation:

```
procedure TMainForm.CustomPreview(Sender: TObject);
begin
  // TQRPrinter is defined in the QRPrntr unit.
  // CustPreview is an auto-created form.
  CustPreview.QRPreview1.QRPrinter := TQRPrinter(Sender);
  CustPreview.Show;
end;
```

Now all you need is to assign this defined method to your QuickRep's *OnPreview* method. For example, if you want to display the QuickRep located on a *Form* object named *SimpleFrm* using your custom previewer, you can execute the following code:

```
SimpleFrm.QuickRep1.OnPreview := CustomPreview;
SimpleFrm.QuickRep1.Preview;
SimpleFrm.QuickRep1.OnPreview := nil;
```

Notice that this code sets the QuickRep's *OnPreview* property to **nil** following the call to *Preview*. This restores the original, default previewer. If you never want to use the default previewer, you can omit this step.

Creating Reports at Run Time

To the extent that you can create reports on-the-fly, you can build an interface that permits your users to generate custom reports at run time. Fortunately, QuickReport supplies you with the ability to do just that.

The QRExtra unit defines two classes with the ability to easily create reports at run time. These are *TQRBuilder* and *TQRListBuilder*. *TQRBuilder* can create a basic report, but lacks database connectivity. *TQRListBuilder* supports data-aware controls, but requires a good deal of programming. Finally, the *QRCreateList* procedure (not an object) permits you to easily create simple list-type reports for some or all fields from a DataSet. The following is the declaration for *QRCreateList*:

```
procedure QRCreateList(AReport: TQuickRep,
  AOwner: TComponent; ADataSet: DataSet;
  ATitle: string;
AFieldList: TStrings);
```

When you call *QRCreateList*, you pass five parameters. The first is a *QuickRep* object, which can be an existing report, or a *QuickRep* instance variable. *AOwner* is the object assigned to the QuickRep's *Owner* property; the *DataSet* provides the underlying data access; and the fourth parameter is a string to use as a default title for the report. The fifth and final argument can either be **nil**, in which case all fields that can fit on the report will be displayed, or a *TStrings* list of field names. By passing a *TStrings* list, you can selectively choose which fields you want to appear on the report.

If the QuickRep component that you pass in the first parameter is an existing report, the generated report is based on the design of the existing report. Consequently, if you call *QRCreateList* twice in a row, the second report generated will have all the fields of the first in addition to all the fields of the second. If you want to reset the QuickRep between calls to *QRCreateList*, you must set the

```
begin
  // Set QRep to nil, in case it was previously created.
  QRep := nil;
  // The string variable RepTitle is declared
  // in this block.
  InputBox('Set Report Title','Report Title',RepTitle);
  QRCreateList(QRep, Self, DataModule1.CustomerTab,
               RepTitle, nil);

  // ReleaseQRep is a method of Self.
  QRep.AfterPrint := ReleaseQRep;
  QRep.AfterPreview := ReleaseQRep;

  case RadioGroup1.ItemIndex of
    0: QRep.Preview;
    1: QRep.Print;
    2: QRep.PrintBackground;
    3:
      begin
        QRep.OnPreview := CustomPreview;
        QRep.Preview;
        QRep.OnPreview := nil;
      end;
  end;

end;
```

Figure 8: A simple report with all the fields from *CustomerTab*.

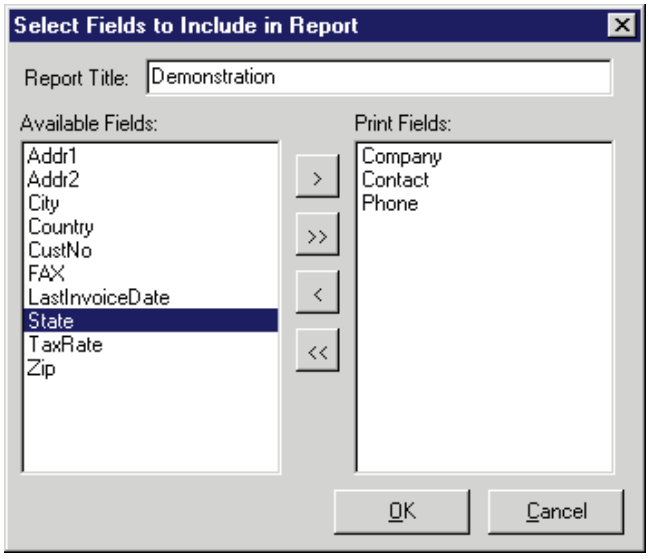


Figure 9: The CustCodeRep form in the QUICKREP project permits the user to select which fields to report.

QuickRep variable to **nil** before calling *QRCreateList* a second time.

After you've created a report using *QRCreateList*, you can print or preview it as you would any other. You can even print it in a background thread, or display it in a custom previewer. One thing to keep in mind, however, is that you must declare the *QuickRep* variable that you pass as the first parameter within scope of the previewer or thread. In other words, you must at least declare this variable in the **implementation** or **interface** blocks of your unit. If you declare the *QuickRep* variable within an event handler, the report will be released as soon as the variable goes out of scope, which will likely raise an exception.

Another issue you should consider is the release of a QuickRep created using *QRCreateList*. Because this QuickRep is not created on a form, you cannot use the form's *OnClose* event handler to release the report. However, you can create a *TNotifyEvent*-type method that calls the report's *Free* method, then assign it to the QuickRep's *AfterPrint* and *AfterPreview* methods. Doing so ensures that the report is freed after it has been printed or previewed.

The code in Figure 8 creates a simple report with all fields (that can fit) from the DataSet named *CustomerTab*, on the data module named *DataModule1*. This code can be found on the QUICKREP project's main form.

Another example of using *QRCreateList* can also be found on the QUICKREP project main form. This one first displays the dialog box shown in Figure 9. This dialog box permits a user to select which records to display on the report being generated. After the desired fields are selected, and a title entered, selecting **OK** causes a custom report to be generated, like the one shown in Figure 10. This report was generated with the code in Figure 11.

The next code is associated with the *AfterPrint* and *AfterPreview* event handler assigned in the preceding code segment:

```
procedure TMainForm.ReleaseQRep(Sender:
  TObject);
begin
  TQuickRep(Sender).Free;
end;
```

Permit me to repeat: the *QRCreateList* is in the QRExtra unit. Consequently, any unit that uses the preceding technique must include both QRExtra and QuickRep units listed in its **uses** clause: QRExtra for the *QRCreateList* procedure, and QuickRep for the *TQuickRep* class, which is required for the *QuickRep* instance variable.

Version 1 vs. Version 2

QuickReport versions 1 and 2 are vastly different. The units involved don't have the same names, and many of the components are defined by different classes. Delphi 3

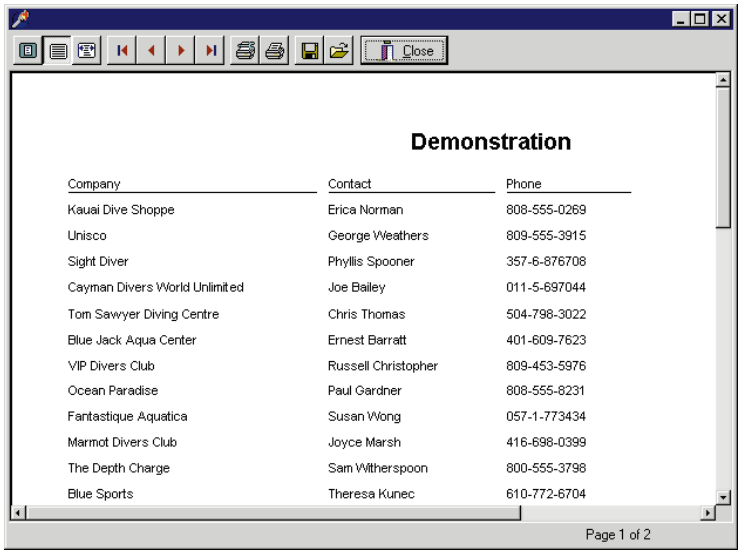


Figure 10: A report generated from user-selected fields.

```
begin
  CustCodeRep := TCustCodeRep.Create(Self);
  if CustCodeRep.ShowModal <> mrOK then
    Exit;

  // Set QRep to nil, in case it was previously created.
  QRep := nil;
  QRCreateList(QRep, Self, DataModule1.CustomerTab,
               CustCodeRep.Edit1.Text,
               CustCodeRep.DstList.Items);
  QRep.AfterPrint := ReleaseQRep;
  QRep.AfterPreview := ReleaseQRep;

  case RadioGroup1.ItemIndex of
    0: QRep.Preview;
    1: QRep.Print;
    2: QRep.PrintBackground;
    3:
      begin
        QRep.OnPreview := CustomPreview;
        QRep.Preview;
        QRep.OnPreview := nil;
      end;
  end;
end;
```

Figure 11: The code that generates the form in Figure 10.

attempts to resolve this by performing a conversion on a version 1 QuickReport when it's opened. While this conversion process is automatic, it does require some manual adjustment to the converted form.

The amount of work you'll need to perform ranges from changing a few properties manually, to wholesale removal of existing objects and replacement by the QuickReport 2 equivalent. For more information on the steps necessary to successfully convert a QuickReport 1 report, see Appendix C of QRPT2MAN.DOC, located in Delphi 3's \QuickRep subdirectory. Always back up your original reports before performing a conversion.

Conclusion

QuickReport version 2 is a dramatic update of this powerful VCL-based report generator. It's easier to use, and is more flexible than its predecessor. Now that ReportSmith is gone

from Delphi, you should take a serious look at QuickReport. It's very likely that you will like what you see.

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\SEP\DI9709CJ.

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]. Cary is also a Contributing Editor of *Delphi Informant,* as well as 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.

By Bill Todd

Maintaining Your Maintenance-Free Database

Inside InterBase: Part IV

"What? I thought InterBase was famous for being the only database server that *doesn't* require maintenance. Why do I need an entire article on maintaining my database?"

InterBase comes as close as possible to being maintenance-free, and InterBase databases have been known to run for years with no maintenance. If you want the best performance from your database at all times, however, read on. There are things you can do — at regular intervals — to keep your database in shape.

Much of what's described in this article will be performed using Server Manager. The screen shots and descriptions are based on the version of Server Manager that ships with InterBase 4.2 for Windows NT. Some of the menu choices and screens may be different if you're using a different version of InterBase.

Shutting Down

Database shutdown is the first topic we'll consider. Sweeping the database (discussed later in this article) is more effective when the database is shut down. Also, some tasks can be done *only* when the database is down. These include:

- validating and repairing the database;
- adding and removing foreign keys; and
- adding and removing secondary files.

Figure 1: Server Manager's InterBase Login dialog box.

To shut down your database, start Server Manager, then connect to your server by choosing **File | Server Login**, and entering the server, protocol, user name, and password (see Figure 1). Next, connect to the database you want to shut down by selecting **File | Database Connect**, and entering the path to the database. If your version of InterBase uses the Super Server architecture (version 4.0 for Netware or 4.2 for NT), you can view a list of the users connected to the InterBase server by selecting **Maintenance | Database Connections**.

You can't shut down a database unless you're logged on as SYSDBA, or you're the owner of the database. To shut down the database, choose **Maintenance | Database Shutdown**; this will display the Database Shutdown dialog box (see Figure 2), which offers three options:

- If you choose **Deny New Connections while waiting**, new users won't be allowed to connect, and the database will shut down as soon as the last user disconnects. The server will wait for the time shown in the **Timeout** field at the bottom of the dialog box. If not all users disconnect within this interval, the database will not shut down.
- If you choose **Deny New Transactions while waiting**, no new transactions can begin,

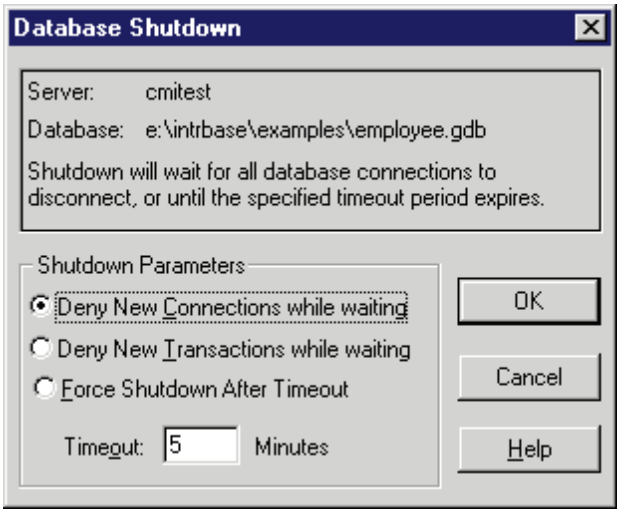


Figure 2: The Database Shutdown dialog box.

and no new users can log on. The database will shut down — and all users will be logged off — as soon as the last transaction is committed or rolled back. Once again, the server waits only for the specified timeout period.

- The third option is **Force Shutdown After Timeout**. If you choose this option, all active transactions will be rolled back — and all users will be logged off — at the end of the timeout period.

To restart the database, choose **Maintenance | Database Restart** from the Server Manager menu.

Removing Old Versions

As multiple users process transactions in InterBase, new record versions are created. Each record version contains the number of the transaction that created it. This ensures that each user always sees a consistent view of the data, as of the moment when his or her transaction started. Something must be done to remove versions no longer needed, or your database will grow continuously. InterBase performs automatic garbage collection to remove such outdated versions. Each time InterBase accesses a row, it compares the transaction number of each version of the row to the oldest interesting transaction (OIT) number. The OIT is the oldest transaction whose status is anything other than committed. All versions whose transaction number is older than the OIT are deleted.

But automatic garbage collection may not be enough — for two reasons: In a large database, a long time may pass between accesses for any particular row. This means that old versions of a row may accumulate, inflating the size of the database unnecessarily. The second and more important problem occurs if the OIT gets stuck. Because the OIT is the oldest transaction with a status other than committed, any transaction rolled back or left in limbo by a crash during a two-phase commit will cause the OIT to stop advancing. This will halt automatic garbage collection, because versions with transaction numbers greater than the OIT can't be deleted.

Sweeping the database when you have exclusive use is one way to correct this problem. You *can* run a sweep while the database is in use; however, this will solve only the first of the problems we examined. Such a sweep deletes all versions with transaction numbers older than the OIT, just as automatic garbage collection does, and also removes all versions of any deleted rows.

If the OIT is stuck, you need to run a sweep with no other users in the database. When you do, InterBase:

1) rolls back any transactions in limbo;
2) resets the OIT to the most recent transaction number;
3) removes all row versions older than the OIT; and
4) removes all deleted rows.

Recovering Limbo Transactions

If you use a two-phase commit to handle transactions that span multiple databases on multiple servers, you may have one or more transactions in limbo. Understanding how a transaction gets placed in limbo requires a look at what happens during a two-phase commit:

1) The controlling server sends a message to all servers involved in the transaction, announcing that it's ready to commit.
2) The other servers change the status of their transaction to "limbo," and acknowledge that they're ready to commit.
3) The controlling server commits its changes, and tells the other servers to commit.
4) The other servers change the status of their transactions from "limbo" to "committed," and acknowledge that they've committed.

The problem occurs between the third and fourth steps. What happens if the controlling server commits its changes and tells the other servers to commit, but one or more of the other servers crashes before it commits? Now you have a transaction that committed on one or more servers, but not on all. This transaction is left in limbo.

Running a sweep when you have exclusive use of the database will roll back any limbo transactions, but this may not be what you want. You can evaluate each limbo transaction individually, and decide whether to commit it or roll it back, by selecting **Maintenance | Transaction Recovery** from the Server Manager menu. If any transactions are in limbo, you'll be able to view them and their affected rows one at a time, and either commit them or roll them back. If no transactions are in limbo, a dialog box will appear with the message, "No pending transactions were found."

Maintaining Indexes

InterBase indexes are balanced B-trees. Adding or deleting a large number of rows can cause an index to become unbalanced and less efficient. You can rebuild any index by using the SQL ALTER INDEX statement to inactivate the index, then activate it. For example:

```
ALTER INDEX IndexName INACTIVE
ALTER INDEX IndexName ACTIVE
```

When you set the index to ACTIVE, it will be rebuilt, and its *selectivity* will be recalculated. You can also recalculate the selectivity for an index without rebuilding it, using the SQL statement:

```
SET STATISTICS INDEX IndexName
```

What is selectivity? It's a measure of the usefulness of an index, computed by dividing the number of rows in the table by the number of unique values in the index. InterBase's query optimizer uses selectivity to determine how useful an index might be in processing a query. Selectivity is computed only when the index is created, or activated by ALTER INDEX or the SET STATISTICS statement. If you make table changes that significantly change either the number of rows in the table or the number of unique values in an index, you should recompute the selectivity of any affected indexes.

Backing Up and Restoring

Backing up and restoring your database accomplishes everything that an exclusive-use sweep does — *and* it rebuilds all indexes. Because a backup is a read-only transaction, it backs up only the most recent committed version of each row, along with the metadata for the database. Because record versions created by transactions in limbo are not backed up, all limbo transactions are effectively rolled back.

When you restore the database, InterBase:

1) recreates the database using the metadata information from the backup;
2) loads the data into each table, renumbering all transactions starting with four (three transactions are used to recreate the metadata);
3) sets the next transaction number to the last transaction number used, plus one;
4) sets the oldest active transaction (OAT) to the next transaction number;
5) sets the OIT to the next transaction number, minus one; and
6) rebuilds all indexes.

A backup and restore is the most comprehensive maintenance for an InterBase database, because it removes all old row versions, resets the OIT, renumbers all transactions, rolls back limbo transactions, and rebuilds all indexes, ensuring that the indexes are balanced and that their selectivity is accurate. Unfortunately, all current versions of InterBase have problems backing up and restoring a database if the metadata includes the SQL PLAN statement. This bug will be fixed in the next release.

To back up your database, select **Tasks | Backup** from the Server Manager menu to display the Database Backup dialog box (see Figure 3). The database path will point to the database that was selected in Server Manager when you selected **Tasks | Backup** from the menu. You can change the database to be backed up, if you wish. In the **Backup File or Device** field, enter the name and path of the disk or tape device to which you want to back up. The Options checkbox group lets you choose any of the six options shown in Figure 4.

After successfully backing up your database, restore it by choosing **Tasks | Restore** from the Server Manager menu.

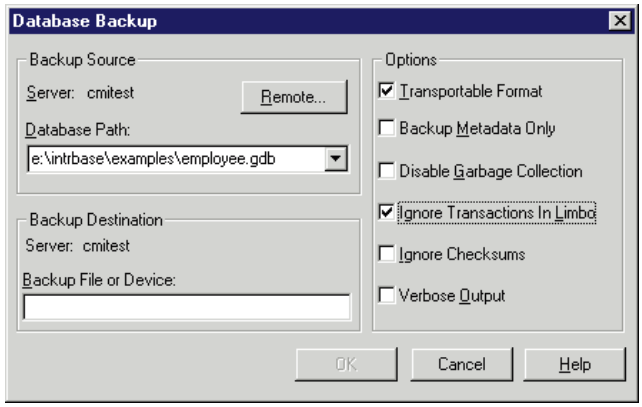


Figure 3: The Database Backup dialog box.

Option	Description
Transportable Format	Backs up your data in a format that can be restored on any InterBase server, regardless of platform.
Backup Metadata Only	Creates a new, empty database from this backup. (You could use this option to create an empty test database from a production database.)
Disable Garbage Collection	Handy if you're trying to back up a corrupt database before attempting repair, or to save time when restoring from a backup. (A backup is simply a read transaction that visits every row in every table; by default, it collects garbage on every row.)
Ignore Transactions In Limbo	Tells InterBase to ignore any limbo transactions, and back up only the last committed version of each row.
Ignore Checksums	Allows you to back up a database that contains invalid checksums. (To detect corrupt data caused by a system crash, InterBase maintains a checksum on each page of the database.)
Verbose Output	Allows you to monitor the progress of the backup. (However, detected errors are always displayed, whether you select **Verbose Output** or not.)

Figure 4: Backup options.

Validating and Repairing Your Database

Because database servers are usually dedicated machines running stable operating systems such as NT or UNIX, and are connected to uninterruptable power supplies, it's not likely that your server will crash. However, if a crash occurs, it's possible that your database may be damaged. You can validate and repair your database using Server Manager by selecting **Maintenance | Database Validation**, to display the dialog box shown in Figure 5.

Select the **Validate record fragments** option if you want InterBase to validate record structures as well as page structures in the database. Choose **Read-only validation** if you want InterBase to validate the database, but not make any repairs. Choose **Ignore checksum errors** if you want the validation to ignore checksum errors in the data, and continue with the validation.

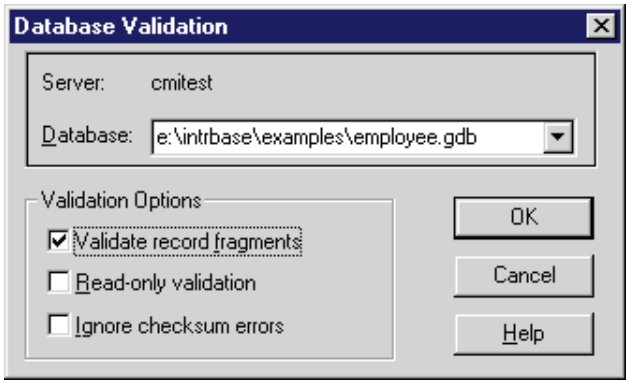


Figure 5: The Database Validation dialog box.

If your database contains checksum errors, the Validation Report dialog box will open, and you can click the **Repair** button to let InterBase attempt to repair the errors. Before you repair checksum errors, be sure to make a copy of the database using an operating-system command or utility; data may be lost during the repair process, and some errors can't be repaired. Remember too that if your database was damaged by a hardware failure, and if you're using a shadow copy of the database, it's probably not damaged.

Conclusion

InterBase databases require very little maintenance, but if you have transactions rolled back or in limbo, either a sweep with exclusive use or a backup and restore is necessary. In addition, bulk changes to a table may unbalance indexes, or make the selectivity for the index invalid. Being aware of the events that can cause problems and the corrective action to take to keep your InterBase system operating at peak efficiency.

Next month, we'll wrap up this series with a look at InterBase's multi-generational architecture. See you then. Δ

Bill Todd is President of The Database Group, Inc., a database consulting and development firm based near Phoenix, AZ. He is a Contributing Editor of *Delphi Informant*; co-author of *Delphi 2: A Developer's Guide* [M&T Books, 1996], *Delphi: A Developer's Guide* [M&T Books, 1995], *Creating Paradox for Windows Applications* [New Riders Publishing, 1994], and *Paradox for Windows Power Programming* [QUE, 1995]; and a member of Team Borland, providing technical support on CompuServe. He has also been a speaker at every Borland Developers Conference. He can be reached on CompuServe at 71333,2146, on the Internet at 71333.2146@compuserve.com, or at (602) 802-0178.

By Dan Ehrmann

The Paradox Files: Part VI

The Finale: Multi-User and Locking Issues

The first two articles in this series explored the internals of Paradox .DB files; they dissected table structure, record and block management, field types, and calculation of record size. The third article examined primary and secondary indices, the fourth discussed validity checks and referential integrity, and the fifth covered password protection and the table-language options. This sixth and final article will cover multi-user access to Paradox tables across a network, and how Paradox locks tables and records.

Using Paradox tables over a network means facing a whole new set of challenges. While a query is extracting information from a table, other users may be adding or modifying the very records being queried. Multi-user access is a balancing act: Minimize conflicts between users to maximize access — all while retaining data integrity.

The Paradox file format is designed to be multi-user "straight out of the box." When you access Paradox tables on a shared device, the BDE automatically places table and record locks on those tables. It places the least restrictive lock compatible with the operation being performed. If an operation requires exclusive access to a table, the BDE will ensure that it has this level of access before proceeding. But if an operation can be performed while other users are modifying the table, the BDE will place a lock with only minimal restrictions.

Before it grants access to an object, the BDE always checks which locks have been placed on that object by other users. Access by any user must be compatible with all existing locks on a table. If prior locks are too restrictive, a user won't be able to access the table, and the BDE will report that other users have placed conflicting locks. If prior locks allow for concurrent access, the BDE will place a new lock — with the network user's name attached — alongside the existing ones, guaranteeing the operation will finish.

Along with the locks the BDE places automatically, you can also place locks preemptively, under program control.

How the BDE Manages Network Locking

Locking options for Paradox tables are significantly more powerful than the default file-locking available in DOS and Windows, and even those in many network operating systems. For this reason, the BDE manages Paradox table and record locks itself, using custom files. These files are created by the BDE when needed, and managed automatically. You will normally not need to worry about them — unless there was a system lockup or abnormal exit from an application — but it helps to understand what these files are and how they work.

PDOXUSRS.NET. This file is known as the Paradox Network Control file, because it serves to track the various users on the network. Every application sharing access to Paradox tables on the network must reference this file, and there should be only one such file on the whole network. The location of this file is stored in the IDAPI.CFG file, which itself can be maintained locally for each user, or shared on the network. It's set in the BDE

Administrator (Delphi 3) or the BDE Configuration Program (Delphi 2). On the Drivers page, select the Paradox driver and specify a shared location for the *Net Dir*, also known as the network control directory.

Newer versions of the BDE (3.x and 4.0) allow you to use different virtual locations for this directory — including mapped drives — as long as each address points to the same physical location. This isn't true in older versions of the BDE.

If a PDOXUSRS.NET file isn't found in the specified location, it's created automatically by the BDE. This file can track up to 300 *virtual* users, including multiple sessions created by a *single* user.

In your Delphi applications, the location of the PDOXUSRS.NET file can be read and set using the *TSession.NetFileDir* property. For the default session, this property is read from the IDAPI.CFG file. To access files controlled by different PDOXUSRS.NET files, create a new *TSession*.

PDOXUSRS.LCK. This file is created by the BDE in each directory containing Paradox tables, when the first user accesses one of these tables. It tracks which activities are permitted in a directory, and what each user is actually doing with the tables in that directory. There are three types of PDOXUSRS.LCK files:

- For a shared directory, the file contains information on the tables and records locked, the user who placed each lock, the session in which the lock was placed, and the type of lock that was placed. When a user attempts to place a lock on a table, this file is checked to see if a conflicting lock exists; if it does not, Paradox places the new lock by making an entry into this file.
- For a user's Private Directory, the file contains a special flag indicating its private status, and the user to whom it belongs. This flag tells the BDE that another user or session can't access Paradox tables in this directory. The Private Directory can be used to hold temporary tables that must remain unique to a specific user, such as intermediate result tables from a series of queries. In your Delphi applications, the location of the Private Directory can be read and set using the *TSession.PrivateDir* property.
- The BDE can also lock a directory as shareable, but read-only. When a directory lock is placed, the BDE knows that no table or record locking need be performed, because all tables in that directory can be read by all users — subject, of course, to the network-access rights granted by the supervisor — but no tables can be modified. This results in significantly better performance.

Lock	Description
Exclusive Lock	The user has exclusive access to, and control of, the table; no one else can even view it. Required for creating or restructuring a table.
Write Lock	Other users can view the table, but only the locked user can modify it. This ensures that updates take place cleanly. Copying a table requires a Write Lock on the destination, as does any kind of bulk change to the table. Only one Write Lock can exist on a table.
Read Lock	The locked user may read the table, and write to it if no other user also has a Read Lock. Other users can also read the table at all times. Multiple Read Locks can coexist on a table, but each will stop the others from writing.
Open Lock	The lowest level of locking. A user may open the table and read its contents; other users can do the same. Required for reading the structure of a table.

Figure 1: Paradox table locks.

A directory lock is handled by the BDE as an exclusive lock on the file PARADOX.DRO (Directory Read Only.) To create a directory lock in Delphi, use the following code:

```
var
  lock_path: array[0..DbiMaxTblNameLen] of Char;
begin
  StrPCopy(lock_path,'C:\MYPATH\PARADOX.DRO');
  Check(DbiAcqPersistTableLock(
          Database1.Handle, lock_path,szPARADOX));
end;
```

The *DbiAcqPersistTableLock* function can be used to acquire an exclusive lock on a non-existent table, where the extension is specified. Use *DbiRelPersistTableLock* to release the lock. The *Check* procedure determines if the return value from the function call indicates an error condition, and if so, calls *DbiError* to raise an exception.

One of the important benefits of using custom files to control locks is that additional information is available in the event of an error. For example, if you try to access a table and a conflicting lock exists, the BDE will return an error message with the name of the user holding the conflicting lock.

PARADOX.NET, PARADOX.LCK, and <tablename>.LCK. As you saw in the first article of this series, the Paradox file format started life back in 1985, within the Paradox for DOS product. Multi-user capabilities were added in 1987 with version 2.0. At the time, Paradox used a slightly different network-locking scheme. In 1992, to coincide with the releases of Paradox 4.0 for DOS and Paradox 1.0 for Windows, Borland revised this scheme to improve performance and reliability. New .NET and .LCK filenames were chosen, but the old ones are also created and flagged to lock old-style applications out of these directories.

Table Locks

Paradox table locks are shown in Figure 1 (listed in order from most to least restrictive); Figure 2 shows how locks can coexist with each other. For each table in a directory, a check-mark indicates that locks of each type can be placed at the same time.

Open Locks can coexist, allowing more than one user to view a table at the same time. Two or more people can also have Read Locks on a table at the same time. In this situation, each stops

the other from writing to the table. A Write Lock provides *exclusive write access* to a table; other people can view the data via an Open Lock, but you're guaranteed that no one else can stop you from reading and writing to the table.

Delphi implements these locks in a slightly different way than you would expect. Through the BDE, Delphi allows you to open a table in Share Mode or Exclusive Mode, controlled by the *TTable.Exclusive* property, as follows:

- If the *Exclusive* property is left at its default of *False*, the table is opened with an Open Lock, and many people can share access to the table — as long as they use the same mode.
- If the *Exclusive* property is set to *True* before the *TTable* is opened, no other application or user can access the table in any way. The table is opened with an Exclusive Lock, and the first user has exclusive control over it.

If another user has a table open in Share Mode (with an Open Lock), and you try to open the same table in Exclusive Mode, you will receive the following error:

```
Table is busy.
Table: <tablename>
User: <other user name>.
```

If another user has an Exclusive Lock on a table, and you try to open the same table in either Share or Exclusive modes, the error message has a slightly different first line:

```
File is locked.
Table: <tablename>
User: <other user name>.
```

A lock conflict is returned as an EDBEngineError, with a code of 10243 for a busy table (conflict with an Open Lock), and 10245 for a locked table (conflict with an Exclusive Lock). You can trap for table-open exceptions using code such as that in Figure 3.

Default Locks for Operations

Every operation performed on Paradox tables has an associated default lock. Figure 4 lists some common operations, and the table-level lock required for each.

For example, if you execute a SQL INSERT, UPDATE, or DELETE statement, the BDE will attempt to place a Write Lock on the table, to ensure that you retain a consistent view for the duration of the operation. If other users have only Open Locks on the table, the Write Lock will be placed. But if other users have Read Locks, the Write Lock will fail.

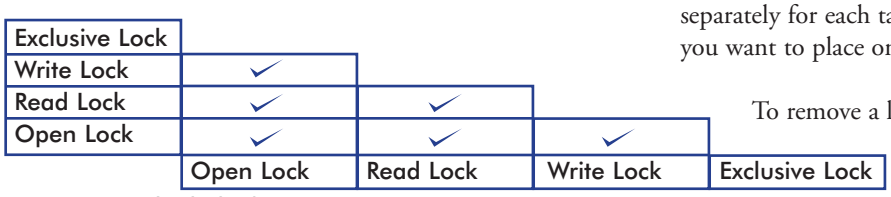


Figure 2: Paradox locks that can coexist.

```
var
  err: Word;
begin

  try
    Table1.Exclusive := True;
    Table1.Open;
  except
    on E: EDBEngineError do
      begin
        err := (E as EDBEngineError).Errors[1].ErrorCode;
        if (err = 10243) or (err = 10245) then
          ShowMessage('Table cannot be opened exclusively')
      end;
  end;

end;
```

Figure 3: Trapping for table-open exceptions.

Operation	Table Lock
Read the structure of a table, including many operations performed in design mode.	Open
Populate a drop-down list control from a field in a table.	Open
Prepare a query.	Open
Execute a SELECT statement.	Open
Execute an INSERT, UPDATE, or DELETE statement.	Write
ALTER a table.	Exclusive
CREATE a table.	Exclusive

Figure 4: Table-level locks required for operations.

Placing Table Locks under Program Control

As you saw in the previous article in this series, the BDE manages access to tables using *TSession* variables. When you run your application, the BDE creates a default *TSession* for you. You can explicitly create additional *TSession* variables to establish independent connections to the BDE and your tables.

The BDE treats each session as a separate *virtual* user. Within the lock file, Paradox considers user IDs and session handles to be fundamentally the same thing. Table locks are placed by Paradox at the object level, but enforced at the session level. If you attempt to place multiple locks on a table within the same session — even if those locks are potentially redundant — you will not be stopped. However, locks placed under different sessions are treated as having been placed by different users, and the concurrency rules shown in Figure 2 will be applied to determine if a lock can be placed.

To lock a Paradox table, use the *TTable.Lock* method. It accepts a single parameter: the lock type to place. LockType is either *ltReadLock* or *ltWriteLock*. You must call this method separately for each table you want to lock, and for each lock you want to place on a table.

To remove a lock already placed, use the *TTable.Unlock* method, which accepts the same parameter. It must be called separately for each table and each lock.

When Are Table Locks *Not* Placed?

In the following situations, table locks will not be placed when you access Paradox tables:

- When the directory is on a local hard disk, unless LOCAL SHARE has been enabled. (This is done under System | Initialization in the BDE Administrator. For it to be functional, you must also enable local sharing at the operating-system level.);
- in the user's Private Directory, because no other users can access Paradox tables in this directory;
- on a read-only device such as a CD-ROM;
- if the table has its Read-Only file attribute set at the operating-system level; or
- if the PDOXUSRS.LCK file indicates a "directory-locked" directory, where all Paradox objects are read-only.

Record Locks

When users start to edit records in the table, Paradox places record locks in the lock file, but the table lock remains an Open Lock to ensure maximum concurrency. When you attempt to post a new or changed record to a table, the Open Lock is escalated to a Write Lock for the duration of the posting operation.

The BDE supports a maximum of 255 record locks per table at any one time. Paradox record locks are exclusive. Only one user at a time can lock a record, although other users can read the record, and even see changes as they occur.

If another user has locked a record that you're attempting to edit, you'll receive the following error:

```
Record locked by another user.
Table: <tablename>
User: <other user name>.
```

To trap for a conflicting record lock, place code in the *OnEditError* event. The record lock is returned as an EDBEngineError, with a code of 10241 (see Figure 5).

Transaction Support

As originally designed, the Paradox file format didn't support transactions. Starting with Delphi 2, however, the BDE is able to support transaction statements on Paradox tables.

It does so by setting up a change log that's private to each user. If you're operating under explicit transaction control and you modify a record, the BDE will place a lock on that record, write a copy of the original record to the local log file, and make the change in the actual table. The record lock isn't released, even if other records and other tables are modified. If you commit your transaction — which happens more often than rollback — the record locks are released, and the old versions in the log file are discarded. If you roll back your transaction, the old version in the log file is written to the table before the record locks are released.

For local transactions to work, the transaction isolation property must be set to *tiDirtyRead*, because this is the only mode supported against Paradox tables.

```
procedure TForm1.Table1EditError(DataSet: TDataSet;
  E: EDatabaseError; var Action: TDataAction);
var
  err: Word;
begin
  if E is EDBEngineError then
    begin
      err := (E as EDBEngineError).Errors[1].ErrorCode;
      if (err = 10241) then
        begin
          ShowMessage('Another user is editing this item');
          Action := daAbort;
        end;
    end;
end;
```

Figure 5: Trapping for a conflicting record lock.

This level of transaction support has some limitations. For Paradox tables, the BDE provides no automatic recovery after a crash. In fact, a crash has the same effect as a commit, because the changes have already been written to the table. Also, because the lock file can handle a maximum of 255 record locks, the size of such transactions is significantly limited.

Transaction support against local tables should be considered as a full implementation. If you need complete transaction control over changes, you should seriously consider InterBase, or other database-server environments.

Multi-User Limits

The mechanics of the Paradox file format impose a practical limitation on the number of simultaneous users. Although the format theoretically can support up to 300 virtual users, including extra sessions and connections, 40 to 50 intensive users is a more practical limit. On the other hand, this compares favorably with Microsoft's Jet engine (used by Access), which starts to break down with 12 to 15 intensive users.

Dealing with Problems

If you're having trouble accessing Paradox tables on a network, examine these areas:

- If users receive BDE errors about multiple .NET files in use, including errors where locks can't be placed because the directory is controlled by a different .NET file, look for multiple IDAPI.CFG files, each pointing to a different location for the Paradox network-control directory.
- A corrupted PDOXUSRS.NET or PDOXUSRS.LCK file is a common culprit. It may occur if a user terminated the program abnormally, leaving phantom connections and locks. To fix the problem, get everyone out of all BDE applications on the network, and delete the damaged file(s). They will be re-created automatically when needed.
- If one user can access the application, but all subsequent users are locked out, check to see if the first user has mistakenly configured the application directory as his or her Paradox Private Directory.

This concludes our six-part exploration of the Paradox file format. With an understanding of how Paradox tables work, you're now better equipped to use them effectively in your Delphi applications.

My thanks to Bill Todd and Barbara Radomsky for technically reviewing each article in this series, and for providing many valuable suggestions. Any errors of fact or omission, however, are entirely my responsibility. Δ

Dan Ehrmann is the founder and president of Kallista, Inc., a database and Internet consulting firm based in Chicago. He's the author of two books on Paradox, and is a member of Team Borland and Corel's CTech. Dan was the Chairman of the Advisory Board for Borland's first Paradox conference, which evolved into the current BDC. He has worked with the Paradox file format for more than ten years. Dan can be reached via e-mail at dan@kallista.com.

Delphi 2 / QuickReport

By Keith Wood

Extending QuickReport: Part II

Further Tweaks to the Database Grid Component

Last month, we endowed our QuickReport add-on with the ability to skip to any page at the press of a key, and to print only the pages we want. We also added a database grid component that automatically displays fields from the data source. This month, we'll continue to improve the database grid with properties for each column. Along the way, we'll encounter component editors and the saving and reloading of extended component properties.

A "Quick" Recap

Last month's *TQRDBGrid* component offers the functionality of a database grid, but is specifically tailored for use with QuickReport. With automatic formatting of the database fields, it allows the use of a single component, rather than separate ones for each field.

This component takes most of its formatting from the field definitions of the data set attached to the grid's data source. The fields' alignment, column headings, display formatting, and column width can be set by altering the appropriate properties. The font and background color for all the columns are set by properties of the grid. A grid showing column headers for the fields can be attached to the detail grid.

But wouldn't it be nice if we had more control? If we could set the alignment, background color, font, and width for each column individually, we could create some stunning effects. Then we might want to have different properties for our column headings. We'll achieve this by adding a list of column definitions to the grid.

Column Definitions

Each column is set up as an object, *TQRDBGridColumn*. The *FieldName* property identifies the field to be displayed. The *Alignment* and *TitleAlignment* properties set alignment values for the field and column heading, respectively. *Colour*, *TitleColour*, *Font*, and *TitleFont* operate in a similar fashion. The *TitleCaption* determines the column heading to be shown, while the *DisplayWidth* property controls the width of the column.

In each case, however, we want the default values to apply unless specifically altered. These come from the field of the data set or from the grid itself, depending on the property. Furthermore, a value set for a column becomes the default for the heading, unless overridden at that level. To control all this, and keep track of what comes from where, an additional property is used: *AssignedValues* is a set of flags corresponding to each property. This is maintained internally, although it's available externally as read-only information. A *RestoreDefaults* method returns all the column properties to their defaults.

To enable a column to pick up these defaults from the field or grid, it must know where to find them. To this end, the column definition, when created, must be passed a reference to its attached grid.

To understand how the object keeps track of which property comes from where, let's examine the *Colour*/*TitleColour* pair. Setting either of these properties stores the new value, and updates the set of non-default values:

```
{ Set colour, and flag that it's valid. }
procedure TQRDBGridColumn.SetColour(
  clrColour: TColor);
begin
  FColour := clrColour;
  Include(FAssignedValues, gaColour);
end;
```

When retrieving the *Colour* value, this set is checked, and the appropriate color is returned from the internal field or the grid default:

```
{ Get colour - default to grid value. }
function TQRDBGridColumn.GetColour: TColor;
begin
  if gaColour in AssignedValues then
    Result := FColour
  else
    Result := FGrid.Colour;
end;
```

Similarly, the *TitleColour* value is returned from the internal field if the flag appears in the set, or from the *Colour* property, which originates from its own field or the grid:

```
{ Get title colour - default to column or field value. }
function TQRDBGridColumn.GetTitleColour: TColor;
begin
  if gaTitleColour in AssignedValues then
    Result := FTitleColour
  else
    Result := Colour;
end;
```

In this way, the defaults flow through multiple levels to create the desired effect.

These column definitions are then placed into a list, the *Columns* property, within the *TQRDBGrid* component. The list derives from *TList*, overriding the *Add* method to ensure that only appropriate column definitions are placed in it. This means that all subsequent code can assume the required objects are present, and process them without further checking.

Maintaining Definitions

All these column definitions can be maintained by manipulating *TQRDBGridColumn* objects, and adding or removing them from the *Columns* property of the *TQRDBGrid*. This would take a fair amount of code, and is perhaps prone to errors. Because Delphi is a visual environment, a better way to achieve the same result is to visually edit these definitions at design time. To do this, we need to create a component editor.

A component editor can add much functionality to the Delphi IDE: adding items to the context menu that appears when the component is right-clicked, changing the default action when a component is double-clicked, and copying additional formats to the Clipboard. In our case, we'll add a column editor to be displayed when the component is double-clicked.

Our new component editor, *TQRDBGridEditor*, is derived from *TComponentEditor*, and needs only to override the *Edit* method to provide the necessary functionality. Within this method, it first checks that the grid is appropriate for editing — that it isn't a header for another grid, in which case the columns should be changed — and that the grid is attached to an active data set, so that field definitions are available. The editor's *Component* property provides access to the component for which the editor was invoked.

```
{ Show the columns editor for the component. }
procedure TQRDBGridEditor.Edit;
var
  dlgGridColumnEditor: TQRDBGridColumnEditor;
begin
  if TQRDBGrid(Component).HeaderFor <> nil then
    MessageDlg('This grid is a header for ' +
      TQRDBGrid(Component).HeaderFor.Name + '.' + #13#10 +
      'Please alter the column definitions' + #13#10 +
      'for the base grid instead.', mtError, [mbOK], 0)
  else
    if (TQRDBGrid(Component).DataSource = nil) or
       (TQRDBGrid(Component).DataSource.DataSet = nil) or
    not TQRDBGrid(Component).DataSource.DataSet.Active then
      MessageDlg('Please attach this grid to a datasource'+
        #13#10 + 'before altering its column definitions.',
        mtError, [mbOK], 0)
    else
      begin
        dlgGridColumnEditor :=
          TQRDBGridColumnEditor.CreateFor(Application,
                                          Component);
        try
          if dlgGridColumnEditor.ShowModal = mrOK then
            begin
              TQRDBGrid(Component).Invalidate;
              Designer.Modified;
            end;
        finally
          dlgGridColumnEditor.Free;
        end;
      end;
end;
```

Figure 1: Invoking the component editor.

Once the grid has passed these tests, the editor creates the dialog-box interface, attaches it to the current component, and displays it modally. If the **OK** button is clicked, the component is told to re-paint itself, and the design environment is informed that the component has been changed. This latter step must be done to keep the IDE synchronized. The dialog box updates the *Columns* property of the component when the changes are accepted. This allows the editor to be used outside the design environment, as shown in the demonstration program (see Figure 1).

The Column Editor

The column editor's dialog box is similar in appearance to that of the column editor for normal database grids in Delphi 2 (see Figure 2). On the left are all the columns defined so far, with buttons to add individual fields, to add all fields from the data set, and to delete one or all columns. The order of the fields can be rearranged by dragging them to a new position. This order is reflected in the grid.

When a column is selected, its properties are displayed on the right. These are arranged on two tabs corresponding to those for the field itself, and those for its column heading. The values within these properties are shown disabled (grayed) when they reflect the default values from the field, grid, or column, as appropriate. Once altered, they are shown in the normal text color. This allows us to easily see which values we've changed, and which are determined by outside properties. Once a value has been altered, the **Restore Defaults** button is enabled. Clicking it returns all the properties for this column to their defaults.

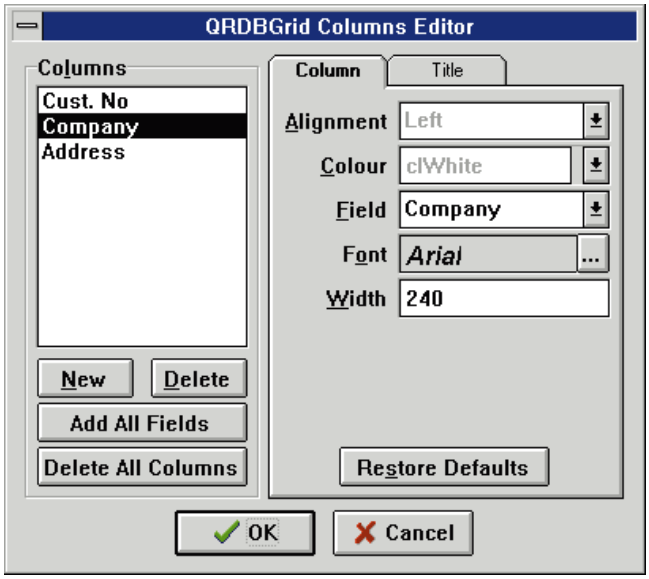


Figure 2: The column editor for the *TQRDBGrid* component.

The color properties can be set by selecting from the drop-down list, by typing in a color as text or hex codes, or by double-clicking the field. This last option invokes the standard Windows color dialog box, allowing the color to be set visually. To obtain the names of the standard Delphi colors, we use the *GetColorValues* callback function, which takes a procedure as a parameter, then passes each color name to the procedure. We want each name to be added to our drop-down list, so we pass along the list's *Add* method. Due to declaration differences between the two, we must first assign the *Add* method to a locally defined function type before casting it as the procedure type expected by *GetColorValues*:

```
type
  TGetStrFunc =
    function(const Value: string): Integer of object;
var
  fnAdd: TGetStrFunc;

{ Add colours by name. }
cmbColour.Items.Clear;
fnAdd := cmbColour.Items.Add;
GetColorValues(TGetStrProc(fnAdd));
```

The font properties can be set by clicking the button at the end of the field, which displays the standard font dialog box. Back on the editor screen, the font is visually displayed in the field.

All the changes can be accepted by clicking the **OK** button, or discarded by clicking the **Cancel** button. Once accepted, the column definitions are copied back to the component for which the dialog box was opened.

Showing Off

The *Paint* and *Print* methods of the *TQRDBGrid* component must be updated to use the new details held in the *Columns* property. In both cases, the code remains substantially the same as before the columns' arrival. We still want to retain the default behavior of getting all the formatting from the data set and its fields. When we process these new columns, only the source of the display parameters changes; their processing does not.

If there are no column definitions, we continue to access the data set and step through each of its fields, looking for and printing those that are visible. If there are column definitions instead, we step through each of these in turn, and use the values to format the fields on the screen or report.

Persistent Columns

All this is great! We can alter each property of each column, and have our grid appear in multiple fonts and colors — all of which are reflected in the report when we preview it by double-clicking on the *TQuickReport* component. But when we close and reopen the report, all has been lost! The carefully crafted definitions in our *Columns* property weren't saved with the rest of the component.

To overcome this, we need to inform Delphi that we want to save additional information with the component, and provide methods for writing and reading these details. This is done by overriding the *DefineProperties* method. By default, this notifies Delphi of all the published properties of the component, and arranges for them to be automatically saved and reloaded.

First, call the inherited method to retain this default processing before adding the *Columns* property to the list. Reading and writing methods need to be supplied, along with a test for when the property contains valid details:

```
{ Set-up for saving Columns property. }
procedure TQRDBGrid.DefineProperties(Filer: TFiler);
begin
  inherited DefineProperties(Filer);
  Filer.DefineProperty('Columns', ReadData, WriteData,
                       Columns.Count > 0);
end;
```

Now describe how to save all the data from the column definitions, then retrieve them. Because they are stored as objects within a list, we need to step though each one and save all its properties before proceeding.

Start the process by telling Delphi that a list will follow, using the *WriteListBegin* method of *TWriter*. Then, for each object in the list, write the name of the field in this column. Next, save only those properties that are different from the defaults. These are identified through elements in the *AssignedValues* set. There is no method to write out a set as such, so we convert it to a string of "Y" and "N" flags by stepping through all the possible elements, and testing for their presence in the set. This string is written out.

Finally, write out a representation of each non-default column property to the component stream. The font properties have embedded values, and are expanded in a similar fashion, when necessary. Last, we close the list with a call to *WriteListEnd*. All this is shown in Listing Five beginning on page 39.

Reading these values back into the property is a similar process. Clear any existing column definitions, and move past the beginning of the property list. Then, while columns are

```
object qrdbgCustomer: TQRDBGrid
  Left = 32
  Top = 0
  Width = 400
  Height = 20
  AlignToBand = False
  BorderStyle = bsVertical
  DataSource = dsCustomer
  Columns = (
    'CustNo'
    'NNNNYYYN'
    0
    'Cust No.'
    'clSilver'
    'Company'
    'NYYYNNYN'
    'clRed'
    'clBlack'
    -13
    'Times New Roman'
    10
    'YNNN'
    140
    'clSilver'
    'Addr1'
    'NNNNNYYN'
    'Address'
    'clSilver'
    'Addr2'
    'NNNNNYYN'
    ''
    'clSilver'
    'City'
    'NNNNNNYN'
    'clSilver'
    'State'
    'NNNNNNYN'
    'clSilver'
    'Zip'
    'NNNNNNYN'
    'clSilver')
end
```

Figure 3: Text representation of the properties for a *TQRDBGrid* component (from the .DFM file) showing the column definitions.

still to be read, retrieve the field name for a column, along with the flags indicating which of its properties are in default. From the stream, read the value of each property that isn't defaulted. Fonts are built up from their constituent parts — the reverse of the writing process.

All these values are used to create a new column-definition object that's added to the grid's list, on completion. Finally, read past the end-of-list marker, and prepare for the next property or object in the component stream. The code for the reading process is shown in Listing Six on page 40.

By opening the .DFM file for a form containing a *TQRDBGrid* whose columns have been defined, we can see the effects of all this processing (see Figure 3). The property is identified by its name, *Columns*, and is followed by a list of internal properties, denoted by the parentheses. Within these are the definitions of each column: the name of the field to show, followed by the flags showing which fields are different from the defaults, and then the values of those fields.

Columns in Action

The demonstration program that accompanied the previous article contained a button, entitled **Columns**, that invoked the columns editor for the detail database grid in the report. Clicking this button has the same effect as double-clicking the grid in the IDE. The resultant dialog box allows columns to be added, removed, or rearranged, and their properties to be altered. Once these changes are saved, they affect the appearance of the report when it's next generated.

Try changing each of the field properties, and observe their effects. Values set for the field flow through to the header, unless overridden. A sample report is shown in Figure 4. Note that the heading on the first column is aligned to the left, while the data is right-aligned. The font for the Company column has also been altered, with a different one again in the heading. The red color of this column stems from the *OnPrint* event for the grid, as described last month.

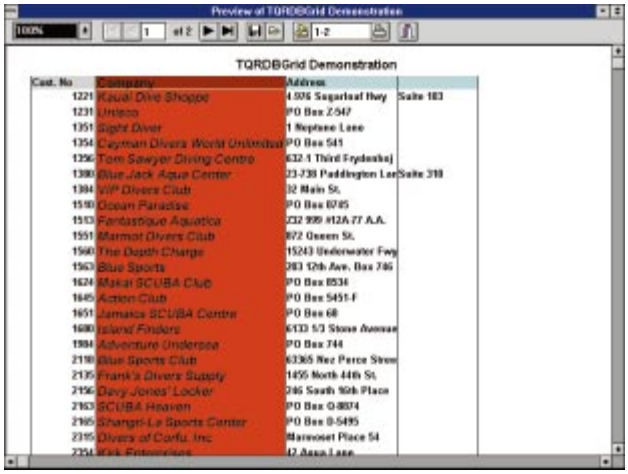


Figure 4: A sample report showing columns with differing alignments, colors, and fonts.

Conclusion

The *TQRDBGrid* component provides a quicker way than ever before for getting data out of the database and onto the page. It easily displays all the fields of the attached data set, taking its formatting instructions from the field definitions.

Now we've extended this component even further, allowing each column to have its own properties. Along the way, we've seen how to create a component editor, and how to save and reload extended properties within Delphi. Of course, the next step is to add code to deal with memo and graphic columns within the grid. But that's another story ... Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\SEP\DI9709KW.

Keith Wood is an analyst/programmer with CSC Australia, based in Canberra. He started using Borland's products with Turbo Pascal on a CP/M machine. Occasionally working with Delphi, he has enjoyed exploring it since it first appeared. You can reach him via e-mail at kwood@netinfo.com.au or by phone (Australia) at 6-291-8070.

Begin Listing Five — Saving column definitions

```
{ Write the contents of the columns list. }
procedure TQRDBGrid.WriteData(Writer: TWriter);
const
  cPresent: array [False..True] of Char = ('N', 'Y');
var
  i: Integer;
  sAssigned: string[10];
  gaValue: TQRDBGridAssignedValues;

  { Write out the details of a font. }
  procedure WriteFont(fntFont: TFont);
  var
    sStyle: string[10];
    fsStyle: TFontStyle;
```

```
  begin
    with fntFont do begin
      Writer.WriteString(ColorToString(Color));
      Writer.WriteInteger(Height);
      Writer.WriteString(Name);
      Writer.WriteInteger(Size);
      sStyle := '';
      for fsStyle := Low(TFontStyle) to High(TFontStyle) do
        sStyle := sStyle + cPresent[fsStyle in Style];
      Writer.WriteString(sStyle);
      { String of N/Y representing styles. }
    end;
  end;

begin

  Writer.WriteListBegin;
  for i := 0 to Columns.Count - 1 do
    { Process all columns. }
    with TQRDBGridColumn(Columns.Items[i]) do begin
      Writer.WriteString(FieldName);
      sAssigned := '';
      for gaValue := Low(TQRDBGridAssignedValues)
        to High(TQRDBGridAssignedValues) do
        sAssigned :=
          sAssigned + cPresent[gaValue in AssignedValues];
      Writer.WriteString(sAssigned);
      { String of N/Y representing attributes. }
      if gaAlignment in AssignedValues then
        Writer.WriteInteger(Ord(Alignment));
      if gaColour in AssignedValues then
        Writer.WriteString(ColorToString(Colour));
      if gaFont in AssignedValues then
        WriteFont(Font);
      if gaWidth in AssignedValues then
        Writer.WriteInteger(DisplayWidth);
      if gaTitleAlignment in AssignedValues then
        Writer.WriteInteger(Ord(TitleAlignment));
      if gaTitleCaption in AssignedValues then
        Writer.WriteString(TitleCaption);
      if gaTitleColour in AssignedValues then
        Writer.WriteString(ColorToString(TitleColour));
      if gaTitleFont in AssignedValues then
        WriteFont(TitleFont);
    end;
  Writer.WriteListEnd;
end;
```

End Listing Five

Begin Listing Six — Reading column definitions

```
{ Read the contents of the columns list. }
procedure TQRDBGrid.ReadData(Reader: TReader);
var
  i: Integer;
  sAssigned: string[10];
  colColumn: TQRDBGridColumn;
  { Read in the details of a font. }
  function ReadFont: TFont;
  var
    sStyle: string[10];
    fsStyle: TFontStyle;
  begin
    Result := TFont.Create;
    with Result do begin
      Color := StringToColor(Reader.ReadString);
      Height := Reader.ReadInteger;
      Name := Reader.ReadString;
      Size := Reader.ReadInteger;
      sStyle := Reader.ReadString;
      { String of N/Y representing styles. }
      for fsStyle := Low(TFontStyle) to High(TFontStyle) do
        if sStyle[Ord(fsStyle) + 1] = 'Y' then
          Style := Style + [fsStyle];
    end;
  end;

begin
  Reader.ReadListBegin;
  Columns.Clear;
  while not Reader.EndOfList do begin
    { Process all the saved columns. }
    colColumn := TQRDBGridColumn.Create(Self);
    with colColumn do begin
      FieldName := Reader.ReadString;
      sAssigned := Reader.ReadString;
      { String of N/Y representing attributes. }
      if sAssigned[Ord(gaAlignment) + 1] = 'Y' then
        Alignment := TAlignment(Reader.ReadInteger);
      if sAssigned[Ord(gaColour) + 1] = 'Y' then
        Colour := StringToColor(Reader.ReadString);
      if sAssigned[Ord(gaFont) + 1] = 'Y' then
        Font := ReadFont;
      if sAssigned[Ord(gaWidth) + 1] = 'Y' then
        DisplayWidth := Reader.ReadInteger;
      if sAssigned[Ord(gaTitleAlignment) + 1] = 'Y' then
        TitleAlignment := TAlignment(Reader.ReadInteger);
      if sAssigned[Ord(gaTitleCaption) + 1] = 'Y' then
        TitleCaption := Reader.ReadString;
      if sAssigned[Ord(gaTitleColour) + 1] = 'Y' then
        TitleColour := StringToColor(Reader.ReadString);
      if sAssigned[Ord(gaTitleFont) + 1] = 'Y' then
        TitleFont := ReadFont;
    end;
    Columns.Add(colColumn);  { Add to columns. }
  end;
  Reader.ReadListEnd;
end;
```

End Listing Six

Delphi 2 / Delphi 3 / Object Pascal

By Robert Vivrette

Dispatches from the Delphi Front

Geometry, Hotkeys, a New Cool Way to Crash, and More...

Handy Math Functions

Delphi 2 and 3 have a very handy unit called Math that adds many powerful mathematical functions to your Delphi repertoire. For example, many Delphi programmers working with geometric equations calculate the Sine and Cosine of an angle using:

```
S := Sin(MyAngle);
C := Cos(MyAngle);
```

But there's a faster way! The Math unit has a procedure called *SinCos*, which does both calculations simultaneously:

```
SinCos(MyAngle,S,C);
```

Here, the resulting Sine and Cosine values are stored in the floating point variables *S* and *C*, respectively. If you need the Sine and Cosine of an angle, using this function is twice as fast as calling the *Sin* and *Cos* functions separately. Many other handy functions cover topics such as angle conversions, exponentials, statistics, and finance.

Another handy function is *MulDiv*, which is located in the file \Source\Rtl\Win\-windows.pas. It multiplies two 32-bit values, then divides the 64-bit result by a third 32-bit value. Sometimes formulas must work with very large numbers, thus a formula can overflow the bounds of a 32-bit integer. This function uses 64-bits to hold an intermediate result, thus avoiding the problem. Also, functions such as *Int32x32To64* and *UInt32x32To64* multiply two signed or unsigned 32-bit values returning a 64-bit result. Check out Delphi's online Help for complete details.

Figure 1: Microsoft's Developer Network Online.

Faster Integer Math

How often do you take an integer and divide it by eight? Or multiply it by two? You might be inclined to do something such as:

```
A := B div 8;  { Integer divide by 8.}
C := D * 2;    { Multiply by 2.}
```

If the result is strictly integers, you can do this much faster by shifting the bits in the variable left or right. Each time you shift an integer left one bit, you are actually *multiplying* it by two. If you shift an integer right one bit, you are *dividing* it by two. Bit shifting is a simple task for the CPU, and it rarely requires more than a clock cycle or two (as opposed to a division or multiplication operation that often requires 10 or more cycles). Each multiple of two is another bit shift left or right, but still takes the same amount of time to complete. So to divide by eight you would shift right three bits. Here is how our code would look using bit shifting instead of multiplication and division:

```
A := B shr 3;
{ Shift right 3 bits; same as integer
  divide by 8.}
C := D shl 1;
{ Shift left 1 bit; same as multiplying by 2.}
```

Note that this technique only works when you are multiplying or dividing in multiples of two, and if the math is on integer values.

Quick Keyboard Shortcuts

Just a quick tip for those who may still be living in the dark ages. There are many shortcuts I find indispensable. For example, selecting a block of text, then hitting

Ctrl Shift I or Ctrl Shift U indents or un-indents the text by a single character respectively.

Also, there is a rudimentary macro capability in all versions of Delphi. Hitting Ctrl Shift R turns on macro recording. Then you can type some text, move the cursor, ect., then hit Ctrl Shift R to turn the recording off. Now to play back the macro, you simply hit Ctrl Shift P. Pretty Handy!

Danger Will Robinson!

A co-worker of mine recently discovered an interesting "feature" of Windows 95 that deserves a nice neon warning label. He was using Explorer to organize a large directory of .PAS files. Once selected, he was going to copy them to another directory when he inadvertently hit Enter. Windows then tried to fire up an instance of Delphi 3 for each of the 78 .PAS files. Initially it was amusing, but the system was quickly brought to its knees after about 54 copies of Delphi were running. Time to hit the big red power button ...

A Wealth of Information

Windows programmers are always looking for "that certain way" to solve a programming problem. Often it's just a matter of knowing where to look. One of the best sources of technical information on Windows programming is the Microsoft Developers Network. This is a subscription-based service that provides developers with quarterly CDs of all sorts of great technical documents on virtually any topic. A powerful search engine allows you to quickly find material.

Microsoft has recently created an online version of this service called Developer Network Online at http://www.microsoft.com/msdn (see Figure 1). Another great site for programming information is the Microsoft Knowledge Base at http://www.microsoft.com/kb/default.asp (see Figure 2).

Creating an Animated Line

Windows has an interesting procedure called *LineDDA* that allows you to create custom line styles. You can even use this technique to create animated lines.

To begin, take a Timer component and drop it on a form. Set the *Interval* to 100. This will control the speed at which the line will animate.

Now, for *LineDDA* to work, we need a callback function. This callback function will define how a line is drawn, pixel by pixel. For each call to *LineDDA*, Windows will call the callback function once for each pixel along the line to be drawn. For example, if the line is 40 pixels long, the callback will be called 40 times with the appropriate x and y values for each of the pixels along the line. All we need to do is come up with some system to determine if a particular pixel is drawn.

To do this, I am using a byte variable called *BitCounter* and repeatedly rotating a single bit through it. Initially, *BitCounter* is set to 1. Each time the callback function is referenced, the bit will be shifted left one position. When it "falls off" the left side of the byte (position 8), it's put back on position 1.

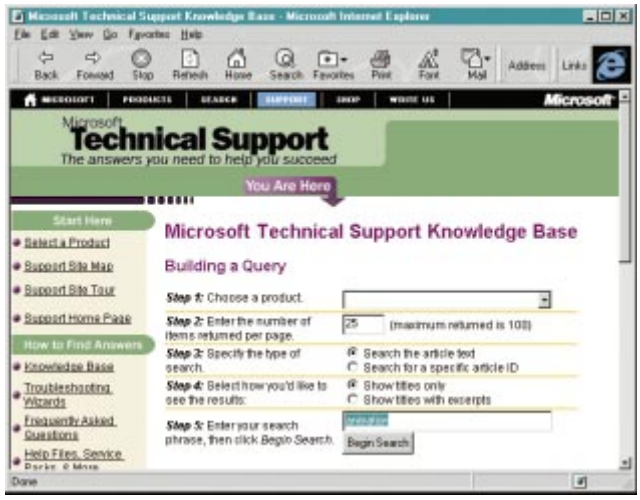


Figure 2: Microsoft Knowledge Base provides great information for developers.

Then all we do is check to see if the bit is in a position that tells us we want a pixel painted.

I am using a mask value of 224, which is the left-most three bits in a byte, i.e. 11100000. When our rotating bit falls in one of those positions, we paint the pixel blue. If not, we erase the pixel by painting it the color of the form. Because we have three bits turned on (the 1's) in our byte, and five bits turned off (the 0's), we'll get a line that alternates three pixels on, then five pixels off. Listing Seven on this page, shows the source code for this technique.

Note also that you don't need to set pixels in the callback. You can draw rectangles, circles, small bitmaps — whatever your imagination can come up with. Remember that the callback must be very fast and efficient, because it's being called so rapidly. Any lengthy processing or painting will quickly slow the application to a crawl. Δ

Robert Vivrette is a contract programmer for Pacific Gas & Electric, and Technical Editor for *Delphi Informant*. He has worked as a game designer and computer consultant, and has experience in a number of programming languages. He can be reached via e-mail at RobertV@csi.com.

Begin Listing Seven — The *LineDDA* procedure

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    { Private declarations }
```

```
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
  BitCounter: Byte;
  CanvasColor: TColor;

implementation

{$R *.DFM}

procedure MovingDots(X,Y: Integer;
                     TheCanvas: TCanvas); stdcall;
begin
  // Shift the bit left one.
  BitCounter := BitCounter shl 1;
  if BitCounter = 0 then
    // If it shifts off left, reset it.
    BitCounter := 1;
  // Are any of the left 3 bits set?
  if (BitCounter and 224) > 0 then
    // Draw the pixel.
    TheCanvas.Pixels[X,Y] := clBlue
  else
    // Erase the pixel.
    TheCanvas.Pixels[X,Y] := CanvasColor;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  BitCounter := 1;
  CanvasColor := Color;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  LineDDA(10,10,80,80,@MovingDots,LongInt(Canvas));
end;

end.
```

End Listing Seven

By *John Gmutza*

Design for Many Applications, etc.

Delphi Tips and Techniques

Don't Design for One Application

A lick and a promise. It's amazing how many programmers write code as if they will only use it once. The illusion is that time is saved by quickly knocking out a routine, planning to return to it later to make it a reusable class. More often than not, however, later doesn't come. You end up programming it again somewhere else, or cutting and pasting — if you can.

Create classes instead. Specialized classes provide a way to concentrate on narrow bits of functionality, and can help clarify your design. Later, you may be able to identify one of those specialized classes for reuse.

Here's a classic example: We were computing sample statistics for a dataset in a graphing application, without a class — only code fragments, as required — throughout the application. After we realized we would also be computing multiple datasets, two alternatives appeared:

1) use parallel arrays (the quick way), or
2) create a class to manage the sample statistics in an array.

The second alternative results in a useful class, a clearer design in the graphing application, and a chance to reuse the class down the road (see Figure 1). The moral is to think in terms of encapsulation. The common task of computing summary data was transformed from many scattered code fragments, to a cohesive class. Many other concepts can be treated in the same way.

When to use a class. First, identify the special case you're working with, then create the base class. At first there doesn't seem to be a motivation for creating a base class *and* one derived class; it appears to be extra work.

```
type
  SeriesData = class
  private
    v_count: Integer;
    v_min: Double;
    v_max: Double;
    v_total: Double;

    function get_Ave: Double;

  public
    constructor Create;
    procedure Init;
    procedure Add(val: Double);

    property Count: Integer read v_count;
    property Min: Double read v_min;
    property Max: Double read v_max;
    property Ave: Double read get_Ave;
  end;

...
implementation
...

constructor SeriesData.Create;
begin
  Init;
end; { Create }

procedure SeriesData.Init;
begin
  v_count := 0;
  v_min   := 999999999.99;
  v_max   := 0.0;
  v_total := 0.0;
end; { Init }

procedure SeriesData.Add(val: Double);
begin
  v_count := v_count + 1;
  v_total := v_total + val;
  if val < v_min then
    v_min := val;
  if val > v_max then
    v_max := val;
end; { Add }

function SeriesData.get_Ave: Double;
begin
  if v_count > 0 then
    Result := v_total/v_count
  else
    Result := 0.0;
end; { get_Ave }
```

Figure 1: A useful class.

Why not just create one class? Because you're banking on the fact that somewhere down the road, quite often in the same application, you will need to create the second and third derived classes. And those second and third instances of reuse might help refine the interface of the base class, further enhancing its ability to be reused.

Don't Auto-Create Forms

Although it's easy and requires no work, don't let Delphi auto-create all the forms in your application. It's a waste of system resources.

To do this, select **Project | Options** to display the Project Options dialog box. On the Forms page, move all forms except the main form (and data module if you have one) to the **Available forms** list box. Then in each unit with a dialog box, remove the Delphi-generated variable at the end of the **interface** section. To access the dialog boxes, you must create and destroy them explicitly:

```
var
  dlg: TSomeDialog;
  Owner: TComponent;
...
dlg := TSomeDialog.Create(Owner);
try
  if dlg.ShowModal = mrOk then
    begin
      ...
    end;
finally
  dlg.Free;
end;
```

Repeat as necessary for each new form you create.

Use Exception Handling

Don't be afraid of exception handling. When seemingly innocuous functions such as *StrToInt* can raise an exception, you don't have a choice. We all know where the program goes when no exception handlers are active. So the benefits of using exceptions are clear; they localize the effects of errors, and provide an easy way to recover.

When you have the debugger set to **Break on Exception**, you can use F8 to step to the exception handler for that exception. If you've done a good job, you should be close to where the error occurred, i.e. in the nearest **try** block. Now you have a place to set a breakpoint for the next time through.

My favorite place to use exceptions is when making changes to a database:

```
var
  Table1: TTable;
try
  Table1.Append;
  Table1Field1.Value := 1.34;
  Table1Field2.Value := 'Delphi';
  Table1Field3.Value := Now;
  Table1.Post;
except
  Table1.Cancel;
  Application.HandleException(Self);
end;
```

This code sequence is quite robust. If anything bad happens while trying to append a new record, the attempt is aborted, and a dialog box is displayed with a system-defined error message.

Another good time to use exceptions is when opening tables:

```
var
  Table1: TTable;
  Table2: TTable;
  Table3: TTable;
  Table4: TTable;
...
try
  Table1.Open;
  Table2.Open;
  Table3.Open;
  Table4.Open;
except
  Application.HandleException(Self);
end;
```

A good place to use the **finally** clause is when performing dynamic memory allocation. A typical use is creating and displaying a dialog box:

```
var
  dlg: TSomeDialog;
  Owner: TComponent;
...
dlg := TSomeDialog.Create(Owner);
try
  if dlg.ShowModal = mrOk then
    begin
      ...
    end;
finally
  dlg.Free;
end;
```

This ensures that no matter what goes wrong while running the dialog box (or after), the dialog object's memory gets deallocated.

Raise Exceptions When They Make Sense

Now that you're fielding exceptions in your code, it's safe to start raising them as well. In certain *TTable* event handlers, such as *BeforeAppend* and *BeforePost*, the only way to abort the append or post operation is to raise an exception:

```
var
  bad_condition: Boolean;
...
procedure SomeClass.Table1BeforePost(Dataset: TDataset);
begin
  if bad_condition then
    raise Exception.Create('Bad Condition is True');
  else
    begin
      { Useful stuff }
    end;
end;
```

When writing components for others, it's good to raise exceptions when the component is not prepared to deal with the current conditions. It's also good to declare unique classes for these exceptions.

For example, let's use a *EBadCompClass* exception from our *SomeComponent*, when it's passed a control it can't work with.

```
type
  EBadCompClass = class(Exception);

...

procedure SomeComponent.SomeMethod(ctl: TControl);
begin
  if ctl is TEdit then
    begin
    end
  else if ctl is TMaskEdit then
    begin
    end
  else
    raise EBadCompClass.Create(
       'Unexpected Control Class: ' + ctl.ClassName);
end; { SomeMethod }
```

Figure 2: Using an exception, *EBadCompClass*, from our *SomeComponent* object.

If the form designer encounters this exception during development, a decision can be made as to whether the form designer works around it, or the component is enhanced to handle the unexpected case (see Figure 2).

Use Return Values

Many applications go awry because they continue execution despite the fact that a crucial function returned a failure code. The application simply didn't check for the error condition. Hours are spent pinpointing these problems, because the errors they create can be subtle, and can surface far from the code that started the "ripple effect." So many hours have been spent that a market's been created for software to help developers track down these errors.

Don't program as if everything will always work. These applications aren't hard to spot. They're fragile; the user is given little room for mistakes. This usually results in a negative experience with the application. "Easy to program" usually translates into "hard to use," and not checking return values is easy.

Don't fall into this trap! You will be well rewarded for the time you spend making your applications more robust. Think of it as reducing the time you'll spend fixing things. Every function that can return a fail code will fail under some circumstances. Otherwise, failure wouldn't be one of the return values.

Cut-and-Paste Components without Losing Event Handlers

How many times have you moved a component from one form to another, only to lose its event handler definitions? I was getting tired of manually hooking things back together, when I discovered this trick. Cut-and-paste the event handlers first, before working on the component. Delphi sees the declarations in the target unit, and leaves all the event handler properties intact.

Here are the steps:

1) On the Events page of the Object Inspector, identify all the component's event handlers you want to cut to the Clipboard.
2) In the component's unit, locate the event handlers' declarations, and cut-and-paste them into the target unit's **class** declaration. Put them in the first section of the **class** declaration, where Delphi places declarations that it generates for you.
3) In the component's unit, locate the event handlers' definitions, and cut-and-paste them into the target unit's **implementation** section. Be sure to change the class name to the target's class name for each event handler.
4) In the target unit, clean up any references to identifiers used in the source unit. Either eliminate them, or add the necessary units to the **uses** statement of the target unit.
5) Finally, cut-and-paste the VCL component from the source form into the target form. Δ

John Gmutza received a BSCS from Western Michigan in 1985. He has been solving tough problems ever since. John works for Postek Inc., a database consulting firm. His interests include guitars, beer cans, and compilers. You can contact him via e-mail at jgmutza@postek.com.

By Paul Kimmel

Is It Really Disabled?

Did you ever notice, when you disable a Delphi Panel or GroupBox, that the controls on these *TWinControl* components don't appear disabled? Well they are. As an advocate of providing user-friendly software (to a point), I like my disabled controls to *look* disabled. This may reduce confusion and frustration among new or harried users of your product.

You could always write a function that disables each control individually:

```
procedure Enabled(const State: Boolean);
begin
  Panel1.Enabled := State;
  Edit1.Enabled := State;
  // and so on.
end;
```

However, this isn't very developer-friendly, nor very extensible. When adding a new component, you would have to add another line of enabling/disabling code.

Writing a **for** loop that iterates through the controls property will work, and it will solve the problem of adding a new line of code each time you add a control:

```
procedure Enabled(Control: TWinControl;
  const State: Boolean);
var
  I : Integer;
begin
  for I := 0 to Control.ControlCount - 1 do
    Control.Controls[I].Enabled := State;
end;
```

The second version of code will disable all controls on a panel, form, or group box. However, if one of these WinControls has child WinControls, as is the case of a group box on a panel, then we're back to square one. A solution can be found using recursion. Don't panic! Delphi 2 and 3 provide a huge stack; what I have in mind won't even be noticed.

To ensure all the controls are disabled — in lieu of the parentage by a disabled control — let's modify the previous algorithm slightly:

```
procedure Enabled(Control: TWinControl;
  const State: Boolean);
var
  I: Integer;
  AWinControl: TWinControl;
begin
  for I := 0 to Control.ControlCount - 1 do
    begin
      Control.Controls[I].Enabled := State;
    if (Control.Controls[I] is TWinControl)
      then
        begin
          AWinControl :=
            (Control.Controls[I] as
             TWinControl);
        if (AWinControl.ControlCount > 0)
          then
            Enabled( AWinControl, State );
        end;
    end;
end;
```

The function now steps down into any parent *TWinControl* recursively. Effectively, the **for** loop will bounce out of the function if there are no controls on the *TWinControl*, but I opted for performing the check before calling the recursive function. The code would be slimmed down by an **if** conditional check, but you would pay for the price of a function call in the event there are no child controls. Considering the nature of a *TWinControl*, what is the likelihood that a form, group box, or panel will have no children?

Applying this technique will ensure — in one fell swoop — that all disabled controls look disabled. Plus, it's extensible and easy to implement. Δ

Paul Kimmel is the founder of the Okemos, Michigan-based Software Conceptions, Inc., which provides professional, object-oriented software-development consulting and training services world-wide. Paul is the author of, or contributor to, several books about Delphi and C++ programming (published by QUE), and is the father of four children, Trevor, Douglas, Alex, and Noah. Contact Software Conceptions at softcon@sojourn.com.

By *Alan C. Moore, Ph.D.*

Sentry Spelling Checker Engine

Wintertree's Spell-Checker Gets a Thumbs-Up

Spell-checking is an essential component of any word processor. Can you imagine buying one that didn't include this vital element? In addition to word processors, there are quite a few other types of applications that spell-checking would enhance. An obvious example is a programmer's editor.

Adding spell-checking to the text-editing parts of applications may be desirable and advantageous, but it is hardly trivial. That is, until the introduction of tools such as the Sentry Spelling Checker Engine (SSCE) from Wintertree Software. This product — which includes a C/C++ and Visual Basic implementation along with those of Delphi 1, 2, and 3 — provides all the tools for adding powerful spell-checking features to any Delphi application. Furthermore, it provides a variety of ways to do this, giving you the flexibility needed in various programming situations. Let's take a detailed look.

A Good Sentry Must ...

What should we expect from a modern spell-checker? Among other things, it should find and display misspelled words, suggest replacements, allow us to add new words to a user dictionary, ignore one or more instances of special words, and — most importantly — insert our corrections. SSCE includes this functionality and more. Some of the more unusual checking options include repeated words, and reporting or ignoring words with digits, mixed case, or all letters capitalized (see Figure 1). SSCE not only allows you to change these options for a particular spell-checking session, but also stores user preferences in an .INI file or (in 32-bit programs) the Windows registry.

You can access the engine directly through various function calls, or use the dialog-box component that ships with the product. The latter provides access to more specialized dialog boxes for program options, maintaining dictionaries, and adding new dictionaries. If you are writing and supporting multi-lingual applications, you can purchase additional dictionaries for many of the major European languages, including Portugese, Danish, Dutch, Finnish, French, German, Italian, Spanish, and Swedish. There's also a medical dictionary available.

SSCE includes options you may not need for English-language spell-checking, but are essential for other languages. Consider the SSCE_SPLIT_CONTRACTED_WORDS_OPT switch. When set, this option allows the user to check the spelling of sub-words in ad hoc contracted words such as *quell'anno*, checking the spelling of *quell* and *anno,* respectively. This would be useful in a French or Italian language application. What about the concatenation of words that are common in German? Again, SSCE has the answer. By setting the SSCE_SPLIT_WORDS_OPT switch to 1, the English composite word "dumptruck-driver" would be recognized as three correctly spelled words combined into one.

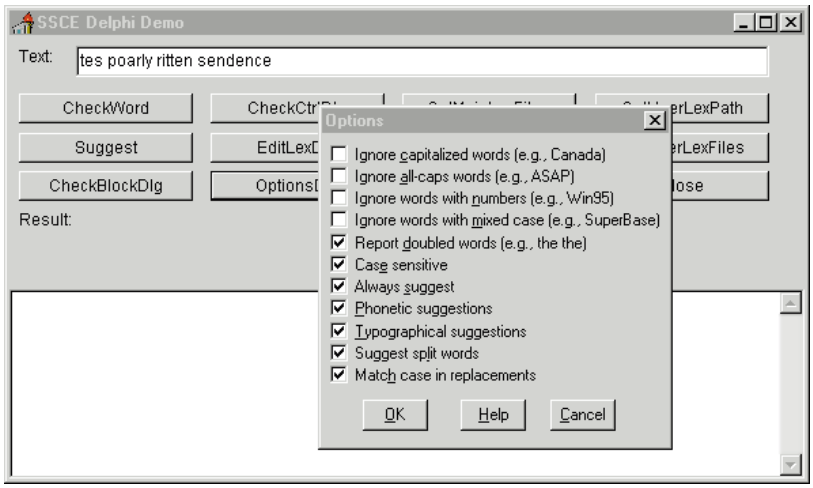


Figure 1: SSCE offers several ready-made options for adding spell-checking functionality.

Selecting Text in a Memo

As powerful as Delphi's Memo components are, considerable functionality isn't implemented. In working with the SSCE, it's essential to be able to select appropriate blocks of text, e.g. words, sentences, and paragraphs. Unfortunately, such capabilities aren't included with this engine, nor is it reasonable to expect that they would be. The example project included with this article shows how to implement such text-selection behavior. Here, we'll select a sentence in a Memo component to display elsewhere.

To set up the application, drop two Memo components and a Button component on a large form (*Height* = `368`, *Width* = `544`.) In both memos, edit the *Lines* property to remove the default text, and set the *ScrollBars* property to *ssVertical*. For Memo1, set the *Top*, *Left*, *Height*, and *Width* properties to `8`, `8`, `233`, and `409` respectively.

For Memo2, set the *Top*, *Left*, *Height*, and *Width* properties to `248`, `8`, `89`, and `409` respectively. Change the caption on Button1 to `Get Sentence`. Double-click on the button or its *OnClick* event in the Object Inspector to create an event handler. Add a single line of code, `GetCurrentSentence`, to this method. Now, with the form having the focus, double-click on the *OnCreate* event to create another event handler, and add the following line of code:

```
Memo1.Lines.LoadFromFile('i:\README.TXT');
```

Be sure to substitute an appropriate text file that actually exists on one of your computer's drives. Rename the unit file as GetSent.pas, and enter the remaining code which is available for download in GetSent.pas (see end of article for details). Now, let's take a look at some of the code used.

The *TMemoInfo* record is the key to locating sentences. It's used throughout to track the absolute location (*AbsOffset*), current line (*Line*), and current column within that line (*Col*). The procedures *GetMemoPos* and *SetMemoPos* find or change the current cursor location in the memo. Note the use of the Windows API functions EM_LINEFROMCHAR and EM_LINEINDEX to find the current line and its cursor offset. The function *FindDelim* searches the current line for an appropriate delimiter (in this instance a period or a blank line). This function is used by the main functions, *FindPreviousPeriod* and *FindNextPeriod*, to return the beginning of the sentence (*SelStart*) and its length (*SelLength*).

One of the challenges in this sort of parsing is tracking white space, e.g. spaces, tabs, etc. Here we need to account for a blank character following a period, or a period that occurs at the end of a line. Many of the key lines of code are commented, explaining the logic involved. While this example is fairly involved, it's still rather crude, as such examples go. To be truly useful, it should be expanded to return paragraphs and words. We could expand our collection of delimiters to include other forms of punctuation. In fact, our sentence parser is somewhat incomplete without the more unusual ending punctuation of "?" and "!". However, this example provides an excellent starting place.

— *Alan Moore*

Installing and Learning the Spell Checker

SSCE installs easily and integrates well into the Delphi environment. It ships with a thorough manual that provides detailed instructions and sample projects. In order to get SSCE to work with Delphi 3, recompile the *.pas file, which is a wrapper for the SSCE DLL. This was not problematic, and by the time you read this article a Delphi 3 version should be available. When you install SSCE in Windows, it sets up a group containing a utility and an example program. Before running these programs, you need to verify that the main .DLL — SSCE4216.DLL for Delphi 1 or SSCE4232.DLL for Delphi 2 or 3 — is installed in the Windows directory. Unfortunately, the installation program didn't accomplish this automatically. Following the directions in the manual, I copied the file manually. (In the new version of SSCE, this is handled automatically.) After everything is properly installed, you can quickly get the feel for SSCE's capabilities and its standard user interface by running the example program.

SSCE includes WinSqLex, a utility that translates a set of words in one or more ASCII text files into a compressed form used by the SSCE. Figure 2 shows this utility application and the text file it's reading, DelpTerm.txt (a list of some of the controls in Delphi's Visual Component Library). For a large list of terms with common suffixes, you can enhance the compression process by adding a suffix file. WinSqLex can compress lexicons by replacing common suffixes with special codes. A compressed lexicon can contain up to 288 suffixes.

Also installed with SSCE, the Dialogs Demo shows the operation of each built-in dialog box in a very simple setting: opening, spell-checking, and saving a .TXT file. It provides access to the preferences dialog box, dictionaries dialog box, and built-in Help system. Keep in mind that if any user-interface elements don't meet your needs, you can create new elements using the basic SSCE API. However, before we discuss the details of the SSCE API, let's take a look at SSCE's lexicon structure, which is the heart and soul of this engine.

Lexiconography 101

The Main Lexicon (which is normally compressed) is an Ignore-Type Lexicon — the words contained are properly spelled. SSCE includes an American English and British English version of the Main Lexicon (in both text and compressed versions). The opposite, an Exclude-Type Lexicon, consists of misspelled words. Such a lexicon is useful in removing certain words (such as slang terms) from other lexicons.

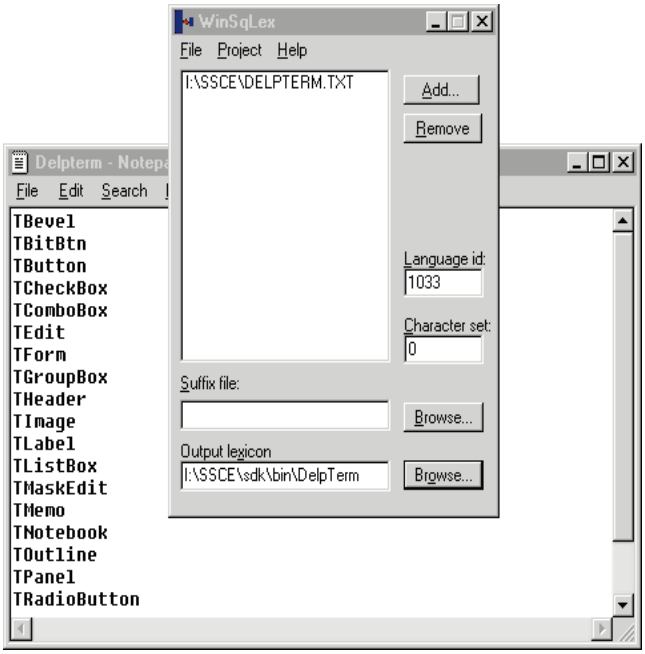


Figure 2: The WinSqLex utility translating a text file into a compressed form for SSCE.

Another useful type, the Change-Type Lexicon, has pairs of words, the first of which is a common misspelling followed by its correction (i.e. "tehn" and "then"). The limitation of such a lexicon is that you're forced to accept the given correction. What if, in the previous example, you intended to use the word "ten"? In that case, yet another type, the Suggest-Type Lexicon, works best (see Figure 3).

SSCE comes with 10 ready-to-use lexicons that can be distributed with any application. In addition to the four main lexicons, this collection includes examples of four user types: Change, Exclude, Ignore, and Suggest.

In working with lexicons, there is one important caveat discussed in the manual that is definitely worth repeating. While you can always compress a lexicon, you can't convert a compressed lexicon back to its text form. So, anytime a new lexicon is created, always keep a copy of its text version.

We have examined the different kinds of lexicons that provide the basic data SSCE uses to spell-check documents, but how do we add such capabilities to our programs? We have two choices: Work directly with the SSCE API, or use the ready-made dialog boxes. While the first approach provides a great deal more flexibility and power, the latter is certainly much easier and is workable in many circumstances. Let's take a brief look at each approach.

The Hard Way: Using the Basic SSCE API

When using the SSCE API, you must perform the required setup and close procedures manually. First you need to open a spell-checking session with *SSCE_OpenSession*. Then open a lexicon by calling *SSCE_OpenLex* (up to 16 lexicons can be open simultaneously).

Now you're ready to get down to the business of checking the spelling. Select a block of text to check; it can be as

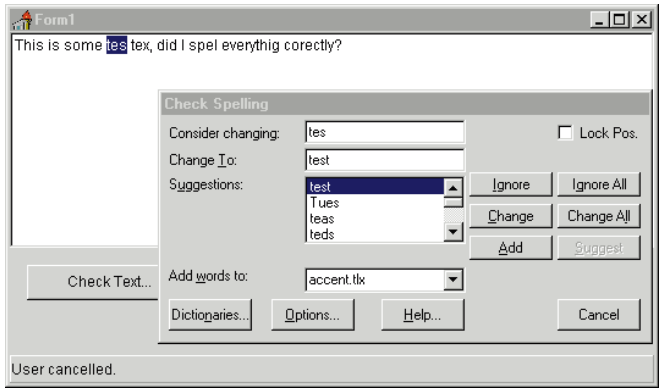


Figure 3: Here the user can pick from several suggested words.

short as a single word, or as long as an entire text file. Unfortunately, there are very few (if any) built-in tools for anything between, such as checking a sentence or paragraph. If you want to make such options available to users, you'll have to write the code to return a sentence or paragraph as the current block. (See the sidebar "Selecting Text in a Memo" on page 49 for a demonstration of how to select text below the cursor.)

Having selected the block of text, open it by calling *SSCE_OpenBlock*. The declaration of this function is:

```
function SSCE_OpenBlock(sid: S16; block: SSCE_PCHAR;
  blkLen, blkSz: S32; copyBlock: S16): S16;
```

Confused? I'm not surprised. This declaration comes from the Delphi file ssce.pas, the main interface to the spell-checking engine. The declaration in the manual is even more confusing:

```
S16 SSCE_OpenBlock(S16 sid, SSCE_CHAR *block,
                   S32 blkLen, S32 blkSz, S16 copyBlock);
```

If you have programmed in C, the syntax of the last declaration should be familiar. In fairness to Wintertree, the manual makes a valiant effort to explain everything, and it does a pretty good job. Because SSCE supports not just Delphi, but VB and C, the manual should have included comparable declarations for the other languages. Also, the presentation of special types is difficult to understand. The reason for using these types is undoubtedly to provide a basis for multi-language programming support. It would have been easier to understand the API declarations if they'd been expressed as basic Delphi types (Shortint, Long, etc.) rather than abstract types (S16, U32, etc.).

The parameters to the *SSCE_OpenBlock* function include the current *Session*, a PChar pointing to the memory block, the block which contains the text to be checked, the initial size of the text block, and so on. It returns a value greater than or equal to zero if successful, or a negative value if there's a problem (e.g. insufficient memory).

Working with Blocks of Text

Once you've opened a block of text, SSCE provides a number of functions for checking the text. The main block-

Wintertree's ThesDB Thesaurus Engine

In addition to its Sentry Spelling Checker Engine (SSCE), Wintertree offers ThesDB, a thesaurus engine. While a lexicon is simply a list of words, an electronic thesaurus is more complex in nature. It consists of word categories: synonyms (words with similar meanings), antonyms (words with opposite meanings), and word classes (parts of speech, such as noun or verb). For example, to build a programmer's thesaurus, you'd include the word "build". Usually, this word is a verb fitting the word category build.v. However, programmers use the word "build" as a noun. So, we'd need a new category, build.n, to designate this meaning (see Figure A).

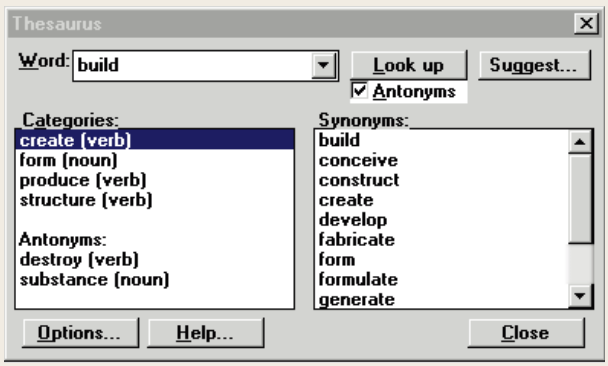


Figure A: A look at the ThesDB in action.

As with its SSCE, Wintertree's ThesDB provides a low-level and high-level interface. The low-level interface gives the most control over access to the inner workings of the engine and thesaurus. The high-level interface provides a built-in dialog box and various Windows function calls that encapsulate a good deal of the engine's functionality. But the similarities don't end there.

If you're familiar with SSCE, learning ThesDB will be very easy, almost a déjà vu experience. Here are a few more similarities:

- Ability to work with text and compressed files.
- Ability to replace misspelled key words.
- Customization features, such as adding or deleting words from user thesaurus.
- Built-in distributable Help file with .RTF source code.
- Configuration with .INI file or the Windows registry.
- Detailed printed and online documentation.
- ANSI C source code available for purchase.

While there are many similarities between the two products, there is one important difference: The structure of a thesaurus is a bit more complex than a lexicon. Thus, there are more details to be concerned about. Fortunately, as with SSCE, most of these details have been taken care of for you. And whichever approach you use, high-level or low-level, there's excellent documentation and example programs to guide you.

— Alan Moore

checking functions are listed in Figure 4. Most of these are quite simple. *SSCE_CheckBlock*, like *SSCE_OpenBlock*, is quite complex with parameters for a session identifier, a block identifier, various bit-masked word-checking options, and four options to manage misspelled words and their replacements. When the end of the block is reached (return value SSCE_END_OF_BLOCK_RSLT), close everything with calls to *SSCE_CloseBlock*, *SSCE_CloseLex*, and *SSCE_CloseSession*.

This probably sounds rather involved, and it is. However, imagine the work involved in coding this from scratch. When you consider there are 14 word-checking options and 10 possible return codes (problem words or errors), the complexity is considerable, indicating the tremendous power of this product. However, to avoid this complexity, there is an easier way ...

The Easy Way: The SSCE Windows API and the SSCEVCL Dialog Box

By using the SSCE Windows API, you can access the functions in the included .DLL through a standard dialog box (SSCEVCL) installed into Delphi's Component palette. This dialog box takes care of most of the details I enumerated previously, making it even easier to use this engine. If you don't need the control offered by using the SSCE API directly, you can simplify your use of the spelling-checker engine considerably by using the SSCE Windows API.

Access to the various dialog boxes (edit lexicons, options, etc.) is provided by the SSCE Windows API. A good deal of the engine's basic functionality is in many of these function calls. This API provides two spell-checking contexts: one is text-block based, the other is control based (i.e. *TMemo*). You can control the configuration files (with .INI files or the Windows registry), work with paths, and gather statistics. Best of all, example files and projects are included to help you learn to use this API quickly and easily. The built-in dialog box (SSCEVCL) has its own Help file included with this product. To my pleasant surprise, the full .RTF source code for the Help file was also included. If you customize the dialog box (rather likely, as we all like to add new functionality), you can explain the added features in an expanded Help file.

Whichever approach you choose — the basic SSCE API or its Windows API and component — Wintertree has included

Function	Purpose
SSCE_CheckBlock	Checks the spelling of a block of text.
SSCE_ReplaceBlockWord	Replaces the current word in a block with another word.
SSCE_DelBlockWord	Deletes the current word in a block.
SSCE_NextBlockWord	Advances to the next word in a block call.
SSCE_GetBlock	Retrieves the text in a block.
SSCE_GetBlockInfo	Obtains a block's size, cursor location, or word count.

Figure 4: The functions for spell-checking a block of text.

an outstanding example program. This certainly helps mitigate the complexity we discussed.

Conclusion

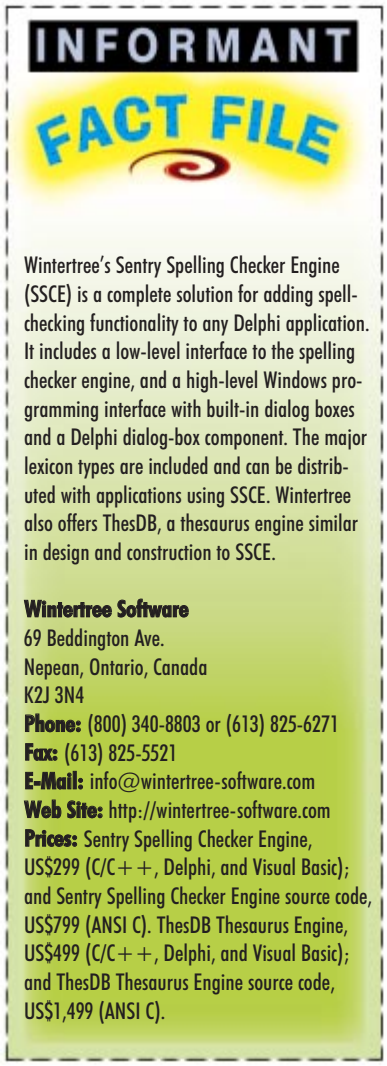
Wintertree's SSCE is a powerful, well conceived, and flexible solution for adding spell-checking capabilities to a Delphi application. With its low-level access to the inner workings of its engine, and its high-level Windows interface, it adapts to a variety of programming situations. This includes building custom spell-checking dialog-box components, adding basic spell-checking functionality to a Memo or RichEdit component, or building a full-featured text editor. Its support for a variety of European languages is excellent.

Wintertree is currently completing a Unicode version of the engine. A Java version of the engine should be ready for release before the end of 1997. Particularly for the developer working in a C/C++ and/or Visual Basic environment that uses Delphi, this product is an excellent choice. If you're writing applications for various platforms other than Windows, the ANSI C source code is available for modification and compilation on any platform that has a C compiler.

For any Delphi programmer looking for a full-featured, powerful, yet flexible spell-checking solution, I recommend this product highly. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\SEP\DI9709AM.

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 on the Internet at acmdoc@aol.com.



By Bill Todd

InfoPower 3

Still the One to Beat

Since its first version, InfoPower from Woll2Woll Software has been the one "must have" add-in product for anyone developing database applications in Delphi. In its third release, InfoPower again brings a host of new features to make developing database applications faster and easier.

What's New

One of the most exciting new components in version 3 is the *TwwRecordViewDialog* component. One of the most useful characteristics of Delphi's grid control is that it dynamically adapts to the dataset it's connected to, so you can easily use a single form with a single grid to edit multiple tables. With *TwwRecordViewDialog*, you get this same flexibility with a form — simply connect this component to any DataSource, call its *Execute* method, and it dynamically builds a data entry form for you.

Figure 1 shows a general-purpose local table browser and editor that uses the *TwwDBGrid* component. After installing InfoPower 3, I dropped a *TwwRecordViewDialog* on the form, set its *DataSource* property, and added a call to its *Execute* method to the grid's *OnDoubleClick* event handler. Now I can double-click the grid and see the current record displayed in a form.

Figure 2 shows some of the flexibility of *TwwRecordViewDialog*. In this example, the Navigator and buttons have been turned off and the form is being shown modelessly. Its style has also been changed from horizontal to vertical to provide a different arrangement of the edit controls.

You get even more control over the form's appearance because the form will automatically use any controls that are defined for a *TwwDBGrid* that is bound to the same dataset. If the grid includes InfoPower's combo box, spin edit, rich edit lookup combo, or check box controls, the same controls will automatically be used for the same fields in the record dialog. Any picture masks you have defined for the dataset will also be applied when you use RecordViewDialog to edit data. If you need a custom menu for the dialog, simply drop a *TMainMenu* component on the form from which you will call the RecordViewDialog, and set the RecordViewDialog's *Menu* property to that menu item.

As with *TwwDBGrid*, RecordViewDialog has a property you can select at design time to set picture masks and change the order in which fields are displayed. This is a real time-saver for any generic editing task, such as lookup tables, where a great deal of specialized functionality is not required.

Figure 3 shows another example of a RecordViewDialog that includes a custom menu and a custom font for the field labels. It also includes another new jewel in InfoPower 3, the *TwwDBRichEdit* component. Of course, this component has all the features found in the Borland rich edit controls, but it also lets the user display an editing window with all the features of a basic word processor, as shown in Figure 4.

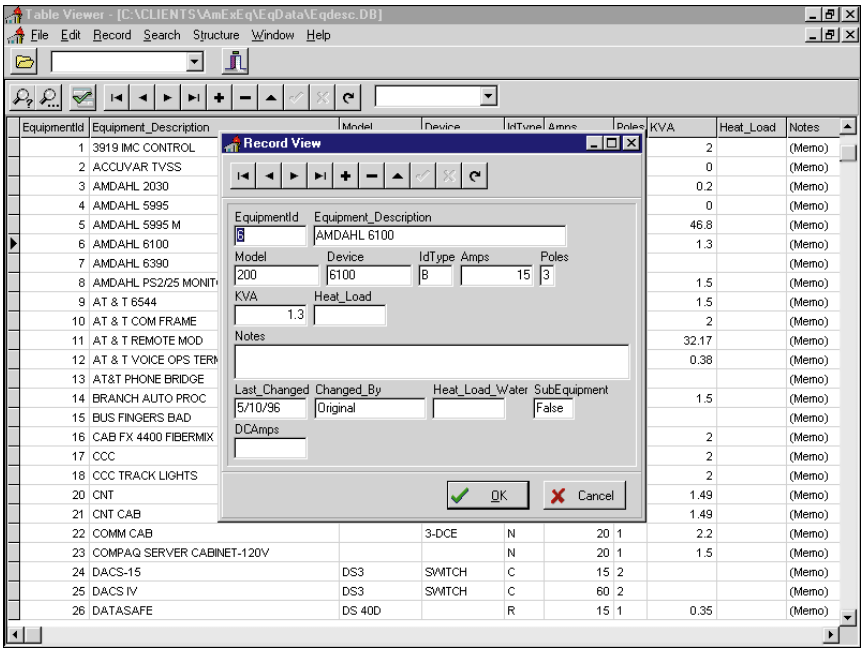


Figure 1: A form generated by *TwwRecordViewDialog*.

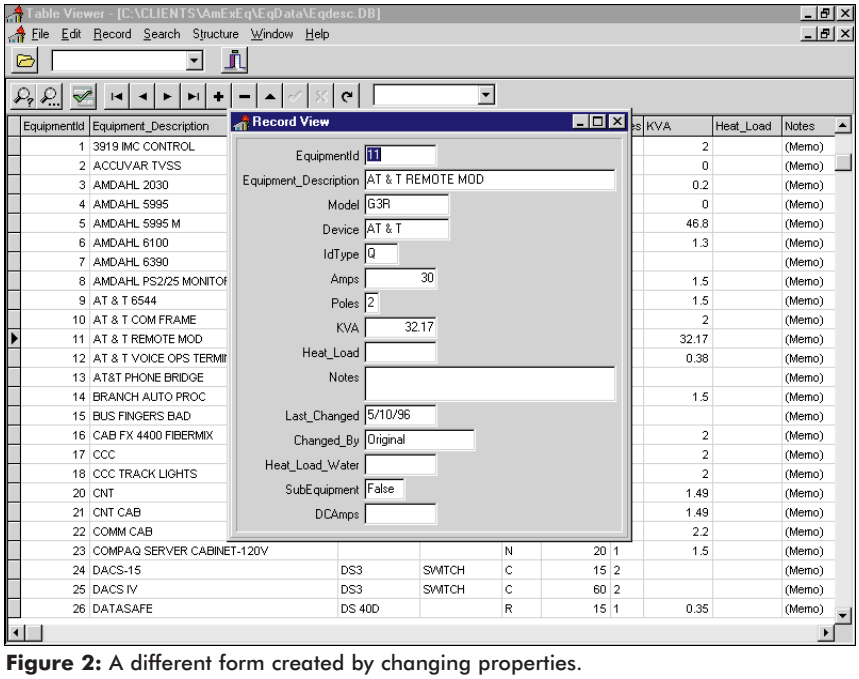


Figure 2: A different form created by changing properties.

The editing window includes a toolbar, format bar, and status bar, which the user can turn on or off at will. Of course, you can set which bars are visible by default when you create your application. In addition, the rich edit component includes the following features:

- Customizable printer margins
- Paragraph indentation
- Tab settings
- Page layout
- An extensive pop-up menu of editing and formatting functions
- Search and replace

InfoPower 3 includes two other new components, *TwwStoredProc* and *TwwClientDataSet*. Both can be attached to *TwwDBGrid*, and both support embedded controls and picture masks (as did the Table, Query, and QBE components in prior versions). Most of the controls from previous versions of InfoPower have new features. All the dataset components now have properties that let you display an hourglass while records are being filtered, thus giving the user the ability to cancel a filter by pressing Esc. This is a welcome addition; filters that make a sparse selection from a large table can be slow. Not only does pressing Esc stop the filter, it also triggers an *OnFilterEscape* event so you can attach code to this action. Another slick new feature is the ability to filter on lookup fields in a dataset.

TwwDBGrid, one of the most powerful components in InfoPower since the first version, has a number of new features. There are two new options in the *MuliSelectOptions* property. The first lets you configure the grid so that clicking on a record without holding down Ctrl will unselect all records in the current selection, and select only the record you clicked. The second enables ⇧ Shift-select support, so you can easily select a contiguous group of records. You can also select or unselect all records in code using the new *SelectAll* and *UnselectAll* methods. When multiselect is enabled in a grid, the new *OnMultiSelect* event is triggered each time a user selects or unselects a record. This lets you easily write code that controls whether the user can select the record or not, or take any other action you wish.

Setting the *EditCalculated* property to *True* lets you edit calculated fields or lookup fields with a single line of code. Now you can embed the *TwwDBEdit* control in the grid so users can edit memo fields directly in the grid — they don't have to display a pop-up editing window to change the memo text, as was necessary in previous versions. Another nice feature is that you can now determine at design time whether the grid will use the dataset's *TFields* properties or its own selected property settings. This lets you customize the grid and have the settings stored with the grid, not the dataset, if you wish.

Another popular component with many new features is *TwwFilterDialog*, which lets users set their own filter crite-

ria. A new feature is support for "and", "or", and "null" keywords within a field. For example, you can now search for records whose state is CA or WA, or a ZIP code field that is null. There is also a new option that adds a **records not matching** checkbox to the dialog box. With this, users can enter selection criteria and see all records that do not match the criteria. Also new is the ability to search on calculated, linked, and lookup fields. One of the problems with filters in the past has been that they can be slow because they do not use indexes. If you are filtering on *TwwTable* and set the new *FilterOptimization* property to *True*, the filter will use indexes to speed the selection process.

The DBLookupCombo and DBCombobox components also have several new features. One feature they share is Quicken-style incremental searching. Simply set the *ShowMatchText* property to *True* and, as users enter text into the edit box, the first matching choice will display automatically. Another welcome feature for the DBLookupCombo is the *AllowClearKey* property. If set to *True*, and the *Style* is set to *csDropDownList*, users can clear the current selection by pressing either Delete or ←Bksp. However, perhaps the most important enhancement to the DBLookupCombo is that it now accepts *TwwQuery*, *TwwQBE*, and *TwwClientDataSet* components as the lookup source. Now you can use parameterized queries or remote datasets as the source of the lookup list.

If You've Never Used InfoPower

If you haven't used InfoPower, here's a look at some of the other components it provides:

- A Table component that fully supports Borland Database Engine (BDE) filters, including the ability to change the filter criteria on-the-fly at run time, a *Pack* method to pack both Paradox and dBASE tables, and a *wwFindKey* method that is faster against SQL tables than Delphi 1's *FindKey* method.
- A QBE component that fully supports QBE queries and includes an *AnswerTable* property to let you easily save your result set to disk, as well as an *AuxiliaryTables* property so you can have the query create Paradox-style KeyViol, Changed, Inserted, and Deleted tables in the user's private directory.
- A data-aware grid component that includes the ability to display a cell as a checkbox, combo box, lookup combo box, custom dialog box, or rich edit control; display the text of a memo field in the grid; double-click a memo field in the grid and display a pop-up memo editor; and display multiple tables in a single grid and define non-scrollable columns in the grid.
- A variety of high-performance search controls that include incremental, exact match, starts with, and substring searching.
- A customizable pop-up memo field editor you can use to edit memo fields displayed in any control.
- A data-aware rich edit control that includes a pop-up WordPad-style editor.
- A DBComboBox component with Quicken-style search and fill, as well as the ability to display one set of values but store a different set in the underlying table.
- A DBComboDialog component that includes an ellipsis button and an event that is triggered when the user clicks the button. This allows you to display custom dialog boxes to help users edit a field in a table.
- A DBLookupCombo component that lets you display any number of fields in the drop-down list; display column separators in the drop-down list; display column headings in the drop-down list; control whether the drop-down list grows to the left or right; lets you control by which column the drop-down list is sorted; lets users incrementally search the drop-down list by typing into the field; supports Quicken-style incremental fill-in; and displays a description instead of a code, even though the code is stored in the table.

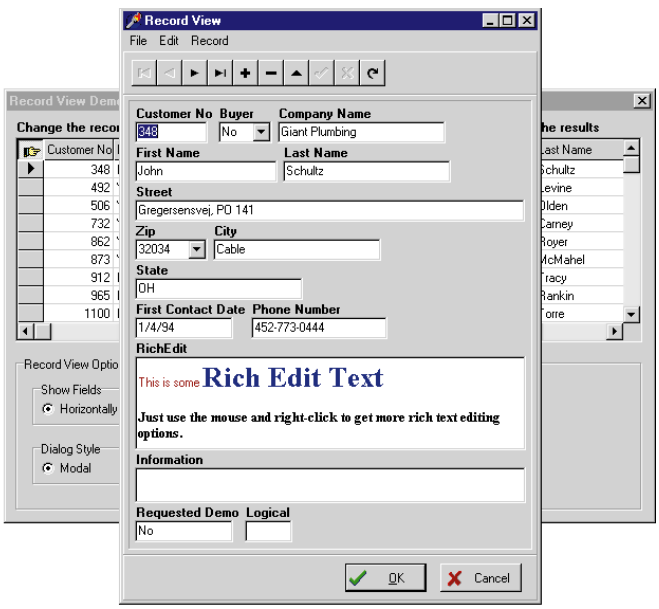


Figure 3: A RecordViewDialog with a custom menu and rich edit control.

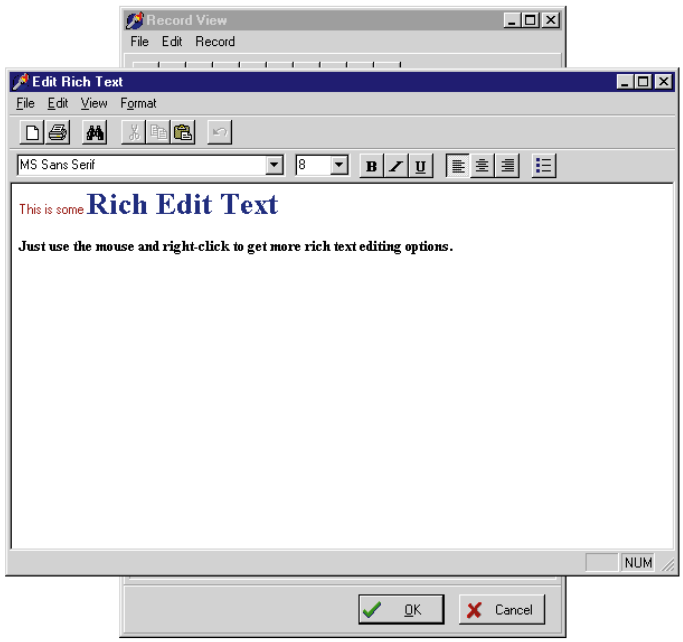


Figure 4: *TwwDBRichEdit*'s editing view.

Character	Description
#	Any digit.
?	Any letter, either upper or lower case.
&	Any letter. Lower-case letters are converted to upper case.
~	Any letter. Upper-case letters are converted to lower case.
@	Any character.
!	Any character. Letters are converted to upper case.
;	Treat the next character as a literal, not a mask character.
*	Repeat count. *# means any number of numbers. *5# means five numbers.
[ ]	Anything in square brackets is optional.
{}	Group of alternatives. {Y,N,U} means the user must enter either Y, N, or U.

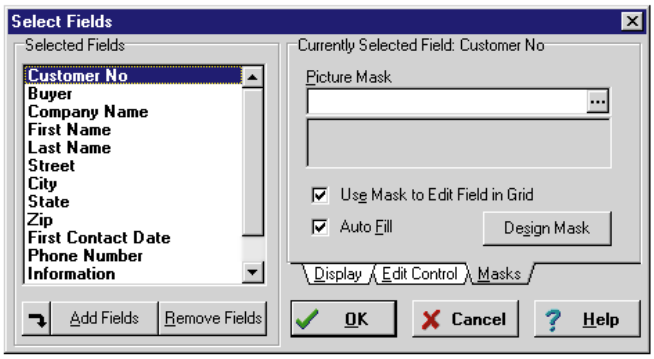
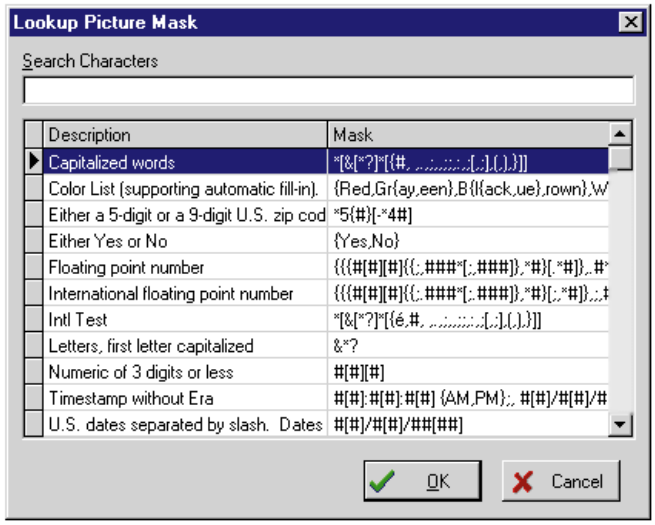


Figure 5 (Top): InfoPower picture characters.
Figure 6 (Bottom): The Select Fields dialog box.



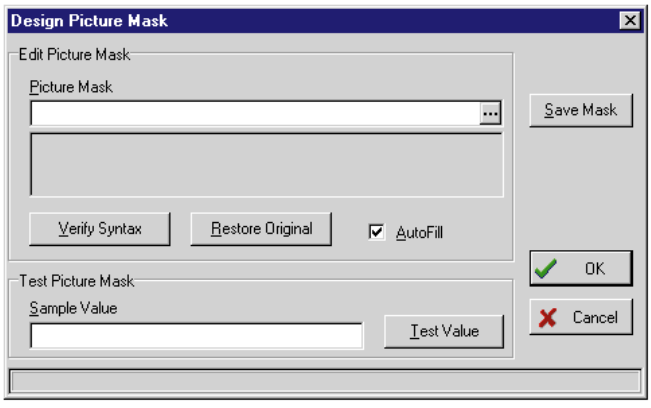


Figure 7 (Top): The Lookup Picture Mask dialog box.
Figure 8 (Bottom): The Design Picture Mask dialog box.

You can also use this component without binding it to a data table.

- A DBLookupComboDialog component with all the features of the DBLookupCombo component, but instead of a drop-down list, this control displays a dialog box with the lookup table displayed in a customizable grid.

Picture Masks

One of the most exciting and useful features in InfoPower is Paradox-style picture masks for controlling what users can enter into a field.

While Delphi's edit masks provide a simple system of templates to control user input, InfoPower's picture masks provide a mask language that supports automatic completion, multiple masks for the same field, and much more. If you are frustrated with the limitations of edit masks, try picture masks — you'll never go back. The table in Figure 5 shows the picture characters used with InfoPower picture masks.

There are several ways to enter pictures when you use InfoPower components, but perhaps the easiest is clicking on the *PictureMasks* property of a *TwwTable* component to display the Select Fields dialog box (see Figure 6).

Clicking the ellipsis button in the Picture Mask field of this dialog box leads to the Lookup Picture Mask dialog box shown in Figure 7. This dialog box lists a number of useful pictures that have already been built; simply select the one you want.

If you need to design a custom picture, click the **Design Mask** button in the Select Fields dialog box to display the Design Picture Mask dialog box shown in Figure 8.

For example, suppose you need a picture that will let users enter in a field either a valid US five- or nine-digit ZIP code or a Canadian postal code. Start by typing the picture `{#&# &#&,#####[-####]}` into the Picture Mask field. Now click the **Verify Syntax** button to verify that your picture is syntactically correct. Then type sample values into the Sample Value field to make sure your picture works as intended.

If you think you'll use this picture again, click the **Save Mask** button to add it to the database of pictures displayed in the Lookup Picture Mask dialog box shown in Figure 7. This is just one example of a mask that is impossible to create in Delphi without InfoPower.

Another good example of the superiority of InfoPower's pictures over Delphi's edit masks is the picture `#[#]/#[#]/##[##]` for a simple date. Not only does this picture allow you to enter either a one- or two-digit month or day, it also lets you enter either a two- or four-digit year. If you've ever tried to use a Delphi edit mask for a date, you have probably discovered that it puts the two literal characters (the slashes), into the field as soon as it gets

focus. You've also discovered you cannot get rid of the literal characters if you then decide to leave the field blank. InfoPower does not insert the literal characters until you reach the point in typing your value where they appear. If you highlight the field and delete the value, the literal characters disappear, so you can easily leave a field blank.

As with all other InfoPower features, pictures work identically in all versions of Delphi. They also apply whether the data is entered by the user through a form or you assign a value to a field in code.

You can allow a user to leave a field that contains an invalid value by setting the *AllowInvalidExit* property to *True*, and you can control whether the pictures are used for interactive editing by setting the *UsePictureMask* property.

InfoPower controls also include *OnCheckValue* and *OnInvalidValue* events to let you control what happens when the user enters an invalid value. Using these events, you can determine if the value entered by the user is valid or not, and if it is invalid when the user tries to post the record you can display an error message that identifies the invalid field. Because the *OnInvalidValue* event identifies the offending field, you could also change the field's color and move focus to the field.

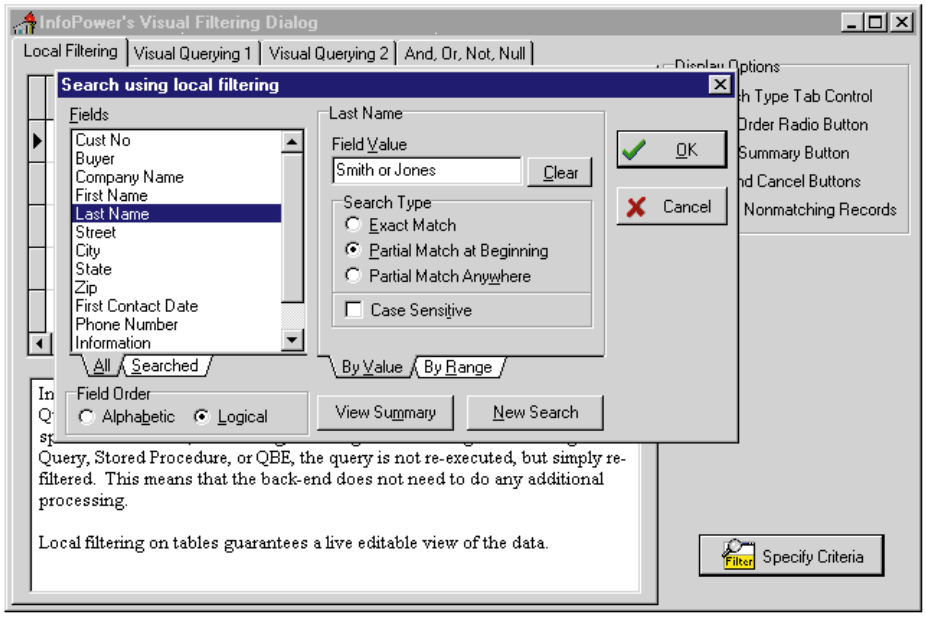


Figure 9: The *TwwFilterDialog* in action.

Filter Dialog

The *TwwFilterDialog* component gives end users an easy way to filter or query a dataset on multiple fields. To use it, simply drop the FilterDialog component on your form, set its *DataSource* property, and provide a button or menu choice to call its *Execute* method.

Calling *Execute* displays the dialog box shown in Figure 9.

Selecting records is done in one of two ways (which you specify). If the dataset being searched is a *TwwTable* or *TwwQBE*, then a BDE filter is applied. If the dataset is a *TwwQuery*, you can either filter the query result set or let the FilterDialog modify the WHERE clause of the query and re-execute it to select the records. You can set the caption of the dialog box, as well as for which fields the user can enter selection criteria.

The **View Summary** button lets users see the field or fields for which they have entered selection criteria. The By Value and By Range tabs let users enter a single value, or a range of values, to search for. Entering a single value lets users choose to search for an exact match, a record that starts with the search value, or a record that contains the search value anywhere in the field. Users can also select a case-sensitive search.

Conclusion

InfoPower continues to be the single-most valuable add-in product for the Delphi database application developer. It will let you develop programs with far more features faster than any other tool I know. Δ

Bill Todd is President of The Database Group, Inc., a database consulting and development firm based near Phoenix, AZ. He is a Contributing Editor of *Delphi Informant*; co-author of *Delphi 2: A Developer's Guide* [M&T Books, 1996], *Delphi: A Developer's Guide* [M&T Books, 1995], *Creating Paradox for Windows Applications* [New Riders Publishing, 1994], and *Paradox for Windows Power Programming* [QUE, 1995]; and a member of Team Borland, providing technical support on CompuServe. He has also been a speaker at every Borland Developers Conference. He can be reached on CompuServe at 71333,2146, on the Internet at71333.2146@compuserve.com, or at (602) 802-0178.

INFORMANT FACT FILE

In its third release, InfoPower continues to be a "must have" add-in product for Delphi developers creating database applications. A host of new features make developing database applications faster and easier.

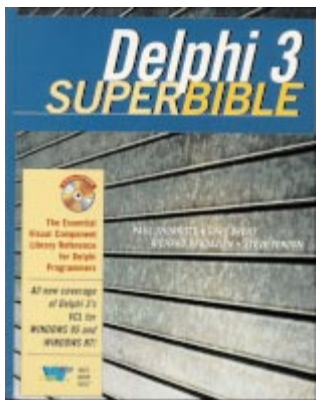
Woll2Woll Software
2217 Rhone Dr.
Livermore, CA 94550

Phone: US (800) 965-2965; International (510) 371-1663
Fax: (510) 371-1664
CompuServe: 76207,2541
CompuServe Forum: Go Woll2Woll
E-Mail: sales@woll2woll.com
Web Site: http://woll2woll.com
Pricing: InfoPower is available for US$199; source code is available for an additional US$99. Contact Woll2Woll for InfoPower upgrade pricing information.

Delphi 3 SuperBible

I've always been happy with Borland's documentation of its programming tools, starting with the one-volume manual for Turbo Pascal 3. Today, things are more complicated. Delphi 3 comes with a large box of reference books, including the expanded *Visual Component Library* (VCL) *Reference* (which now requires two volumes and nearly 1,700 pages of text). Now *that* should meet the needs of most programmers, right? Well, perhaps quite a few, but consider some of the following criticisms that could be made: The VCL is a hierarchical structure, so why doesn't Borland arrange its VCL reference in a way that reflects that structure, rather than alphabetically? I would like to learn more about the support classes (*TCollection*, *TStrings*, etc.) that are used in many of the components; why doesn't Borland put them in the same section of its reference? The code examples that Borland does include get me started on using these classes and components quickly, but I wish there were more. I could go on, but you get the idea.

If you've used Delphi for a while, and are familiar with the VCL structure, then Delphi 3's *VCL Reference* probably meets your needs. If not, then you should consider Waite Group Press' *Delphi 3 SuperBible* by Paul



Thurrott, Gary Brent, Richard Bagdazian, and Steve Tendon. It begins with an excellent introduction, then launches into a full disclosure of the contents of the VCL, beginning with *TObject*. (Borland buries this most basic of all constructs midway through Volume II). *SuperBible* then presents the most basic objects, including classes used with .INI files and the Windows registry (*TIniFile*, *TRegistry*, *TRegIniFile*), *TList*, *TThread*, and the Exception classes. The latter are dealt with rather extensively in their own 12-page chapter.

The remaining four sections of the book are closely related to the structure of the VCL, covering persistent objects (*TPersistent* and its descendants), components, non-windowed controls, and windowed controls. A careful examination of the *TPersistent* entry and a comparison to the similar entry in the *VCL Reference* shows one of *SuperBible's* limitations — its acknowledged emphasis on the more commonly used prop-

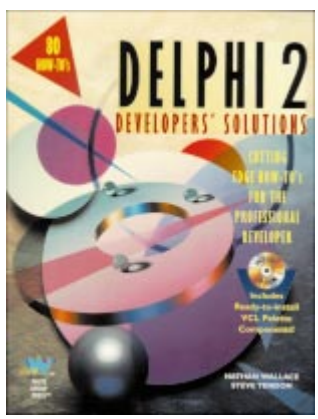
"Delphi 3 SuperBible" continued on page 60

Delphi 2 Developers' Solutions

At first glance, *Delphi 2 Developers' Solutions* appears to be a collection of tips and techniques, similar in nature to its predecessor, *Borland Delphi How-To* (see Larry Clark's review in the January 1996 issue of *Delphi Informant*). On closer examination, I found a carefully thought-out structure where most of the 80 "how-to's" are, in fact, building blocks of some impressive sample applications. Beginning with three utility programs in the first chapter, developing a full-featured file manager in the next few chapters, and concluding with an in-depth exploration of Internet programming, *Solutions* is a *tour de force* of programming problems and their solutions.

Written by Nathan Wallace and Steve Tendon, *Solutions* has a clear and consistent style. The "how-to's" in each chapter are steps in creating a fairly elaborate Windows application. At the beginning of each chapter, the authors provide a list and a capsule summary of the "how-to's" that follow. Each begins with a statement of the problem (in real-world terms), a brief overview of the technique to solve the problem, and a step-by-step description of how to develop the code needed. Refreshingly, you get all the code. (This is not one of those books that provides a few snippets of code, then refers you to the CD-ROM.) In addition, the CD-ROM includes all the code, as well as several collections of custom components and other utilities and files.

The target audience is intermediate to advanced. However, the authors note that even a relatively new Delphi



programmer could benefit from studying the code.

Most of the book is devoted to building two major applications: a file manager and an Internet control center. The chapters devoted to the file manager cover the techniques for building a Windows 95-style graphical file manager with drag-and-drop support; specialized mouse and key-

"Delphi 2 Developers' Solutions" continued on page 61

Delphi 3 SuperBible (cont.)

erties and methods of a class. In this instance, *SuperBible* provides an excellent description of the *TPersistent* class and its most important method, *Assign*. However, it does not provide any information on the protected methods, *AssignTo* or *DefineProperties*, as the *VCL Reference* does. So, don't put your Delphi 3 manuals away yet.

An important difference between the two references is the way they explain a control's properties. Not surprisingly, there is overlap and similarity in the information itself. The difference lies in the *approach* to presenting that information. While the *VCL Reference* rarely uses code examples to show a property's uses, such examples are common in *SuperBible*. And while the *VCL Reference* is careful to restrict its discussion to closely related classes (while sometimes providing more information about these), *SuperBible* attempts to anticipate some of the other techniques that might be of interest to a programmer using the property in question. Let's examine one example: the *OnKeyUp* event of *TWinControl*.

The *OnKeyUp* event is handy if you need to monitor the release of keystrokes, then take some kind of action based upon the key or combination of keys just released. The entry in *SuperBible* provides a nice example of how to create a specialized *KeyUp* event handler. You are reminded that the *Chr* standard function can be used to convert the *Key* parameter to a character; then you are referred to the example code. Unfortunately, the *Chr* function is not used in the stated example, although it is used in the earlier *OnKeyDown* example. (With the current desire of publishers to get new programming books on the shelves as soon as possible, I don't think you will find many books that are totally free from this kind of error.) While the *VCL Reference* doesn't provide sample code for this event, it does provide additional information that might be needed by the advanced programmer, including the type definitions for *TShiftState* and *TKeyEvent*.

As with many Delphi books I have read, I must take issue with the user level suggested by the publisher, here specified as Intermediate to Advanced (emphasis on the latter.) My assessment is that this book would be of *least* interest to the more advanced Delphi programmers (who should find Borland's *VCL Reference* more than adequate), but would be of greater use to relatively new Delphi programmers who want to quickly learn the structure and uses of the VCL. If you want to explore the essentials of the VCL and have an excellent resource to guide you, then I highly recommend this book.

— Alan C. Moore, Ph.D.

Delphi 3 SuperBible by Paul Thurrott, et al., Waite Group Press, 200 Tamal Plaza, Corte Madera, CA 94925, (800) 368-9369 or (415) 924-2575.
ISBN: 1-57169-027-1
Price: US$54.99
(1,312 pages, CD-ROM)

Delphi 2 Developers' Solutions (cont.)

board functionality; printer capabilities; a Help file; and techniques for creating thumbnails of graphics files, then using them in the file manager. The Internet control center is the focal point of the last three chapters. They provide the techniques needed to use Winsock, perform FTP operations, how to connect to — and use — the various UseNet Newsgroups, and add e-mail capabilities to the application.

One of the strongest aspects in *Solutions* is the attention given to error handling. For instance, in their discussion of the file manager, the authors re-introduce some of the venerable Pascal file-handling procedures. For example, how many of you DOS veterans remember the *FindFirst* and *FindNext* procedures? These procedures, and others, are used in this file manager. To provide complete error handling, file-manipulation calls are enclosed in **try..except** blocks to provide complete error information when something goes wrong (as opposed to the use of Boolean function returns in some of the newer file routines that merely tell you it didn't work — not why).

Besides the file manager and Internet control center, there are other interesting and useful tools discussed in other sections of the book. For example, the first chapter shows how to build three utilities: a Windows screen saver, a Windows wallpaper changer, and an application launcher in the style of a NeXT taskbar. Two other chapters discuss interesting aspects of database programming, such as interfacing with the BDE, performing a phonetic search with the Soundex algorithm, and using the InterBase server.

Solutions has many virtues — and only a few problems. The book promises to offer components to make the functionality of these projects readily available. In fact, many of the "how-to's" conclude with a discussion of such components. Unfortunately, these components are nowhere to be found. After visiting Waite Group Press' Web site, I discovered the reason was that the publisher was "unable to acquire all of the permissions needed to put these components on the CD-ROM." This was a disappointment, but not a big deal. All, or most, of the code necessary to wrap such functionality into a component is there.

The other criticism is a matter of personal preference: While I appreciated having the full source code in the book, I really didn't expect or need to see page after page of form-file text. The text of the first .DFM file in Chapter 9 goes on for 25 pages. Is there anyone who is actually going to type in all of this? Please, just tell me where to drop the components and what their non-default property values are!

While the shortcomings are, at most, mildly annoying, the book's strengths are considerable. The authors provide excellent models for building large applications with full error checking, particularly for relatively newer programmers. Delphi 1 programmers, or those working with various Delphi versions to support both 16- and 32-bit environments, will appreciate the care given to explicating Delphi 1 issues. Despite the title, a handful of the projects are specific to Delphi 1, and are intended to provide some of the functionality of Delphi 2 for its predecessor.

In sum, I highly recommend this carefully crafted and thor-

"Delphi 2 Developers' Solutions"
continued on page 60

Delphi 2 Developers' Solutions (cont.)

oughly delightful programming treatise for serious Delphi developers. It's a must for those who need to enter the Internet programming arena.

— *Alan C. Moore, Ph.D.*

Delphi 2 Developers' Solutions by Nathan Wallace and Steve Tendon, Waite Group Press, 200 Tamal Plaza, Corte Madera, CA 94925, (800) 368-9369 or (415) 924-2576.
ISBN: 1-57169-071-9
Price: US$59.99
(892 pages, CD-ROM)

Delphi 3: The Ultimate ActiveX Factory?

Delphi 3 and ActiveX control development are synonymous — especially if you believe Borland's marketing material. Just about anyone who can point-and-click can create an ActiveForm, and view it from within a Web browser. But how much substance is behind this hype? Is Delphi truly a foundry for developing professional-strength ActiveX controls?

When Delphi 3 was released, I wanted to find out. Not only was I curious about bending this ActiveX envelope, but I had a legitimate need as well. Specifically, I used Delphi 3 to create a new breed of ActiveX control called a design-time control. Unlike typical ActiveX controls you use within a Web browser, Delphi, or Visual Basic, design-time ActiveX controls don't surface at run time; rather, they are employed exclusively at design time to create content for Web pages. (Technically speaking, a design-time control is an ActiveX control that has the IActiveDesigner interface implemented.) After spending a great deal of time and energy on this task, I decided to use this month's column to share my impressions of using Delphi to create ActiveX controls.

Delphic Approach. To begin, Delphi 3 does to ActiveX what Delphi 1 did to RAD development: It provides a natural, intuitive approach to ActiveX control development creation. This was no small task, mind you. Component development is nothing new to Delphi, but ActiveX has a completely different architecture and philosophy behind it than Delphi's VCL architecture. Thus, Borland engineers faced a supreme challenge when it came to ActiveX: how to bridge the "Delphi way" of creating components with the ActiveX specification. For the most part, Borland succeeded in this attempt. Experienced VCL component developers will undoubtedly agree, as the base object for an ActiveX control is a VCL. In sum, with its intuitive Type Library editor, inline code instructions, and online Help, Delphi 3 enables able VCL component developers to create standard ActiveX controls with minimal knowledge of the ActiveX architecture.

Off the Beaten Path. But that isn't the end of the story. The operable word here is "standard," because once you venture off the beaten path, you're often on your own. As a result, serious ActiveX development in Delphi can become a real trial-and-error process. This isn't necessarily due to Delphi itself, but to the fact that little documentation and few examples exist on implementing ActiveX interfaces with Delphi 3. (Frankly, I'm flabbergasted that Delphi 3 shipped without a serious ActiveX demo.) I'm hopeful this will change in the coming months as more is written about Delphi from third-party sources.

Given Microsoft's dominance in Windows programming, Delphi developer's may often feel like they're going against the grain when creating ActiveX controls. That's most certainly my case: Even after faithfully monitoring Delphi's activex.writing newsgroup and talking with Borland, I discovered no one else used Delphi to create design-time controls. So, as I plodded ahead on my own, I had to hitch a ride with Microsoft. Because much of the documentation on ActiveX is Microsoft's, I found myself translating Visual C++ examples into the Object Pascal equivalent. In the end, I probably spent as much time in the Visual C++ IDE working with C++ samples as I did within Delphi.

There's a downfall to forging your own ground. When you encounter a problem, it can be difficult to know when you or your tool are wrong. Only after many hours of frustration and confirmation from Borland did I realize that the source of my difficulty was due to two bugs in the ActiveX.pas source file in its declaration of the IActiveDesigner interface.

The Old Stand-By. Finally, in spite of these Lone Ranger issues, I still prefer using Delphi than Visual C++ to create ActiveX controls. Delphi's ActiveX programming prowess lies not in its wizardry, but in its ability to simplify Windows programming at the code level. And ultimately, that is what gives Delphi 3 the edge in terms of ActiveX control development. Δ

— Richard Wagner

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

table of CONTENTS