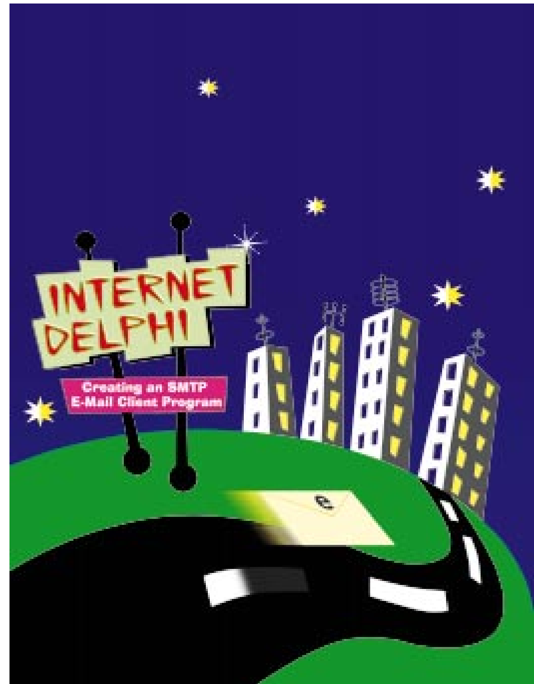
July 1997, Volume 3, Number 7

Internet Delphi

Creating an SMTP E-Mail Client Program



Cover Art By: *Tom McKeith*

ON THE COVER

6 Internet Delphi: Part I — Gregory Lee
Adding Internet e-mail capabilities — for automated registration, program support, even database reporting — could wake up your next ho-hum application. This series of articles tracks the development of a Delphi e-mail program, beginning with Simple Mail Transfer Protocol.

FEATURES

10 Informant Spotlight
What's New with Experts? — Ray Lischner
You might not know that Delphi 3's Open Tools API sports some enhancements. Discover how project creators and module creators, when combined, let you create experts and wizards with ease.

19 Delphi at Work
Automated Access — Ian Davies
First Word, then Excel, and now Access. In this third installment, Mr Davies shows how Automation may be ideal for manipulating Access systems that include more than a database.

23 Greater Delphi
InterBase Indexes — Bill Todd
InterBase uses indexes more flexibly than do most databases. Learn how careful index creation can yield the best possible performance.

27 Columns & Rows
The Paradox Files: Part IV — Dan Ehrmann
Although the Paradox file format has extensive features for validity checks and referential integrity, Delphi doesn't support some, but goes others one better. Here's how it all shakes out.

32 DBNavigator
Cached Updates: Part III — Cary Jensen, Ph.D.
In previous installments, you learned the advantages of cached updates. Now Dr Jensen explains the use of two event properties for those times when you want complete control.

38 Sights & Sounds
Optimizing Graphics — Peter Dove and Don Peer
Speed is a primary concern in graphics programming, and this ongoing example project is no exception. This month, the authors weigh several optimization methods to increase speed by 50 percent.

43 On the Net
NetCheck: Part II — John Penman
If you develop for the Internet or intranets, you need a network debugging tool — and Mr Penman has just the thing. This month, he adds Echo processing and packet tracing to May's NetCheck tool.

49 At Your Fingertips
Displaying Shortened Pathnames — Robert Vivrette
Suppose you want to retain the right and left portions of a too-long pathname, and eliminate characters from the middle. The remedies for this and other puzzlers are at hand this month.

REVIEWS

51 AdHocery for Delphi
Product Review by Bill Todd

DEPARTMENTS

2 Delphi Tools
5 Newslines
53 File | New by Richard Wagner

Delphi TOOLS

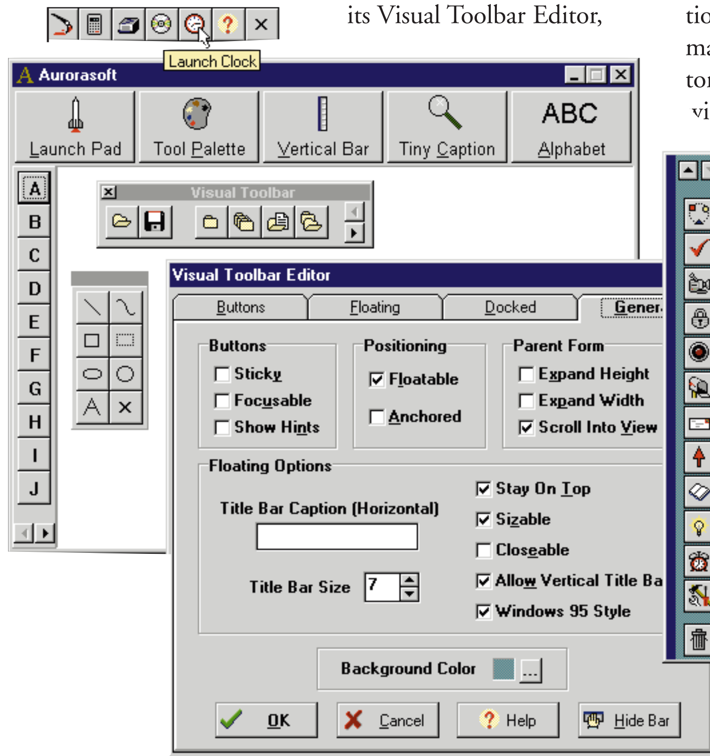
New Products
and Solutions

Aurorasoft Releases New Visual Toolbar for Delphi

Aurorasoft of Danville, CA has released *Visual Toolbar* for Delphi, a component that allows developers to create toolbars for any type of application.

Visual Toolbar, along with its Visual Toolbar Editor, allows users to drag-and-drop between a customized floating window and a stationary position. It retains size, position, and visible state automatically, and users can customize its floating toolbar windows.

Toolbars can also be developed by creating up to 10 buttons at once, dragging and dropping buttons onto the toolbar, setting the toolbar size and position visually, and setting other properties through a design interface.



Price: US$79
Contact: Aurorasoft, P.O. Box 104, Danville, CA 94526-0104
Phone: (800) 987-2426 or (510) 939-3788
Fax: (510) 939-3779
E-Mail: sales@aurorasoft.com
Web Site: http://www.aurorasoft.com

DemoShield Ships ActiveX

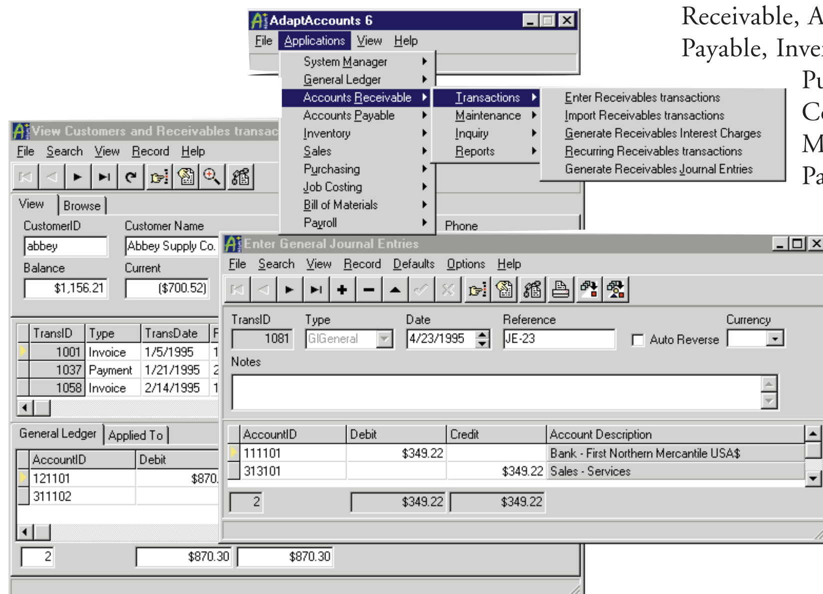
DemoShield Corp. has released Demo-X, an ActiveX version of its DemoShield product. Demo-X contains tools for creating drag-and-drop interactive buttons, screen shots, hot spots, and animated transitions — without scripting. Demo-X is available free from http://www.demoshield.com.

Adapta Software Launches AdaptAccounts 6.0

Adapta Software Inc. of Victoria, Canada has shipped *AdaptAccounts 6.0*, a family of modules that can be installed together or separately. This version combines Adapta's integrated accounting database applications with Windows 95 and Windows NT environments using Delphi.

Adapta's line of modular applications includes System Manager, General Ledger and Financial Reporter, Accounts Receivable, Accounts Payable, Inventory, Sales, Purchasing, Job Costing, Bill of Materials, and Payroll.

All modules are available in multi-user or single-user versions, with source code available. Adapt-Accounts' modules can also be integrated with other applications, and include transaction importing facilities and validation of externally loaded data.



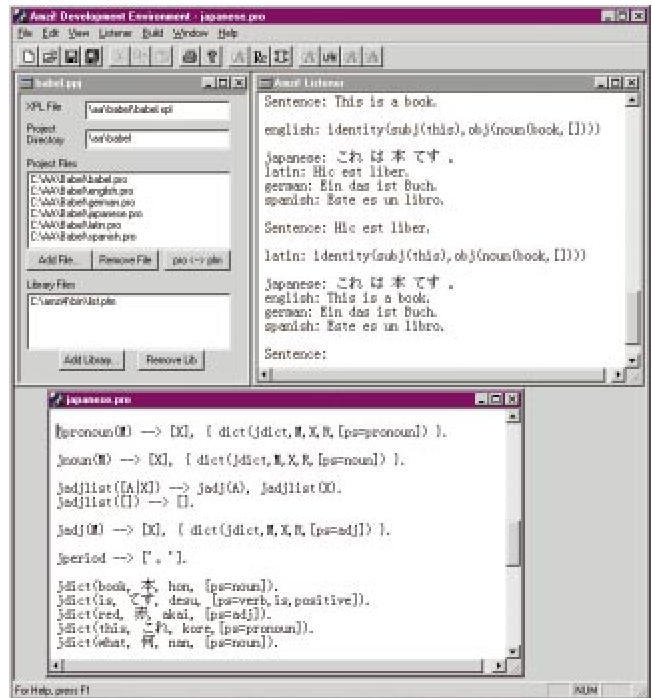
Price: The General Accounting pack (includes System Manager, General Ledger and Financial Reporter, Accounts Receivable, and Accounts Payable) ranges from US$1,495 to US$4,195. The Accounting/Distribution pack (includes the Inventory, Sales, Purchasing, and general accounting modules) ranges from US$2,695 to US$7,695. Typical individual module prices range from US$495 to US$1,295. Existing users of AdaptAccounts 5.0 or 5.7 can upgrade for 40 percent plus US$75 per module.
Contact: Adapta Software Inc., 4608 Cliffwood Place, Victoria, BC, Canada V8Y 1B5
Phone: (250) 658-8484
Fax: (250) 658-2108
E-Mail: sales@adapta.com
Web Site: http://www.adapta.com

Delphi
T O O L S

New Products and Solutions

Amzi! Releases Intelligent Components

Amzi! Inc. of Stow, MA has released ***Amzi! Prolog + Logic Server 4.0***, a C++ application that integrates Prolog's catch/throw mechanism to the catch/throw mechanisms in Delphi, C++, and Java. Because it's designed as a C++ application, it allows other programs to create multiple instances of the Logic Server running in the same or separate threads.



Amzi! Prolog + Logic Server allows the execution of multiple Prolog engines for applications such as Web server-side applications, telephony systems, and Java-based and database server components. It also supports Unicode character sets, allowing the creation of Prolog components that can reason over and speak any language.

The Amzi! Prolog engine is a native Unicode application similar to Java and NT 4.0. The Prolog source code can be written in Unicode or ASCII, allowing predicates and variables, as well as strings to be represented in the 16-bit Unicode character set.

Support of Internet and Web-based applications has been expanded with three new components, the CGI interface, the Java Class, and the Sockets Logic Server Extension.

The CGI interface allows users to write server-side applications for the Web, making it possible to embed all manner of expert systems, advisors, problem solvers, and intelligent components on Web pages. This interface consists of a C program, the Prolog framework, and a library that provides an interface between the Prolog script and CGI/HTTP protocols.

The Java Class allows Prolog components to be embedded in Java programs and applets. They include support for exception handling.

The Sockets Logic Server Extension allows users to write clients and/or servers in Prolog that communicate with other Internet clients and servers for e-mail, FTP, news, HTML, and more.

Version 4.0 has an enhanced IDE, including improved project support. Console versions of the command-line tools are available for building Amzi! Prolog components while compiling and linking the main application. The latest development environments from Borland and Microsoft are also supported.

Price: Personal edition, US$298; Professional edition US$598.
Contact: Amzi! Inc., 40 Samuel Prescott Dr., Stow, MA 01775
Phone: (508) 897-7332
Fax: (508) 897-2784
E-Mail: info@amzi.com
Web Site: http://www.amzi.com

Pretty Objects Computers, Inc. Announces Polyglot 2.24

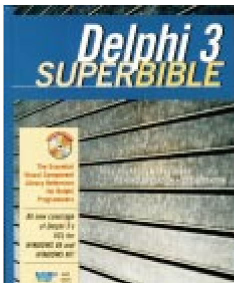
Pretty Objects Computers, Inc. of Outremont, Quebec has announced ***Polyglot 2.24***, an internationalization expert for Delphi 2 that enables users to create multilingual applications.

Polyglot allows users to determine what needs to be changed to present applications in other languages by exporting all character strings to a table, and re-importing them.

Polyglot also manages elements that are less visible, such as the user's Help file, date formats, numbers, and common Windows dialog boxes.

Price: US$200 for first license; US$100 for second to fifth license; US$50 for additional licenses.
Contact: Pretty Objects Computers, Inc., 5158 Hutchison, Outremont, Quebec, Canada H2V 4A9
Phone: (514) 990-7026
Fax: (514) 990-7026
E-Mail: info@prettyobjects.com
Web Site: http://www.prettyobjects.com

Delphi 3 Superbible
Paul Thurrott, Gary Brent, Richard Bagdazian, & Steve Tendon
Waite Group Press

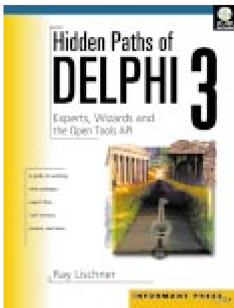


ISBN: 1-57169-027-1
Price: US$54.99
(1,312 pages, CD-ROM)
Phone: (800) 368-9369

Delphi TOOLS

New Products and Solutions

Hidden Paths of Delphi 3
Ray Lischner
Informant Press



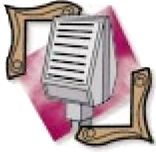
ISBN: 0-9657366-0-1
Price: US$39.99
(300 pages, CD-ROM)
Phone: (800) 884-6367 or (916) 686-6610

Comparing Delphi 3 Versions

Features	Standard	Professional	Client/Server
Visual drag-and-drop RAD	X	X	X
32-bit, optimizing, native-code compiler	X	X	X
Royalty-free, stand-alone EXEs and reusable DLLs	X	X	X
Packages compiler technology for EXEs	X	X	X
Interfaces for native COM and ActiveX support	X	X	X
Access to Win32 API, ActiveX, multi-threading, OLE, COM, DCOM, ISAPI, NSAPI	X	X	X
Creates multi-threaded Windows 95/NT applications	X	X	X
Professional IDE with Editor and Debugger	X	X	X
Object-oriented, extensible component and application architecture	X	X	X
Object repository for storing and reusing forms, data modules, and experts	X	X	X
Visual form inheritance and form linking	X	X	X
Suite of Windows 95 common controls	X	X	X
Visual component Library with over 100 drag-and-drop reusable components	X	X	X
Create and use OLE automation controllers and servers	X	X	X
Visual components creation for making component templates	X	X	X
CodeTemplates Wizard	X	X	X
CodeCompletion Wizard	X	X	X
CodeParameter Wizard	X	X	X
ToolTip Expression Evaluation	X	X	X
DLL debugging	X	X	X
Multiple database engine support	X	X	X
Native drivers for MS Access, FoxPro, Paradox, and dBASE	X	X	X
Data-aware components	X	X	X
Separate business rules from application code with Data Module Objects	X	X	X
Database Explorer for managing tables, aliases, and indices	X	X	X
Integrated reporting	X	X	X
Delphi 1 for 16-bit Windows 3.1 applications	X	X	X
One-step ActiveX creation for maximum reusability (100 percent compiled high-performance ActiveX controls with no run-time redistributables)		X	X
One-step ActiveForm creation to Web-enable applications		X	X
Live graphs and charting		X	X
Additional 30 VCL components		X	X
VCL source code and printed manual		X	X
ODBC connectivity		X	X
Maintain data integrity with scalable Data Dictionary		X	X
Develop and test SQL applications with Local InterBase		X	X
Cached updates		X	X
Internet Solutions Pack		X	X
InstallShield Express		X	X
Open Tools API		X	X
Printed documentation		X	X
SQL Links native drivers, with unlimited deployment license for Oracle, Sybase, Informix, MS SQL Server, InterBase, and DB2			X
SQL Database Explorer			X
SQL Monitor			X
Visual Query Builder			X
Develop and test multi-user SQL applications with InterBase (4-user license)			X
Decision Cube Crosstabs for multi-dimensional data analysis			X
Remote DataBroker			X
ConstrainBroker			X
Business ObjectBroker			X
WebServer			X
Support for Netscape NSAPI and Microsoft ISAPI with WebBridge			X
WebModules for information publishing			X
WebDispatch for responding to Web client requests			X
WebDeploy			X
Integrated Intersolv PVCS Version Manager			X
CASE Tool Expert			X
Data Pump Expert			X

News LINE

July 1997



Borland Announces Spin-Off of Open Environment Consulting Group

Scotts Valley, CA — Borland has announced NetNumina Solutions Inc., a spin-off of its Open Environment Division. Located in Boston, MA, NetNumina has been providing consulting services on behalf of Borland to Open Environment and Borland customers, as well as servicing new customers, since April.

Under the terms of the agreement, consulting services provided to Borland and its customers by NetNumina include pre- and post-sale technical support, training, custom development projects, and placement and management of contract personnel at customer sites.

The principals of NetNumina were previously the consulting arm of the Open Environment Division. They produced over 150 multi-tier applications at Fortune 1000 companies. NetNumina is currently extending its multi-tier technology knowledge to Web-deployed, distributed object solutions.

Borland acquired the Open Environment Corp. in November 1996. Borland is currently preparing Entera 4.0 (previously developed by Open Environment) for release, and is expanding the Entera development team by adding engineers.

Secondarily, Entera is an enabling technology within Delphi, Borland C++Builder, JBuilder, and IntraBuilder.

Borland Improves Support for Corporate Developers

Scotts Valley, CA — Increasing its focus on corporate IT, Borland announced a new developer support system. This new support structure includes several programs differentiated by levels of support, rather than by product lines. Borland hopes this unified process will speed response times and reduce the number of support contracts needed.

Since May, Borland has been taking orders for its new support contracts. Borland has, at the same time, continued to honor all existing contracts. The new support programs apply to customers in the US and Canada, and are scheduled to be in operation this month. They include: Installation Assist, offering customer support via telephone on Borland workstation software products; and Primary Assist, providing per-minute telephone support on local installation and product usability.

Developers can choose among several programs, including Developer Incident Assist, Priority Developer Assist, and Extended Developer Assist.

Developer Incident Assist provides phone support for questions concerning installation, programming, connections to database servers, and usability of Borland products in a workstation network or client/server environment, on a per incident basis.

Priority Developer Assist is an annual developer support service with priority hotline assistance, during service hours, on Borland workstation products. The service covers questions concerning installation, programming, connections to database servers, and usability of Borland products in a workstation, network, or client/server environment for a predefined number of 15 incidents or 12 months.

Extended Developer Assist offers priority hotline services during service hours on Borland workstation products, including extended products such as Delphi/400. Questions concerning the installation, programming, connections to database servers, and usability for Borland products in a workstation, network, or client/server environment are covered for a predefined number of 15 incidents or 12 months.

For more information on these programs, call Borland at (408) 431-1064.

Droege's 1997 Developers Competition

Durham, NC — Droege Computing Services, Inc. has announced the 1997 Developers Competition, to be held October 7 to 9 in Durham, NC.

In the past six years, this competition has attracted one- or two-developer teams from 16 countries. Contestants build a typical business application for a charity, and may use any product on any platform. Past events have produced applications for a Child Protection Team, the Duke Primate Center, the American Dance Festival, Habitat for Humanity, Sunshares, Rainbow House, and others.

This year developers can select from special categories, including Internet, object-oriented, RAD, GUI, and client/server.

The results are judged by industry experts, and the winners split US$10,000 in cash. Typically, another US$500,000 of sponsor-donated software products is also distributed among the contestants.

For more information, visit http://www2.interpath.net/devcomp.

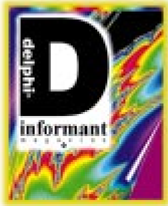
Apple Computer's Worldwide Vice President Joins Borland

Borland has appointed John Floisand, 52, to vice president of US Sales. Floisand has over 25 years of sales experience, most recently with Apple Computer, Inc. where he was responsible for worldwide sales.

Floisand plans to use his experience in sales, customer service, operations and support to help build Borland's direct support business. This involves improving relationships with VARs and systems integrators to extend support for corporate customers. Additionally, Floisand plans to improve the segmentation of Borland's product range, packaging, prices, distribution, and support to better serve customers.

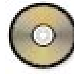
During Floisand's 11 years at Apple Computer, he held various management positions, including: senior vice president of Worldwide Sales; president of Apple Pacific; vice president of Sales, Customer Services, Operations and Support; and director of UK Sales.

table of CONTENTS



ON THE COVER

Delphi 2 / SMTP / Winsock



By Gregory Lee

Internet Delphi: Part I

Creating an SMTP E-Mail Client Program

E-mail is being used in some ingenious and non-traditional ways that go beyond sending a message to a friend or business acquaintance. Automated product registrations, program support, even database reporting can be tied into the global e-mail distribution system. Adding Internet e-mail capabilities could be just the thing to turn your next ho-hum application into an exciting and useful product.

Over the course of the next few months, we'll follow the development of a fully functional Internet e-mail program written entirely within Delphi. Although we'll focus on traditional e-mail functions, you can adapt the underlying code to just about any application.

Getting Started

If you haven't used Delphi to write an Internet application before, you may want to do a little homework. A few good books are available that focus on programming for the Internet, but ones that help you do it in Delphi are few and far between. The August 1996 issue of *Delphi Informant* contains an article I wrote that describes a Finger program written in Delphi. Truth be told, there's not a lot to implementing the Finger protocol itself, and the bulk of that article is really about Winsock.

Winsock is the Windows version of the original Berkeley sockets interface. The sockets interface was developed to provide a simple API for network applications based on the TCP/IP network protocol. You don't need to understand what TCP/IP is, or how it works, to use Winsock. What you do need is the WINSOCK.DLL and a basic knowledge of the functions available. In the Delphi Finger article, I touched on some of the more commonly used functions. If you want to understand the low-level stuff, the Finger article is a good place to start. To keep things simple here, however, we won't discuss the Winsock interface much more.

Pick a Protocol

Internet e-mail is governed by two basic protocols:

1) The Simple Mail Transfer Protocol (SMTP) lays out the rules for sending messages, from the e-mail clients' point of view.
2) The Post Office Protocol (POP) defines the process for retrieving messages.

In this installment, we'll focus exclusively on the implementation of SMTP. POP is a little more involved, but we'll get around to it. You'll be able to apply a lot of what you learn here when we get to POP.

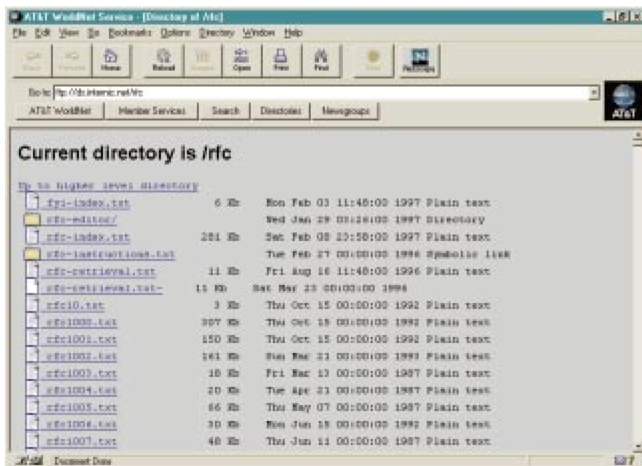


Figure 1: The ftp site ds.internic.net/rfc contains an index to all RFC documents, as well as the documents themselves.

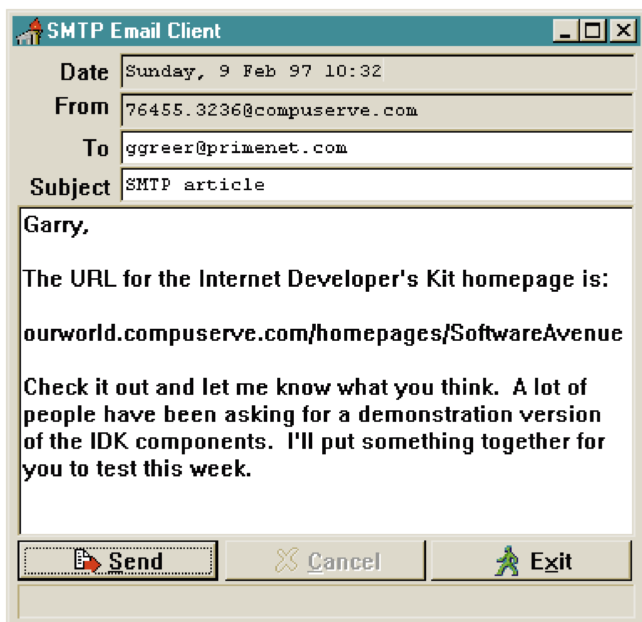


Figure 2: A typical SMTP e-mail client session.

If you want a complete description of SMTP, you should read RFC 821, "Simple Mail Transfer Protocol" by J. Postel (see Figure 1). RFC stands for Request For Comment, and 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.

Connecting on Cue

In a typical e-mail session, users enter their address and the address of the person to whom they're sending the message. They will then enter the subject and body of the message (see Figure 2). When everything is ready to go, they'll click the Send button. This is our application's cue to initiate the SMTP conversation.

Before we can connect to the SMTP server, we must know where to find it. Part of the server's location can be taken directly from the e-mail address of the person to whom the message is being sent. In a typical e-mail address, everything to the right of the @ sign is referred to as the *host name*. For example, in the e-mail address garry@goodnet.com, the host name is goodnet.com. If you want an explanation of how the host name is translated into an IP address, you can pore through the Windows Socket Specification (or get a copy of the Finger article mentioned earlier). From our perspective, the process itself is less important than the result.

The host name gets us most of the way there, but we still need to know where on the host system we can find the SMTP mail server. RFC 821 tells us that the mail server will be listening for calls at port number 25. You can think of Internet port numbers as the extension numbers in a telephone system. Most established protocols have a fixed port number where clients and servers can count on hooking up. These fixed port assignments are often referred to as "well-known ports," and the "well-known port" for SMTP happens to be port 25.

With the host name and port number now in place, we're ready to establish the connection. Unfortunately, the connection itself doesn't happen instantaneously. Depending on the route the connection takes and the amount of network traffic, it may take a few seconds. Sooner or later though, we should get a notification message indicating a successful connection.

Welcome to the Machine

According to RFC 821, when we connect to the SMTP server, it will send us a greeting. Again, Winsock sends a notification message to indicate that the data has arrived. We'll use the Winsock **recv** function to retrieve the greeting message.

The greeting could contain just about anything, but according to SMTP, we can count on one constant element: the first three characters in the greeting message will be 220. Now we could check the first three characters of every message we get during an SMTP session for these three magical characters, but it's more efficient to create something called a *state machine*, so we only look for the 220 prefix when appropriate. The state machine is also handy because it lets the program remember how far along our session has progressed at any given time. Are we waiting for the greeting? Have we sent the response? Are we waiting for the server to acknowledge something we've sent?

A simple state machine can be implemented with one global variable (the current state), and a **case** statement that executes whatever code is appropriate, given the current state. At each stage in the protocol, our program will proceed by sending something new to the server, then waiting for a response. When the response or acknowledgment message is received, the state machine is bumped to the next level.

In our SMTP Email Client program, an SMTP state machine is implemented using the global variable *State* and the **case** statement inside the *SmtpEngine* procedure. To make the code a little clearer, we've also defined a new type for the *State* variable. Appropriately enough, we've called this new type *TState* (see Figure 3). With the new type established in this way, the function of the code in *SmtpEngine* is fairly obvious.

```
TState = (Inactive, Connected, HelloSent, FromSent,
          ToSent, DataStart, SendingData, DataEnd,
          QuitSent, UnChanged);
```

Figure 3: The definition of the *TState* variable type.

A Warm HELO

Initially, our *State* is *Inactive*. Once we've connected to the SMTP server, our *State* changes to *Connected*. The *Connected* case handles this state by scanning incoming messages for the 220 prefix we're expecting. When we get the response we're looking for, we can proceed to the next step, which is to send the server a warm HELO. No, that's not a typo. RFC 821 indicates that we must respond to the server's greeting by sending a line back to the server with the keyword `HELO`, followed by the name of the host system. Once we've sent it, *State* is bumped to *HelloSent*.

The next time we receive a message from the server, it will filter through our state machine, and the new message will be handled by the code in the *HelloSent* case. This code checks for a reply: `250`. Again, SMTP doesn't dictate exactly what the server will send us, but it does require that the first three characters be `250`. As soon as we receive that sequence, we can proceed to the next step.

Return to Sender

One of the basic requirements of any e-mail system is that you let the e-mail server know where a particular message is coming from and where it's headed. The need for the receiver's address should be obvious; however, the need for the sender's may not be so clear. The simple answer is that the SMTP server needs a return address in case things go bad. For example, what if the user you're sending this e-mail to just moved to another system? Typically, the host system will notify you by return mail, if an e-mail you've sent is undeliverable.

The format of the line we send to the server indicating the return address is:

```
MAIL FROM:<user@host>
```

where *user@host* is our Internet e-mail address. After this line is sent, we set the global state variable to *FromSent*, so the state machine will know what to look for as a response.

The code in the *FromSent* case will look for the appropriate reply code, which, in this case is another 250. If you haven't bought into the importance of this state-machine scheme by now, the fact that we're getting a second 250 reply message should close the deal. Up until this point, we could have gotten away with using the reply number to indicate our next move; but if we're getting the same reply to MAIL FROM and HELO, that approach would clearly be inadequate.

Once the `250` reply is detected, we can send the destination address. The format of this line is:

```
RCPT TO:<user@host>
```

where *user@host* is the Internet e-mail address of our intended receiver. If there are multiple receiver addresses, we simply wait for a reply and send another RCPT TO:<user@host> message with the next address. Once all the receiver lines are sent, the *State* variable becomes *ToSent*, so the state machine will be ready to move forward when a response arrives.

The *ToSent* case looks for another `250` and, once we get it, we're nearly ready to send the text of our message.

Are We There Yet?

Before we can actually send the text of the message, we must give the SMTP server a little warning. The keyword used to accomplish this is `DATA`. After the `DATA` message has been sent, the *State* variable is set to *DataStart*, and incoming messages are filtered through the *DataStart* case.

This time we're looking for a return message of `354`. Once we get it, we can start sending the text of our e-mail. Sort of. According to RFC 822, "Standard for the format of ARPA Internet text messages" by D. Crocker, Internet e-mail messages must conform to some additional rules. Most of these rules have to do with something called *headers*. Message headers are those things at the top of every Internet e-mail you get; they tell you when and where the message originated, who it was sent to, and how it came to your system. Some of that stuff is tacked on along the way, but a few things must be included from the beginning.

Specifically, we need to include a From header, a To header, a Date header, and a Message-ID header. There are a lot of other, optional, header lines such as Subject, Cc, Bcc, Reply-To, as well as an undefined number of Extension headers. Extension headers are easy to spot, because they all start with a leading X-. Among other things, they're often used to identify the e-mail program and version that generated the message.

We already have most of the information needed for these headers, so it's just a matter of building some strings and shoving them into the queue. Using the Delphi functions *Time* and *DateTimeToString*, we can build the Date header easily enough, but where does this Message-ID header come from?

The Message-ID header uniquely identifies a specific piece of e-mail. There's no way to be positive a message identifier we generate will be unique. However, we can greatly increase the odds by using something like our e-mail address and the current date and time. After all, what are the odds that somebody else is going to use our e-mail address to identify a piece of their e-mail? If you combine that with the current time, even we would have to try pretty hard to create another message with exactly the same identifier.

After we place all the headers into the pipeline, a blank line is sent after them to identify the end of the header section. Now we're ready to send the text of the message.

Make It So

The text of the message is sent one line at a time. At this point, we don't even have to wait for the server to respond after each line. The only problem we have to be even remotely concerned about is the possibility that a line of the message will start with the period character. That's a potential cause for alarm, because that happens to be the signal we use to tell the mail server that we've reached the end of the message text. To send a line of text that starts with a period, we must quote the period character by preceding it with yet another period character. Simple enough.

Each of the lines are sent, and the state machine remains in the *SendingData* mode until we reach the end of the text. Then we send the termination signal — a line containing a single period character — and advance the *State* variable to *DataEnd*. If all goes well, the mail server responds to our signal with another `250` reply code.

To finish, we use the message `QUIT`, and advance the *State* variable again to *QuitSent*. The mail server reacts by sending us a `221` reply code, then closes the connection. All we have to do now is close our end of the connection, and let the user know the e-mail has been sent.

Your Mileage May Vary

The most common error message you're likely to encounter with the sample program is:

```
Valid name, no data record of requested type
```

This message is a direct pass-through from Winsock. This usually indicates that one of two name lookups has failed.

When we're attempting to connect with the SMTP server, among the first low-level tasks we must complete is to translate the given host name into an Internet IP address. Every registered Internet domain has at least one unique IP address.

The Internet uses the IP address for identification and routing. The host name is essentially a convenient alias for this address. The mechanism used to store and report the relationship between host names and IP addresses is called the Domain Name System (DNS). If the DNS lookup for a host name fails, Winsock will give the error code for:

```
Valid name, no data record of requested type
```

You may also see this message on a LAN where the 'HOST-NAMES' file has not been set up properly, or does not contain an entry for the host name you've given.

After the host name has been resolved, a second name lookup may be used to translate the service name 'smtp' into its associated "well-known" port number. The translation between common Internet service names and their "well-known" port numbers is handled by way of another (much smaller) lookup table. If this translation fails, the Winsock error code again indicates:

```
Valid name, no data record of requested type
```

To make things even more confusing, you may also see this error message if you have one or more conflicting versions of WINSOCK.DLL and/or WSOCK32.DLL on your system. This often happens when one version of the DLL exists in your \Windows\System directory and is used to establish your local TCP/IP network connection, and another version is used by the dialer that was provided with your Internet account.

Different versions of Winsock and WSock32 are generally not compatible, so the golden rule is: Whatever WINSOCK.DLL or WSOCK32.DLL is used to establish your Internet connection must also be the first and only version available to your Internet client program. Typically, the first hint that this is a problem occurs when you see:

```
Valid name, no data record of requested type
```

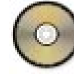
for a host name that you know is a valid registered domain.

Conclusion

To turn this into a full-featured mail program, we'll probably want to log the message at this point, or at least record somewhere the fact that a message has been sent. And then there's receiving messages — but we'll get to that next time. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUL\DI9707GL.

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 their Delphi development tool, Internet Developer's Kit. Greg can be reached by e-mail at 76455.3236@compuserve.com.



By Ray Lischner

What's New with Experts?

A Look at Project and Module Creators

Delphi 3 has many new, exciting features — packages, interfaces, ActiveX support, and more. What you might not know is that Delphi's Open Tools API also sports some enhancements. This article examines two of the major new features for writing experts and wizards: project creators and module creators.

The Open Tools API lacks formal documentation, so this article refers to specific source files for the Open Tools API. These source files are in the \Source\Toolsapi directory, provided you have Delphi 3 Professional or higher. If you're not familiar with the Open Tools API and writing experts in Delphi, several books provide solid information: *Hidden Paths of Delphi 3* [Informant Press, 1997] and *Secrets of Delphi 2* [Waite Group Press, 1996] by Ray Lischner, *Delphi Component Design* [Addison-Wesley Developers Press, 1997] by Danny Thorpe, and *The Revolutionary Guide to Delphi 2* [WROX Press, 1996] by Paul Hinks, et al.

Start by looking at the *TIToolServices* class in ToolIntf.pas, where you will see several new methods. This article discusses the *ModuleCreate* and *ProjectCreate* methods and their related classes, *TIModuleCreator* and *TIProjectCreator* (which you can find in EditIntf.pas).

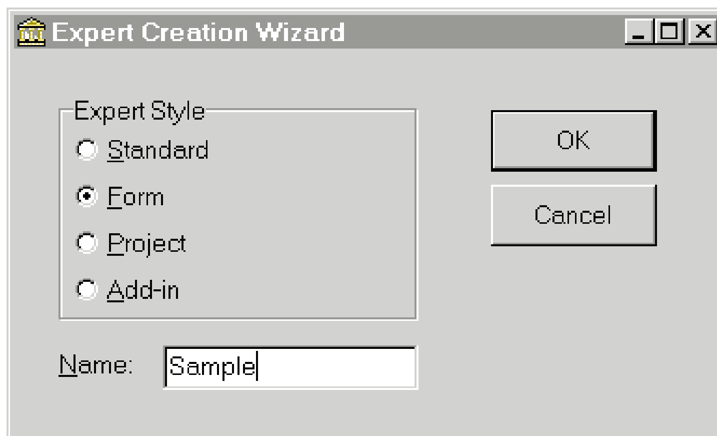


Figure 1: The Expert Creation Wizard.

Jump Right In

First, let's look at how project and module creators work, using an expert-creation wizard as an example. The wizard creates a skeleton for an expert, based on a few tidbits of information from the user. Specifically, it creates a library project to create and register the wizard. It also creates a unit that declares the expert interface class. Figure 1 shows the expert-creation wizard at work. It prompts the user for the kind of expert to create and an expert name, then uses this information to create its files.

The first step is the same for any expert: declare the expert interface class. Listing One (beginning on page 14) shows the Expert unit, which declares the *TExpertCreator* class. The expert-creation wizard is a project expert, so it defines an author, comment, repository page, and so on. The *Execute* method calls *RunExpert*, which is in the ExptDlg unit, shown in Listing Two (beginning on page 15). *RunExpert* opens the main form to get the expert style and name from the user. If the user clicks the OK button, the expert employs the user's information to instantiate a *TExpertInfo* object, which it passes to *CreateExpert* in the ExptGen unit.

The interesting part of the expert is the ExptGen unit (see Listing Three, beginning on page 15). This unit declares the project creator class, *TProjectCreator*, and a module creator, *TModuleCreator*. The project creator creates the library source file. Notice that most of its methods return empty strings, which tell Delphi to use its default settings.

For example, *GetFileName* must return the filename for the project's source file. An empty string, however, tells Delphi to use its default — Project1.dpr — in the current directory. The two most interesting methods are *NewProjectSource* and *NewDefaultModule*.

Flag Literal	Description
cpCustom	Custom project (empty project file).
cpApplication	Application (project starts with default **program** declaration).
cpLibrary	DLL (project starts with default **library** declaration).
cpCanShowSource	Delphi displays the project source file.
cpExisting	The project source file already exists.

Figure 2: The flags for *ProjectCreate*.

NewProjectSource returns the source code for the project's .DPR file. In this case, the source code is relatively short, so this function calls *Format* to produce the result string. The parameters for *Format* are the project name and the expert interface class. Delphi supplies the project name as an argument to *NewProjectSource*. Your expert supplies the class name, from the *TExpertInfo* object. You must ensure that the source code is correct, or your expert's user might be upset when he or she is unable to compile the resulting project.

After Delphi creates the project source file, it calls *NewDefaultModule*, which creates the sole unit for the project. As you might expect, given the topic of this article, *NewDefaultModule* uses a module creator for the expert interface unit.

The module creator, *TModuleCreator*, has a more difficult job than the project creator. Just like *TProjectCreator*, most of the methods of *TModuleCreator* return empty strings. As you might have already guessed, that tells Delphi to use its defaults for the filename, and so on. *NewModuleSource*, however, is complex. It returns the complete source code for the module file — in this case, the expert interface unit. You can use the same *Format* technique that you used in the project creator, but it's hard to work with. Instead, the module creator writes its text to a stream.

One of the advantages of using a stream is that you can define a custom class — in this case, *TTextStream*. With a custom stream class, you can define methods that work the way you want. The *TTextStream* class defines methods that make it easier to write to the stream one line at a time.

To summarize, the *CreateExpert* procedure does the real work of creating the project. *CreateExpert* constructs an instance of *TProjectCreator* and passes that instance to *ToolServices.ProjectCreate*. Delphi calls the methods of the project creator to obtain information about the project, and uses that information to create and open a new project. The project creator adds units to the project in the *NewDefaultModule* method. The sample expert has one unit, so *NewDefaultModule* creates one module creator instance and passes that object to *ToolServices.ModuleCreate*. Delphi calls back to the module creator to obtain information about the unit file, and creates the unit's source file accordingly. Now it's time to look more closely at the creator classes, and understand what they do.

Project Creators

A project creator puts you in control when creating a new project. The project creator instructs Delphi how to create the project's source (.DPR) and resource (.RES) files. It can tell Delphi to use existing files, or to create new ones. Note that Delphi doesn't actually create a file on disk when the project creator runs. Instead, it creates files in memory. When the user saves the project, Delphi saves its in-memory buffers to disk. This is the same behavior as when Delphi creates its default project. Delphi assigns the name Project1 to the project and Unit1 to the blank form, but doesn't create any files on disk. This gives the user the greatest degree of flexibility to rename the project or units when saving files, or to abandon the project without saving anything.

To use a project creator, you first derive a class from the interface class, *TIProjectCreator*, overriding all its methods. The expert creates an instance of your project creator class, and passes that object as the first argument to *TIToolServices.ProjectCreate*. When your expert calls *ProjectCreate*, Delphi calls back to your project creator object to learn the project's name, source file, resources, and so on. The declaration for the *ProjectCreate* method of *TIToolServices* is:

```
function ProjectCreate(ProjectCreator: TIProjectCreator;
  CreateFlags: TCreateProjectFlags): TIModuleInterface;
```

If you don't need to specify the project's source file and resources, you can let Delphi create a default project by passing a **nil** pointer for the *ProjectCreator* argument. In this case, the *CreateFlags* argument tells Delphi what kind of project to create. Figure 2 lists the project creation flags; except for *cpCanShowSource*, the flags have meaning only if *ProjectCreator* is **nil**. When *ProjectCreator* is **nil**, you must choose one of *cpCustom*, *cpApplication*, or *cpLibrary* to specify what kind of project you want Delphi to create.

The *cpApplication* project is Delphi's default application and default main form. The *cpLibrary* project is Delphi's default DLL project. The *cpCustom* flag is for any other kind of project. For a *cpCustom* project, Delphi starts with an empty file, which is not what you usually want your expert to do. Instead, when using *cpCustom*, your expert can create the project source file explicitly, and use the *cpExisting* flag to tell Delphi that the file exists.

If you set the *cpExisting* flag (and *ProjectCreator* is **nil**) and the project source file doesn't exist, Delphi will create a default project file, according to the *cpApplication* (default project file, but with no forms), *cpLibrary* (default DLL project), or *cpCustom* (empty source file) flag.

Regardless of whether *ProjectCreator* is **nil**, you can use the *cpCanShowSource* flag when you call *ProjectCreate*. This flag tells Delphi to display the project source file in its Code Editor. If you exclude *cpCanShowSource*, Delphi hides the project source file. For an application with forms, you

would typically omit *cpCanShowSource*; but for projects such as a library, you would most likely want to include this flag. *ProjectCreate* returns a module interface object for the new project file. You use the module interface to further modify the project file. No matter how your expert uses the interface, it must call *Free* to release the interface.

TIProjectCreator Class

The first argument to *ProjectCreate* is the project creator object. If you want to create a default project, pass a **nil** pointer. If you want to define the resource, project source file, filename, or other aspects of the project, you must define a project creator. The following sections describe the functions and procedures of *TIProjectCreator*. Remember to override all of these, because *TIProjectCreator* declares them as virtual, abstract methods. (You can find the declaration for *TIProjectCreator* in the EditIntf.pas file.)

Existing Function

```
function Existing: Boolean;
```

Override the *Existing* function to return *True* if the project files exist, or *False* if your expert is creating new files. If you define *Existing* to return *False*, Delphi assumes the files don't exist, and calls *NewProjectSource*, *NewDefaultModule*, and *NewProjectResource*. If one or more files exist, define *Existing* to return *True*. Delphi looks for the files; if a file exists, Delphi doesn't call the corresponding function. That is, if the .DPR file exists, Delphi doesn't call *NewProjectSource*. If the .RES file exists, Delphi doesn't call *NewProjectResource*. Delphi doesn't look for any unit files — if *Existing* returns *False*, Delphi doesn't call *NewDefaultModule*.

GetFileName Function

```
function GetFileName: string;
```

Override the *GetFileName* function to return the full path to the project's .DPR file. If the file doesn't exist, Delphi doesn't create the file yet, but marks the project as modified, so the user can choose to save the file, or abandon the new project without creating any files. The *GetFileName* function can also return an empty string, in which case Delphi uses a default project name (e.g. Project1) in the current directory.

GetFileSystem Function

```
function GetFileSystem: string;
```

Override the *GetFileSystem* function to return the name of a registered file system that will store the project's source and resource files. In most cases, your project creator will return an empty string, telling Delphi to use the default file system.

NewDefaultModule Procedure

```
procedure NewDefaultModule;
```

Delphi calls the *NewDefaultModule* procedure after it calls *NewProjectSource*. You can use *NewDefaultModule* to add units and forms to the project. If *Existing* returns *True*, Delphi doesn't call *NewDefaultModule*. If you want a project that doesn't have any units or forms, write *NewDefaultModule* so it returns without doing anything.

NewProjectResource Procedure

```
procedure NewProjectResource(Module:
  TIModuleInterface);
```

Delphi calls the *NewProjectResource* procedure after *NewDefaultModule* to report to your expert about the project's resource file. If *Existing* returns *True*, and the project resource file exists, Delphi doesn't call *NewProjectResource*. If *Existing* returns *False*, or if the resource file doesn't exist, the *Module* argument is the module interface for the new resource file. You can call its *GetProjectResource* method to obtain a resource file interface.

When Delphi calls *NewProjectResource*, the resource file is empty. Remember that when you save a project, and you haven't specified an application icon, Delphi supplies a default MAINICON resource. In other words, if you're satisfied with the default application icon, you can write *NewProjectResource*, which will do nothing except free the module interface.

One important aspect of a module interface object is that Delphi creates a single *TIModuleInterface* object for each file, and shares that object with all the experts that need it. Delphi uses reference counting to make sure it doesn't free a module interface object while an expert still retains a reference to it. When your expert finishes using any module interface object, it must free the object reference so Delphi can keep an accurate reference count. Thus, even if *NewProjectResource* doesn't use its *Module* argument to add any resources, it must call *Module.Free* to release the object reference.

NewProjectSource Function

```
function NewProjectSource(const
  ProjectName: string): string;
```

Delphi calls the *NewProjectSource* function to obtain the contents of the project's source (.DPR) file. If *Existing* returns *True,* and the project source file exists, Delphi doesn't call *NewProjectSource*. Otherwise, Delphi calls *NewProjectSource*, and passes the name of the project as the *ProjectName* argument. Typically, your expert would use the name in the source file's **program** or **library** statement. This function must return the full contents of the file as a single string.

Module Creators

A module creator is similar to a project creator, but they differ in many of their details. The biggest difference is that a project creator creates a project, while a module creator creates a module — that is, a unit source file and possibly a form or data module. The methods of *TModuleCreator* work slightly differently than the corresponding methods of *TProjectCreator*, so you can't always apply to module creators what you learned from project creators.

To use a module creator, you must derive a class from *TIModuleCreator*, and override its methods. Your expert creates an instance of your derived module-creator class, and passes that object to the tool services *ModuleCreate* function. The declaration for the *ModuleCreate* function is:

```
function ModuleCreate(ModuleCreator: TIModuleCreator;
  CreateFlags: TCreateModuleFlags): TIModuleInterface;
```

The *ModuleCreate* function creates a new unit, form, or data module, in a manner that your expert defines. You define how to create the module by deriving a class from *TIModuleCreator*. Your expert creates an instance of your derived, concrete class. This creator object returns the module name, file system, and so on. Delphi calls back to your module creator object after it creates a form, passing a form interface object as an argument.

Flag Literal	Description
cmAddToProject	Add the new module to the currently open project.
cmMainForm	Make the new module the main form. Requires *cmAddToProject*. Ignored if there is no form.
cmMarkModified	Mark the module as modified so Delphi will prompt the user to save it before closing the file.
cmShowForm	Show the form in the form designer (ignored if there is no form).
cmShowSource	Show the unit in the Code Editor.
cmUnNamed	Mark the module as unnamed. When the user closes or saves the file, Delphi will prompt the user for a filename.

Figure 3: The significant flags for *ModuleCreate*.

ToolIntf.pas lists several flags in the *TCreateModuleFlags* type. The *ModuleCreate* function heeds only some of the flags in its second argument. The other flags are there for *CreateModule* and *CreateModuleEx*. Figure 3 describes the flags that matter to *ModuleCreate*.

When your expert calls *ModuleCreate*, Delphi calls back to your module creator object, calling *GetAncestorName*, *GetFormName*, *GetFileName*, *Existing*, *GetFileSystem*, *NewModuleSource*, and *FormCreated* (in that order). Delphi creates the unit and form files and returns a *TIModuleInterface* object. Most form experts can free the module interface without doing anything else with it.

If you return an empty string from *GetFileName*, make sure you include the *cmUnNamed* flag when calling *ModuleCreate*. Delphi generates a default name, but the user probably wants to choose a different name when saving the file. The *cmUnNamed* flag tells Delphi that the current name is just a placeholder, so Delphi prompts the user for a real filename when it saves the file.

If *Existing* returns *False*, make sure you include the *cmMarkModified* flag. This tells Delphi to mark the new files as modified, forcing the user to save them before closing the files or project. If you omit *cmMarkModified*, Delphi will let the user close the project without saving the new unit or form files. In most cases, a form expert will use the *cmShowSource* and *cmShowForm* flags. Without them, Delphi creates the new files, but doesn't show the files to the user. A form expert's job is to create new units and forms, and the user probably wants to see the newly created source code and form.

A project expert, on the other hand, might omit *cmShowSource* and *cmShowForm* for certain units. Perhaps the project expert creates an application with several forms, including an About dialog box. The About box is less important than the application's main form. The project expert can use *cmShowForm* and *cmShowSource* when creating the main form, but omit these flags when creating the About box. This avoids cluttering the screen with too many forms.

TIModuleCreator Class

Find the declaration of the *TIModuleCreator* class in the EditIntf unit. To use this class, derive a class from *TIModuleCreator*, overriding all its methods. Your expert uses your derived class in a call to *TIToolServices.ModuleCreate*, as previously discussed. This section describes the *TIModuleCreator* class in depth.

Existing Function

```
function Existing: Boolean;
```

Override the *Existing* function to return *True* if the module source file exists, or *False* if your expert will create a new file. If *Existing* returns *True*, the file must exist, or Delphi raises an *EFOpenError* exception. Note that this behavior is different from that of a project expert, which checks whether the file exists, but continues regardless.)

Note that Delphi ignores the *cmExisting* flag in the call to *TIToolServices.ModuleCreate*. If the module files exist, make sure *Existing* returns *True*.

FormCreated Procedure

```
procedure FormCreated(Form: TIFormInterface);
```

Delphi calls the *FormCreated* procedure after it has created the form. If the source file (as returned from *NewModuleSource*) doesn't contain a *{$R *.DFM}* directive, Delphi doesn't call *FormCreated*. If the unit source file contains a *{$R *.DFM}* directive, Delphi creates the form, and passes the form interface to *FormCreated*. Typically, your expert would use the form interface to set the form's properties, add components, and so on.

GetAncestorName Function

```
function GetAncestorName: string;
```

Override the *GetAncestorName* function to return the name of the ancestor form. If your new form doesn't use form inheritance, return an empty string or `Form`. To create a data module, use `DataModule` as the ancestor name. If your module creator is creating a source unit without a form, return an empty string.

GetFileName Function

```
function GetFileName: string;
```

Override the *GetFileName* function to return the path to the unit's source (.PAS) file. Your module creator can also return an empty string, in which case Delphi chooses an appropriate default filename, e.g. Unit1.pas in the current directory. Delphi extracts the unit name from the filename by stripping the drive, directory, and extension.

If *Existing* returns *True*, the file named by *GetFileName* must exist. If the file doesn't exist, Delphi will raise an *EFOpenError* exception.

GetFileSystem Function

```
function GetFileSystem: string;
```

Override the *GetFileSystem* function to return the name of the file system where you want Delphi to store the form and source file. You must name a registered file system or return an empty string to use the default file system.

GetFormName Function

```
function GetFormName: string;
```

Override the *GetFormName* function to return the name of the unit's form. Return an empty string if your module creator is not creating a form, or if you want to use a default form name, e.g. Form1. Another way to obtain a form name is by calling *TIToolServices.GetNewModuleAndClassName*.

If you define *GetFormName* so it doesn't return an empty string, you must ensure the form name is a valid Delphi/Pascal identifier, and is unique in the project. Your expert can test whether a form name is unique by comparing it with the names of all the other forms in the current project.

NewModuleSource Function

```
function NewModuleSource(UnitIdent, FormIdent,
  AncestorIdent: string): string;
```

Delphi calls the *NewModuleSource* function to obtain the source code for the new module. The *UnitIdent* argument is the name of the new unit, *FormIdent* is the name of the form, and *AncestorIdent* is the name of the ancestor form. Your module creator must return the contents of the unit's .PAS file as a string.

If you're defining a form or data module, make sure the unit's source code contains a proper declaration of the form's class, and a reference to the form description (.DFM) file. If you want to create a plain unit without a form, feel free to ignore the *FormIdent* and *AncestorIdent* arguments.

Remember to preface the form and ancestor names with "T" to turn the names into type names. If your form doesn't use form inheritance, the ancestor name is `Form`. This makes your job easier when creating the source code for the form; you can use the same code to generate the source string for all situations.

Conclusion

Project and form experts are ideal solutions for defining standardized projects, units, and forms. Templates in the Object Repository are static, and offer little flexibility. Experts, on the other hand, have the full power of Delphi to ask questions of the user and create customized projects, units, and forms. Delphi 3 makes your job easier with project and module creators.

Project creators give your expert access to Delphi's default application and library projects; or your expert can take control, and define the project's source and resource files. Module creators give you control for individual units and forms. Combine the two, and you can create experts and wizards with ease. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUL\DI9707RL

This article was adapted from *Hidden Paths of Delphi 3* [Informant Press, 1997], Ray Lischner's latest book covering undocumented aspects of Delphi. *Hidden Paths of Delphi 3* reveals the hitherto undocumented Open Tools API. Lischner's first Delphi book, *Secrets of Delphi 2* [Waite Group Press, 1996], continues to be pertinent for Delphi 3.

Ray Lischner is a contributor to several Delphi periodicals, and is a familiar figure on the Delphi Usenet newsgroups. He 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, is the president of the Corvallis chapter of the Software Association of Oregon, and serves on the board of directors for the Pacific Northwest Software Quality Conference. You can reach him at delphi@tempest-sw.com.

Begin Listing One — The Expert Unit

```
unit Expert;
{ Expert-creation wizard. }

interface

uses Windows, Classes, SysUtils, Forms,
  Dialogs, ExptIntf, ToolIntf;

type
  TExpertCreator = class(TIExpert)
  public
    procedure Execute; override;
    function GetAuthor: string; override;
    function GetComment: string; override;
    function GetGlyph: HICON; override;
    function GetIDString: string; override;
    function GetMenuText: string; override;
    function GetName: string; override;
    function GetPage: string; override;
    function GetState: TExpertState; override;
    function GetStyle: TExpertStyle; override;
  end;

implementation

uses ExptDlg;

resourcestring
  sAuthor = 'Tempest Software';
  sComment = 'Create an expert';
  sName = 'Expert Wizard';
  sPage = 'Projects';

procedure TExpertCreator.Execute;
begin
  RunExpert;
end;

function TExpertCreator.GetAuthor: string;
begin
  Result := sAuthor;
end;

function TExpertCreator.GetComment: string;
begin
  Result := sComment;
end;
```

```
{ Use the application's icon as the expert's icon. If you
  define a package for this expert, make sure it has
  the MAINICON resource. }
function TExpertCreator.GetGlyph: HICON;
begin
  Result := LoadIcon(hInstance, 'MAINICON');
end;

function TExpertCreator.GetIDString: string;
begin
  Result := 'Tempest Software.ExpertCreator';
end;

function TExpertCreator.GetName: string;
begin
  Result := sName;
end;

function TExpertCreator.GetPage: string;
begin
  Result := sPage;
end;

function TExpertCreator.GetStyle: TExpertStyle;
begin
  Result := esProject;
end;

function TExpertCreator.GetMenuText: string;
begin
  Result := '';
end;

function TExpertCreator.GetState: TExpertState;
begin
  Result := [];
end;

end.
```

End Listing One

Begin Listing Two — The ExptDlg Unit

```
unit ExptDlg;
{ Expert-creation wizard. }

interface

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

type
  TMainDlg = class(TForm)
    ExpertStyle: TRadioGroup;
    Label1: TLabel;
    ExpertName: TEdit;
    OkButton: TButton;
    Button1: TButton;
    procedure ExpertStyleClick(Sender: TObject);
    procedure ExpertNameChange(Sender: TObject);
  private
    { Private declarations }
    procedure EnableOkButton;
    function GetExpertInfo: TExpertInfo;
  public
    { Public declarations }
  end;

procedure RunExpert;

implementation

uses ToolIntf, ExptIntf;

{$R *.DFM}

{ Run the expert when the user invokes it. }
procedure RunExpert;
var
  Dlg: TMainDlg;
  Info: TExpertInfo;
begin
  Dlg := TMainDlg.Create(Application);
  try
    if Dlg.ShowModal = mrOK then
      begin
        Info := Dlg.GetExpertInfo;
        try
          CreateExpert(Info);
        finally
          Info.Free;
        end;
      end;
  finally
    Dlg.Free;
  end;
end;

{ Enable the OK button only when the user has selected
  an expert style and entered a name. }
procedure TMainDlg.EnableOkButton;
begin
  OkButton.Enabled := (ExpertStyle.ItemIndex >= 0) and
                      (ExpertName.Text <> '');
end;

procedure TMainDlg.ExpertStyleClick(Sender: TObject);
begin
  EnableOkButton;
end;

procedure TMainDlg.ExpertNameChange(Sender: TObject);
begin
  EnableOkButton;
end;

{ Create a TExpertInfo object to store the information
  the user supplied. }
function TMainDlg.GetExpertInfo: TExpertInfo;
begin
  Result := TExpertInfo.Create(
    TExpertStyle(ExpertStyle.ItemIndex),ExpertName.Text);
end;

end.
```

End Listing Two

Begin Listing Three — The ExptGen Unit

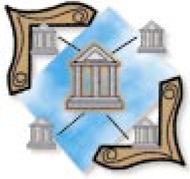
```
unit ExptGen;
{ Expert-creation wizard. }

interface

uses ToolIntf, ExptIntf;

type
  TExpertInfo = class
  private
    fStyle: TExpertStyle;
    fName: string;
    function GetClassName: string;
  public
    constructor Create(Style: TExpertStyle; Name: string);
    property Style: TExpertStyle read fStyle;
    property Name: string read fName;
    property ClsName: string read GetClassName;
  end;

procedure CreateExpert(Info: TExpertInfo);

implementation
```

```
uses SysUtils, Classes, EditIntf, TypInfo;

resourcestring
  sInvalidStyle = 'Invalid expert style, %d';
  sMainLogic = '  { Fill in the main logic here. }';
  sMainIcon1 = '{ In a package, you must explicitly add the';
  sMainIcon2 = '  MAINICON resource to the package. In a DLL,';
  sMainIcon3 = '  set the icon in the project options. }';
  sAuthor = 'Author';
  sOrganization = 'Organization';
  sFormPage = 'Forms';
  sProjectPage = 'Projects';

type
  TProjectCreator = class(TIProjectCreator)
  private
    fInfo: TExpertInfo;
  public
    constructor Create(Info: TExpertInfo);
    function Existing: Boolean; override;
    function GetFileName: string; override;
    function GetFileSystem: string; override;
    function NewProjectSource(
      const ProjectName: string): string; override;
    procedure NewDefaultModule; override;
    procedure NewProjectResource(
      Module: TIModuleInterface); override;
    property Info: TExpertInfo read fInfo;
  end;
  TModuleCreator = class(TIModuleCreator)
  private
    fInfo: TExpertInfo;
  public
    constructor Create(Info: TExpertInfo);
    function Existing: Boolean; override;
    function GetAncestorName: string; override;
    function GetFileName: string; override;
    function GetFileSystem: string; override;
    function GetFormName: string; override;
    function NewModuleSource(UnitIdent, FormIdent,
      AncestorIdent: string): string; override;
    procedure FormCreated(Form: TIFormInterface); override;
    property Info: TExpertInfo read fInfo;
  end;


{ Create the expert. }
procedure CreateExpert(Info: TExpertInfo);
var
  ProjectCreator: TProjectCreator;
begin
  ProjectCreator := TProjectCreator.Create(Info);
  try
    ToolServices.ProjectCreate(ProjectCreator, []);
  finally
    ProjectCreator.Free;
  end;
end;


{ TExpertInfo class }
constructor TExpertInfo.Create(Style: TExpertStyle;
  Name: string);
begin
  inherited Create;
  fStyle := Style;
  fName := Name;
end;

{ Build a class name from the expert name by removing all
  non-alphanumeric characters and prepending 'T' for Type. }
function TExpertInfo.GetClassName: string;
var
  I: Integer;
begin
  Result := 'T';
  for I := 1 to Length(Name) do
    if Name[I] in ['a'..'z','A'..'Z','_','0'..'9'] then
      Result := Result + Name[I];
end;

{ TProjectCreator class. Remember the expert info object. }
constructor TProjectCreator.Create(Info: TExpertInfo);
begin
  inherited Create;
  fInfo := Info;
end;

const
  CRLF=#13#10;

{ Create and return contents of project's source file. }
function TProjectCreator.NewProjectSource(
  const ProjectName: string): string;
begin
  Result := Format(
    'library %s;' + CRLF + CRLF +
    'uses ShareMem, Forms, ExptIntf, ToolIntf;' +CRLF+CRLF+
    '{$R *.RES}' + CRLF + CRLF+
    'function InitExpert(ToolSvc: TIToolServices;' + CRLF +
    '  RegProc: TExpertRegisterProc;' + CRLF +
    '  var Terminate: TExpertTerminateProc):' + CRLF +
    '    Boolean; stdcall;' + CRLF +
    'begin' + CRLF +
    '  Result := True; { return False for error }' + CRLF +
    '  ToolServices := ToolSvc;'+CRLF+
    '  Application.Handle := ToolSvc.GetParentHandle;'+CRLF+
    '  RegProc(%s.Create)' + CRLF + 'end;' + CRLF + CRLF +
    'exports InitExpert name ExpertEntryPoint resident;' +
    CRLF + CRLF + 'begin' + CRLF + 'end.' + CRLF
    , [ProjectName, Info.ClsName]);
end;

function TProjectCreator.Existing: Boolean;
begin
  Result := False { New files. }
end;

function TProjectCreator.GetFileName: string;
begin
  Result := '' { Default filename. }
end;

function TProjectCreator.GetFileSystem: string;
begin
  Result := '' { Default file system. }
end;

{ Add the expert interface to the project. }
procedure TProjectCreator.NewDefaultModule;
var
  ModuleCreator: TModuleCreator;
begin
  ModuleCreator := TModuleCreator.Create(Info);
  try
    with ToolServices.ModuleCreate(ModuleCreator,
         [cmAddToProject,cmShowSource,cmUnNamed,
         cmMarkModified]) do
      { Free the module interface. }
      Free;
  finally
    ModuleCreator.Free;
  end;
end;

{ Default resources. Remember to free the
  module interface. }
procedure TProjectCreator.NewProjectResource(
  Module: TIModuleInterface);
begin
  Module.Free;
end;

{ TModuleCreator class. Remember the expert info object. }
```

```
constructor TModuleCreator.Create(Info: TExpertInfo);
begin
  inherited Create;
  fInfo := Info;
end;

{ The TTextStream class inherits from TStringStream.
  It makes it a little easier to write lines of text
  to a stream. }
type
  TTextStream = class(TStringStream)
  public
    constructor CreateEmpty;
    procedure FormatLn(Fmt: string; Args: array of const);
    procedure WriteLn(Line: string);
    procedure NewLine;
  end;

constructor TTextStream.CreateEmpty;
begin
  inherited Create('');
end;

procedure TTextStream.NewLine;
begin
  WriteString(CRLF)
end;

procedure TTextStream.FormatLn(Fmt: string;
  Args: array of const);
begin
  WriteString(Format(Fmt, Args));
  NewLine;
end;

procedure TTextStream.WriteLn(Line: string);
begin
  WriteString(Line);
  NewLine;
end;

{ Return a string containing all source code for module. Use
  a TTextStream to keep this function manageable. }
function TModuleCreator.NewModuleSource(
  UnitIdent, FormIdent, AncestorIdent: string): string;
var
  Stream: TTextStream;
  ClsName: string;
begin
  ClsName := Info.ClsName;
  Stream := TTextStream.CreateEmpty;
  with Stream do
    try
      FormatLn('unit %s;', [UnitIdent]);
      NewLine;
      WriteLn('interface');
      NewLine;
      WriteLn('uses Windows, ToolIntf, ExptIntf;');
      NewLine;
      WriteLn('type');
      FormatLn('  %s = class(TIExpert)', [ClsName]);
      WriteLn('  public');
      WriteLn('    procedure Execute; override;');
      WriteLn('    function GetAuthor: string; override;');
      WriteLn('    function GetComment: string; override;');
      WriteLn('    function GetGlyph: HICON; override;');
      WriteLn('    function GetIDString: string; override;');
      WriteLn('    function GetMenuText: string; override;');
      WriteLn('    function GetName: string; override;');
      WriteLn('    function GetPage: string; override;');
      WriteLn('    function GetState: TExpertState; override;');
      WriteLn('    function GetStyle: TExpertStyle; override;');
      WriteLn('  end;');
      NewLine;
      WriteLn('implementation');
      NewLine;
      FormatLn('procedure %s.Execute;', [ClsName]);
      WriteLn('begin');
      if Info.Style in [esForm, esProject, esStandard] then
        WriteLn(sMainLogic);
      WriteLn('end;');
      WriteLn('');
      FormatLn('function %s.GetAuthor: string;', [ClsName]);
      WriteLn('begin');
      if Info.Style in [esForm, esProject] then
        FormatLn('  Result := ''%s'';', [sAuthor])
      else
        WriteLn('  Result := '''';');
      WriteLn('end;');
      NewLine;
      FormatLn('function %s.GetComment: string;', [ClsName]);
      WriteLn('begin');
      if Info.Style in [esForm, esProject] then
        FormatLn('  Result := ''%s'';', [Info.Name])
      else
        WriteLn('  Result := '''';');
      WriteLn('end;');
      NewLine;
      if Info.Style in [esForm, esProject] then
      begin
        WriteLn(sMainIcon1);
        WriteLn(sMainIcon2);
        WriteLn(sMainIcon3);
      end;
      FormatLn('function %s.GetGlyph: HICON;', [ClsName]);
      WriteLn('begin');
      if Info.Style in [esForm, esProject] then
        WriteLn('  Result := LoadIcon(hInstance, ''MAINICON'');')
      else
        WriteLn('  Result := 0;');
      WriteLn('end;');
      NewLine;
      FormatLn('function %s.GetIDString: string;', [ClsName]);
      WriteLn('begin');
      FormatLn('  Result := ''%s.%s'';', [sOrganization, Info.Name]);
      WriteLn('end;');
      NewLine;
      FormatLn('function %s.GetMenuText: string;', [ClsName]);
      WriteLn('begin');
      if Info.Style in [esStandard] then
        FormatLn('  Result := ''%s'';', [Info.Name])
      else
        WriteLn('  Result := '''';');
      WriteLn('end;');
      NewLine;
      FormatLn('function %s.GetName: string;', [ClsName]);
      WriteLn('begin');
      FormatLn('  Result := ''%s'';', [Info.Name]);
      WriteLn('end;');
      NewLine;
      FormatLn('function %s.GetPage: string;', [ClsName]);
      WriteLn('begin');
      if Info.Style in [esForm] then
        FormatLn('  Result := ''%s'';', [sFormPage])
      else if Info.Style in [esProject] then
        FormatLn('  Result := ''%s'';', [sProjectPage])
      else
        WriteLn('  Result := '''';');
      WriteLn('end;');
      NewLine;
      FormatLn('function %s.GetState: TExpertState;', [ClsName]);
      WriteLn('begin');
      if Info.Style in [esStandard] then
        WriteLn('  Result := [esEnabled];')
      else
        WriteLn('  Result := [];');
      WriteLn('end;');
      NewLine;
      FormatLn('function %s.GetStyle: TExpertStyle;', [ClsName]);
      WriteLn('begin');
      FormatLn('  Result := %s;',
        [GetEnumName(TypeInfo(TExpertStyle), Ord(Info.Style))]);
      WriteLn('end;');
      NewLine;
      WriteLn('end.');

      Result := DataString;
    finally
```

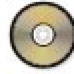
```
      Free;
    end;
end;

function TModuleCreator.Existing: Boolean;
begin
  Result := False; { Create new files. }
end;

function TModuleCreator.GetAncestorName: string;
begin
  Result := ''; { Not using inheritance. }
end;

function TModuleCreator.GetFileName: string;
begin
  Result := ''; { Use the default filename. }
end;

function TModuleCreator.GetFileSystem: string;
begin
  Result := ''; { Use the default file system. }
end;

function TModuleCreator.GetFormName: string;
begin
  Result := ''; { Not a form. }
end;

procedure TModuleCreator.FormCreated(
  Form: TIFormInterface);
begin
  { This unit has no form, so Delphi should never call
    this method. If you define a module creator that
    defines a form, remember to free the form interface,
    as shown below. }
  Form.Free;
end;

end.
```

End Listing Three



Delphi 2 / Object Pascal / Microsoft Access / Visual Basic for Applications

By *Ian Davies*



Automated Access

Creating Automation Clients: Part III

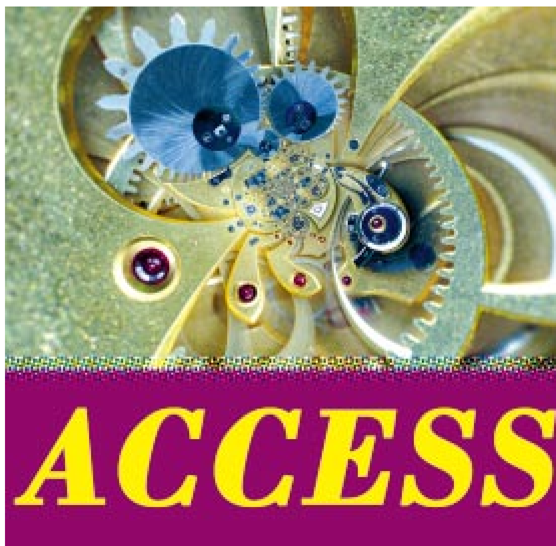
In this third and final installment of the Automation series, we'll cover the use of Microsoft Access as an Automation Server. (In the May issue of *Delphi Informant* I discussed using Word as an Automation server; last month I covered using Excel.)

Access version 2 for Windows 3.1 uses Access Basic as its underlying programming language, but versions 7 and 8 for Windows 95 (supplied with the Professional Editions of Office 95 and Office 97, respectively) use Visual Basic for Applications (VBA). Because Access version 2 cannot act as an Automation server, I will concentrate on the use of VBA for the remainder of this article.

Access as an Automation Server

Access exposes the *Application* object, which can be used for Automation. It can be used in the following way:

```
Acc := CreateOLEObject('Access.Application');
```



where *Acc* is declared as a Variant elsewhere in the program. The instance data of the Automation object is stored in *Acc*, and it is through this that you gain access to its underlying functionality. For example, to open an existing database, you would call the *OpenCurrentDatabase* method of the *Application* object, as follows:

```
Acc.OpenCurrentDatabase(
  filepath := '...path and filename of database...');
```

Similarly, the *NewCurrentDatabase* method will create a new, empty database (the *filepath* variable is implied in this case):

```
Acc.NewCurrentDatabase(
  '...path and filename of new database...');
```

Each open database has an associated *DoCmd* object that exposes various properties and methods specific to that database. For example, to open a pre-defined query in the current database, use the *OpenQuery* method of the *DoCmd* object:

```
Acc.DoCmd.OpenQuery('Sales by Category');
```

Similarly, the *OpenReport* method will open and print the report specified by the argument it is passed.

Figure 1 shows a Delphi form, in design mode, that demonstrates the principles discussed

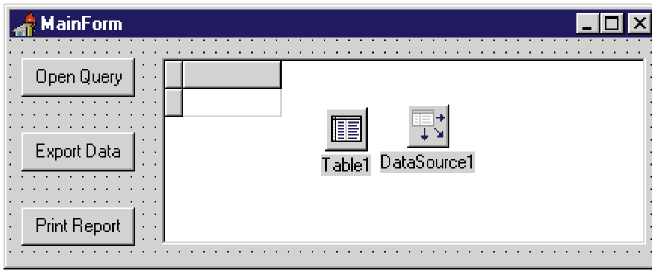


Figure 1: An Automation example.

here. The program uses the Northwind sample database that ships with all versions of Access.

One important point concerning the Northwind database is that of its splash screen: When the database is opened — normally or through Automation — it displays a splash screen. If you were using Access directly, the splash screen must be cleared before any progress can be made, because it's shown modally. When controlling Access using Automation, however, this isn't necessary, because the commands directly reference the underlying object, so your application proceeds as if the splash screen wasn't there.

(It's possible to prevent the splash screen from appearing by sending a [+] keystroke, which simulates holding down [Shift], before the database is opened. However, because the splash screen doesn't inhibit the functionality of Access when used through Automation, and considering that the databases you're likely to use probably won't have a splash screen, I won't delve into the complexity of sending keystrokes to other applications.)

The **Open Query** and **Print Report** buttons implement the previous two examples. I chose to use the "Ten Most Expensive Products" query in the example, because this demonstrates a nice feature of Access — its implementation of SQL that makes generating statistics of this kind very simple. The **Export Data** button executes a pre-defined query, exports the data returned by that query in dBASE format, and finally opens and displays the table using Delphi's Table and DataSource components. The export is performed using Access' *TransferDatabase* method, and is called with various parameters describing the type of transfer to be carried out, the format of the export, the path of the destination table, the type of the source object, the name of the source object, and the destination filename:

```
Acc.DoCmd.TransferDatabase(acExport, 'dBase 5.0',
  'C:\Program Files\Borland\Delphi 2.0\Demos\Data\', acQuery,
  'Ten Most Expensive Products', 'qry.dbf');
```

All the methods used here, together with their respective parameters, are fully documented in Access' online Help, and need only slight modifications to get them to work via Automation from Delphi.

When closing the application, the instance of Access also needs to be closed. This is performed by calling the *Quit* method of the *Application* object.

Using Access, macros that perform some local function can be stored within the database and executed remotely using Automation. This is achieved by calling the *RunMacro* method of the *DoCmd* object, passing the name of the macro as a string parameter. For example:

```
Acc.DoCmd.RunMacro('Macro1');
```

This means that processes not available, nor appropriate to be performed via Automation, can still be used. They are implemented directly in Access and controlled from Delphi.

Data Access Objects

Technically speaking, Data Access Objects (DAO) is a COM (Component Object Model) interface to the JET database engine (Microsoft's counterpart to the Borland Database Engine). DAO to a Delphi programmer is, in some respects, similar to Access, insofar as it's an Automation server. However, it's typically used for a different purpose. In general, DAO is used for manipulating data, and Access is used for presenting it. DAO is more efficient than Access at getting data stored in an Access database (a .MDB file), because it comprises — and therefore loads into memory — only the functionality necessary for manipulating the data, not any visual functions such as reporting, graphical querying, table manipulation, etc.

DAO exposes the *dbEngine* object as its top-level object used in Automation as follows:

```
dbEngine := CreateOLEObject('DAO.dbEngine');
```

where *dbEngine* has been declared as a Variant. The *dbEngine* object can be used to control various functions, such as implementing transaction control, creating new databases, opening existing databases, setting access privileges, and returning details of error messages. In the following example, we are using the *OpenDatabase* method to open an existing database, the instance details of which are stored in the *dbs* variable (which has previously been declared as a Variant):

```
dbs := dbEngine.OpenDatabase(
         '... path and filename of database...');
```

Having established a reference to a particular database, we can now create what is known in the Microsoft world as a *recordset*. You can think of a recordset as a collection of records in a database, such as a table, part of a table, a query, or part of a query. This object is also stored in a Variant and can be used as follows:

```
rst := dbs.OpenRecordset('Ten Most Expensive Products');
```

Rather than a specific pre-defined object in a database, recordsets can take as a parameter a SQL query statement. For example:

```
rst := dbs.OpenRecordset('Select * From Employees',
                            dbOpenSnapshot);
```

As previously stated, DAO is useful for manipulating data, rather than displaying it. The previous examples demonstrated how queries can be executed, but there was no way for the results to be viewed. The functionality discussed

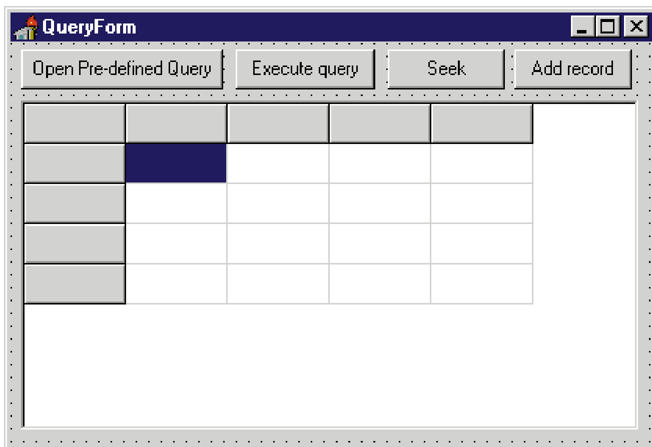


Figure 2: A demonstration form that uses DAO to access data.

```
procedure TQueryForm.FillStringGrid(Data: Variant);
var
  NumRows, NumColumns, l, M: Integer;
begin
  { Move to the last record in the recordset to ensure the
    RecordCount method references all the records. }
  Data.MoveLast;
  Data.MoveFirst;

  { Retrieve the number of fields and records, then
    set the size of the string grid. }
  NumRows := Data.RecordCount;
  NumColumns := Data.Fields.Count;

  StringGrid1.RowCount := NumRows + 1;
  StringGrid1.ColCount := NumColumns;

  { Add Captions to first row of string grid. }
  for l := 0 to NumColumns - 1 do
    StringGrid1.Cells[l, 0] := Data.Fields[l].Name;

  { Cycle through each cell in the recordset placing its
    value in the appropriate place in the StringGrid. }
  for l := 0 to NumRows - 1 do begin
    for M := 0 to NumColumns - 1 do
      if Data.Fields[M].Value <> Null then
        StringGrid1.Cells[M, l + 1] := Data.Fields[M].Value
      else
        StringGrid1.Cells[M, l + 1] := '';
    { Move to the next record in the recordset. }
    Data.MoveNext;
  end;  { for }
end;
```

Figure 3: A function to display the contents of a dataset in a string grid.

here has been implemented in the example shown in Figure 2. It includes the two queries already mentioned, plus a function that returns the results and displays them in a string grid (see Figure 3), a facility that performs an indexed search on a particular table, and an example of inserting a new record with some sample data into a database table. A more complex example using DAO would involve having multiple databases and multiple recordsets within those databases open at once, possibly stored in a Variant array.

If you've used Delphi's standard facilities for accessing tables and queries, using DAO will be reasonably familiar to you. To add a record to a database, for example, you would call the *Add* method of the recordset, set the values of each field, and post the changes using the *Update* method. Similarly, to search for a particular value in a database, you would set the active index and call the *Seek* method. The steps involved are identical to those you would use if you were using a database native to Delphi, but the syntax is different. After these differences are overcome, DAO is an ideal way to get to existing data stored in an Access database.

Licensing

It's appropriate to mention the licensing implications of using third-party products as Automation servers. In general, you have no rights to distribute any part of the Automation server with your application unless permission is obtained from the author of the server software (which will typically involve a licensing fee). However, if you used Visual Basic or Visual C++ (or another similar Microsoft development tool) as the development language, you have a royalty-free license to distribute the Microsoft JET database engine and its associated DAO with your application.

Unfortunately, this doesn't extend to developers using non-Microsoft tools, such as Delphi. If you know your clients have the rights to use a product that can be used as an Automation server, you are permitted to make full use of its capabilities, provided the number of system users doesn't exceed the number of licenses of the Automation server. Different license agreements have different restrictions, so it's always wise to check with the author of the Automation server before implementing any system based on it.

Conclusion

Access (and its associated file format) is an extremely popular database management system, largely because it's part of the Microsoft Office Professional suite. Using Automation is only one way to get at data in its databases; various third-party add-ons are available for Delphi that offer native access to the data without the overhead of using an Automation server. However, Automation is the ideal solution if you're interested in manipulating systems that exist in Access that include more than a database.

Referring generally to office suites, you can choose the component of the suite that best suits a particular sub-task and call (what are likely to be) pre-defined functions to produce an extremely rapid solution. For example, Excel provides many complex functions that would be pointless and difficult to recreate in Delphi. Using Automation, they are only a function call away, yet, at all times, control is maintained by your application. An ideal application of Automation is if your client wants to maintain control over the content of, for example, printed reports, to make changes at will, yet still be able to access them via a program pre-written in Delphi (this was described in detail in Part I of this series). This removes the need to rebuild reports using a specialized reporting tool for what could only be a minor amendment. Furthermore, if implementing a new system, the creation of the reports (or conversion of existing ones) could be carried

out by the system users, while development of the code to access them could proceed in parallel — only a small interface routine would be required to bridge the gap.

Although I have referred specifically to Microsoft Office and its components in this series, the principles discussed can be applied to any Automation server. Even if you aren't interested in implementing such a system in a practical situation, you should at least have an idea of how complex Automation servers are designed, should you ever decide to create your own in Delphi. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUL\DI9707ID.

Ian Davies is a developer of 16- and 32-bit applications for the Inland Revenue in the UK. He began Windows programming using Visual Basic about four years ago, but has seen the light and is now a devout Delphi addict. Current interests include Internet and intranet development, inter-application communication, and sometimes a combination of the two. Ian can be contacted via e-mail at 106003.3317@compuserve.com.

By Bill Todd

InterBase Indexes

Inside InterBase: Part II

In Part I of this series (presented in last month's issue), we explored creating and using InterBase triggers and generators. In this installment, we continue looking inside InterBase, focusing on the use of its indexes.

Why Indexes?

The reasons to create indexes in an InterBase database are the same as for any other: Indexes provide a fast way to find specific rows in a table, and the means to enforce uniqueness. Because any index is a sorted list — unlike the table data itself — the database management system (DBMS) can find any value quickly. The ability to enforce uniqueness stems directly from the ability to find a value quickly; each time you add a row to the table, the DBMS must first search for the unique value to see if it already exists.

You can create an index on a single column, or on multiple columns in a table. InterBase allows a maximum of 16 columns in an index. By maintaining a sorted list of the values in the column(s) being indexed, an index lets you find rows fast. Therefore, you can use a multi-column index to search for a value or values in the first *n* columns.

For example, assume you have an index on an Items table that includes the CustomerNumber, OrderNumber, and PartNumber columns. A DBMS can use this index to find rows by:

- customer number,
- customer number and part number, or
- customer number, part number, and order number.

However, the index is useless if you need to find a row by part number alone; the index is sorted first by customer number. The only way to find a part number in the index is to scan it sequentially — you might as well scan the table.

Creating Indexes

Creating indexes has advantages and disadvantages. For the best performance from InterBase, you must understand the *type* of indexes to create and *when* to create them. By indexing all the columns you may use to select records, you'll garner the best performance locating rows. However, indexes hurt performance when you are inserting, modifying, or deleting rows. This is because the indexes must be updated each time you insert or delete a row, or modify an indexed column in a row.

Indexes also consume disk space. However, this is usually a minor consideration because disk space is relatively inexpensive compared to users' lost time with a slow application.

Therefore, you should create an index under these three conditions:

1) The column(s) are used frequently in the WHERE clause of a query.
2) The column(s) are used frequently to join tables in a query.
3) The column(s) are used frequently in the ORDER BY clause of a query.

With InterBase, including the same column in multiple indexes is a bad idea. For example, if you have three columns to index, you'll need six indexes to include all the different combinations of column order. However, InterBase can use multiple, single-column

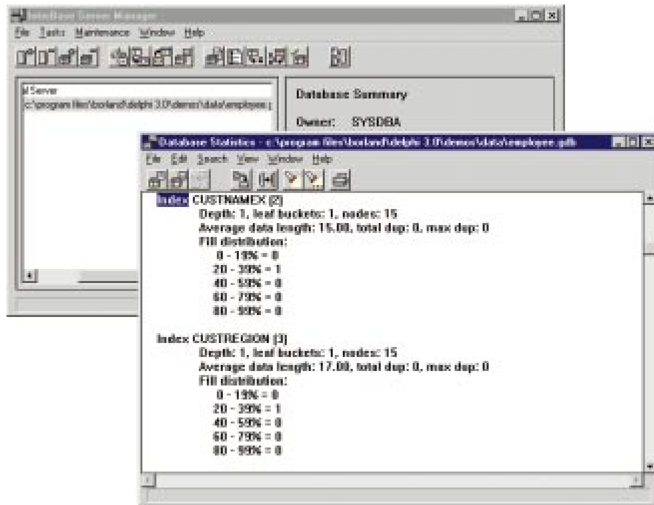


Figure 1: The InterBase Server Manager's Database Statistics window.

indexes when a query includes selection criteria for both columns. So you'll get better overall performance creating a single-column index on each of the three columns, than by creating six, three-column indexes.

However, this isn't true if you use the columns in only one order. For example, if you always select rows from the Items table by specifying a customer number, an order number, and a part number, you'll get the best performance by creating a single index on all three columns. The same is true if you always order a query's results by the same columns. If the ORDER BY clause exactly matches the fields and order of an existing index, InterBase will use the index to order the result set, instead of retrieving the requested rows and sorting them.

Additionally, you'll get better performance using single-column indexes if a query's WHERE clause includes the OR operator to connect selection criteria on two columns. This is because InterBase indexes are used for moving through records in order, and as bitmaps. In an OR operation, InterBase will use single-column indexes to evaluate the selection conditions, then combine them to retrieve the required rows. (Yes, bitmapped indexes existed long before FoxPro started using them.)

The database page size also plays a role in index performance. A larger page size means each index page holds more entries. That, in turn, means fewer pages must be read from disk to search the index. The page size also affects the depth of the index tree. If an index is more than four levels deep, consider increasing the page size. If the index depth on data that changes frequently is less than three levels, consider decreasing the page size. However, don't decrease the page size to the point that one record will not fit on a page.

To determine the number of levels in the index, use the InterBase Server Manager. Select **Tasks | Database Statistics** from the menu to open the Database Statistics window. In the Database Statistics window, select **View | Database Analysis** to display the statistics for all the tables and indexes in the database. To find the information for a particular index, select **Search** from the main menu, and search for the index name. Figure 1 shows the statistics for the CUSTNAMEX and CUSTREGION indexes. The table in Figure 2 explains these statistics.

InterBase automatically creates an index whenever you define a primary key, foreign key, or unique constraint. You can create additional indexes using the SQL CREATE INDEX statement. Its syntax is:

```
CREATE [UNIQUE] [ASCENDING | DESCENDING] INDEX IndexName
  ON TableName (column1, column2, ...)
```

For example, this statement creates a unique index on the Items table:

```
CREATE UNIQUE INDEX ITEMX
  ON ITEMS (CustomerNumber, OrderNumber, PartNumber)
```

You can abbreviate ASCENDING as ASC and DESCENDING as DESC. The index's name must be unique within the database. You can create multiple indexes on the same column. So if you need to order records both ways, you may want to create an ascending and a descending index on a date column. You cannot create a unique index if the column already contains duplicate values.

Modifying Indexes

To permanently remove an index, use the SQL DROP INDEX command. For example, this command permanently removes the index named ITEMX from the database:

```
DROP INDEX ITEMX
```

You can also temporarily deactivate an index using the ALTER INDEX command. For example, these commands deactivate the ITEMX index, then reactivate it:

```
ALTER INDEX ITEMX INACTIVE
ALTER INDEX ITEMX ACTIVE
```

Setting the index to ACTIVE rebuilds the index; on a large table, this may take some time. One case where you'll want to deactivate indexes is when importing a large number of

Item	Definition
Depth	Number of levels in the index tree.
leaf buckets	Number of leaf pages in the index.
nodes	Total number of pages in the index.
Average data length	Average length of each index entry in bytes.
total dup	Total number of rows in the index with duplicate values.
max dup	Number of rows for the value that has the most duplicate entries in the index.
Fill distribution	Histogram showing the number of pages in each of the percent fill ranges.

Figure 2: Explaining the numbers in the Database Statistics window shown in Figure 1.

records into a table. Deactivating the indexes on the table, importing the data, then rebuilding the indexes by activating them is faster than incurring the overhead of updating the indexes for each new row as it's imported.

Although indexes normally require no maintenance on your part, some attention is occasionally necessary to ensure optimum performance. InterBase indexes use a balanced B-tree structure; over time, they can become unbalanced if many records are added and deleted. Deactivating and reactivating the indexes will rebuild them, and correct this problem.

Selectivity

For each index, InterBase also generates a statistic referred to as *selectivity*. Selectivity indicates the number of unique values in the index, in relation to the number of rows in the table. The query optimizer uses this number to determine if an index should be used to process a particular query, or if scanning the table will be faster. Adding and/or deleting large numbers of records could change the index's selectivity.

The selectivity of the index is calculated when the index is built. It is not updated as rows are added and deleted from the table. Although selectivity is re-computed any time the index is rebuilt, you don't have to rebuild the index to update the selectivity. You can update the selectivity using the SET STATISTICS command:

```
SET STATISTICS ITEMX
```

None of these steps are necessary if you periodically back up and restore your database. A backup and restore rebuilds all indexes, and thus recalculates their selectivity. Additionally, a backup and restore performs a sweep to remove outdated record versions, and re-packs the database so the pages are uniformly filled.

When you use a SQL SELECT statement to retrieve records from an InterBase database, the optimizer examines your query and formulates an execution plan. To determine which indexes to use, and how to use them to provide the best performance, the optimizer looks at:

- the size of each table involved in the query,
- the available indexes,
- the selectivity of the indexes,
- the contents of the WHERE clause, and
- the contents of the ORDER BY clause.

A Definite Plan

Although the InterBase optimizer is very good, it's always possible it won't use an index when you think it should. If you experience poor performance from a query, you should examine whether the query plan created by the optimizer appears to make optimal use of available indexes.

To see the optimizer's plan for executing a query, start ISQL and select **Session | Basic Settings** to display the Basic ISQL Set Options dialog box (see Figure 3). Check the **Display Query Plan** box; ISQL will display the query plan for each query you execute. Figure 4 shows the result of the following query:

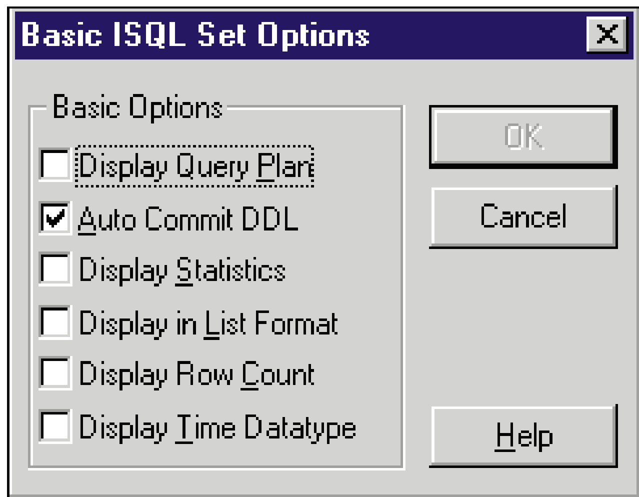


Figure 3: The Basic ISQL Set Options dialog box.

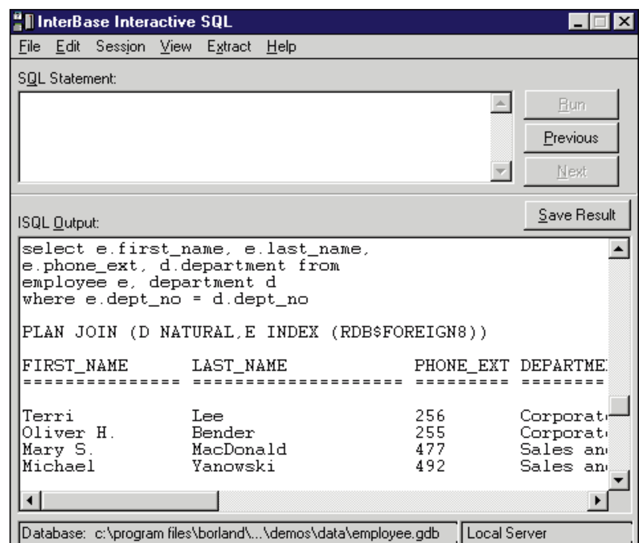


Figure 4: Displaying the query's plan.

```
SELECT E.FIRST_NAME, E.LAST_NAME, E.PHONE_EXT, D.DEPARTMENT
  FROM EMPLOYEE E, DEPARTMENT D
 WHERE E.DEPT_NO = D.DEPT_NO
```

The results window contains the query, plan, and query result, respectively. In this example, the plan is:

```
PLAN JOIN (D NATURAL, E INDEX (RDB$FOREIGN8))
```

This shows the optimizer will perform the natural join between the Employee and Department tables using the index, RDB$FOREIGN8. The index has a system-generated name because it was created automatically when the foreign key on the DEPT_NO field in the Employee table was defined.

To override the plan created by the optimizer, use the PLAN clause of the SQL statement (see Figure 5). An InterBase extension to standard SQL, the PLAN clause lets you specify your own plan instead of letting the optimizer create one. You should use this capability sparingly. First, it's unlikely you'll

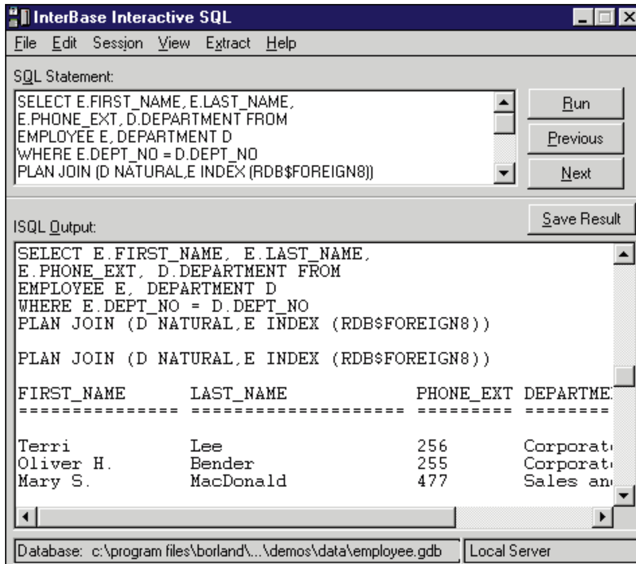


Figure 5: Specifying a plan.

find a case where you can create a better plan than the optimizer. Second, one of the great benefits of dynamic query optimization is that the query plan may change over time, as the characteristics of the tables change. For example, if an index's selectivity drops dramatically, the optimizer will recognize this, and stop using the index. If you add an index to a table, the optimizer will detect the new index automatically, and use it when appropriate. On the other hand, if you specify a plan in your SQL statement, the plan will never change — no matter what happens to the database — unless you change it.

Conclusion

You will get the best possible performance from your InterBase application by carefully creating indexes on fields, or combinations of fields, that you use to select and/or order rows. InterBase is more flexible in its use of indexes than most databases, in that it can effectively use multiple single-column indexes to select rows, based on values in multiple columns. Therefore, creating single-column indexes is generally better than creating multi-column indexes, unless you're sure you'll always select and/or order by the same columns in the same order.

Next month, we'll look at managing security for your InterBase databases. ∆

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: A Developer's Guide* [M&T Books, 1995], *Delphi 2: A Developer's Guide* [M&T Books, 1996], and *Creating Paradox for Windows Applications* [New Riders Publishing, 1994]; and a member of Team Borland providing technical support on CompuServe. He is also a nationally known-trainer, and has been a speaker at every Borland Developers Conference and the Borland Conference in London. 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 IV

Validity Checks and Referential Integrity

Developers use the Paradox file format every day, yet the Delphi documentation offers little information about it. To help fill that gap, the first two articles in this series explored the internals of Paradox .DB files: table structure, BDE record and block management, field types, and record size calculation. The third article examined primary and secondary indexes. In this, the fourth article of the series, we'll examine validity checks and referential integrity.

What Is a Validity Check?

A validity check, abbreviated as *ValCheck*, is a data formatting requirement that you define for a field, and that's enforced for you by the BDE whenever you add or modify the value in that field.

ValChecks are defined and modified from the Restructure dialog box of the Database Desktop. When this dialog box first opens, the panel on the right side allows you to set ValChecks for each field (see Figure 1). ValChecks are saved in the *TableName*.VAL file, a member of the table's family of files.

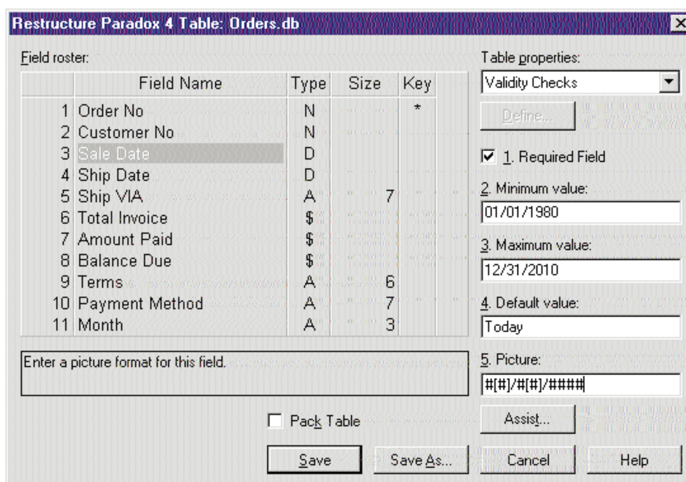


Figure 1: The Database Desktop's Restructure dialog box.

This file has an undocumented binary format, so you can't modify it directly.

The Paradox file format supports the following types of ValChecks:

- required value (the field cannot be left blank);
- minimum value in a field;
- maximum value in a field;
- default value in a field when a record is inserted; and
- a format mask that can also control allowable values in a field.

Figure 2 lists each Paradox field type, and the ValChecks that are valid for that type.

ValChecks have been available in the Paradox file format since its earliest versions. A former limit of 64 ValChecks per table was removed with the advent of Level 7 tables. (For more information on Paradox table "levels," see the "The Paradox Files: Part I," in the April 1997 *Delphi Informant*.)

ValChecks provide a number of benefits, because they're defined and enforced at the database level, rather than within your application. This allows you to maintain them in one place, and make them available

Field Type	ValChecks
Alpha	All valid
Number	All valid
Money (Currency)	All valid
Short	All valid
Long Integer	All valid
BCD	All valid
Date	All valid
Time	All valid
Timestamp	All valid
Autoincrement	Minimum only
Logical	Required and Default only
Memo	Required only
Formatted Memo	Required only
Graphic	Required only
OLE	Required only
Binary	Required only
Byte	Required only

Figure 2: Field types and valid ValChecks.

to every application that uses the same set of tables. As you'll see, however, Delphi provides only limited support for ValChecks, reducing their benefit in your applications.

The Minimum and Maximum ValChecks

The purpose of ValChecks is self-evident. You can define one to set a lower or upper boundary, or define them both to set a range. The boundary value itself is a valid entry (i.e. with a Minimum ValCheck defined, allowable values are equal to or greater than the defined value.) Obviously, the Maximum ValCheck must be larger than the Minimum.

When you use Delphi's field editor to add static field objects to your form, Delphi doesn't pick the underlying ValChecks for those field types that have *MinValue* and *MaxValue* properties. This is an unfortunate omission on Borland's part.

When you specify these ValChecks with an Alpha field, the BDE gets the sort order from the table language. (If you don't specify a custom table language, each table is defined with a default language driver that you specify in the BDE Configuration program.) When you define a Minimum ValCheck for an Autoincrement field, the BDE uses this value to seed the counter. New records increment from the specified value.

Delphi and the BDE do not test these ValChecks until you post the new or updated record. If one of them is violated, you'll receive the following error:

```
EDBEngineError
Minimum (or Maximum) validity check failed.
Field: <name>.
```

The minimum or maximum value isn't shown, and there's no easy way to obtain it. If you use the *MinValue* and *MaxValue* properties instead, you'll receive the following error when the value limits aren't met:

```
EDatabaseError
X is not a valid value for field <name>.
The allowed range is <min> to <max>.
```

As you can see, the second error is more informative for the user.

The Default ValCheck

Use this option to specify a starting value for the field when a new record is inserted. You can then overwrite the default value with any other valid entry. The Default ValCheck is fully supported by Delphi. Delphi 3 includes a *DefaultValue* property for many of the *TField*-derived objects, which can be used in place of the ValCheck.

Special keywords. For Date fields, you can enter the keyword `Today` in the Default ValCheck to place the current system date into the field. You can also enter this keyword in the Minimum or Maximum ValChecks, to set the current date as the lower or upper limit. (This option has been supported since the file format's earliest versions.)

For Time and Timestamp fields, you can enter the keyword `Now` in the Default ValCheck to place the current system time or date/time in the field. You can also enter this keyword into the Minimum or Maximum ValChecks, to set the current time or date/time as the lower or upper limit. (This option was added in Level 7.)

The Required ValCheck

This ValCheck specifies that a value is required in the field before the record can be posted. If you post without a value, you'll receive the following error message:

```
EDatabaseError
Field <name> must have a value.
```

Delphi's static field objects include a *Required* property. When you instantiate one of these objects, and if the DataSet connection is active, Delphi will set the *Required* property to *True*, if there is a Required ValCheck. You can also set this property manually without the underlying ValCheck being set, but you must then write code in the *OnValidate* event, to enforce the property.

The Picture ValCheck

Pictures are format strings that control which values can be entered in a field, and how those values should be displayed. Figure 3 shows the characters in Paradox's picture "language," while Figure 4 shows some sample pictures.

Character	Meaning
@	Any character
#	Any number (0-9)
?	Any letter (uppercase or lowercase)
&	Any letter (converted to uppercase)
~	Any letter (converted to lowercase)
!	Any character (letters converted to uppercase)
[ ]	Bracket-optional entries
{ }	Group-required entries
*<num>	Specify a number of repetitions of a group or character
*	Specify any number of repetitions
;	Literal escape character
Other chars	Treated as literals

Figure 3: The Paradox picture language.

Picture	Meaning
###-##-####	US Social Security Number
*3#-*2#-*4#	US Social Security Number
#&#-&#&	Canadian postal code
red,green,blue,yellow	One of four listed colors.
red,b{rown,l{ack,ue}}	One of four listed colors; grouping ensures the first matching entry isn't filled.
!*@	Anything, but capitalize the first character if it's a letter.
##/##/####	Date with a four-digit year.
!*{ !,@}	Capitalize the first letter in each word ("proper case").

Figure 4: Picture examples.

Delphi provides no support for Paradox's Picture ValCheck on data entry or posting. Instead, Delphi provides properties with some of the same capabilities, specific to each field type:

- *TFloatField*, *TCurrencyField*, *TSmallIntField*, *TIntegerField*, *TBCDField*, and *TAutoIncrementField* objects provide the *DisplayFormat* and *EditFormat* properties. (*EditFormat* is ignored for Autoincrement fields.)
- *TDateField*, *TTimeField*, and *TTimeStampField* objects provide the *DisplayFormat* and *EditMask* properties.
- *TStringField* objects use only the *EditMask* property, because the displayed value is the same as the entered value.
- *TBooleanField* objects use the *DisplayValues* property to control how True and False are represented in data-bound controls.

Referential Integrity

When one or more fields in a table refer to the primary key of another table, those fields are said to be a *foreign key*, i.e. a key "imported" from another table. Foreign keys are necessary to enforce data consistency among the various tables in the database. The Referential Integrity (RI) rule states that when two sets of fields in two tables are in a primary key/foreign key relationship, every foreign key value in one table must match an existing primary key value in the other table.

For example, a typical Stock table might contain fields for VendorID, StockType, and PackagingType. Each of these fields is likely to be a foreign key to a supporting table containing the valid vendors, stock types, and packaging types. RI ensures that only entries in each of these supporting tables can be used to populate the appropriate fields in the Stock table.

Certain types of operations on a database might cause existing foreign-key data to become invalid. For example, if a vendor goes out of business, what happens to the stock items supplied by that vendor? And if your company decides to exit from a specific line of business, what happens to all the items with that stock type?

For update and delete operations, the RI rule defines three possible outcomes:

1) A change should be cascaded through the database. For example, if you change a vendor ID or delete a vendor, you might choose to cascade this change through the Stock table.
2) A change should be prohibited if records depend on the value being updated or deleted. For example, you might stop the user from deleting a vendor if products sold by that vendor are still within their warranty period.
3) A change should cause linked values to be set to null or blank. For example, if you decide to no longer support a particular type of packaging, all stock items that use the packaging method might have this field set to null.

The Paradox file format doesn't support all these outcomes. In fact, only the following limited options are available:

- For update operations, you can elect to cascade or prohibit the change. Paradox does not support changing to null for update operations.
- For delete operations, only prohibit is supported. Paradox tables do not support "cascaded delete" or "change to null." This means that you must manually delete linked rows from the foreign-key table before you can delete the matching row from the primary-key table. (This operation will succeed, because there will then be no dependent rows.)

This limited support for RI is one of the more serious limitations in the Paradox file format. It's unfortunate that Borland doesn't offer more.

Defining RI

RI is defined from the foreign-key table back to the primary-key table. For example, an RI relationship between the Stock and Vendor tables is defined from the Stock table, referring back to the primary key of the Vendor table.

To set up RI, use the Restructure dialog box in the Database Desktop. To display the appropriate panel, click on the **Table properties** drop-down list in the top right corner. To define an RI rule, click the **Define** button to see the dialog box shown in Figure 5. On the left side, select the field or fields that define the foreign key. On the right side, select the table whose primary key matches this foreign key. You can also define how the RI relationship should behave for updates. When you close the dialog box, you'll be prompted to enter a name for the RI relationship; this name follows the same rules as the field and index names described in earlier articles.

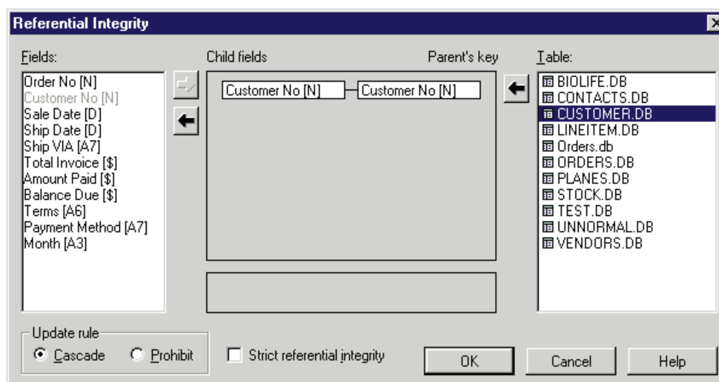


Figure 5: The Database Desktop's Referential Integrity dialog box.

Strict RI. In Figure 5, notice the Strict referential integrity check box. This setting is a holdover from the days when Paradox for DOS and Paradox for Windows applications coexisted on networks. Paradox for DOS didn't support the RI features added to the Paradox file format with Level 4. When this box was checked, the table could be accessed only by Paradox for Windows, to protect against the possibility that a Paradox for DOS application would subvert RI. This setting is ignored by the BDE.

How Does RI Work?

RI information is saved in the .VAL files for both tables. The BDE first places a maintained secondary index on the field or fields comprising the foreign key. This index allows the BDE to quickly sort and filter the table by these fields, and to manage the foreign-key side of the relationship. It also allows Delphi to create a one-to-many relationship within the primary-key table as the master, and the foreign-key table as the linked detail. When you're prompted to specify the name of the RI relationship, you're actually specifying the name of this secondary index.

For the foreign-key table, the BDE stores the name of the field or fields comprising the foreign key, the name of the table whose primary key forms the other end of the relationship, and the name of the index used to manage the link. It also stores the prohibit or cascade setting for the update rule.

For the primary key table, the BDE stores the name of the dependent foreign-key table, and the name of the index used to manage the link. This information allows the BDE, Database Explorer, and Database Desktop to list the tables that depend on this table. A data-modeling tool could use this information to trace the RI relationships in either direction.

RI and Delphi

Because its database operations are performed by the BDE, Delphi fully supports Paradox's implementation of RI. If you try to insert a value into a field that has an RI relationship to another table, and the inserted value isn't in the other table, you'll receive the following error message:

```
EDBEngineError
Master record missing.
```

Unfortunately, Delphi doesn't tell you which field contains the errant value, nor does it move you to that field once you've cleared the error dialog box.

If you try to change the primary-key value in the other table, or delete the record with that value — and if there are records using that value — you'll receive the following error message:

```
EDBEngineError
Master has detail records. Cannot delete or modify.
```

Again, Delphi doesn't tell you the name of the table where the detail records reside. (There could be more than one.)

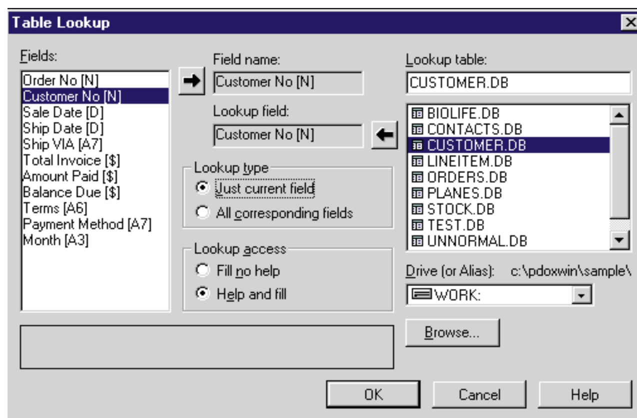


Figure 6: The Database Desktop's Table Lookup dialog box.

No additional information is available from the *EDBEngineError* object, or from the *TDBError* objects in the BDE error stack.

The Table Lookup "ValCheck"

In its earliest days, the Paradox file format didn't support RI at all. Instead, Paradox provided a Table Lookup feature that worked in a similar fashion. A Table Lookup can be established on any field in a table, linking that field to the single-field primary key of another table. (There's no support for composite primary keys.) A value entered in that field must exist in the lookup table.

Table Lookup information is also stored in the .VAL file. To configure a Table Lookup, use the Restructure dialog box in the Database Desktop. Click the Table properties drop-down list in the top-right corner to display the appropriate panel. Click the Define button to see the dialog box shown in Figure 6. On the left side, specify the single field on which to perform a lookup. On the right side, select the lookup table, whose primary key must match the field already selected. Two additional options can be set:

- A Lookup access parameter is designed for table view in the Database Desktop and Paradox itself, and for native Paradox forms. When set to Help and fill, this option provided a Ctrl Spacebar hot key to display the lookup table in a dialog box, so that a value could be selected. When set to Fill no help, this option only validated the entry, and didn't provide the hot key or pop-up dialog box.
- A Lookup type parameter controls which fields are filled when a lookup is performed. When set to Just current field, only the lookup field itself is filled. When set to All corresponding fields, the lookup field and any additional fields with the same name and type are filled.

Table Lookup and Delphi

If you have a Table Lookup defined, Delphi will validate the lookup value on record-post (not on field-depart, as the Database Desktop and Paradox do.) If the value isn't in the lookup table, you'll receive the following error message:

```
EDBEngineError
Field value out of lookup table range.
```

Delphi doesn't tell you which field has the bad value. Also, Delphi doesn't provide native support for the Lookup access

and Lookup type options described previously. There is no automated way to display the lookup table, nor to fill in additional values based on matching field names.

In general, don't bother setting up table lookups unless you plan to manipulate data with the Database Desktop or Paradox itself. The RI features described previously provide the same validation as an entered value in a lookup table. To display an actual lookup list, use a *DBLookupComboBox* component instead of a *TDBEdit* component. The *ListSource* and *ListField* (Delphi 3) or *LookupSource* and *LookupField* (Delphi 1 and 2) properties allow you to bind this component to a lookup table. You even have explicit control over the fields displayed in the drop-down list, as well as the size of the list.

Should You Use ValChecks and RI with Delphi?

Although the Picture ValCheck exists in the Paradox file format, Delphi ignores it, and newer, better-integrated features have replaced its functionality. The situation for Table Lookup is similar; Delphi and the BDE support it, but RI and built-in components have replaced it.

While Delphi and the BDE support the Minimum and Maximum ValChecks, you can also set minimum and maximum properties at the *TField* level. When you do this, Delphi provides more-informative error messages. For this reason, many developers don't use these ValChecks, either.

Delphi fully supports the Default and Required ValChecks. It provides analogs in the *DefaultExpression* and *Required* properties for *TField* objects, so you can set these properties in either place. However, as I noted previously, if you set the *TField.Required* property to *True*, and don't set the underlying Required ValCheck, you must write your own code in the *OnValidate* event to trap for a blank field.

RI is an essential part of any database definition. It links normalized tables to ensure that the data in each table remains consistent and fully synchronized. It's unfortunate that the Paradox file format doesn't support all RI options, but the available support shouldn't be ignored.

And in the Future?

Borland is adding advanced database features to Delphi and the BDE. For example, Delphi 3 supports the following new properties on *TField*-descendent objects:

- *CustomConstraint*, to specify and test application-specific constraints imposed on a field's value, using SQL-based search expressions. For example: Value >= 0 AND Value <= 100.
- *ConstraintErrorMessage*, to specify a custom error message when the constraint is violated.
- *ImportedConstraint*, a read-only property that holds server-based constraints.

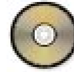
Borland also intends to enhance a BDE-based, centralized database catalog that can draw from heterogeneous data sources. This catalog will support complex constraints programmed in SQL and host languages. These constraints will serve as business rules that function across different data sources, and for any application using that catalog. With this feature, Borland implements the theory behind Paradox's ValChecks in a far more powerful and flexible way that's independent of file formats.

Next Time

The next article in this series will examine the Paradox file format's encryption and security mechanisms, and how the BDE manages passwords for Paradox tables. It will also discuss table-language options that allow you to define tables with different character sets and sort orders. ∆

Dan Ehrmann is the founder and President of Kallista, Inc., a database and Internet consulting firm based in Chicago. He is 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 10 years. Dan can be reached via e-mail at dan@kallista.com.

By *Cary Jensen, Ph.D.*



Cached Updates: Part III

The *OnUpdateRecord* and *OnUpdateError* Event Handlers

In the past two issues, this column has explained the use of cached updates in Delphi 2 and 3. This third and final installment of the series will look at event handlers related to cached updates: *OnUpdateRecord* and *OnUpdateError*.

A Quick Recap

Update caching is a mechanism by which all changes made to a DataSet are stored locally, then applied simultaneously using the *ApplyUpdates* method of a DataSet or Database component. Cached updates are enabled by setting a DataSet's *CachedUpdates* property to *True*. The cached edits are applied only specifically to the corresponding DataSet; closing a table, or setting *CachedUpdates* to *False* without actually applying the updates, cancels all cached edits.

When updates are being cached, you can offer editing options that would otherwise be unavailable. For example, you can permit the user to view all edited records in the cache, including those that have been deleted. Also, you can display both the original and new field values for records that have been modified. Finally, you can permit the user to revert any cached edit, restoring the record to its original state.

Last month, we looked at the UpdateSQL component. This component can be used with a DataSet to allow a user to edit read-only DataSets, such as SELECT queries that make use of the DISTINCT keyword. The UpdateSQL component permits you to define up to three SQL queries — one each for delete, insert, and modify updates. These SQL statements can be quickly and easily constructed using the UpdateSQL Editor, a component editor for UpdateSQL components.

You might remember that there are four primary advantages to using cached updates:

- a decrease in network traffic,
- an increase in performance,
- more user interface options, and
- greater programmatic control over the posting of individual records.

This month's DBNavigator will concentrate on the final advantage, programmatic control over applying updates.

Using Cache-Related Event Handlers

For those instances where you want complete control over the application of cached updates, the DataSet class provides two event properties: *OnUpdateRecord* and *OnUpdateError*. From *OnUpdateRecord*, you replace Delphi's attempt to update the records with your own; in other words, you supply completely customized update behavior. *OnUpdateError* is triggered if an exception is raised during the attempt to update a record, whether the exception originated from Delphi's own update process, or from *OnUpdateRecord*.

Using *OnUpdateRecord*. As mentioned, if you need to define custom update behavior, you assign an event handler to the *OnUpdateRecord* event property. However, although this series' preceding two articles demonstrated how to turn on cached updates with an open table, the technique can't be used with *OnUpdateRecord*. Specifically, if

Value	Description
ukModify	Current record has been modified.
ukInsert	Current record is newly inserted.
ukDelete	Current record is scheduled for deletion.

Figure 1: The values of *UpdateKind* and their meanings.

Value	Description
uaFail	Record couldn't be updated; raise an exception and abort updating. (This is the default value.)
uaAbort	Same as *uaFail*, except that a silent exception is raised, meaning that no error message is displayed to the user.
uaSkip	Current record was not updated. Continue applying updates.
uaRetry	Not meaningful in an *OnUpdateRecord* event. Used for *OnUpdateError* to make another attempt at updating a record whose previous attempt generated an exception.
uaApplied	The event handler successfully updated the record.

Figure 2: The values of *UpdateAction* and their meanings.

Cached Updates with OnUpdateRecord and OnUpdateError Events

Start Caching | Empty Cache | View Types: All, Inserted, Deleted, Modified

CustNo	Company	Addr1	Addr2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 1
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 3
1384	VIP Divers Club	32 Main St.	
1510	Ocean Paradise	PO Box 8745	
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	
1551	Marmot Divers Club	872 Queen St.	

Figure 3: The CACHE5 project demonstrates the use of *OnUpdateRecord* and *OnUpdateError*.

you open a table, then set its *CachedUpdates* property to *True*, code associated with *OnUpdateRecord* won't be executed. However, if you set *CachedUpdates* to *True* at design time, or before opening a table (either by setting its *Active* property to *True*, or by calling the table's *Open* method), the table's *OnUpdateRecord* event handler will execute properly. This behavior is not a bug, but rather an artifact of how *OnUpdateRecord* is implemented.

In the following example, a Query component will be cached, rather than a Table component. With a Query component, *OnUpdateRecord* is triggered whether it is set before or after the activation of the Query. Consequently, the following example can be similar to the preceding examples.

Here's the declaration for the *OnUpdateRecord* event handler:

```
procedure(DataSet: TDataSet; UpdateKind: TUpdateKind;
  var UpdateAction: TUpdateAction)
```

The first parameter, *DataSet*, identifies the DataSet component to which updates are being applied. More specifically, the updates are being applied to the current record of this DataSet. Because *DataSet* controls which record is current when *OnUpdateRecord* is executing, it's extremely important not to perform any record navigation during the execution of *OnUpdateRecord*. (This same caution applies to *OnUpdateError*.)

The second parameter, *UpdateKind*, identifies the type of update that needs to be applied. This parameter has three possible values, as shown in Figure 1.

The third parameter, *UpdateAction*, informs the Query component of what you've done with the current record. There are five valid values for this property, as shown in Figure 2. The default value of *UpdateAction* is *uaFail*. This value causes the entire update to fail. Therefore, if you are successful in updating the current record, it's imperative that you assign a value of *uaApplied*.

The use of the *OnUpdateRecord* event handler is demonstrated in the CACHE5.DPR project, shown in Figure 3. The DataSet being cached in this example is a Query. In addition, a Table component appears on this form. This table points to the same table as the Query, and is used to perform the updates for individual records when the cached update is applied. Also, the Query's *RequestLive* parameter doesn't need to be set to *True*, because the updates will occur via the *OnUpdateRecord* event handler, which edits the Table, not the Query. In fact, leaving the Query's *RequestLive* property set to *False* ensures that updates are performed only within a cached-update mode.

Figure 4 shows the *OnUpdateRecord* event handler assigned to the Query. It includes a **case** statement to test the value of *UpdateKind*. If the current record is being inserted, and a new record is added to the table, then the fields of the Query are assigned to the fields of the table, and the record is posted. If the record has been modified, then the existing record is located in the table, and its fields are updated. If the record has been deleted, again the corresponding record is located in the table and deleted. If the actions of this event handler don't raise an exception, the final statement sets the *UpdateAction* parameter to *uaApplied*. If an exception is raised, the *UpdateAction*'s default value of *uaFail* remains unchanged, causing the update to fail.

Using *OnUpdateError*. If you create an *OnUpdateError* event handler, it will be executed if the attempt to update a particular record fails. This is true whether the update is being performed by the default DataSet behavior, an UpdateSQL object, or code attached to *OnUpdateRecord*. Unlike *OnUpdateRecord*, which doesn't work properly with Table components under all conditions, *OnUpdateError* works properly for any DataSet.

The following is the declaration for the *OnUpdateError* event handler:

```
procedure(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
```

```
procedure TForm1.Query1UpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
var
  i: Integer;
begin
  // Post pending cached edits.
  if DataSet.State in [dsEdit,dsInsert] then
    DataSet.Post;
  // Update the record.
  if UpdateKind = ukInsert then
    begin
      Table1.Insert;
      for i := 0 to Table1.FieldDefs.Count - 1 do
        if DataSet.Fields[i].Value <> Null then
          Table1.Fields[i].Value := DataSet.Fields[i].Value;
        try
          Table1.Post;
          UpdateAction := uaApplied;
        except
          Table1.Cancel;
          raise;
        end;
    end
  else
    // Not an insert. Locate the existing record.
    if Table1.Locate('CustNo',
                     DataSet.Fields[0].OldValue,[]) then
      case UpdateKind of
        ukModify:
          begin
            Table1.Edit;
            for i := 0 to Table1.FieldDefs.Count - 1 do
              if Table1.Fields[i].Value <>
                 DataSet.Fields[i].Value then
                Table1.Fields[i].Value :=
                  DataSet.Fields[i].Value;
              try
                Table1.Post;
                UpdateAction := uaApplied;
              except;
                Table1.Cancel;
                raise;
              end;
          end;
        ukDelete:
          begin
            Table1.Delete;
            UpdateAction := uaApplied;
          end;
      end;
end;
```

Figure 4: The *OnUpdateRecord* event handler assigned to the Query.

As with *OnUpdateRecord*, *OnUpdateError* is passed parameters for a *DataSet*, an *UpdateKind*, and an *UpdateAction*. These parameters contain the same information as the corresponding parameters for *OnUpdateRecord*. In addition, the *E* parameter points to the exception raised during the failure. You use this exception to determine how to handle the error.

Unlike *OnUpdateRecord*, where it makes no sense to use the *UpdateAction* value of *uaRetry*, *uaRetry* plays an important role in handling update errors. Specifically, in addition to aborting the update or skipping the current record, you can also make any changes necessary to the record, then attempt to update it again. Setting *UpdateAction* to *uaRetry* causes the DataSet to attempt to post the current record again, either through its internal code, or by calling your *OnUpdateRecord* event handler again.

```
procedure TForm1.Query1UpdateError(DataSet: TDataSet;
  E: EDatabaseError; UpdateKind: TUpdateKind;
  var UpdateAction: TUpdateAction);
var
  BdeError: PChar;
  NewVal: string;
begin
  GetMem(BdeError, 1024);
  try
    BDE.DbiGetErrorString(DBIERR_KEYVIOL,BdeError);
    // Test if a key violation message is in the exception.
    // If a key violation occurred, permit the user
    // to enter a new key.
    if E.message = copy(StrPas(BdeError),1,
                        Length(E.message)) then
      begin
        NewVal := DataSet.Fields[0].NewValue;
        if InputQuery('Key violation',
                      'Enter new key',NewVal) then
          begin
            DataSet.Edit;
            DataSet.Fields[0].Value := NewVal;
            DataSet.Fields[0].NewValue := NewVal;
            UpdateAction := uaRetry;
          end
        else
          // Use uaAbort since the user has clicked Cancel.
          // No error message is necessary.
          UpdateAction := uaAbort;
      end
    else
      ; // The default UpdateAction is uaFail.
      // Permit the update to fail.
  finally
    FreeMem(BdeError);
  end;
end;
```

Figure 5: The *OnUpdateError* event handler of the CACHE5 project.

The use of *OnUpdateError* is demonstrated in the CACHE5 project (see Figure 5). Here the event handler is looking for a specific type of error: a key violation. If this error is encountered, the user can assign another key value, after which the update to the record is retried.

Within this event handler, the invalid key value is stored in the variable *NewVal*. This value is then displayed in an *InputQuery* dialog box, which allows the user to enter a correct key value.

Once the value is entered, the DataSet for which the event handler is executing is placed into the *dsEdit* state using the *Edit* method, and the new value is assigned to both the *Value* and *NewValue* properties of the key field. The *OnUpdateRecord* event handler is then re-triggered by setting the *UpdateAction* parameter to *uaRetry*.

If you want to permit a user to edit changes to a record being posted during a cached update, there are two reasons for caution:

- If the user happens to leave the workstation during the update, a transaction may be left open, awaiting the user's return.
- With the preceding code example, I've found that if the user enters a second invalid key in response to an initial invalid key, the record can't be posted, and an exception will be raised when a valid key is entered. This behavior may be due to a bug in the cached-updates feature.

Consequently, the best approach may be to use *OnUpdateError* to copy any invalid record to a temporary table for manual processing by an administrator, setting the *UpdateAction* parameter to *uaSkip* when the record has been copied.

Alternatively, you can either validate and post the updated record from within *OnUpdateError*, or use an UpdateSQL object to update the record instead of an *OnUpdateRecord* event handler. Regardless, you should rigorously test any *OnUpdateError* event handler if you plan to use it for correcting invalid records.

Executing UpdateSQL Queries from *OnRecordUpdate*

In the preceding example of the *OnUpdateRecord* event handler, Table methods such as *Locate*, *Insert*, and *Delete* were used to perform the updates of the cached edits. Furthermore, you learned that the *UpdateObject* property of a DataSet is ignored if a procedure is assigned to *OnUpdateRecord*. This doesn't mean, however, that you can't use an UpdateSQL object with an *OnUpdateRecord* event handler. In fact, it's possible, and in some cases highly desirable, to assign one or more UpdateSQL components from your code to *OnUpdateRecord*.

The most common reason for using UpdateSQL components from within *OnUpdateRecord* is that the DataSet is a read-only query that contains a join between two or more tables. A single UpdateSQL component can't be used to update both tables; you need one UpdateSQL component for each table that needs updating.

When two or more UpdateSQL components are in use (unlike when they're used singly), don't use the *UpdateObject* property of the DataSet. Instead, call methods of the *TUpdateSQL* class to execute the queries associated with the various UpdateSQL components.

The most common technique is to call the UpdateSQL method *Apply*, which has the following declaration:

```
procedure Apply(UpdateKind: TUpdateKind);
```

When you call *Apply*, it executes the UpdateSQL statement associated with the type of update. For example, if you pass this method a parameter of *ukDelete*, the *DeleteSQL* statement of the specified UpdateSQL component is executed.

The other alternative is to call *SetParams*, followed by *ExecSQL*. The following are the declarations of these two methods:

```
procedure SetParams(UpdateKind: TUpdateKind);
procedure ExecSQL(UpdateKind: TUpdateKind);
```

Actually, the call to *Apply* encapsulates calls to *SetParams*, which performs the run-time binding of the query parameters, followed by *ExecSQL*. The only time you would want to use *SetParams* and *ExecSQL* is when a particular query includes parameters in addition to the defaults.

For example, to perform a *ukDelete* action on an orders table without deleting records that have shipped within the last month, you could include — in the DELETE query — a parameter used in the WHERE clause to test the shipping-date field. Then, when applying the updates, you would begin by calling *SetParams* to perform the default run-time parameter binding, then assign an appropriate date value to your date parameter, and finally call *ExecSQL*.

If the SQL statements you assign to your UpdateSQL components don't use aliases (*DatabaseNames*), you'll also need to assign UpdateSQL's *DataSet* property before calling *Apply* or *SetParams*. *DataSet* is a run-time property that points to the DataSet affected by the SQL statements. This permits the UpdateSQL component to use the Database being used by the DataSet.

The use of multiple UpdateSQL components is demonstrated in the project CACHE6, shown in Figure 6. This project includes a read-only query that joins two tables. These tables, CUSTOLY1.DB and RESERVA1.DB, are copied from the CUSTOLY.DB and RESERVAT.DB tables supplied with Delphi, from within the *OnCreate* event handler for the main form.

Also from within the *OnCreate* event handler, Query1 is assigned to the *DataSet* properties of both UpdateSQL components. The following SQL statements are associated with Query1:

```
SELECT  d.ResNo, d.EventNo, d.CustNo, d.NumTickets,
        d.Amt_Paid, d.Pay_Method, d.Card_No, d.Card_Exp,
        d.Pay_Notes, d.Purge_Date, d.Paid,
        d1.Last_Name, d1.First_Name, d1.VIP_Status,
        d1.Address1, d1.Address2, d1.City, d1."State/Prov",
        d1.Post_Code, d1.Country, d1.Phone, d1.Fax,
        d1.EMail, d1.Remarks
  FROM  "Reserva1.db" d, "Custoly1.db" d1
 WHERE  (d1.CustNo = d.CustNo)
```

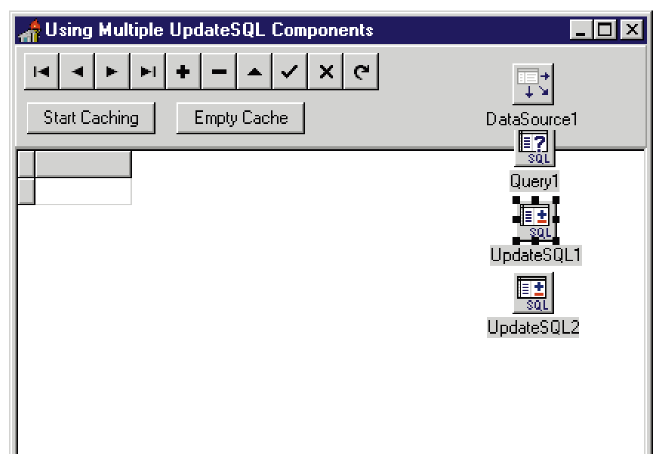


Figure 6: The CACHE6 project demonstrates the use of two UpdateSQL components to update the result set of a join.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  OldTable: TTable;
  NewTable: TTable;
begin
  OldTable := TTable.Create(Self);
  NewTable := TTable.Create(Self);
  try
    OldTable.DatabaseName := 'DBDEMOS';
    OldTable.TableName    := 'CUSTOLY.DB';
    NewTable.Tablename    := 'CUSTOLY1.DB';
    NewTable.DatabaseName := 'DBDEMOS';
    NewTable.BatchMove(OldTable,batCopy);
    NewTable.AddIndex('','CustNo',[ixPrimary, ixUnique]);
    OldTable.TableName    := 'RESERVAT.DB';
    NewTable.Tablename    := 'RESERVA1.DB';
    NewTable.BatchMove(OldTable,batCopy);
    NewTable.AddIndex('','ResNo',[ixPrimary, ixUnique]);
    Query1.DatabaseName   := 'DBDEMOS';
    Query1.Open;
    UpdateSQL1.DataSet    := Query1;
    UpdateSQL2.DataSet    := Query1;
  finally
    OldTable.Free;
    NewTable.Free;
  end;
end;
```

Figure 7: The *OnCreate* event handler for the CACHE6 project's main form.

```
UPDATE "Reserva1.db"
   SET EventNo    = :EventNo,
       CustNo     = :CustNo,
       NumTickets = :NumTickets,
       Amt_Paid   = :Amt_Paid,
       Pay_Method = :Pay_Method,
       Card_No    = :Card_No,
       Card_Exp   = :Card_Exp,
       Purge_Date = :Purge_Date,
       Paid       = :Paid
 WHERE ResNo = :OLD_ResNo
```

Figure 8: The *ModifySQL* property of UpdateSql1.

The code in Figure 7 is the *OnCreate* event handler for the CACHE6 project's main form. When the cached updates are applied, the *OnUpdateRecord* event handler calls the *Apply* methods of the UpdateSQL1 and UpdateSQL2 components.

Here's the *OnUpdateRecord* event handler for the Query:

```
procedure TForm1.Query1UpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  UpdateSQL1.Apply(UpdateKind);
  UpdateSQL2.Apply(UpdateKind);
  UpdateAction := uaApplied;
end;
```

As you can see, the event handler is very simple.

Probably the hardest part of using UpdateSQL components from within *OnUpdateRecord* is ensuring the queries associated with the component make sense. While this may seem obvious, it can be much harder than you think. In CACHE6, for instance, the UpdateSQL2 component, which is used to update the CUSTOLY1.DB table, does not include a DeleteSQL query. As you can imagine, the fact that a reservation is deleted for a customer doesn't mean that the user also wants to delete the customer. Likewise, the memo fields and autoincrement fields can't be modified. Your SQL statements must take this into account.

```
UPDATE "Custoly1.db"
   SET "Custoly1.db"."Last_Name"  = :"Last_Name",
       "Custoly1.db"."First_Name" = :"First_Name",
       "Custoly1.db"."VIP_Status" = :"VIP_Status",
       "Custoly1.db"."Address1"   = :"Address1",
       "Custoly1.db"."Address2"   = :"Address2",
       "Custoly1.db"."City"       = :"City",
       "Custoly1.db"."State/Prov" = :"State/Prov",
       "Custoly1.db"."Post_Code"  = :"Post_Code",
       "Custoly1.db"."Country"    = :"Country",
       "Custoly1.db"."Phone"      = :"Phone",
       "Custoly1.db"."Fax"        = :"Fax",
       "Custoly1.db"."EMail"      = :"EMail"
 WHERE "Custoly1.db"."CustNo" = :"OLD_CustNo"
```

Figure 9: The *ModifySQL* property of UpdateSQL2.

```
INSERT INTO "Custoly1.db"
  ("Custoly1.db"."Last_Name", "Custoly1.db"."First_Name",
   "Custoly1.db"."VIP_Status", "Custoly1.db"."Address1",
   "Custoly1.db"."Address2", "Custoly1.db"."City",
   "Custoly1.db"."State/Prov", "Custoly1.db"."Post_Code",
   "Custoly1.db"."Country", "Custoly1.db"."Phone",
   "Custoly1.db"."Fax", "Custoly1.db"."EMail")
VALUES
  (:"Last_Name", :"First_Name", :"VIP_Status", :"Address1",
   :"Address2", :"City", :"State/Prov", :"Post_Code",
   :"Country", :"Phone", :"Fax", :"Email")
```

Figure 10: The *InsertSQL* property of UpdateSQL2.

The *ModifySQL* property of UpdateSQL1 contains the SQL statements shown in Figure 8. Here's the *InsertSQL* property from the same component:

```
INSERT INTO "Reserva1.db"
  (EventNo, CustNo, NumTickets, Amt_Paid, Pay_Method,
   Card_No, Card_Exp, Purge_Date, Paid)
VALUES
  (:EventNo, :CustNo, :NumTickets, :Amt_Paid, :Pay_Method,
   :Card_No, :Card_Exp, :Purge_Date, :Paid)
```

The *DeleteSQL* property of UpdateSQL1 is shown here:

```
DELETE FROM "Reserva1.db"
WHERE ResNo = :OLD_ResNo
```

Let's now consider the SQL statements from UpdateSQL2. The *ModifySQL* property is shown in Figure 9. Finally, Figure 10 is the *InsertSQL* property from UpdateSQL2.

Conclusion

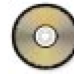
Cached updates greatly increase the number of options you have when it comes to editing records. In addition, they can generally improve your application's performance. In its simplest case — that of editing a single table — cached updates are easy to employ. Even when your caching needs are complex, however, Delphi's cached-update capabilities provide the tools necessary to get the job done right. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUL\DI9707CJ.

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://gramercy.ios.com/~jdsi. You can also reach Jensen Data Systems at (281) 359-3311, or via e-mail at cjensen@compuserve.com.

Delphi 2 / Object Pascal

By *Peter Dove and Don Peer*



Optimizing Graphics

Delphi Graphics Programming: Part V

Speed is a primary concern in graphics programming, and optimization is one of the most studied areas of games programming. This month, we'll discuss several methods of optimization, and apply them to our example program (see Figure 1).

To keep *TGMP* as object-oriented as possible, we'll exclude some of the more "exotic" optimization methods. Our goal is to increase the speed that *TGMP* rotates the Shaded Textured Cube, at its default position, by 50 percent. So first, we'll optimize the DIB class, then *TGMP*.

Note that optimization involves knowing *where* to optimize. A function called only once during an object's life is probably not worth optimizing. However, a method that's called repeatedly is worthy of attention. An extreme example is a deeply nested loop. If the inner loop contains inefficiencies, they're magnified in exact relation to the number of times the loop is executed.

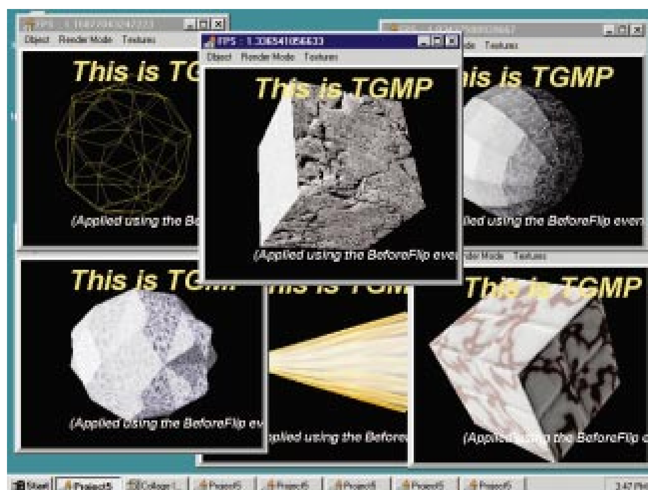


Figure 1: The sample application — times six.

Multiplication and Division

The first optimization method we'll cover is a quick way to divide and multiply integers. Using the **shl** and **shr** operators enables you to shift integers bitwise, to the left or right. If you bit-shift an integer 1 to the left, you multiply its value by 2 (binary is base 2). If you bit-shift an integer 1 to the right, you divide its value by 2.

Bit-shifting is quick — it only takes one clock cycle of the processor to perform a bit-shift. Multiplication and division operations will take many times longer depending on the processor you use. Here's an unoptimized line of code, and its optimized equivalent using a bit-shift operation in place of conventional multiplication (from the *DrawHorizontalLine* procedure in DIB16.PAS):

```
{ Previous statement. }
BasePointer := Pointer(FScanWidthArray[Y] +
                       (X * 2));

{ Optimized statement. }
BasePointer := Pointer(FScanWidthArray[Y] +
                       (X shl 1));
```

Two Timing

One of the DIB class' main tasks is to clear the backpage. Unfortunately, the backpage is cleared only one pixel at a time. Because a pixel is 16 bits of data, and the largest integer size accessible in Delphi 2 is 32 bits, this begs the question, "How can we clear the backpage in 32-bit hits rather than in 16-bit

```
procedure TDIB16bit.ClearBackPage(Color : Word);
var
  X, XColor   : Integer;
  BasePointer : ^Integer;
begin
  { Loop through bitmap, setting every pixel
    to Color parameter. }
  BasePointer := FPointerToBitmap;

  XColor := Color and (Color shr 16);

  // Optimization: Shift division.
  // Replace div 2 with shr 1.
  for X := 0 to (((FScanWidth shr 1) *
           (FBHeader.bmiHeader.biHeight))-1) shr 1 do
begin
    BasePointer^ := XColor;
    { Inc increments a pointer by the size of the type that
      the pointer points to; in other words, 4 bytes. }
    Inc(BasePointer);
  end;
end;
```

Figure 2: The new, optimized *ClearBackPage* procedure.

hits?" Theoretically, this would cut the operation time in half. Incidentally, the *ClearBackPage* procedure is also a good candidate for optimization; it's called numerous times and consumes large chunks of program time.

So how do we do it? Recall in Part IV, we talked a lot about bit-shifting. Figure 2 shows the fully-commented *ClearBackPage* procedure with all optimizations in place. Note that *BasePointer* is now a pointer to a 32-bit integer value. To write two pixels at a time, we move the Word *Color* into the top 16 and bottom 16 bits of a 32-bit integer. The new *XColor* variable is 32-bit, so we perform an and operation on *Color*, with *Color* shifted to the left 16 bits. We incorporated bit-shifting optimization methods with the bit-shifting, replacing the two division operations. Also, we are "blitting" 32 bits at a time, instead of 16 bits.

Lookup Tables

Lookup tables offer a quick way to pre-calculate multiple variables and place them into an array. Thus, the program only has to view a place in memory for a result, rather than calculate it. In this section, we'll implement lookup tables in the *SetPixel* and *DrawHorizontalLine* procedures of the DIB class.

This unoptimized code from *SetPixel* is called every time a pixel is drawn in the
texture-mapping modes:

```
Integer(BasePointer) :=
  Integer(FPointerToBitmap) +
        (Y * FScanWidth) + (X * 2);
```

Also consider the following calculation. It only returns the beginning of the DIB class, then the *Y* offset into it:

```
Integer(FPointerToBitmap) + (Y * FScanWidth)
```

An array of pre-calculated *Y* positions can do a better job. We easily implement the arrays by adding this code to the DIB class:

```
{ Private section of the class. }
FScanWidthArray : array [0..800] of Integer;

{ Place at the end of the Create constructor. }
for X := 0 to 800 do
  FScanWidthArray[x] :=
    (FScanWidth * X) + Integer(FPointerToBitmap);
```

This new optimized line in the *SetPixel* procedure now uses the lookup table:

```
{ Using a lookup table and bit-shifting to increase speed. }
BasePointer := Pointer(FScanWidthArray[Y] + (X shl 1));
```

You probably noticed we also included another optimization, converting:

```
(X * 2)
```

to:

```
(X shl 1)
```

Now let's optimize *TGMP*. The optimized code for DIB16.PAS is available for download (see end of the article for details).

Reciprocals

There are still many ways of optimizing the code to obtain more speed. In this vein, we'll further optimize the GMP unit and the *TGMP* class.

The floating-point processor doesn't take the same amount of time for all calculations. For example, floating-point division takes much longer than floating-point multiplication. Thus, we should favor the use of floating-point multiplication wherever possible. A good way to do this is by using reciprocals. The *reciprocal* of a number is 1 divided by that number. For example, the reciprocal of 31 is 1/31, or 0.032258.

Before optimization, the *RemoveBackfacesAndShade* procedure continued the following:

```
R := Round(((255 - GetRValue(AnObject.Color)) / 31) *
      Intensity) + GetRValue(AnObject.Color);
G := Round(((255 - GetGValue(AnObject.Color)) / 31) *
      Intensity) + GetGValue(AnObject.Color);
B := Round(((255 - GetBValue(AnObject.Color)) / 31) *
      Intensity) + GetBValue(AnObject.Color);
```

Here's the optimized code which uses reciprocals:

```
R := Round(((255 - GetRValue(AnObject.Color)) * 0.032258) *
      Intensity) + GetRValue(AnObject.Color);
G := Round(((255 - GetGValue(AnObject.Color)) * 0.032258) *
      Intensity) + GetGValue(AnObject.Color);
B := Round(((255 - GetBValue(AnObject.Color)) * 0.032258) *
      Intensity) + GetBValue(AnObject.Color);
```

Fixed-Point Math

Integer math is much faster than floating-point math (especially when it comes to division), but how can we use integer math when we need to handle decimal values? We can simulate floating-point precision with integer math, using a number of bits to represent the fraction.

```
{ Unoptimized code without floating-point division. }
if RenderMode = rmSolidTexture then
  begin
    for Y := 0 to 479 do begin
      if YBuckets[Y].StartX = -16000 then
        continue;
      Length := (YBuckets[Y].EndX - YBuckets[Y].StartX) + 1;
      TextXIncr := ((TextureBuckets[Y].EndPosition.X –
                     TextureBuckets[Y].StartPosition.X)) /
                     Length ;
      TextYIncr := ((TextureBuckets[Y].EndPosition.Y –
                     TextureBuckets[Y].StartPosition.Y)) /
                     Length ;
      TextX := TextureBuckets[Y].StartPosition.X;
      TextY := TextureBuckets[Y].StartPosition.Y;
      for I:=YBuckets[Y].StartX to YBuckets[Y].EndX do begin
        if I < 0 then
          begin
            TextX := TextX + TextXIncr;
            TextY := TextY + TextYIncr;
            Continue;
          end;
        if I > Width then
          begin
            TextX := TextX + TextXIncr;
            TextY := TextY + TextYIncr;
            Continue;
          end;
        { Use the FDib SetPixel method instead of the
          Windows GDI SetPixel. }
        FDib.SetPixel(I, Y, FCurrentBitmap[Round(TextX),
                                           Round(TextY)]);
        TextX := TextX + TextXIncr;
        TextY := TextY + TextYIncr;
      end;
    end;
  end; { if RenderMode = rmSolidTexture ...}
```

Figure 3: From the unoptimized *RenderYBuckets* procedure.

For example, let's say you select 16 bits for the precision; you have just placed a decimal point in the middle of the 32-bit integer. Using eight bits of a 32-bit integer to represent the fraction provides a precision of 1/256.

Converting a normal integer value into a fixed-point value is the easiest way to perform this optimization. The code in Figure 3 is from the *RenderYBuckets* procedure. The *Length* variable is a normal integer. To convert it (or any normal integer) into a fixed-point integer, shift the bits left by the number of bits of precision (e.g. eight) you have decided to use. You can see this effect with these two statements:

```
TextXIncr := ((TextureBuckets[Y].EndPosition.X -
  TextureBuckets[Y].StartPosition.X) shl 8) div Length ;
TextYIncr := ((TextureBuckets[Y].EndPosition.Y -
  TextureBuckets[Y].StartPosition.Y) shl 8) div Length ;
```

The start position is subtracted from the end position, and the result shifted eight bits to the left. The divisor, *Length,* remains untouched, i.e. it was not shifted. If you converted length into a fixed point, the equation would suffer from incorrect scaling — the value would be too low. The opposite occurs when using fixed-point multiplication — you must downscale the answer by eight bits.

The results, *TextXIncr* and *TextYIncr*, are correctly formatted, fixed-point numbers. If length were converted into a fixed-point number (by shifting left eight positions), you would take *TextXIncr* and *TextYIncr,* then shift right eight bits to normalize it to a fixed-point number.

To reconvert a fixed-point number into a normal integer, simply reshift it right by the precision you selected (in this case, eight bits). Of course, the problem with fixed-point math is that the highest value available is the highest value (32), minus precision (24 bits in this case).

Let's look at an integer multiplication example that assumes eight bits of precision. Let's say we have a variable, *X*, with a value of 55; and a variable, *Y*, with a value of 10. First we convert each into an 8-bit fixed-point value using the **shl** operator:

55 **shl** 8 = 14080

and:

10 **shl** 8 = 2560

Next, we multiply:

14080 * 2560 = 36044800

Then we normalize:

36044800 **shr** 8 = 140800

Last, we obtain the conventional integer value:

140800 **shr** 8 = 550

which we can check using good ol' multiplication:

10 * 55 = 550

Now let's look at division. This formula, (2 / 10) * 20 = 4, for example, just can't be done with integer math. First, let's evaluate the expression in the parentheses. Sixteen bits of precision is needed to ensure this example is correct. Convert 2 to a fixed-point value:

2 **shl** 16 = 131070

The divisor, however, doesn't need to be converted:

131070 **div** 10 = 13107

Now we convert 20 into a fixed-point value:

20 **shl** 16 = 1310700

The final equation is:

13107 * 1310700 = 17179344900

```
if RenderMode = rmSolidTexture then
  begin
    for Y := 0 to Height do begin
      if YBuckets[Y].StartX = -16000 then
        continue;
      Length := (YBuckets[Y].EndX - YBuckets[Y].StartX) + 1;

      { Floating-point to fixed-point. }
      TextXIncr := ((TextureBuckets[Y].EndPosition.X -
                     TextureBuckets[Y].StartPosition.X)
                     shl 8) div Length ;
      TextYIncr := ((TextureBuckets[Y].EndPosition.Y -
                     TextureBuckets[Y].StartPosition.Y)
                     shl 8) div Length ;

      { Turns TextX and TextY into fixed-point
        integers with 8 bits of precision. }
      TextX := TextureBuckets[Y].StartPosition.X shl 8;
      TextY := TextureBuckets[Y].StartPosition.Y shl 8;
      for I:=YBuckets[Y].StartX to YBuckets[Y].EndX do begin
        if I < 0 then
          begin
            TextX := TextX + TextXIncr;
            TextY := TextY + TextYIncr;
            Continue;
          end;
        if I > Width then
          begin
            TextX := TextX + TextXIncr;
            TextY := TextY + TextYIncr;
            Continue;
          end;

        { To turn a fixed point integer to a normal integer,
          shift the bits right by the same amount shifted
          left when turning them into fixed point integer. }

        { Uses the FDib SetPixel method instead of
          the Windows GDI SetPixel. }
        FDib.SetPixel(I, Y, FCurrentBitmap[TextX shr 8,
                                           TextY shr 8]);

        { Addition of fixed point numbers is the same as
          normal integers and it is the same with
          subtraction of fixed point numbers. }
        TextX := TextX + TextXIncr;
        TextY := TextY + TextYIncr;
      end;  { for }
    end;  { for }
  end;  { if RenderMode = rmSolidTexture }
```

Figure 4: From the fully-optimized *RenderYBuckets* procedure.

Now correct the multiplication overflow:

shr 16 = 262140

Next, convert it into a conventional number by shifting right 16 bits:

262140 **shr** 16 = 4

The fully-optimized code from the *RenderYBuckets* procedure, with additional comments, is shown in Figure 4.

More on Optimization

Optimizing with with. Delphi supports other optimizations, such as with its **with** statement (see Figure 5). Here, the **with** statement has the compiler save a pointer to *Object3D.Polystore[P]*, so the pointer doesn't have to be evaluated each time through the **for** loop.

```
with Object3D.PolyStore[P] do begin
  if Z <> 0 then
    begin
      for I := 0 to NumberPoints - 1 do begin
        NewX :=
          Point[I].X * PreCalCos - Point[I].Y * PreCalSin;
        NewY :=
          Point[I].X * PreCalSin + Point[I].Y * PreCalCos;
        Point[I].X := NewX;
        Point[I].y := NewY;
      end;
    end;
end;  { with }
```

Figure 5: The Delphi **with** statement optimizes your code by resolving a pointer (to *Object3D.PolyStore[P]* in this case) once for an entire block of code.

Speed, speed, speed. Our original goal was to improve the engine's speed by 50 percent. Did we achieve it? On one machine, the code from Part IV ran at 10 frames per second (fps) with fully-lighted texturing. The new, optimized code runs at 14 fps, so we're not quite there yet. However, many optimizations remain. For now, we'll leave the final fps up to you; look at the different optimization sections provided and see if you can apply them to other portions of *TGMP*. And don't worry, we'll continue to implement optimizations in our next article.

It's in the cards. Note that the code will only run as fast as your graphics card can run. On a system with a Matrox Mystique graphics card, the fully-lighted textured cube ran at an amazing 30 fps. The commercial version of *TGMP* (due for release soon), ran at over 70 fps. Optimization software has shown most of the overhead in *TGMP* is consumed by the blitting-to-screen process. Thus, the faster your card, the faster the cube will spin. This can also be proven by reducing the window size in ARTICLE5.EXE while it's running. Each time you reduce the window size, the fps will increase proportionately.

From our readers. A reader, Fred Mitchell, found that the *CrossProduct* function was wrong, and we have corrected it accordingly. The *RemoveBackfacesAndShade* method was also altered to ensure it consistently refers to the new *PolyWorld* array instead of sometimes referring to the *PolyStore* or *PolyWorld* array. This was an oversight in the upgrade to the World Coordinate System (from Part IV).

Our Fifth Application

Essentially, our fifth application is the same as that presented in Part IV, with two major additions:

1) We added code to the *Timer1.Timer* event that analyzes the frames per second, running a type constant counter within the event. When the counter reaches 50, it evaluates the time elapsed since it arrived at 50. The rest is simple math. The result is displayed in the title bar to indicate any speed improvements you may want to make.
2) The *SetLightSourcePosition* procedure in *TGMP* configures the light vector's exact orientation. The method accepts two parameters: the position of the light in space, and its direction, i.e. a point in space to which the light "looks."

The complete source for our fifth application developed with the *TGMP* component is available in ARTICLE5.PAS.

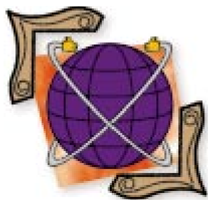
Conclusion

Aside from adding the camera coordinate system, next month we'll add more optimizations (so you can compare those you devised with the ones we implemented), allow for embedding sprites into the 3D engine, and change the system to cope with displaying more than one object at a time. Finally, we'll provide the ability to display and build the scene at design time. See you then. ∆

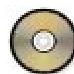
The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUL\DI9707DP.

Peter Dove is a partner in Graphical Magick Productions, specialists in graphics, training, and component development. He can be reached via the Internet at peterd@graphicalmagick.com.

Don Peer is a Technical Associate with Greenway Group Holdings Inc. (GGHI). He can be reached via the Internet at dpeer@mgl.ca.



By John Penman



NetCheck: Part II

Completing the 32-Bit Network Debugging Tool

In the whirls of the Internet and intranets, developers must ensure their network programs remain robust. A network debugging tool, therefore, is essential for any developer building Internet and/or intranet applications.

In Part I of this series (presented in the May *Delphi Informant*), we began examining NetCheck, a simple network debugging application that uses three, non-visual Delphi components, each encapsulating a well-known debugging service:

1) Sonar, a wrapper for the ping application;
2) EchoC, an echo client wrapper for the Echo service, similar to ping; and
3) Trace, which encapsulates the TraceRoute service. Trace maps the route of the packets between the sending and receiving machines. This mapping can help find any bottlenecks or "breaks" in the network.

In May, we implemented Sonar in NetCheck. This month, we're extending the utility by adding the EchoC and Trace components (see Figure 1).

Kicking Out the Jams

Recall that Sonar operates in blocking mode, causing the user interface to "freeze" and preclude user interaction. Because Trace descends from Sonar, it inherits this not-too-serious, but inconvenient, problem.

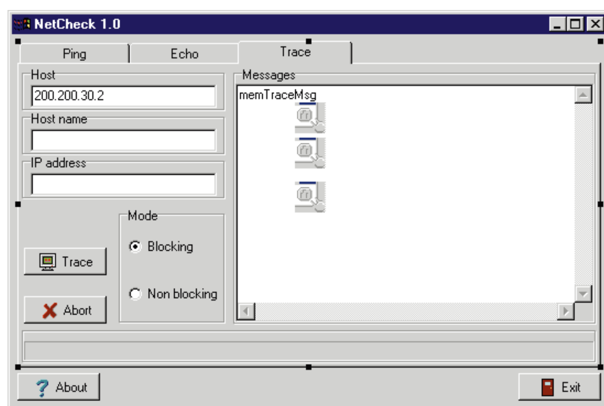


Figure 1: NetCheck showing Sonar, EchoC, and Trace at design time.

To overcome the effects of blocking, we'll use the *TThread* class to add multi-tasking capability to Trace and Sonar. However, we'll take a different approach when handling the Echo service. To work asynchronously, EchoC uses the Winsock function, *WSAAsyncSelect*. This permits you to work with NetCheck while processing the background, *without* using threads.

Full of Echoes

The Echo service is used to verify the connection between the target host and client machine, and that the server is operating at the application level. In contrast, Sonar and other ping applications test the connection only at the target machine's network interface; this doesn't indicate the target machine's operating system is functioning. The downside of using the Echo service is that an echo server must be running on the target host. Ping applications, including Sonar, do not require such a server.

When the host responds positively to an echo request using the Echo service, you know the server is working with a viable connection. (EchoS.pas is the source to implement the EchoS component, used by the echo server program, EchoServe.exe. Because EchoS is similar to EchoC, we won't cover it in detail. EchoS.pas is available for download; see end of article for details.)

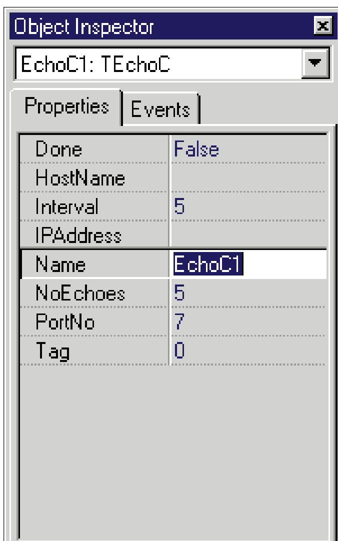


Figure 2: The Object Inspector showing EchoC's published properties.

The operation of the echo client is simple: It sends a test message at predetermined intervals to the echo server, on port seven (IPPORT_ECHO). The server then reflects the test message back to the client. If the echo server doesn't respond, either the network interface or target machine's operating system is down, or both. Another possibility is that the echo server program hasn't started. When this happens, Sonar can be used to verify the connection at the target machine's network interface, to narrow the number of possible causes.

The EchoC Component

The *TEchoC* class descends from the *TComponent* class to form the basis of the EchoC control (see Listing Four beginning on page xx). The *TEchoC.Create* constructor includes the *CheckWS* function to initialize WinSock.DLL before use. If the .DLL isn't available, EchoC posts an error message, and closes NetCheck.

Two methods, *Start* and *Stop*, are declared in *TEchoC*'s **public** section:

```
public
  { Public declarations }
  procedure Start;
  procedure Stop;
```

The *Start* procedure calls *GetHost* to resolve a given host name before beginning the echo process. The *Stop* procedure halts the echo process.

You control EchoC's behavior through its published properties, *PortNo*, *Interval*, and *NoEchoes*. They allow you to specify a port, the time lapse between transmissions, and the number of times to send a message, respectively. Although the Echo service uses the standard IPPORT_ECHO port to transmit messages, you can change it to any port number. However, you must also change the echo server's port; otherwise, communication is not possible.

You can alter the values of published properties at design time (see Figure 2) or run time. For flexibility, these properties are available through appropriate controls on the Echo page in NetCheck (see Figure 3). For example, to change the interval between transmissions from 0 to 999 seconds, use the UpDown component for **Interval (secs)**. In the same way, you can indicate the number of messages to send in **No of echoes**.

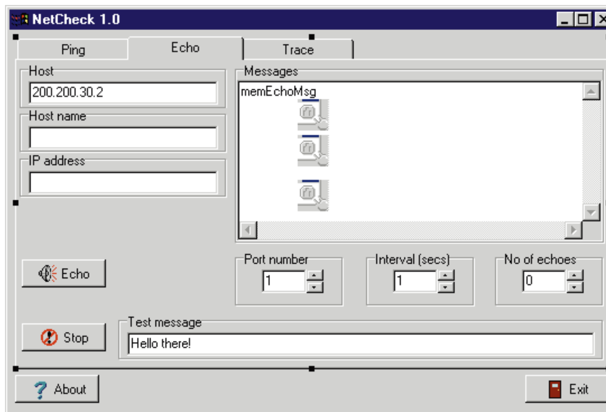


Figure 3: The controls on the Echo page allow you to change the default values before attempting an echo.

The Echo service can use the Transmission Control Protocol (TCP), or User Datagram Protocol (UDP) to send messages. TCP, a streaming protocol, guarantees reliable delivery of data, using a virtual circuit between the sender and receiver machines across the network. UDP, a connectionless protocol, does not require a virtual circuit. Unlike TCP, UDP contains no error checking, and consequently, has lower overhead.

TCP is analogous to using registered snail mail to send information safely, requiring the recipient to acknowledge receipt. UDP is similar to using regular snail mail; the recipient doesn't acknowledge receipt. In this case, EchoC only uses UDP; however, you can easily implement the TCP version of the Echo service. (TCP is partially implemented in the EchoC and EchoS components. I leave this enhancement to you.)

Processing Echoes

The *GetHost* method configures a socket to transmit messages and receive message echoes. A socket is an endpoint in a communication link, used to send data and listen for incoming data.

After *GetHost* creates the socket, *Start* calls *AllocateHwnd* to create the *FWnd* and *FTWnd* handles to invisible windows for the *EchoEvent* and *TimerEvent* event procedures, respectively. When the socket receives notification that data is ready to read or send, the *EchoEvent* procedure responds. To make triggering of *EchoEvent* possible, *WSAAsyncSelect* is called through *StartAsyncSelect* to put the socket, *FSocketNo*, into non-blocking mode. EchoC is now ready to work asynchronously.

After sending a message, the socket waits in the background for an echo reply. When the socket receives the data (from the target host), Winsock sends an FD_READ notification. This triggers *EchoEvent* to call *GetData* to read the data. EchoC then posts the echo reply to *memEchoMsg*, a Memo control on NetCheck's Echo page. After an interval of *Interval* seconds, Windows sends a WM_TIMER message. This triggers the *TimerEvent* method to call *SetData* to send another message.

Each time Windows triggers *TimerEvent*, *TimerEvent* compares *FWriteCount* — a counter that is incremented with

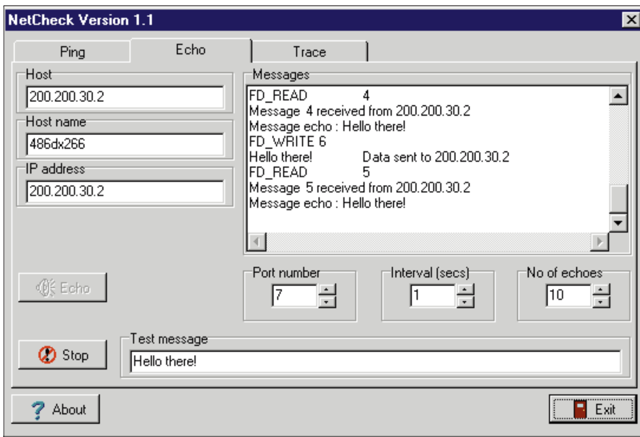


Figure 4: EchoC in action.

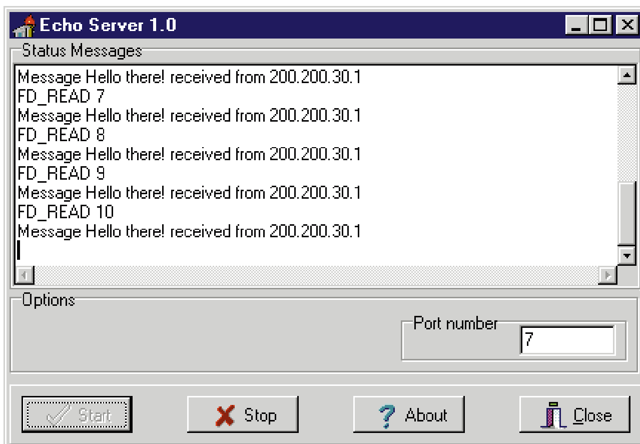


Figure 5: The Echo Server program responding to echo requests from NetCheck.

each transmission — with *FNoEchoes*. If *FWriteCount* exceeds *FNoEchoes*, *TimerEvent* calls the following code in *TimerEvent* to halt transmission:

```
if FWriteCount > FNoEchoes then
  begin
    KillTimer(FTWnd,1);
    DeallocateHWND(FTWnd);
    FDone := True;
    OnDoneEvent;
    Exit;
  end;
```

OnDoneEvent then posts a message to the *Stop* method to halt the echo process.

To use the EchoC component, select the Echo page in NetCheck and enter the target host's name or IP address in **Host name** or **IP address**. Then, alter the default settings for the port number, interval, and number of echoes. Finally, enter the test message in the Edit control, *edEchoTestMsg*.

To start the echo process, click the **Echo** button. The exchange of data immediately appears in the *memEchoMsg* Memo control (see Figure 4). Figure 5 shows the Echo Server program responding to echo requests from NetCheck. You can halt transmission at any time by clicking the **Stop** button, which activates the *Stop* method. The asynchronous nature of EchoC provides the freedom to cease the process.

Tracing Packets

Trace checks the connectivity and the route between two machines, tracking any bottlenecks or "breaks" between client and server. Trace can also be used to determine if a routing problem is causing a network application failure. Although you can't solve any network problems that cause the packets to disappear en route, you can obtain evidence of a routing problem.

Like all TraceRoute programs, Trace uses the Time To Live mechanism (TTL) in the Internet Protocol (IP). TTL indicates the number of hops a packet can travel before expiring. Recall from Part I in May that a *hop* is a link between any two machines on a path over the network. Twenty hops may exist between the sending and target machines.

Let's say a packet's TTL is 32, and the target host is 33 hops from the sending machine. This packet will expire before reaching its destination. However, if a packet's TTL is greater than 32, it will probably reach that destination. All TraceRoute (a.k.a. hopcheck) programs use this principle to map the route between the sender and receiver.

When you start a trace, the initial TTL of an ICMP packet is one. The first machine on the route receives it, and decrements the packet's TTL. When the router sees the packet's TTL is zero, it returns an error message, indicating the packet's TTL has expired.

Next, note the first machine's address and send another packet with a TTL of two. The first machine decrements the TTL by one, then forwards the packet to the next machine when it sees the TTL is non-zero. When the second machine receives the packet, it decrements the packet's TTL by one. After seeing this packet's TTL is zero, the second machine returns an error message. Again, note the address of the second machine and increase the TTL by one to three. Continue this cycle of sending the packet with increasing TTL until it reaches the destination host, or dies because of a network problem.

Using the TTL mechanism isn't foolproof, because the routes can vary between each packet sent. In spite of this, TTL is a useful tool.

The Trace Component

The Trace component was created by deriving the *TTrace* class from the *TSonar* class. Therefore, Trace inherits *TSonar*'s *Create* constructor. This method checks the status of the ICMP and Winsock DLLs. *Create* sets the TTL to a default value of 128, a reasonable value to cover most routes (see Figure 6).

TTrace uses *TSonar*'s *GetHost* method to obtain the address of the target machine to trace. Like the Sonar component, Trace must initialize the *TIPOptions* and *TICMPEchoReply* records before starting a trace. (For more information on this, refer to Part I.)

The heart of the Trace component is the *DoTrace* method. First, *DoTrace* obtains the addresses of the *IcmpCreateFile*, *IcmpCloseHandle,* and *IcmpSendEcho* functions, exported

```
type
  TTrace = class(TSonar)
  private
    { Private declarations }
    FTimeToLive : Byte;
    FIPFound, FHostFound : string;
    procedure ResolveHost;
  protected
    { Protected declarations }
    pEchoReply : pIcmpEchoReply;
    procedure DoTrace; virtual;
    procedure Stats; override;
  public
    { Public declarations }
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
    procedure Trace;
  published
    { Published declarations }
    property TimeToLive : Byte
      read FTimeToLive write FTimeToLive default 128;
end;
```

Figure 6: The *TTrace* class.

by the ICMP.DLL. Then, *pEchoReply* and *FIPOptions* are initialized, with the *FIPOptions.TTL* field set to 1.

Before starting the *IcmpSendEcho* function, *DoTrace* creates a handle for *IcmpCreateFile*. As with the calls to the ICMP and Winsock DLLs, *DoTrace* checks the result of the *IcmpCreateFile* function. If *IcmpCreateFile* returns a value of INVALID_HANDLE_VALUE, *DoTrace* aborts with an error message, and exits to NetCheck.

A **while** loop executes the *IcmpSendEcho* function until the control variable, *Finished*, is set to *True* by the following code:

```
if (pEchoReply.Status = IP_SUCCESS) or
   (FIPOptions.TTL > FTimeToLive)   then
  Finished := True
else
  Inc(FIPOptions.TTL);
```

After the *IcmpSendEcho* function is executed, the code examines the *pEchoReply.Status* field. If it contains an IP_TTL_EXPIRED_TRANSIT value, the packet's TTL has expired. Next, the application checks that the address returned by *IcmpSendEcho* is valid, then calls *ResolveHost* to resolve the name of the machine that dispatched the error message. Then TTL is increased by 1, provided the value of *FIPOptions.TTL* is less than that of *FTimeToLive* (set at design time).

If *pEchoReply.Status* contains the value IP_REQ_TIMED_OUT, a time out has occurred, perhaps because of heavy network traffic. When the ICMP packet finally reaches the target host, *IcmpSendEcho* returns a value of IP_SUCCESS, then calls *ResolveHost* to determine the host's name. Then, *Stats* is called to post the number of hops to reach the host, and target the host's name. Finally the *Finished* flag is set to *True* to terminate the **while** loop.

Mapping a Path

In NetCheck, select the Trace tab. First, enter the target host's name or IP address, then click the **Trace** button to begin the trace. The **Trace** button calls the Trace component's *Trace*

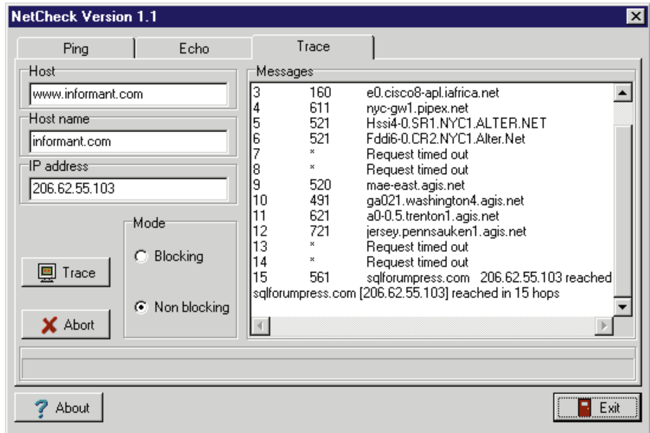


Figure 7: The Trace page, after tracing a route between www.informant.com and the sender.

method, which in turn, calls *GetHost* to obtain the target host's address. *Trace* then calls *DoTrace* to send the packets. Each time the Trace component receives data from a machine, it posts a message to the *memTraceMsg* Memo control, through the component's *Msg* property (see Figure 7).

You can press the **Abort** button to cancel the trace at any time. However, Trace operates in blocking mode. This causes NetCheck's user interface to be unresponsive, preventing you from using **Abort**. To enable the use of the **Abort** button, we must put Trace to work in non-blocking mode. This is where threads can help.

Using Threads

We aren't implementing *true* non-blocking versions of Sonar and Trace. The intent is to add multi-threading capability without changing these components. Sonar and Trace remain blocking in nature; they're placed on a thread separate from the primary thread, which is usually the user interface. Thus, NetCheck's interface can be used to perform other tasks. For example, we can exercise Trace's capabilities while simultaneously using EchoC.

Adding threading capability to Sonar and Trace without compromising their integrity, however, involves breaking a cardinal rule: I don't use the *Synchronize* method of the *TThread* class. Any messages the components send to update NetCheck's Memo and ProgressBar components should be done through *Synchronize*. I didn't use the *Synchronize* method because it would have required me to "hardwire" the locations of these controls (i.e. Memo, Edit, etc.) from within the Sonar and Trace components. Therefore, be careful when using Sonar and Trace components in non-blocking mode. (In testing these components, I didn't encounter any problems by not using *Synchronize*.)

To add multi-threading capability to Trace, use Delphi's New Items dialog box to create a new class, *TTraceThrd*, in the TraceThrd unit (see Figure 8). The *Create* constructor initializes a private copy of the *TTrace* class in *FTrace*. To make this *TTraceThrd* class available to NetCheck, declare *Tracer* in the **interface** section of the TraceThrd unit, and add `TraceThrd` to

```
type
  TTraceThrd = class(TThread)
  private
    { Private declarations }
    FTrace : TTrace;
  protected
    procedure Execute; override;
  public
    constructor Create(TTracer : TTrace; Name : string);
  end;
```

Figure 8: The *TTraceThrd* class enables Trace to be non-blocking.

```
procedure TpdMain.bbtnTraceClick(Sender: TObject);
begin
  with Trace1 do begin
    if Mode = Blocking then
      begin
        HostName := edTraceHost.Text;
        Trace;
      end
    else
      begin
        TimeToLive := OldTimeToLive;
        Tracer := TTraceThrd.Create(Trace1,
                                    edTraceHost.Text);
      end;
  end;
end;
```

Figure 9: Attach this code to the **Trace** button.

the **uses** clause in Main.pas. Next, add the code in Figure 9 to the **Trace** button. (These techniques are applicable to Sonar as well.)

Now when you need to abort a trace, click the **Abort** button. It simply resets Trace's *TimeToLive* property to 1, halting the trace completely. Before starting Trace in non-blocking mode, set the *Mode* property to **Non blocking** in the *rgTraceMode* RadioGroup control.

Conclusion

With these enhancements, NetCheck is a basic debugging tool that can be used as-is, or extended to include more features. For example, you may want to include the option to select a TCP or UDP for the Echo service. Additionally, you may want to enhance the interface by adding a pick list of favorite hosts that could be stored in the Windows 95 or Windows NT 4.0 registry.

We covered the internals of a Delphi application for network debugging. However, network debugging techniques is a large topic, far beyond the scope of this article. To help you, a list of references is included here. (I particularly recommend Chapter 13 in *Windows Sockets Network Programming*.) ∆

References

Chapman, Davis, *Building Internet Applications with Delphi 2* [QUE, 1996].
Dumas, Arthur, *Programming Winsock* [SAMS, 1995].
Quinn, Bob, and David Shute, *Windows Sockets Network Programming* [Addison-Wesley, 1996].
Stevens, W. Richard, *UNIX Network Programming* [Prentice Hall, 1990].
Taylor, Don, et al., *KickAss Delphi Programming*, Chapters 4 and 5 [Coriolis Group, 1996].
Verbruggen, Martien. Source code for the demonstration ping program is available on the Web from http://www.tcp.chem.-tue.nl/~tgtcmv and http://www.dephi32.com/apps.

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUL\DI9707JP.

John Penman is the owner of Craiglockhart Software, which specializes in providing Internet and intranet software solutions. John can be reached on the Internet at jcp@iafrica.com.

Begin Listing Four — The *TEchoC* Class

```
type
  CharArray = array[0..MaxBufferSize] of char;
  TConditions = (Success, Failure, None);
  TTransport = (TCP, UDP);
  TEchoC = class(TComponent)
  private
    { Private declarations }
    FParent : TComponent;
    FStatus : TConditions;
    FVersion, FVersionDate, FComponentName,
    FDeveloper : string;
    FStatusWS, FOkay : Boolean;
    FProgress : Integer;
    Fh_addr : pChar;
    FMsgBuf : CharArray;
    function CheckWS : Boolean;
  protected
    { Protected declarations }
    FNoSent, FNoRecv, FNoEchoes, FMin, FMax, FAve,
    FwMsg, FRTTSum : Word;
    FHostName, FHostIP : string;
    FOnRecv, FOnNewData, FOnProgress,
    FOnDone : TNotifyEvent;
    FSocket : TSocket;
    FHwnd : THandle;
    FSockAddr, FSockAddrIn  : TSockAddrIn;
    FAddress : DWord;
    FHost : PHostent;
    FSocketNo : TSocket;
    FProtocol : PProtoEnt;
    FService : PServent;
    FReadCount, FWriteCount : Integer;
    FTransport : TTransport;
    FWnd, FTWnd: HWND;
    FMessage, FMsg, FTestMsg : string;
    FInterval, FEchoPortNo : Integer;
    FDone : Boolean;
    procedure StartAsyncSelect;
    procedure EchoEvent(var Mess : TMessage);
      message SOCK_EVENT;
    procedure TimerEvent(var Mess : TMessage);
      message WM_TIMER;
    procedure OnRecvEvent;
    procedure OnNewDataEvent;
    procedure OnProgressEvent;
    procedure OnDoneEvent;
    procedure SetUpAddress;
    procedure SetUpAddr;
    procedure GetHost;
    procedure SetPortNo(ReqdPort : Integer);
    function GetPortNo : Integer;
```

```
    function GetMessage : string;
    procedure SetMessage(ReqdMsg : string);
    function GetData : string;
    procedure SetData(DataReqd : string);
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
  public
    { Public declarations }
    procedure Start;
    procedure Stop;
    property Status : TConditions
      read FStatus write FStatus default Success;
    property Transport : TTransport
      read FTransport write FTransport default UDP;
    property Msg : string read FTestMsg write FTestMsg;
    property StatusMsg : string read FMsg write FMsg;
  published
    { Published declarations }
    property Done : Boolean
      read FDone write FDone default FALSE;
    property Interval : Integer
      read FInterval write FInterval default 1;
    property PortNo : Integer
      read FEchoPortNo write FEchoPortNo
      default IPPORT_ECHO;
    property HostName : string
      read FHostName write FHostName;
    property IPAddress : string read FHostIP write FHostIP;
    property NoEchoes : Word
      read FNoEchoes write FNoEchoes default 5;
    property OnRecv : TNotifyEvent
      read FOnRecv write FOnRecv;
    property OnNewData : TNotifyEvent
      read FOnNewData write FOnNewData;
    property OnProgress : TNotifyEvent
      read FOnProgress write FOnProgress;
    property OnDone : TNotifyEvent
      read FOnDone write FOnDone;
  end;
```

End Listing Four

Delphi / Object Pascal

By *Robert Vivrette*

Displaying Shortened Pathnames

... and Other Brief Treats

There are times when you may need to display a long pathname in a short space; for example, the Panel caption in Figure 1. This panel has been placed on the form with its *Align* property set to *alClient*. As a result, the space available for the caption changes when the form is resized. When the width of the form is reduced, the caption is truncated on the left and right.

Suppose, however, that you want to retain the right and left portions of the path, and eliminate characters from the middle. The Win32 API provides this capability with the *DrawTextEx* API call. We won't be using the function to actually draw text, but to modify the string we send.

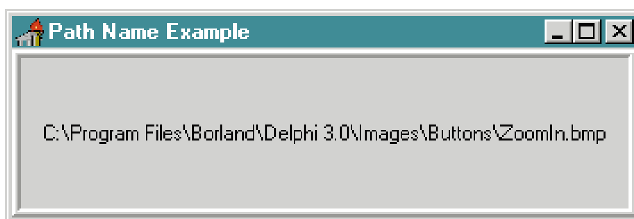


Figure 1: What will happen to the pathname when the form is resized?

```
procedure TForm1.FormResize(Sender: TObject);
var
  B : array[0..255] of Char;
  R : TRect;
begin
  StrCopy(B,'C:\Program Files\Borland\Delphi 3.0\' +
          'Images\Buttons\ZoomIn.bmp');
  R := ClientRect;
  InflateRect(R,-10,-10);
  DrawTextEx(Canvas.Handle,B,-1,R,
    DT_PATH_ELLIPSIS or DT_MODIFYSTRING or DT_CALCRECT,nil);
  Panel1.Caption := B;
end;
```

Figure 2: The pathname-shortening technique.

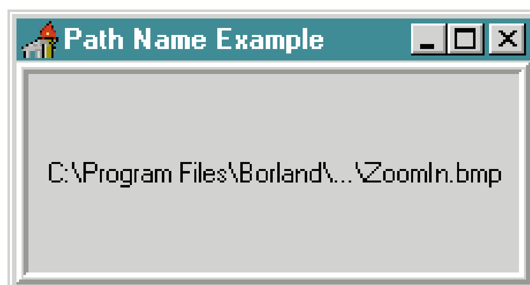


Figure 3: The shortened pathname.

The first few parameters of *DrawTextEx* are the Canvas' *Handle* (a Windows Device Context), the string to display, the length of that string (-1 calculates it for us), and the rectangle defining the text area. Then comes a combination of several flags, only three of which interest us here. The first is DT_PATH_ELLIPSIS, which eliminates some characters in the string and replaces them with three dots (an ellipsis). The second is DT_MODIFYSTRING, which modifies the passed string so that we get an altered string back after the function returns. The last is DT_CALCRECT, which is used to calculate the rectangle occupied by the text. We're not interested in this value, but DT_CALCRECT has the side effect of telling *DrawTextEx* not to draw the text. If we left out this flag, the path would be displayed on the Panel Canvas, separate from the Caption. The result? The function modifies the passed string, making it a shortened form of the original.

Figure 2 demonstrates the technique, and Figure 3 shows the result. The *InflateRect* call simply reduces the size of the rectangular area, so the text has a bit of a "margin." I use *StrCopy* to copy from a constant, so that we start with a fresh, undisturbed copy of the string each time. (Remember — it's modifying the return value.)

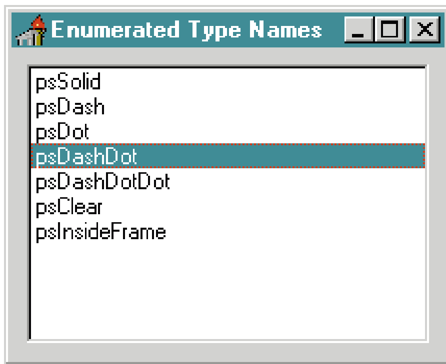


Figure 4: The element's string value is added to a list box.

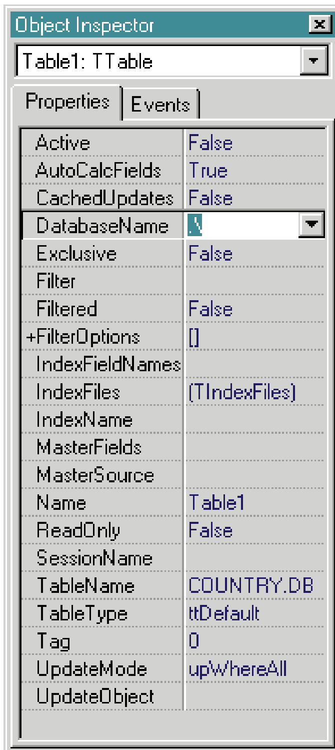


Figure 5: This syntax tells Delphi to look right under its nose.

Speeding *TList* Memory Allocation

Here's a tip for fans of the Delphi *TList* object. In looking through Delphi's online Help, you may have noticed that a *TList* has a *Capacity* property. This property isn't used to set the maximum size of the list, but to give Delphi an idea of how many items the list will contain.

As you may already know, when you add items to a *TList*, the list dynamically allocates memory for the new items. If you add an item to the list when there's no free space, Delphi requests a chunk of memory for several more items (between four and 16, depending on how many are currently in the list). But what if you need to add, say, 50,000 items? Delphi would request little chunks of memory over 12,500 times.

Instead, you can estimate how many items the list will have, then set its *Capacity* property accordingly. Afterward, if you didn't add as many as you thought, you can reset *Capacity* to reflect the real number. The result is that you will have one memory allocation at the start, none during the adding of the items, and one small deallocation afterward. If the starting allocation isn't sufficient, Delphi will generate a memory exception at the outset.

Look at the following pseudo-code to see how this might be done in practice:

```
ListOfBooks.Clear;
ListOfBooks.Capacity := 50000;
repeat
  blah ... blah ...
  ListOfBooks.Items.Add(TheNewBook);
  blah ... blah ...
until DoneAddingBooks;
ListOfBooks.Capacity := ListOfBooks.Count;
```

Enumerated Types

Have you ever created a set type and wanted to access the actual names of each element? Granted, it doesn't happen often, but it's nice to know how when you need to. For example, if you wanted to access the names of a *TPenStyle* set, you might be inclined to do something like this:

```
case ThePenStyle of
  psSolid       : ShowMessage('psSolid');
  psDash        : ShowMessage('psDash');
  psDot         : ShowMessage('psDot');
  psDashDot     : ShowMessage('psDashDot');
  psDashDotDot  : ShowMessage('psDashDotDot');
  psClear       : ShowMessage('psClear');
  psInsideFrame : ShowMessage('psInsideFrame');
end;
```

However, there's a better way. Delphi's RTTI (Run-Time Type Information) can obtain this information for you:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  a : Integer;
begin
  for a := Ord(Low(TPenStyle)) to Ord(High(TPenStyle)) do
    ListBox1.Items.Add(GetEnumName(TypeInfo(TPenStyle),a));
end;
```

This cycles through the elements of *TPenStyle*. You get the first and last elements with *Low* and *High*, respectively, then use *Ord* to get the element's ordinal number. Then we use *GetEnumName* to get the name of the enumerated type. *GetEnumName* wants the type information record for the type as the first parameter; we get this value by calling *TypeInfo*. The result is the string value of the element, and we add this value to a list box (see Figure 4).

Navigating in Close Quarters

Sometimes, database applications must access Paradox data files (*.DB) in the currently logged directory. The developer has several options, the best of which is to set up a database alias that gives access to the data file. A second option would be to enter a fully qualified path into the *Database* property. Of course the disadvantage is that if the application is moved to another machine, the database file might not reside in that location and the application wouldn't be able to find it.

Wouldn't it be nice if you could tell Delphi to look for the database in the same directory as the application regardless of where it is? Well, you can! All you need to do is put a period and a backslash in the *DatabaseName* property of a Table component (see Figure 5). When you access the *TableName* property, you'll see the database files in the currently logged directory. Even if you move the application elsewhere, it will still be able to find database files in its own directory. ∆

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.

By Bill Todd

AdHocery for Delphi

The Key to Seizing Query Control

Need to provide ad-hoc query capabilities for your users? AdHocery from Nevrona Designs not only makes providing such capability easy, it also provides control of the user interface. With AdHocery, your users can see only the data they want, either on screen or in reports.

Understanding AdHocery

The easiest way to understand AdHocery is to examine a sample program that uses it. Figure 1 shows a form with a Query, DataSource, and DBGrid connected to display the result set of a query. The SQL statement is:

```
SELECT CustNo, Company, City, State,
       Country, Addr1, Addr2
  FROM Customer
```

The goal of this project is to let the user select the records returned by the query, using any combination of Company, City, State, and Country. To provide the ad-hoc query capability, start by placing an AdHocSource component — one of the five AdHocery components — on the form. Next, set the AdHocQuery component's *BaseQuery* property to `Query1`, to connect it to the Query component on the form. Double-click **AdHocSource** to display its Fields Editor, as shown in Figure 2. Now right-click, select **Add Fields** from the menu, and add the Company, City, State, and Country fields.

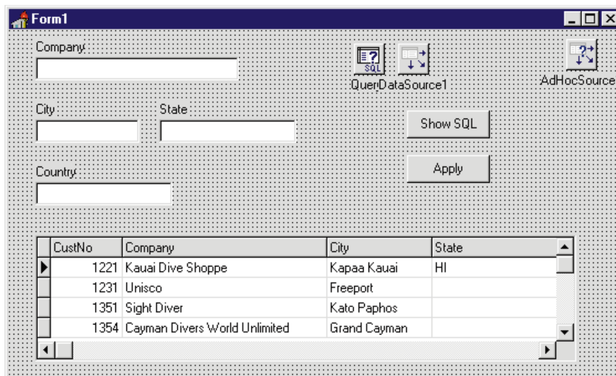


Figure 1: A sample ad-hoc query form.

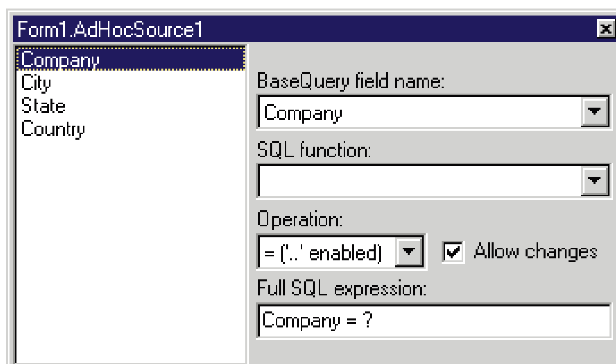


Figure 2: The AdHocSource Fields Editor.

Now drag the four fields from the Fields Editor window, and drop them onto your form. AdHocery automatically creates four DBEdit components with labels, then sets their *DataSource* properties to the AdHocSource component, and their *DataFields* property to their respective fields. The Fields Editor also lets you specify the comparison operator for each field. By default, the comparison operator is =, but you can specify other conditions, such as <, <=, >, or >=. You can also use the Fields Editor to add multiple instances of a field, so you can specify ranges in the query. To allow the user to enter a range for a numeric field, you would add the field twice in the Fields Editor — specifying >= as the comparison operator for one instance, and <= for the second. This allows you to drop two instances of the field on the form — one for the minimum value, and one for the maximum.

If the operation is:

```
= ('..' Enabled)
```

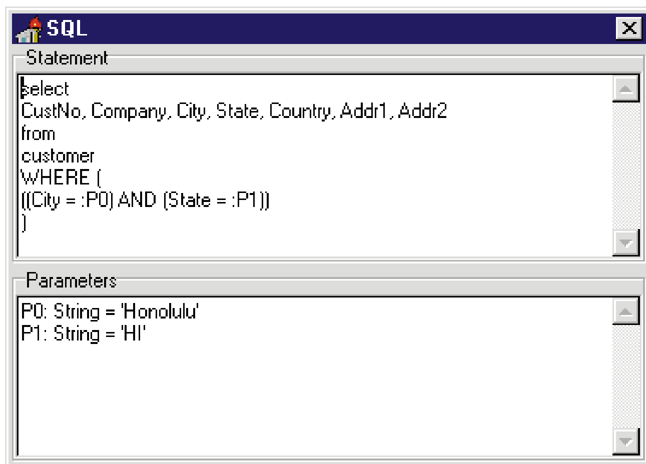


Figure 3: The AdHocery SQL dialog.

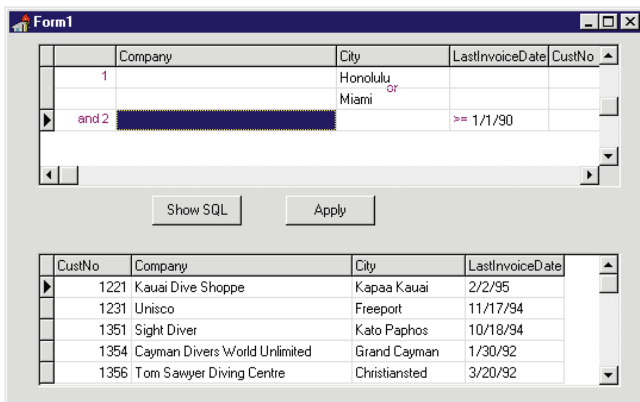


Figure 4: Operations between forms can be specified.

the user can specify a "starts with" value for that field by adding two periods to the end of the value. Behind the scenes, AdHocery changes the operator from = to `LIKE`, and replaces the .. with `%`. To give users the full power of the SQL LIKE operator, change the operation to `LIKE` in the Fields Editor. Now users can enter the SQL % wildcard anywhere in the search value.

Finally, drop two buttons on the form, and set their captions to `Show SQL` and `Apply`, respectively. Add the following lines of code to the button's *OnClick* event handlers:

```
// The Show SQL button's OnClick event handler.
AdHocSource1.ShowSQL;

// The Apply button's OnClick event handler.
AdHocSource1.ExecuteSQL;
```

If you compile and run this program, you can enter any combination of values in the DBEdit components, then click the **Apply** button, and see the result of your query in the DBGrid. To see the SQL generated by AdHocery, click the **Show SQL** button, and you'll see a display similar to Figure 3.

While this example provides basic query capability, it doesn't support logical AND and OR operations, or compound conditions other than "and-ing" between fields. To make more complex queries easy, AdHocery provides two other components: AdHocTreeView and AdHocGrid. The AdHocTreeView component works with the DBEdit components to allow you to build a complex query condition using AND, OR, and grouping (parentheses).

The AdHocGrid component also lets users build complex queries, as shown in Figure 4. The grid approach resembles query-by-example in many respects, and enables users to build complex queries in an intuitive format. A particularly nice feature of AdHocGrid is the ability to specify logical AND and OR operations between cells. This makes building lists, such as:

```
City = Honolulu OR Miami
```

(the condition in Figure 4), very easy for users. Another nice feature of the AdHocTreeView and AdHocGrid components is that you can change the text on the context menus that appear when the component is right-clicked, to suit your taste and the sophistication of your users.

AdHocery includes one more component, AdHocLookupGrid, which, like a multi-select list box, lets users easily specify a list of values to match in one field of a query. For example, you could present a list of states, and let users select only those that interest them. AdHocery also provides a powerful developer interface — via public methods and events — that lets you control every aspect of the query-generation process in code.

INFORMANT FACT FILE

Nevrona Designs' AdHocery 1.0 offers unsurpassed control of ad-hoc query potential, offering powerful interfaces for both the developer and the user.

Nevrona Designs
1930 S. Alma School Rd., Ste. B214
Mesa, AZ 85210-3041

Phone: (888) 776-4765; (602) 491-5492
Fax: (602) 530-4823
E-Mail: info@nevrona.com
Web Site: http://www.nevrona.com/designs
Price: US$149

Conclusion

AdHocery is a great tool. It gives you the ability to let the users of your programs construct complex and powerful queries easily, with a choice of user interface. This allows you to exploit query generation in a way most useful to the users of each program you write. AdHocery is the best ad-hoc query tool I've seen — it will certainly remain in my toolbox. ∆

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.

File | New

Directions / Commentary

Delphi on the Web

Software development need not be an isolated process. Macho programmers may continue to insist on coding every line themselves, but the rest of us can take advantage of the Web as a resource for components, tips, techniques, and other technical information. The Web offers a plethora of Delphi sites. This month, I've given "gold stars" to the top five Delphi-related sites on the Web.

Delphi Super Page ☆☆☆☆☆

Maintained by Robert Czerwinski, this is my favorite site to visit when looking for components and other Delphi resources. It has the largest library of VCL components I've seen, and can be quickly searched using a variety of criteria (e.g. 16-bit and/or 32-bit, freeware and/or shareware, etc.). Component descriptions, while not as extensive as *Torry's Delphi Pages* or *The Delphi Deli*, are more than adequate. Finally, Robert does a terrific job of updating the site regularly.

Torry's Delphi Pages ☆☆☆☆☆

Running neck and neck with the *Delphi Super Page*, this site is administered by Maxim Peresada and Victor Gvozdev. Not only is its library extensive, it also provides helpful comments — by a dog named Torry — about many of the components; awards the best components with a special "Torry's Top" award; and notes the most popular downloads. Another nice feature is a page that lets you know what's new on other Delphi sites.

The Delphi Deli ☆☆☆☆

If the two previous sites provide the "meat and potatoes" for Delphi developers (components), *The Delphi Deli*, maintained by Sylvia Lutnes, strives to provide a full menu. Yes, it offers a component library, but it also has information on Delphi mailing lists, book reviews, FAQs, a chat room, and more. This is a well-rounded site, particularly for those folks who want to do more than just download the latest VCL.

The Delphi EXchange ☆☆☆☆

Maintained by Brad Choate, this site offers a large collection of VCLs, as well as additional resources such as Delphi news and announcements, volunteer Delphi experts (DEXperts), and programming tips. *The Delphi EXchange* also features perhaps the most exhaustive list of Delphi-related Web links I've seen. When searching for components, you can take advantage of its powerful search engine, or drill down by category, using an outline view of its file library. My one "wish" for this site is better file descriptions, e.g. is a file freeware? Does it include source code? This shortcoming aside, *The Delphi EXchange* is a source I visit often.

The Delphi Information Connection

☆☆☆ This site, maintained by David and Susan Bernard, deserves special mention. It's an "all-around site," containing not only a wealth of components, but also links and other resources. Its design is notable, featuring a graphic Table of Contents in the form of Delphi's Object Inspector. Unfortunately, it looks as though the site hasn't been updated since late 1996; hopefully it will be revived soon, before it becomes outdated.

While each of these "gold star" sites deserve special merit, there are a vast array of other Delphi-related Web sites that also prove helpful to developers (see table below). You owe it to yourself to check 'em out. ∆

— Richard Wagner

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

(all URLs begin with http://)	Number of Components	Searching	File Descriptions	Additional Resources	Links to Other Delphi Sites
Delphi Super Page sunsite.icm.edu.pl/delphi/	Excellent	Excellent	Good	Fair	Very good
Torry's Delphi Pages carbohyd.siobc.ras.ru/torry/	Very good	Listing only	Excellent	Very good	Very good
The Delphi Deli www.intermid.com/delphi/	Good	Good	Excellent	Excellent	Good
The Delphi EXchange www.delphiexchange.com	Very good	Excellent	Fair	Good	Excellent
The Delphi Information Connection www.delphi32.com	Good	Very good	Very good	Very good	Good
The Delphi Temple simtel.coast.net/~jkeller/	Fair	Listing only	Good	Average	Good
The Delphi Companion www.xs4all.nl/~dgb/delphi.html	N/A	N/A	N/A	Very good	Very good
Delphi Source www.doit.com/delphi/	Good	N/A	Fair	Good	Very good
Delphi Station www.technosoftinc.com/delphi.shtml	Limited	Listing only	Fair	Good	Very good