October 1995 - Volume 1, Number 6

DB Under Construction

Getting Started with Delphi Databases

Cover Art By: *Michael Tanamachi*

ON THE COVER

FEATURES

REVIEWS

DEPARTMENTS

magazine MENU

Symposium

"It was on a visit to Delphi, as he said, that he had left our land; and he came home no more after he had once set forth."

— Sophocles, *Oedipus the King*

Or to paraphrase "Once you do Delphi, you can never go back." Which, oddly, reminds me of one of the early letters-to-the-editor I received regarding the Premier issue. And since I can't find the letter, I'll paraphrase again (and please pardon me for referring to the writer of the letter as "the reader"). The reader was positive for the most part (i.e. he liked the magazine), but did take umbrage at what he characterized as "Borland cheerleading," saying that it "cheapened the magazine."

Despite the fact that the letter did not live on in our files, the comments have lived on in my memory. I completely agree with the sentiment, and never want to be legitimately accused of "cheerleading" for Borland (or any other vendor for that matter). The reader didn't specify what got his goat, but my guess is that he was referring to our coverage of Delphi's debut at Software Development 95 in San Francisco. The news item described how people literally ran to the Borland booth and quoted some Borland employees' surprised reactions to the general melee/party atmosphere. It probably sounded like marketing hype, but there wasn't an ounce of exaggeration in the account. I've been to enough of these conferences to have some perspective, and believe me, the news was accurate. If anything, the piece failed to convey the genuine excitement with which Delphi was met at that event. It was a good day for Delphi and Borland, and we were just reporting the news. Honest.

There was a different type of excitement at the 6th Annual Borland Developers Conference, but it too was generated by Delphi. Different in that it wasn't the frenzy of SD 95; Delphi was then an essentially unknown quantity. The mood at the BDC was more studied — most of the attendees had been working with Delphi for some months. Instead, the feeling was a sort of thrilled satisfaction. Most developers (and I include myself) were, and are, simply delighted to be working with Delphi and involved with the product at such an early stage of its life cycle.

Which brings me to Delphi32. Yup, good as the Windows version is (let's call it Delphi16), the 32-bit version for Windows 95 is better. It's even faster, has many more features, and can turn your Delphi16 apps into their 32-bit equivalents with a simple compile.

But something's wrong — this doesn't feel right. We haven't had a decent opportunity to whine and complain about Delphi16, and a newer shinier model is already here. I can't think of another instance when a new version of the product came out *before* the consumers started clamoring for the next rev. Where's the fun in that?

Like Athena bursting full grown from Zeus' head, Delphi16 dazzled us with its leapfrog technology that blends the best visual 4GL tools with a cutting-edge compiler. Now, the Delphi R&D team stands prepared to repeat the trick for Windows 95. Thrilled as we may have been, by the end of the conference we were all taking it for granted that most Delphi sessions would feature Delphi32. And not just for quick peeks on well-planned excursions that avoided known weak spots in the software. Some of the presenters went on full-blown expeditions into the new 32-bit environment. And I never saw it crash.

And yes I'm a little red faced about the September "Component" issue. Well, not the whole issue, but a stupid error I made in arranging the articles. For some reason, I had locked onto the idea that Gary Entsminger's article was a "cover story." The only problem, of course, is that Gary's article doesn't describe a component. Now, as I write this in early September (before I receive the first "What the hell?" letter), please allow me to apologize for misplacing the article. It's a great article, but like the letter from "the reader", it was misfiled.

— Jerry Coffey, Editor-in-Chief
CIS: 70304,3633

table of CONTENTS

Delphi TOOLS

New Products and Solutions

InfoPower Upgrade

Woll2Woll Software has released an upgrade to *InfoPower*, a library of data-aware Delphi VCL components. In version 1.1, InfoPower enhanced several Delphi components including DataSource, Table, Query, DBGrid, and DBLookupCombo. In addition, InfoPower features a Table Index Selection combo box, Incremental Search combo box, Incremental Search dialog box, a Lookup dialog box, and more.

Free demonstration and trial versions are available in the Informant CompuServe forum in Library 14 (Filenames INFODEMO.ZIP and INFOTRI.ZIP).

For more information, contact Woll2Woll at (800) 965-2965; or e-mail 76207.2541@compuserve.com.

DFL Software Releases Light Lib Images VCL for Delphi

DFL Software of Toronto, ON has released ***Light Lib Images VCL for Delphi***, a native professional imaging VCL. It enables Delphi programmers to add document and image management capabilities to their applications.

Light Lib Images is the first imaging library to support the PNG format (public domain replacement for GIF), and support for this format is included in the Delphi VCL.

Light Lib Images supports all Windows printers and video resolutions, and features industry leading TWAIN scanner support. Scanning, displaying, saving, retrieving, printing, zooming, rotating, flipping, scaling, converting, gray-scaling, and dithering are fully supported for .BMP, .PCX, .TIF, .GIF, .JPG, and .PNG file formats. It also supports Huffman, RLE, LZW, CCITT 1D/Group 3 and 4 fax, and JPEG compression systems.

Price: US$149

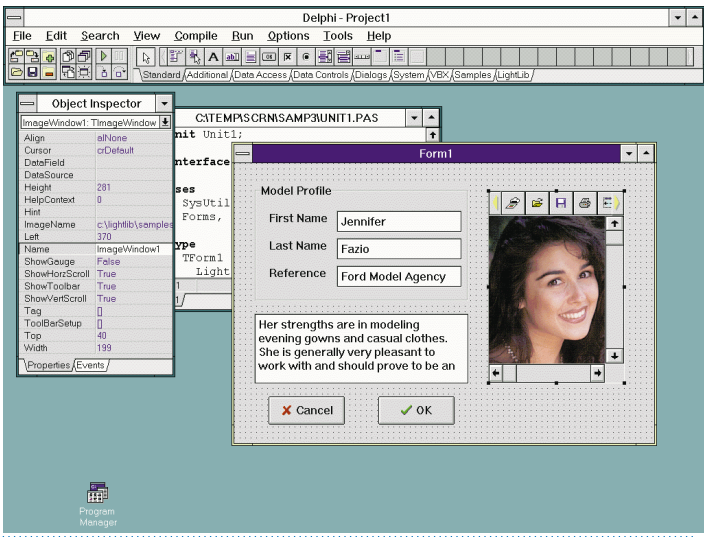
Contact: DFL Software Inc., 1712 Avenue Road, Box 54616, Toronto, ON, M5M 4N5, Canada

Phone: (416) 789-2223

Fax: (416) 789-0204

BBS: (416) 784-9712

E-Mail: CIS: 74723,3321



Borland and Brainstorm Technologies Ship Delphi/Link for Lotus Notes

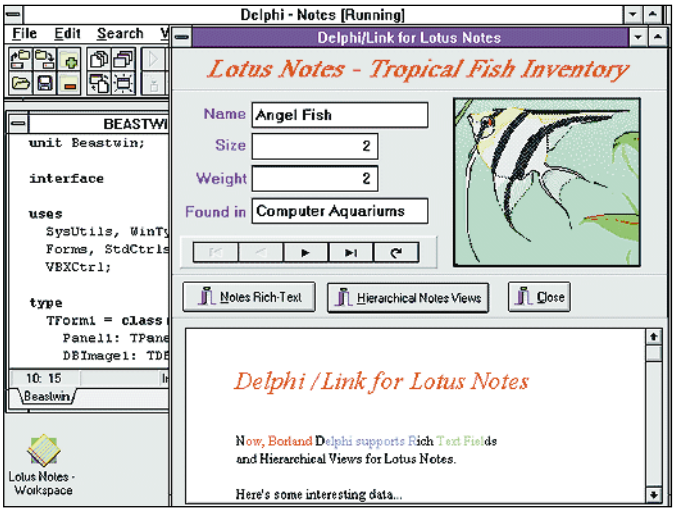
At its Sixth Annual Developers Conference, **Borland International Inc.** announced it has begun shipping ***Delphi/Link for Lotus Notes***, a tool developed by **Brainstorm Technologies**.

Delphi/Link is a set of customized native links to Lotus Notes for Delphi and Delphi Client/Server. It allows customers to use Delphi's object-oriented architecture while taking advantage of Notes replication for wide-area distribution of data. This tool also gives BC++ 4.5 and Visual dBASE developers the same capabilities.

According to Brainstorm Technologies, this product supports the move to client/server computing, and uses existing Borland tools and databases while leveraging the Notes installed base.

Delphi/Link gives you the ability create graphical front-ends to Notes databases, develop Notes client/server mail-enabled applications, integrate Notes data and applications with all major relational and SQL databases, and create, delete, read, and update Notes documents. It supports full-text searches on Notes documents, views, and databases. In addition, Delphi/Link allows developers to access, display, and manipulate Notes rich-text information, including file attachments, doc-links, and embedded/linked OLE objects.

Delphi/Link is one of several products marketed through Borland's newly formed Companion Product Group designed to provide additional products that extend across Borland's core development tools.



Price: US$399. All prices apply only in the US and Canada.

Contact: Borland International, 100 Borland Way, Scotts Valley, CA 95066-3249

Phone: (800) 932-9994

Web Site: http//:www.borland.com

Delphi TOOLS

New Products and Solutions

Delphi Training in Canada

As a standing Borland Connection Training Member, **InfoCan Management Consultants Group** (Canada) Inc. is the first in Canada to offer Delphi and dBASE hands-on training and consulting services nationwide. Both the three-day Introductory (US$1200) and the two-day Advanced (US$850) classes are being offered monthly in Victoria BC, Vancouver BC, Edmonton AB, Calgary AB, Montreal PQ, Quebec City PQ, Ottawa ON, Toronto ON, and Moncton NB. (Classes conducted in French are available in Montreal and Quebec.)

On-site training is also available upon request. InfoCan Management offers a wide variety of consultation services such as mentoring, programming, user interface design, system design, and project management. For more information or to register, contact InfoCan Management at (800) 715-5355; fax (604) 432-1799; or e-mail at 76307.720@compuserve.com.

Add Graphics to Delphi with ProEssentials

Gigasoft, Inc. of Keller, TX has released ***ProEssentials***, a set of three interactive controls — a Graph, Scientific Graph, and Pie Chart — for information-system, scientific, quality control, financial, and data-acquisition implementations.

ProEssentials is designed for adding a complete graphing sub-system via a 200-plus property interface. It features image construction with no overlapping labels independent of size, shape, style, or amount of data being graphed. Images can be prepared in memory so there are no flashing redraws.

ProEssentials has built-in zooming capabilities. After zooming, the user can horizontally pan to view the remaining data. The ProEssentials Graph object has an integrated table synchronized with the graph. As the user zooms or pans, the table's data moves to coincide with the graph.

Metafiles can be exported to the Clipboard, file, and printer via a built-in print dialog box, bitmaps can be exported to the Clipboard or file, textual data can be exported to the Clipboard or file via a built-in text export dialog box, and OLE representations can be exported to the Clipboard.

ProEssentials' built-in interface also provides an automatic and comprehensive hot-spot/drill-down mechanism. Possible hot-spots include subset and point labels, data points, and graph and table coordinate feedback.

ProEssentials ships with example projects for Delphi, Visual Basic, Visual C++, and FoxPro. Example code is also supplied for Clarion, Gupta, and PowerBuilder.

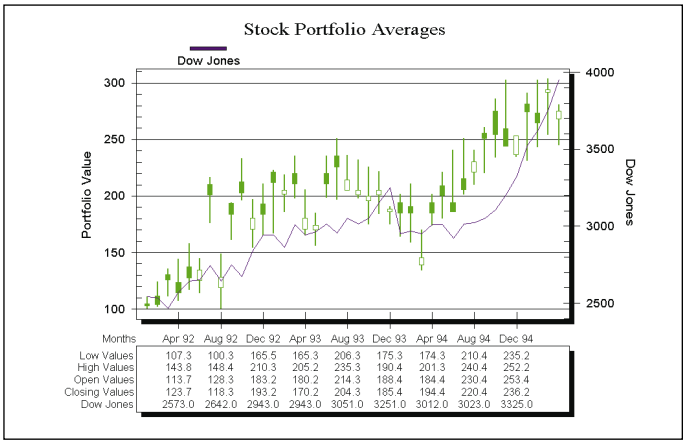
Price: US$249

Contact: Gigasoft, Inc., 696 Lantana Dr., Keller, TX 76248

Phone: (817) 431-8470

Fax: (817) 431-9860

E-Mail: CIS: 71561,315



ProtoView Visual Help Builder for Delphi

ProtoView Development Corp. of Cranbury, NJ has released ***ProtoView Visual Help Builder Version 2.11*** for Delphi, a RAD help development environment.

Unlike a help authoring system, ProtoView Visual Help Builder captures all forms, controls, and embedded menu items of a specified Delphi application. It then creates the corresponding topics, links, and search keywords for an on-line help system. The user is left with the sole task of text entry.

To enable text entry, ProtoView Visual Help Builder embeds itself within Microsoft Word 2.0 or 6.0. In addition, ProtoView Visual Help Builder contains a help authoring system that supports multimedia and visual logic design features.

ProtoView Help Builder also features the Help Eyes Technology. This feature allows the user to supply a help system to an existing application, even if the application source code is not available. The user can also build on-line help for any commercial application, including context-sensitive help for applications such as Microsoft Office, Lotus Notes, or any other Windows application.

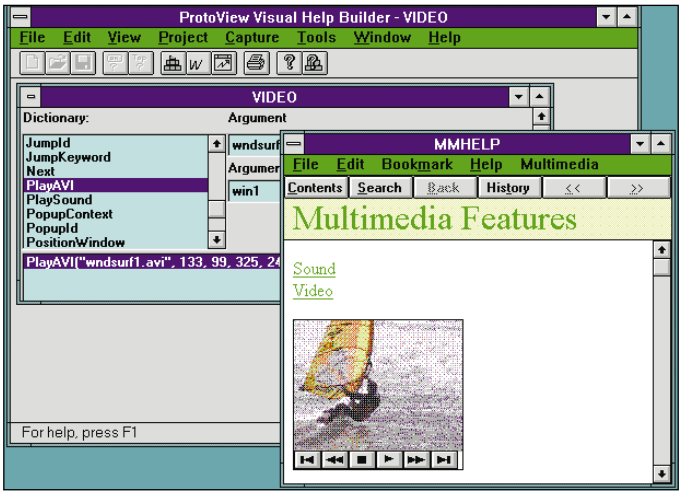
Help Builder works with Delphi, Visual Basic, C/C++, Pascal, PowerBuilder, and SQL Windows. It is compatible with Intersolv's PVCS Version Manager.

Price: US$395

Contact: ProtoView Development Corp., 2540 Route 130, Cranbury, NJ 08512

Phone: (800) 231-8588, or (609) 655-5000

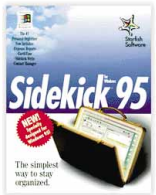
Fax: (609) 655-5353



Delphi TOOLS

New Products and Solutions

Starfish Ships Sidekick 95



Starfish Software has released *Sidekick 95*. This 32-bit version features an expense report view, a world-wide clock, a rich-text document editor, and more.

Sidekick 95 uses the TAPI (telephony applications standard) support available in Windows 95, and features a contact management and cardfile link with the telephone dialer. According to Starfish, Sidekick 95 was built for Windows 95 and Windows NT versions 3.51 and later (not just ported from Windows 3.1).

Sidekick 95 is available for download from the Internet Shopping Network (ISN), ZiffNet, CompuServe, and software.net. It sells for US$49.95 (prices may vary). An upgrade directly from Starfish Software runs US$39.95 (plus US$6 shipping and handling). To order call Starfish at 1-800-765-7839.

PowerTCP for Delphi Adds SMTP and POP3

Dart Communications of Cazenovia, NY has added support for the Post Office Protocol Version 3 (POP3) and Simple Mail Transfer Protocol (SMTP) to their ***PowerTCP Standard Toolkit for Delphi v1.3***. The first commercially available TCP/IP libraries for the Delphi environment, PowerTCP now features UUENCODE and MIME encoding/decoding. This allows developers to send and retrieve "attached documents" with minimal application programming.

The PowerTCP Standard Toolkit for Delphi includes Delphi components that support TCP (client/server), FTP, POP3, TELNET, SMTP, and a VT220 terminal emulator component. The PowerTCP Specialty Toolkit for Delphi includes Delphi components supporting UDP (client/server), TFTP (client/server), and SNMP. Each toolkit ships with sample applications with source code for each of the included components. The toolkits also include sample applications with source code for other TCP/IP protocols such as RLOGIN, REXEC, and RSH.

Price: The Standard Toolkit (TCP, TELNET, FTP, SMTP, POP3 and VT220 components) and Specialty Toolkit (UDP, TFTP and SNMP) are priced at US$598 each, including documentation. OEM and end-user run-time licenses are available.

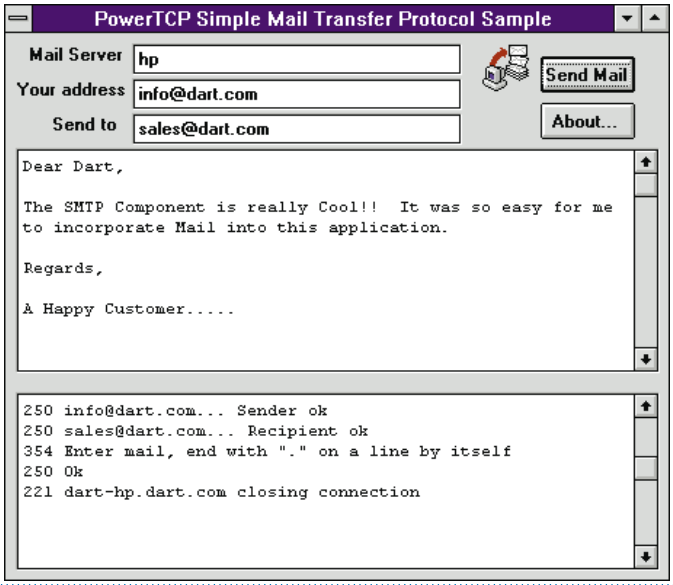
Contact: Dart Communications 61 Albany Street, PO Box 618, Cazenovia, NY 13035

Phone: (315) 655-1024

Fax: (315) 655-1025

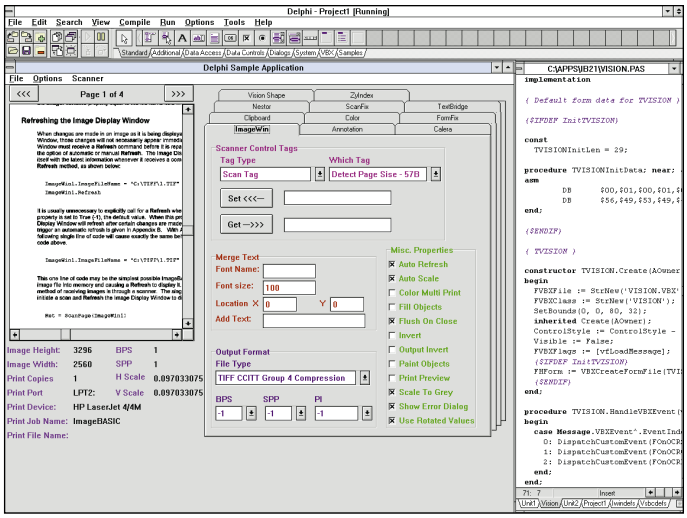
E-mail: info@dart.com

Web Site: http://www.dart.com



Diamond Head Software Launches ImageBASIC for Delphi

Diamond Head Software of Richardson, TX has launched ***Image BASIC for Delphi***, an integrated suite of VBXes for creating production-level document imaging applications.



With ImageBASIC for Delphi, end-users and professional developers can create production-level, customized document imaging applications. Because of ImageBASIC's underlying component architecture, developers can incorporate the imaging functionality into new or existing business applications.

The ImageBASIC document imaging components support a wide choice of OCR-based automatic image indexing options that further speed development time. These components can also be integrated with Lotus Notes, ActionWorkflow, and all leading business applications.

ImageBASIC's core system includes scan, display, and print functionality, and supports leading imaging engine vendors.

Delphi developers can also integrate additional components to support forms processing, color scanning and display, and image annotation.

Price: Standard edition US$1,750; additional modules begin at US$295 with additional per-seat licensing fees. Bundled systems are also available.

Contact: Diamond Head Software, Inc., 1217 Digital Drive, Suite 125, Richardson, TX 75081

Phone: (214) 479-9205

Fax: (214) 479-0219

table of CONTENTS

News LINE

October 1995

Borland Announces CodeGuard for Borland C++ 4.5

Borland International Inc. has announced CodeGuard for Borland C++ 4.5, a debugging tool that allows software developers to automatically locate and diagnose memory bugs in 16-bit Windows applications.

CodeGuard is designed to find bugs such as memory leaks, invalid memory references, memory overruns, and uninitialized data access. Unlike traditional interactive debuggers, CodeGuard automatically catches errors most likely to cause data corruption as they occur. With CodeGuard, developers can work directly within the Borland C++ integrated development environment, providing a single environment to detect, diagnose, investigate, and fix bugs.

CodeGuard for Borland C++ 4.5 is priced at US$149.95, with a special introductory price of US$99.95 available until Nov. 30, 1995. The product is shipped on CD-ROM; disks are available for an additional charge of US$19.95. All prices, including special offers, apply only in the US and Canada. For more information, call Borland at (800) 233-2444.

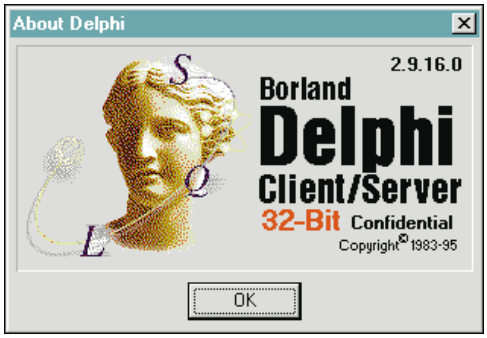
Delphi32 Previewed at Sixth Annual Borland Conference

San Diego, CA — Borland used their Sixth Annual Borland Developers Conference (August 6-9) to reveal the next version of Delphi. Called Delphi32, the new version is designed to operate on Windows 95. [Richard Wagner takes a *Delphi Informant* "First Look" at Delphi32 beginning on page 25.]

Delphi32 was previewed in two standing-room-only "Product Announcement" sessions and was met with an extremely enthusiastic response. The announcements themselves were presented by: Borland's Director of Product Marketing, Gillian Webster; Delphi Research and Development Manager, Gary Whizin; Delphi Product Manager, Zack Urlocker; and Delphi Chief Architect, Anders Hejlsberg.

Webster kicked things off by chronicling the phenomenal growth of Delphi third-party products. These products include more than 40 books; 6 newsletters and magazines; a "slew" of worldwide training seminars; a community of consultants; a large and rapidly-growing number of on-line groups; and many third-party tools and libraries.

Whizin discussed the primary goals of the Delphi product. These include: full operating system support, world-class code generation, extending the Delphi environment to provide all the tools and information necessary for development, and next-generation database support with a 32-bit BDE, live queries, new SQL Monitor utility, and Database Explorer.



Urlocker demonstrated the new components that add Win 95 support: PageControl, TreeView, ProgressBar, TrackBar, RichEdit, TabControl, UpDown, etc.

Hejlsberg then demonstrated the new compiler, stressing that it is "completely backward compatible". Unless a Delphi 1.0 program contains in-line assembler, performs 16-bit math, or makes Windows 3.1 API calls that aren't supported in the Windows 95 API, it can be simply recompiled in Delphi32 to make it a 32-bit Win 95 application.

Hejlsberg then moved on to the Delphi32 optimizing code generator, quipping that there are three keys to performance optimization on the Intel platform: "registers, registers, and registers". Among many register-related performance enhancements, the compiler performs lifetime analysis on local variables. If two local variables don't having overlapping scope they can be assigned to the same register.

Delphi32 also features new data types: a String type that can be up to 4GB in size; a Variant data type that can contain integer, Boolean, float, string, and OLE object values; and two wide character (Unicode) types.

Hejlsberg finished the presentation with a demonstration of Delphi32's ability to create OLE automation controllers and servers.

ICG Announces *Delphi Informant* on CD-ROM

Elk Grove, CA — Informant Communications Group, Inc. (ICG) has announced the company will release its second magazine, *Delphi Informant* on CD-ROM. Available in mid-December 1995, *Delphi Informant Works: 1995* will contain the text to all the articles appearing in 1995.

"We are very pleased to be able to offer *Delphi Informant* on CD-ROM" said Mitchell Koulouris, Publisher of *Paradox Informant* and *Delphi Informant*. "The demand for *Delphi Informant* has exceeded our supply of printed copies. With *Delphi Informant Works: 1995* we can make back issues that have been sold out for months available."

Delphi Informant Works features a new word index that allows users to find all the articles with specific keyword references or phrases.

The new CD-ROM will also contain all code and supporting files for every article. *Delphi Informant Works* will also contain demonstration versions of third-party tools for Delphi; an electronic version of *Delphi Power Tools*, a catalog of third-party tools and services available for Delphi; and a CompuServe starter kit with the latest version of WinCim and a US$15 usage credit for new members of the service.

Delphi Informant Works will sell for US$39.95 plus US$5 shipping and handling. The CD price is for a single-user license. For international customers the shipping charge is US$15. *Delphi Informant Works* will be updated annually.

ICG is now taking orders for *Delphi Informant Works: 1995*. To purchase the CD, call toll-free (800) 88-INFORM in the US. Outside the US, dial (916) 686-6610, or order by fax at (916) 686-8497.

News LINE

October 1995

Preferred Solutions' Monthly Delphi Component Contest

Preferred Solutions Ltd., a software consulting company specializing in object-oriented development of cross-platform applications, is running a monthly Delphi Component Contest to foster the development of Delphi components.

Each month Preferred Solutions invites Delphi application developers to enter suggestions for the perfect component. Entries are judged for originality and desirability, and published on the World Wide Web. Many excellent and creative ideas have already been submitted. Whether you are an application or component developer, you should try your hand at winning a prize. You'll find all the details at: http://www.topia.com.

Informant CompuServe Forum Opens Oracle® Sections

Informant Communications Group has opened new Oracle message and library sections on their CompuServe forum.

In Oracle Development (message section 13), members can discuss Oracle Workgroup/2000® issues. Also, Oracle Tools and Utilities (library section 16) has been added. This library is available for you to upload your favorite utilities, or view and download other Oracle Workgroup/2000-related items (i.e. code samples, mini applications, etc.).

To visit the Informant CompuServe forum, enter "GO ICGFORUM" at any command prompt.

BDC 1996: Wetsel Keynote Sets New Tone for Borland

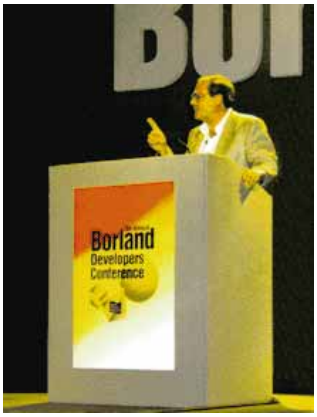
San Diego, CA — More than 2,000 people attended the 1995 Borland Developers Conference. It was highlighted by previews of the Windows 95 versions of Delphi and Paradox.

Another highlight was the Opening Keynote address delivered by Gary Wetsel, CEO of Borland. In it, Wetsel stated that he had taken over a "financially challenged" company, and that Borland had made mistakes with its pricing assumptions for Paradox and dBASE. Expectations were for approximately US$200 a unit. Instead the price was closer to US$100. According to Wetsel, there were two main reasons for the price difference: a long term industry trend of dropping software prices, and the impact of software suites.

Wetsel's view as an outsider coming in was that Borland had "good technology" but their "cost model was not in sync with revenues". In fact, the revenues were half of what was expected. He analyzed the company and found it had loaded cost structures stemming from the Ashton-Tate takeover. He also found that selling to two markets — developers and end-users — was expensive from a marketing standpoint. Borland needed to narrow its focus and concentrate on one type of customer — the developer.

Continuing with his bottom-line analysis, Wetsel said Borland has been moving away from an entrepreneurial model and rebuilding the company's infrastructure. This entailed some "tough times" — laying off approximately 650 employees and closing manufacturing facilities in Scotts Valley, CA and Dublin, Ireland. Borland has also greatly reduced marketing expenses and closed many of its international offices. The effect, according to Wetsel, has been to increase the gross margin to 85 percent (from somewhere in the low 70s). The changes to the infrastructure are expected to take nine months, ending in October. Borland will then reveal plans for the future — plans currently being assembled by their top 60 managers.

When asked "Did Delphi save Borland?" Wetsel replied that Delphi's success has "gone a long ways towards bridging the gap" during Borland's re-organization, but that Borland C++ was an important part of the



equation as well. He said that BC++ has a 50 percent market share in an extremely competitive arena, and that "dBASE and Paradox are also important". He also stated that Borland's strategy with InterBase has been unclear and the product was not very well known. Now that InterBase is bundled with Delphi, it's their "fastest growing product".

Wetsel then introduced Paul Gross, Borland's Vice President of Research and Development. Gross began by discussing Delphi, saying it was developed by 10 people in two and a half years and that it has sold approximately 125,000 units [as of early August]. He also stated that Windows 95 is a "tremendous opportunity" for Borland. Continuing in the frank vein of the entire presentation, Gross then gave a demonstration of Delphi32 and shared the results of benchmark tests that show Delphi is dramatically faster than Microsoft's Visual Basic and PowerSoft's PowerBuilder. In the Sieve benchmark for example, Delphi32 was 15 times faster than Visual Basic and more than 800 times faster than PowerBuilder.

In closing, Wetsel said: "Quite clearly, beyond a shadow of a doubt — Borland is back."

Borland Launches Windows 95 Web Site

Scotts Valley, CA — Borland International Inc. has launched its Smooth Sailing to Windows 95 World Wide Web Site in conjunction with Microsoft's introduction of Windows 95. The Smooth Sailing to Windows 95 Web Site offers developers a way to learn about Borland's Windows 95 products and how to smoothly migrate applications to the new operating system.

Customers can access the site either through the Microsoft Windows 95 Launch site (http//:www.windows.microsoft.com/launch95/), or Borland Online (http: //www.borland.com), Borland's World Wide Web Site.

In addition, customers visiting the site can win a cruise or other prizes including Borland T-shirts.

Borland's Windows 95 products are scheduled to ship in the third and fourth quarters of its fiscal year ending March 31, 1996.

table of CONTENTS

On the Cover

Delphi / SQL / Object Pascal

By C. Rand McKinney

Questions and Answers

An Introduction to Structured Query Language in Delphi

Structured Query Language (SQL) is widely known as a powerful language for creating and querying relational databases. Over the years, SQL has gained acceptance as a standard interface to relational database servers. All the major database server vendors, including Oracle, Sybase, Informix, etc., support versions of SQL. Although each server has a slightly different implementation, they all share a core of standard statements codified in standards such as ANSI SQL92.

Despite its name, SQL is:

- not a "structured" language, in the same sense as Pascal or other modern programming languages.
- not just for queries. SQL can be used for creating and modifying databases, as well as a multitude of other more specialized functions.
- not a stand-alone language. SQL statements must be embedded within other programming language code or applications to feature conditional logic (e.g. an **if** statement) or looping (e.g. a **while** loop).

Nevertheless, SQL can provide relational database applications with a great deal of power and portability, and is widely used for client/server applications.

While SQL is a complex subject, it need not be intimidating to the Delphi programmer. Delphi is both a powerful tool for developing SQL applications, and an ideal environment for programmers who want to learn SQL. This article is an introduction to SQL and shows you how to use it in Delphi. This article assumes you can build a basic database application with Delphi.

Delphi Client/Server enables you to build applications that access remote SQL servers across a network using Borland's SQL Links. However, even the desktop edition of Delphi provides substantial SQL capabilities with the Local InterBase Server (LIBS). LIBS is a 16-bit version of the Borland InterBase server. This unique feature enables you to create stand-alone desktop SQL applications without the need to access a remote server. Because LIBS comes with Delphi, this article will focus on its use. All the standard SQL syntax used will port (i.e. is applicable) to other SQL servers.

Why Use SQL?

Aside from a great buzzword to put on your resume, why would you want to use SQL in a Delphi application? If you've toyed with Delphi database applications, you know that *TTable* is a key data access component. Building database applications with *TTable* is relatively straightforward. Doesn't *TTable* provide all the database functionality you would ever need?

While the *TTable* component is fine for many purposes, using SQL with a *TQuery* enables you to:

- harness the power of database servers in client/server applications.
- search for records based on values in non-indexed fields.
- perform heterogeneous joins. That is, queries that include tables from different types of databases (e.g. a Sybase table on a remote server and a local Paradox table).
- perform complex queries that are not possible with *TTable*, including sub-selects and joins.

A SQL Overview

SQL can be divided into several distinct areas:

Data Definition Language (DDL): The statements that create, change, and delete databases, tables, and other relational objects (i.e. CREATE, ALTER, and DROP statements). You typically use DDL at the beginning of a project to define the database that the application will use. If the database already exists, then you need not be concerned about DDL.

Data Manipulation Language (DML): The statements to add, change, and delete data (i.e. INSERT, UPDATE, and DELETE statements), and to perform queries (i.e. the SELECT statement). These statements are the SQL workhorses that applications use to perform their tasks.

SQL also provides syntax for advanced features such as transaction control, granting access privileges, and executing stored procedures. These important features are often lumped with DML, but are not part of the core of the language. In general, Delphi applications will need to use DML statements, since most applications must retrieve and manipulate data and *not* define databases. For this reason, this article will concentrate on DML statements; defining databases is outside the scope of this article.

In SQL parlance, *metadata* are the data structures and procedural code that define a SQL database — so called because they define the structure of the database, and are therefore "data about the data" or meta-data. The metadata are sometimes also referred to as *database objects*, but should not be confused with Delphi objects.

When creating a database, you use DDL statements to define the metadata. Figure 1 lists the different types of objects that can be defined for an InterBase database (other types of databases may have slightly different types of objects).

Table	The fundamental SQL database object that defines the structure of a database. A table consists of columns (sometimes called fields). Each item of data in the table is known as a row (sometimes called a record). Each column has a name, a data type, and special constraints, including nullability (whether the column is required to have data), referential integrity (restrictions based on data in other tables), and check constraints (any general restrictions on the value of the data). The Delphi component *TTable* corresponds to a table.
View	A virtual table, based on the definitions of other tables.
Domain	A global column definition that can be used in defining tables; something like a user-defined data type.
Stored procedure	A self-contained program that can receive input parameters from, and return values to, applications. The Delphi component *TStoredProc* corresponds to a stored procedure.
Trigger	Conceptually similar to a Delphi event, a self-contained routine associated with a table or view that automatically performs an action when a row is inserted, updated, or deleted.
Index	Used to improve the speed of data retrieval. Identifies columns that can be used to retrieve a unique row and sort rows. Slightly different than the concept of an index in desktop databases like Paradox and dBASE.
Generator	A simple function that generates unique numbers (usually for use in an index).
Function	A user-defined function.
Filter	A BLOb filter.
Exception	Unrelated to Delphi exceptions.

Figure 1: InterBase database objects (metadata).

Introducing Windows Interactive SQL

If you are new to SQL, you can become familiar with it by using the Windows Interactive SQL program (WISQL.EXE) shown in Figure 2. This interactive SQL tool ships with Delphi, and enables you to enter SQL statements and then interactively see the results. Primarily intended as a DDL tool for creating and modifying InterBase databases, WISQL is also an excellent tool for learning SQL. Its extensive on-line help (file name WISQL.HLP) provides a complete reference

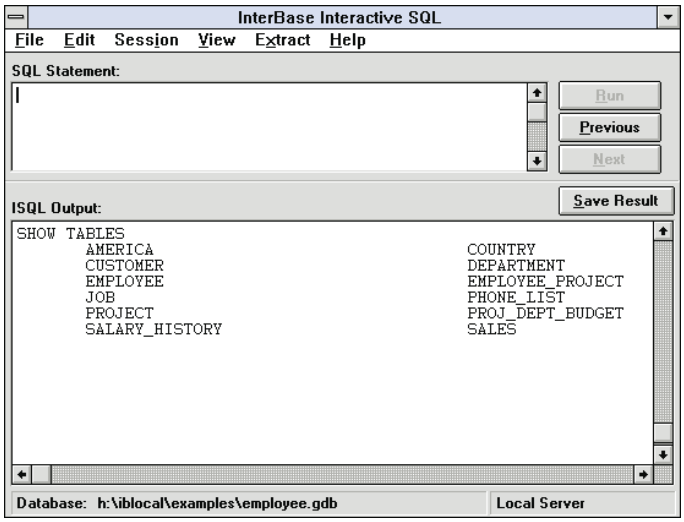


Figure 2: The Windows Interactive SQL program (WISQL.EXE) that ships with Delphi.

to SQL statements. The InterBase documentation that comes with Delphi also has a detailed SQL tutorial that uses Windows ISQL.

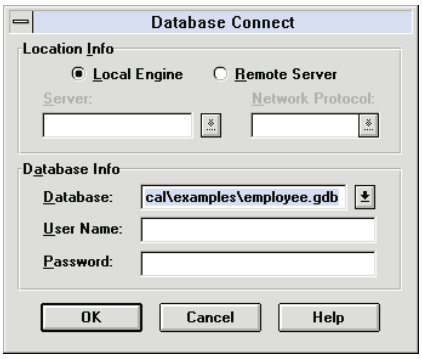


Figure 3: The Database Connect dialog box.

To start WISQL, double-click on the Windows ISQL icon in the Delphi program group. When using WISQL, you must first connect to a database by selecting **File | Connect to Database**. The Database Connect dialog box will be displayed (see Figure 3). Connecting to Local InterBase is easy. Just enter the location of a local database (e.g. C:\IBLOCAL\EXAMPLES\EMPLOYEE.GDB) in the **Database** edit box and click the **OK** button to connect.

To connect to databases on SQL servers, you are usually required to supply a user name and password to provide security. However, to connect to a local database (a LIBS database), you don't need to enter a password and user name and can leave these fields blank. Note however, that if you enter any user name, you are required to provide a password. Therefore, make sure the **User Name** and **Password** edit boxes are blank before selecting **OK** to connect to the sample database.

When WISQL is connected to a database, you'll see the path of the database file at the bottom of the window (again, see Figure 2). Every action you perform in WISQL then affects that database. Note that unlike desktop databases such as Paradox and dBASE, all the tables in a SQL database are contained in a single file. The EMPLOYEE.GDB file we're using as an example here, *is* the employee database. That is, it contains all the data and metadata of the sample database. There are no auxiliary files as for Paradox and dBASE.

WISQL's interface is simple. At the top of the window there is a small area for entering SQL statements, and below there is larger area that displays the statements' results. To type a statement in the **SQL Statement** area, click the mouse in that area. To execute the statement, click the **Run** button. To recall a previously entered statement, click on the **Previous** button.

The pull-down menu items enable you to perform additional actions, including: the ability to run SQL scripts created with an editor, extract all the DDL statements to define the database's metadata, and view all the metadata in the database.

Viewing Metadata

Use the **View** menu to display metadata information. For example, to see a list of tables in the database, choose **View | Metadata Information**. In the View Metadata dialog box, leave the **Object Name** edit box blank to display all the names of objects of that type. If you want detailed information on an object, simply fill in the name. For example, to see information about the EMPLOYEE table, select **Table** from the combo box and enter EMPLOYEE in the **Object Name** edit box, as shown in Figure 4. Then click on **OK**. You'll see the results in the **ISQL Output** area (see Figure 5).

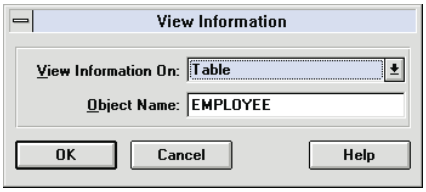


Figure 4: Selecting which metadata to display using the View Information dialog box.

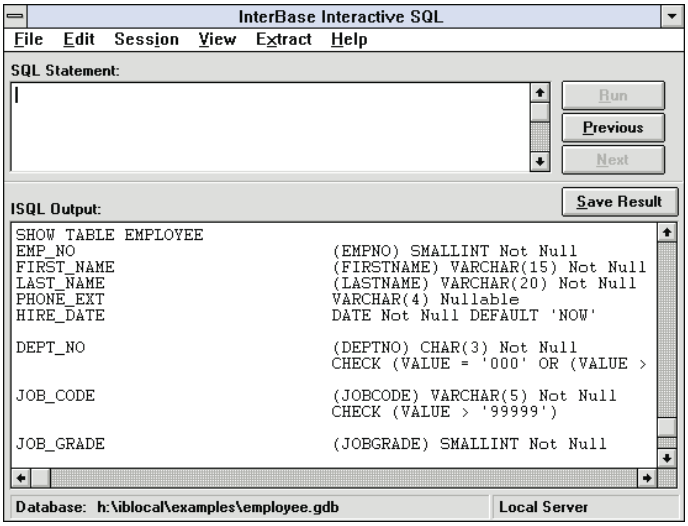


Figure 5: Detailed information about the EMPLOYEE table, the results of the View Information dialog box selections made in Figure 4.

The results list the name of each column in the table followed by its data type, and in parenthesis the domain from which it was derived (if any). A *domain* is like a user-defined data type for columns. The table in Figure 6 shows all the information provided. For example, the LASTNAME domain is defined as a VARCHAR(20), or up to twenty characters long. Some columns also have check constraints. These are conditions that the data entered in the column must meet. For example, JOB_GRADE must be an integer between 0 and 6.

In general, SQL tables can have very rich structures with many interconnecting conditions. Don't be too concerned about all the details right now. Just try to familiarize yourself with the structure of a SQL table. Try displaying some of the other tables in the database. If you're feeling adventurous, try displaying other objects such as domains or triggers.

Querying with the SELECT Statement

Now, you can start experimenting with SQL. A good place to start is probably the most well-known and powerful SQL statement: SELECT. SELECT is used to query a database,

Column	SQL Description
EMP_NO	(EMPNO) SMALLINT Not Null
FIRST_NAME	(FIRSTNAME) VARCHAR(15) Not Null
LAST_NAME	(LASTNAME) VARCHAR(20) Not Null
PHONE_EXT	VARCHAR(4) Nullable
HIRE_DATE	DATE Not Null DEFAULT 'NOW'
DEPT_NO	(DEPTNO) CHAR(3) Not Null. CHECK(VALUE = '000' OR (VALUE > '0' AND VALUE <= '999') OR VALUE IS NULL)
JOB_CODE	(JOBCODE) VARCHAR(5) Not Null. CHECK (VALUE > '99999')
JOB_GRADE	(JOBGRADE) SMALLINT Not Null. CHECK (VALUE BETWEEN 0 AND 6)
JOB_COUNTRY	(COUNTRYNAME) VARCHAR(15) Not Null
SALARY	(SALARY) NUMERIC(15, 2) Not Null DEFAULT 0

Figure 6: A detailed description of each of the columns (fields) in the EMPLOYEE table.

that is, to retrieve data. In the **SQL Statement** area, type:

```
SELECT * FROM EMPLOYEE
```

The asterisk (*) indicates that we want all the columns in the table. You'll see a display of all the records (usually called *rows* in SQL terminology) in the EMPLOYEE table. The output will scroll off the top of the **ISQL Output** area. You can scroll the output window up to see all the records retrieved by the query.

If you only want to retrieve some of the fields (or *columns* in SQL terminology), just provide a list of the column names instead of the asterisk. For example, to retrieve only the columns for the employee name and employee numbers, use the following statement:

```
SELECT LAST_NAME, FIRST_NAME, EMP_NO
FROM EMPLOYEE
```

The full syntax of the SELECT statement is complex, as you can see if you look it up in the on-line help under SQL Statement Reference. This complexity yields a great deal of power for querying with SQL. The main features of the SELECT statement can be summarized in the following syntax diagram:

SELECT [DISTINCT] *columns*
FROM *tables*
[WHERE *search_condition*]
[GROUP BY *columns* HAVING *search_condition*]
[ORDER BY *sort_order*]

In this syntax notation, keywords are capitalized, variables are italicized, and optional clauses are enclosed in brackets.

Use the optional keyword DISTINCT to eliminate duplicate rows in the query. The variable *columns* can be either an asterisk to retrieve all columns in the table, or a comma-delimited list of columns and aggregate functions. Aggregate functions include SUM, AVG, MIN, and MAX, that provide the total, average, minimum, and maximum (respectively) of all the rows retrieved. The special aggregate function COUNT returns the number of rows retrieved. For example, a query could select the average of employee salaries and the number of employee with the following statement:

```
SELECT AVG(salary), COUNT(emp_no)
FROM EMPLOYEE
```

The variable *tables* is a comma-delimited list of tables that can include joins between tables. Joins enable you to select columns based on comparisons between values in different tables. (A discussion of joins is beyond the scope of this article. Joins are powerful and are useful for advanced querying.)

The WHERE clause and the HAVING clause include a *search_condition* that can be any number of conditions linked by the Boolean operators AND and OR. The conditions can be any expression that equates to a Boolean value, including comparisons between column values, constants, using the standard arithmetic operators: equal to (=), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=).

For example, to display the employee records for all engineers hired on or after the beginning of 1994, enter:

```
SELECT * FROM EMPLOYEE
WHERE JOB_CODE = "Eng" AND
      HIRE_DATE >= "1-JAN-1994"
```

The *sort_order* is a list of column names, followed by either the keyword ASCENDING or DESCENDING. For example, the query could be displayed in ascending order of last names, and then by employee number by appending this to the SELECT statement:

```
ORDER BY last_name ASCENDING,
         emp_no    ASCENDING
```

Using Other DML Statements

The SELECT statement is great for retrieving data from the database tables, but how do you get the data into the tables? There are three other basic DML statements that enable you to add, modify, and delete data:

- INSERT adds a row (record) to a SQL table.
- UPDATE modifies existing rows.
- DELETE eliminates entire rows.

Figure 7 summarizes these DML statements. It is not as easy to experiment with these statements using the EMPLOYEE database because of the *referential integrity* rules that are defined for the database. Referential integrity is a weighty topic. In essence, it restricts changes to the database according to rules defined by the database designer. For example, if you try to delete a department that has employee information in it, you will get a "violation of for-

SQL Statement Syntax	Description	Example
INSERT INTO *table* (*col1*, *col2*, ...) VALUES (*val1*, *val2*, ...)	Inserts new rows into a table.	`INSERT INTO COUNTRY (COUNTRY,CURRENCY) VALUES ('Indonesia','Rupiah')`
UPDATE *table* SET *column* = *value* WHERE *condition*	Modifies values of existing rows.	`UPDATE COUNTRY SET CURRENCY = 'Rp' WHERE COUNTRY = 'Indonesia'`
DELETE FROM *table* WHERE *condition*	Removes rows from a table.	`DELETE FROM COUNTRY WHERE COUNTRY = 'Indonesia'`

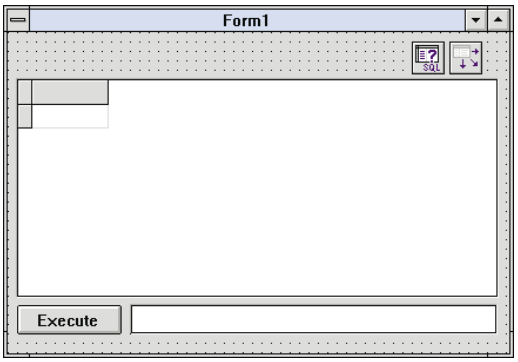


Figure 7 (Left): SQL Data Manipulation Language (DML) statements. **Figure 8 (Right):** Creating the sample application.

eign key constraint" message because the employee records refer to the department record.

To see how these statements are used, look at the INSERTS.SQL and UPDATES.SQL files located (by default) in the IBLOCAL\EXAMPLES\TUTORIAL directory. These files are used in the SQL tutorial and illustrate how to use INSERT and UPDATE statements. To experiment with one of these statements, cut it to the Windows Clipboard (using Ctrl X), paste it into WISQL (with Ctrl V), and then edit the statement appropriately before executing it.

Using SQL with Delphi

Before we discuss SQL in Delphi, you must first understand the two major ways a Delphi application can access a SQL database. The first uses the Borland Database Engine (BDE), the second uses passthrough SQL. In general, these two methods correspond to using the Table and Query components respectively. With just a few mouse clicks, you can create a simple working database application with a component. When you use a Table component, you're going through the BDE.

The other way to access a database is with a Query component. This requires you to use SQL syntax, which is then "passed through" to the SQL database engine. While the Table component is quick and simple to use, it has limited functionality. The Query component, on the other hand, enables you to access all the features of SQL.

Creating a Simple Interactive SQL Application

To understand how Delphi handles SQL, we'll create a simple application that will execute any SQL statement entered by the user. Essentially, this application will do the same thing that WISQL does, but on a rudimentary level.

Start a new project called SQLapp and place the following components on the main form:

- Edit
- Button
- Query
- DataSource
- DBGrid

Arrange the components so the form resembles Figure 8.

Set the Query component's *DatabaseName* property to `IBLOCAL`. This is the built-in alias for the Local InterBase EMPLOYEE database. Then connect the DataSource component to the Query component (i.e. set the DataSource's *DataSet* property to `Query1`), and connect the grid to the DataSource (i.e. set the grid's *DataSource* property to `DataSource1`).

Change the Button's *Caption* property to `Execute`, and its *Default* property to *True*. (This will enable the user to simply press Enter instead of clicking on the button — a nice convenience.) Add the following code to the Button's *OnClick* event:

```
Query1.Close;
Query1.SQL.Clear;
Query1.SQL.Add(Edit1.Text);
Query1.Open;
```

The *SQL* property of *TQuery* holds the text of the SQL statement to be executed by a Query component. One of the strengths of Delphi is that the SQL property can be set at run-time. The code shown above does the following:

- It closes the query, in case it was previously open. If was not open, there is no harm done. In general, it is good practice to do a *Close* before running any query, just to be safe.
- It clears the *SQL* property if it had some text remaining from a previous query.
- It adds the text in the *Edit1* component to the *SQL* property. Since *SQL* was just cleared, this becomes the entire text of the Query's SQL statement.
- It opens the Query — that is, attempts to execute the statement in the *SQL* property.

Compile and run the project. Then type the following code in the Edit component and click the **Execute** button:

```
SELECT * FROM EMPLOYEE
```

Before the application can execute a query, it first must connect to the EMPLOYEE database, just as you did when you used WISQL. Therefore, when the application runs the *Open* method, the Database Login dialog box will appear (see Figure 9). The user name `SYSDBA` (the default login name for the IBLOCAL alias), will be displayed. Either clear the **User**

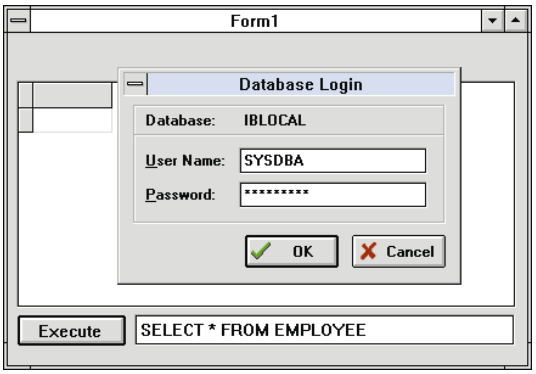


Figure 9: The sample application at run-time. Executing the SELECT statement triggers the display of the Database Login dialog box.

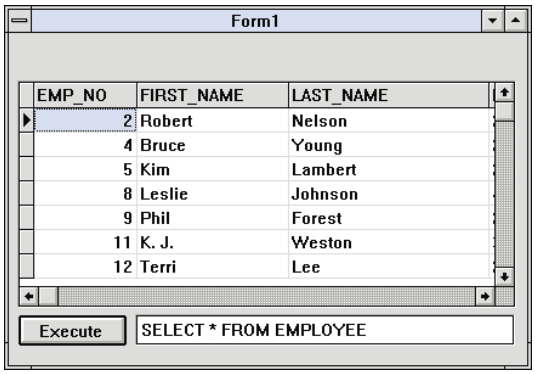


Figure 10: The sample application showing the result of a successful SELECT statement.

Name and **Password** fields, or enter the password `masterkey` (the default InterBase password for the `SYSDBA` account) and click on **OK**.

The results of the query will be displayed in the grid (see Figure 10). Try selecting from several different tables, such as DEPARTMENT or COUNTRY. You can scroll the records both vertically and horizontally in the grid. Notice that you cannot edit the records in the grid. This is because the query is not "live", but is a read-only query. By default, all queries in Delphi are read-only. (Making a query "live" is beyond the scope of this article. For now, we'll just use the grid to display records.)

The Trouble with *Open*

At this point, the application will generate an error if you enter any SQL statements other than SELECT. This is because Delphi expects the *Open* method to return a set of records from the database with a cursor. A *cursor* is essentially a pointer into the set of records retrieved. Since other SQL statements such as INSERT or DELETE do not return a set of records, the application will generate an error as it attempts to get the cursor back from the database.

To see this, try entering an INSERT statement in the edit field, for example:

```
INSERT INTO COUNTRY VALUES ('Finland','Markka')
```

When you click on the **Execute** button, you cause an error. If you are running the application from within the Delphi IDE, you will first see an exception raised (see Figure 11). After you click on **OK**, click the **Run** button in the Delphi

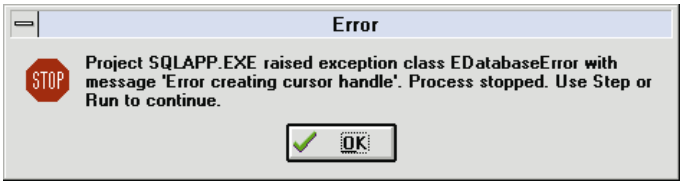


Figure 11: This error message is returned by Delphi when it does not receive a cursor handle as it expects to when the *Open* method is used to execute a SQL statement.

IDE to continue. Next, you'll see the error message passed from the local InterBase SQL engine: "Error creating cursor handle." By calling the *Open* method, you told the application to expect a cursor and a result set back from the database. However, an INSERT statement (or *any* statement except SELECT) does not return a result set.

To make things even more interesting, *Open* actually performs the statement you entered, even though the application complained. Enter `SELECT * FROM COUNTRY` again, and you'll see that the record (Finland, Markka) was actually inserted into the Country table. The application performed the action, but generated an error because it mistakenly thought it was going to receive a result set from the database.

Fortunately, *TQuery* has another method — *ExecSQL* — that executes a SQL statement and does not expect to get a result set and cursor handle back from the database. Thus, you want to use *Open* whenever executing a query and *ExecSQL* when executing any other statement.

But how can the application know what kind of statement you have entered? It could parse the text and look for the keyword SELECT, but that's not a very elegant solution. The best way is to use Delphi's exception-handling capabilities. Replace the call to the *Open* method with the following call:

```
try
  Query1.Open
except
  on E:EDatabaseError do
    if (E.Message <> 'Error creating cursor handle') then
      MessageDlg(E.Message,mtInformation,[mbOK],0);
end;
```

This code will **try** to call *Open*, and when it meets an exception, will handle it with the code in the **except** block that simply says to display a message dialog box with the exception message for any error but the "Error creating cursor handle" message. Since we know the *Open* method actually executes the statement, the net effect is that the application simply ignores the "Error creating cursor handle" message.

(Note: If you run the application from the Delphi IDE, you'll still get the IDE's notification of the exception, but not the database error message as before. If you compile the application and run it outside the IDE, you won't see any notification.) Run the application, and notice that you can enter any SQL statement, including: SELECT, INSERT, UPDATE, and DELETE.

The Database Component

By now, you are probably getting pretty tired of entering the password each time you want to run the application and connect to the database. The work-around to this is to add a Database component to the form, and set its properties as follows:

- *DriverName*: `INTRBASE`
- *DatabaseName*: `EmployeeDB`
- *LoginPrompt*: *False*

Now double-click on the Database component. The Database Properties Editor will open. Click on the **Defaults** button to load the default database login parameters based on the INTRBASE alias. All the parameters are displayed in the **Parameter overrides** box (see Figure 12).

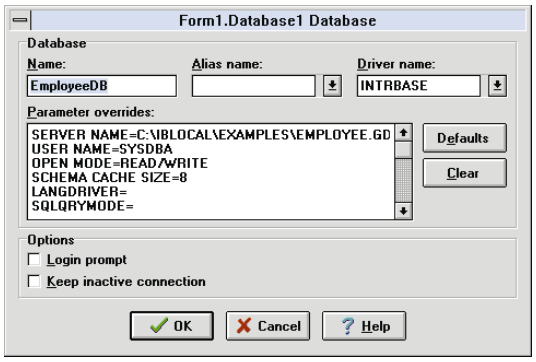


Figure 12: The Delphi Database Properties Editor is displayed by double-clicking on a Database component.

You're going to keep all the defaults. The only thing you need to add is the password. Scroll the edit box down to the parameter entry `PASSWORD=` and add `masterkey`. Now, because the *LoginPrompt* property is False, and you have entered the password here, you won't be bothered with the database login.

Now modify the *DatabaseName* property of the Query component to match that of the Database component (`EmployeeDB` in this case). Then run the application to confirm that you don't have to log into the database. Circumventing security is okay when you are developing an application. However, when you deploy an application, you will probably want to keep the login requirement to prevent unauthorized access to the database. To remove the password, simply click on the **Clear** button in the Database Properties Editor.

Using a *TDatabase* object also enables an application to maintain a persistent database connection. Starting a database connection incurs some overhead, even if the user name and password are automatically provided by the application. Without a Database component, whenever an application does not have any tables in a database open (i.e. no active *TTable* objects), Delphi automatically closes the database connection. Although that is not a problem for this little sample application, in practice, applications may be opening and closing tables right and left, and you will usually want to maintain a connection to reduce some of the connection overhead. (The Database component also enables you to provide your application with transaction control, another topic that is outside the scope of this article.)

Conclusion

You have now created a simple Delphi application that lets you enter any SQL statement against the InterBase EMPLOYEE database. It uses a Database component to avoid database logins and exception-handling to distinguish between SELECT statements and other statements.

Paradoxically, the simplicity of the application developed in this article displays some of the robust features of structured query language. In the next article of this series, we'll exploit more SQL features and discuss transactions, live versus read-only queries, and parameterized SQL statements. Δ

The SQL project demonstrated in this article is available on the Delphi Informant Works CD located in INFORM\95\OCT\RM9510.

C. Rand McKinney is a Senior Technical Writer for the Delphi team at Borland International. Previously, he helped to document the InterBase 4.0 Workgroup Server and client tools. He has also worked as an AI researcher and a space systems analyst. He can be reached at rand@borland.com.

On the Cover

Delphi / Object Pascal

By Cary Jensen, Ph.D

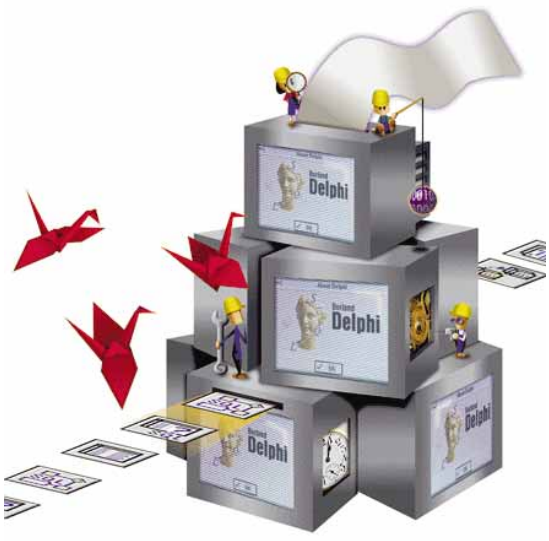
Tables Under Construction

Creating Tables at Run-Time

In most database applications, you will create tables before your application is deployed. To do this you typically use the Database Desktop, or a full-scale database, such as Paradox for Windows or Visual dBASE. There are times, however, when you have to create tables at run-time. This article demonstrates how to do this using Delphi's Table component

The Table component is not the only object you can use to create tables at run-time. For instance, you can use a Query component and execute a CREATE TABLE statement using Structured Query Language (SQL). Although this provides you with most of the capabilities you need in table creation, using the Query component is a subject for a full article. Another alternative is to create a table using the BatchMove component. BatchMove creates a new table when the destination is a table that does not yet exist. The drawback to this technique is that it only permits you to create a table based on an existing one.

In most cases, you will create a table by using the *CreateTable* method of a Table component. Using *CreateTable* requires that you first assign several properties of the Table component. At a minimum you must assign the name of the table that you want to create to the table's *TableName* property. You must also assign values to the *FieldDefs* property. If you are creating a table associated with an ODBC or SQL Link driver alias, you will also need to assign a value to the *DatabaseName* property. Finally, if the table type is not obvious from the *Filename* property, you should also assign a value to the *TableType* property.



The *FieldDefs* Property

The most complicated task in creating a table is defining its fields. To do this, you use the *FieldDefs* property.

The *FieldDefs* property defines a list of *TFieldDef* objects, each of which defines a field in the Table. When you open an existing table, the Table component automatically loads the *TFieldDef* objects based on information supplied by the BDE (Borland Database Engine). However, when you create a table, you must use code to create the *TFieldDef* objects.

Although there are several techniques you can use to set the *FieldDefs* property of a Table component, the easiest is the *Add* method of the *TFieldDef* object. *Add* has the following syntax:

```
procedure Add( const Name:  String;
                     DataType:  TFieldType;
                     Size:  Word;
                     Required:  Boolean );
```

The first argument, *Name*, defines the name of the field you are adding to the *FieldDefs* property. This name needs to conform to the naming conventions for the table type you are creating.

The second argument, *DataType*, is a value of the type *TFieldType*. The following are the available *TFieldType* values: *ftBoolean*, *ftBCD*, *ftBlob*, *ftBytes*, *ftCurrency*, *ftDate*, *ftDateTime*, *ftFloat*, *ftGraphic*, *ftInteger*, *ftMemo*, *ftSmallint*, *ftString*, *ftTime*, *ftUnknown*, *ftVarBytes*, and *ftWord*.

The third argument, *Size*, applies to only some of the *TFieldType* types. For *ftBCD*, use *Size* to identify the number of decimal places the field will support. For *ftBlob*, *ftBytes*, *ftGraphic*, *ftMemo*, *ftString*, and *ftVarBytes*, *Size* indicates how many bytes will be allocated within the primary table for storing the value. For all other field types, any non-negative integer value can be passed.

The final argument enables you to define whether the field you are adding will be a required field. To make the field required, pass the value *True*. Otherwise, pass the value *False*.

Tables on the Run

The following example demonstrates how to create a table at run-time. In Delphi, begin by creating a new form. Place the following components on the form: a DataSource, Table, DBGrid, and Button. When you are done, your form should look similar to that in Figure 1.

Next, set the DataSource's *DataSet* property to `Table1`. Set the DBGrid's *DataSource* property to `DataSource1`. Set the Button's *Caption* property to `Create Table`. Then, double-click the button to create an *OnClick* event handler and add the code shown in Figure 2.

Run the form. When you click on the button labeled **Create Table**, the table is created and activated. The table then appears in the DBGrid as shown in Figure 3.

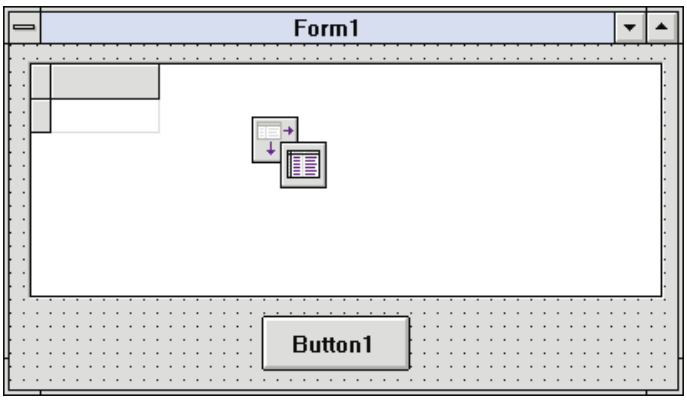


Figure 1: The basic form before adding the code in Figure 2.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.Active    := False;
  Table1.TableName := 'TEST.db';
  Table1.TableType := ttParadox;
  Table1.FieldDefs.Clear;
  Table1.FieldDefs.Add('FullName',ftString,20,True);
  Table1.FieldDefs.Add('DateOfBirth',ftDate,0,False);
  Table1.FieldDefs.Add('NumberOfChildren',ftInteger,
                       0,False);
  Table1.FieldDefs.Add('Salary',ftCurrency,0,False);
  Table1.CreateTable;
  Table1.Active := True;
end;
```

Figure 2: Add this code to the *OnClick* event of `Button1` shown in Figure 1.

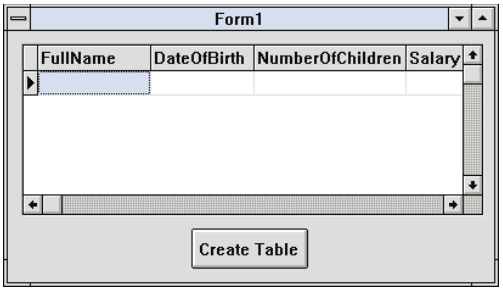


Figure 3: When you click the **Create Table** button, the new table will appear in the DBGrid.

Notice that in the code shown in Figure 3, the `Table1` object appears on every line. This is unnecessary. The code example in Figure 4 produces the same result, but is sometimes more efficient due to the Delphi compiler's pointer-load optimization.

Adding Indexes

Just as there are several ways to define the *TFieldDef* objects for the table you are creating, you have several viable options for adding indexes. The easiest to use is the *AddIndex* method of the *TTable* object. This method has the following syntax:

```
procedure AddIndex( const Name,
                    Fields: String;
                    Options: TIndexOptions );
```

The first parameter you pass is the name you want to assign to the index. (Remember that some table types place restrictions on the names you can assign to indexes.) The second argument is a list of fields that are included in the index. If the index is composed of more than one field, separate the field names with semicolons.

The final argument is a set of values that define the characteristics of the index. The valid values are: *ixPrimary*, *ixUnique*, *ixDescending*, *ixNonMaintained*, and *ixCaseInsensitive*. Remember that sets are enclosed in brackets. You can define two or more characteristics of the index by enclosing a comma separated list of *TIndexOptions* within the brackets.

The following is an example of a statement that will define an index for `Table1` created in the last example:

```
Table1.AddIndex('test',
                'fullname;dateofbirth',
                [ixPrimary,ixUnique]);
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
with Table1 do
  begin
    Active    := False;
    TableName := 'TEST.db';
    TableType := ttParadox;
    with FieldDefs do
      begin
        Clear;
        Add('FullName',ftString,20,True);
        Add('DateOfBirth',ftDate,0,False);
        Add('NumberOfChildren',ftInteger,0,False);
        Add('Salary',ftCurrency,0,False);
      end;
    CreateTable;
    Active := True;
  end;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  myTable: TTable;
begin
  myTable := TTable.Create(Form1);
  with myTable do
    begin
      TableName := 'demotab';
      TableType := ttParadox;
      FieldDefs.Clear;
      FieldDefs.Add('First',ftString,10,True);
      FieldDefs.Add('Second',ftString,10,False);
      AddIndex('mainindex','first',[ixPrimary,ixUnique]);
      CreateTable;
      Free;
    end;
end;
```

Figure 4 (Top): By using **with** statements, the code in this *OnClick* event is more efficient than that shown in Figure 2. It eliminates repeating `Table1` on each line (which allows the compiler to create more efficient code), and is easier to read. **Figure 5 (Bottom):** You can create the Table component at run-time by placing this code in the *ButtonClick* procedure.

Although the Table component was created as part of the form in the preceding example, this isn't necessary. You can also create the Table component itself at run-time. This is demonstrated in Figure 5. If you create a new form, place a single button on it, and then add this code to the button's *OnClick* event handler, a Paradox table named DemoTab.DB will be created when you click on the button.

Note however, that because you did not add a Table object to the form to begin with, Delphi did not automatically add the necessary support units to your unit's **uses** clause. You must do this manually. To make this code work, at a minimum you will need to add the `DBTables` and `DB` units to your **uses** clause. The following is an example of how this **uses** clause will appear when you're finished:

```
uses
  SysUtils, WinTypes, WinProcs, Messages,
  Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, DBTables, DB;
```

A Practical Example

Figure 6 shows the form for a project called CARDS.DPR. It is designed to act as a simple phone list. An application such as this typically allows a user to maintain multiple lists of phone numbers. Furthermore, it is also common for a user to be able to create a new list on demand. As shown in the menu in Figure 6, CARDS.DPR provides both of these capabilities.

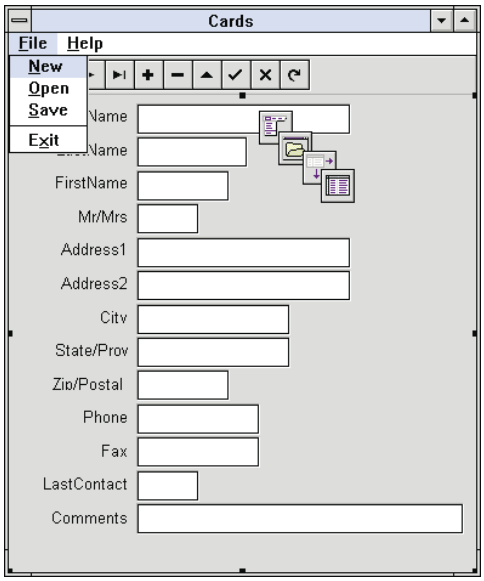


Figure 6: The **File** menu for the CARDS.DPR application.

While it is not practical to consider all the code associated with CARDS.DPR, there are several issues that apply to creating tables. The first issue is creating a table at run-time. This is produced in CARDS.DPR with the use of an event handler for the **File | New** menu selection, and a procedure named *CreateCardTable*. The event handler is shown in Figure 7. Figure 8 shows the code for *CreateCardTable*.

If you inspect the code for the **File | New** event handler, you will also notice that it makes calls to two additional procedures: *CheckState* and *ShiftMRU*. *CheckState* is called when the user closes the application or changes the current table. It tests to see if the current table has been edited, and asks the user if they want to save the changes:

```
procedure TForm1.New1Click(Sender: TObject);
begin
  OpenDialog1.Options :=[];
  if OpenDialog1.Execute then
    begin
      CheckState;
      if FileExists(OpenDialog1.Filename) then
        begin
          if MessageDlg('Replace ' + OpenDialog1.Filename,
                        mtConfirmation,[mbOK,mbCancel],
                        0) = mrCancel then
            Exit;
        end;
      CreateCardTable(Table1,OpenDialog1.Filename);
      Table1.Active := True;
      ShiftMRU(OpenDialog1.Filename);
      Self.Caption := 'Cards: ' + OpenDialog1.Filename;
    end;
end;
```

Figure 7: This event handler is associated with the **File | New** menu selection and calls the *CreateCardTable* procedure shown in Figure 8.

```
procedure TForm1.CreateCardTable(Tab: TTable;
                                 const tabName: String);
begin
  with Tab do
  begin
    Active    := False;
    TableName := tabName;
    with FieldDefs do
      begin
        Clear;
        Add('CompanyName',ftString,35,True);
        Add('LastName',ftString,18,False);
        Add('FirstName',ftString,15,False);
        Add('Mr/Mrs',ftString,10,False);
        Add('Address1',ftString,35,False);
        Add('Address2',ftString,35,False);
        Add('City',ftString,25,False);
        Add('State/Prov',ftString,25,False);
        Add('Zip/Postal Code',ftString,15,False);
        Add('Phone',ftString,20,False);
        Add('Fax',ftString,20,False);
        Add('LastContact',ftDate,0,False);
        Add('Comments',ftMemo,10,False);
      end;
    CreateTable;
    AddIndex('CompanyIndex','CompanyName',
             [ixPrimary,ixUnique]);
  end;
end;
```

Figure 8: The *CreateCardTable* procedure creates an indexed table at run-time.

```
procedure TForm1.CheckState;
begin
  if (Table1.State = dsEdit)   or
     (Table1.State = dsInsert) then
    if MessageDlg('Save changes to this record?',
                  mtConfirmation,
                  [mbOK,mbCancel],0) = mrOK then
      Table1.Post;
end;
```

ShiftMRU is a procedure that updates the most recently used (MRU) file list. Upon startup of the application, the most recently used files are loaded into the **File** menu from an .INI file. Each time a new file is opened or created, it is added to the MRU list, and the .INI file is updated. [For a demonstration of using .INI files with Delphi, see Douglas Horn's article "Initialization Rites" in the August 1995 *Delphi Informant*.]

The second interesting part of this application is the code that executes when a user opens an existing table. You must verify that this table conforms to the proper file structure. The code in Figure 9 executes when the user selects **File | Open**.

Notice that this code includes a call to a procedure named *ValidateTable*. If *ValidateTable* (see Figure 10) determines that the selected table does not have the proper structure, it raises an exception, which is handled by a **try...except** block in the **File | Open** event handler.

ValidateTable starts by creating a new, temporary table using the *CreateCardTable* procedure. The next step is to compare the number of *TFieldDef* definitions between the temporary table and the one being opened. If the tables have a different number

```
procedure TForm1.Open1Click(Sender: TObject);
var
  oldTable: String;
begin
  { Permit selection of existing tables only }
  OpenDialog1.Options := [ofFileMustExist];
  if OpenDialog1.Execute then
    begin
      CheckState;
      with Table1 do
        begin
          FieldDefs.Clear;
          oldTable := Tablename;
          if Active then
            Active := False;

          Tablename := OpenDialog1.FileName;

          try
            ValidateTable;
          except
            on EBadTable do
              begin
                Tablename := oldTable;
                Active    := True;
                Raise;
              end;
          end;

          Active := True;
          ShiftMRU(OpenDialog1.Filename);
          Self.Caption := 'Cards: '+OpenDialog1.Filename;
        end;
    end;
end;
```

```
procedure TForm1.ValidateTable;
var
  tempTable: TTable;
  i: Integer;
begin
  tempTable := TTable.Create(Form1);
  CreateCardTable(tempTable,'__temp.db');

  if Table1.FieldDefs.Count <>
     tempTable.FieldDefs.Count then
    begin
      tempTable.Free;
      raise EBadTable.Create('Invalid number of fields');
    end;

  for i := 1 to tempTable.FieldDefs.Count-1 do
    begin
      if (tempTable.FieldDefs.Items[i].Size <>
          Table1.FieldDefs.Items[i].Size)       or
         (tempTable.FieldDefs.Items[i].FieldClass <>
          Table1.FieldDefs.Items[i].FieldClass) or
         (tempTable.FieldDefs.Items[i].Name <>
          Table1.FieldDefs.Items[i].Name)       then
        begin
          tempTable.Free;
          raise EBadTable.Create('Invalid table structure');
        end;
    end;

  tempTable.Free;

end;
```

Figure 9 (Top): This event handler is associated with the **File | Open** menu selection and calls the *ValidateTable* procedure shown in Figure 10. **Figure 10 (Bottom):** The *ValidateTable* procedure will raise an exception if the table the user is attempting to open does not have the appropriate structure.

of *TFieldDef* definitions, an exception is raised. If the number of *TFieldDef* definitions match, *ValidateTable* then steps through each of the *TFieldDef* objects defined in the *FieldDefs* property, comparing their *Size*, *FieldClass*, and *Name* properties. Again, if any inconsistencies are found, an exception is raised. (This exception is a custom exception declared in the **implementation** section of this unit.)

Conclusion

Creating a table at run-time is an essential feature of many applications. This article has described one way to do this using a Table component. It also demonstrated the use of this technique in a practical application. Δ

The demonstration Delphi projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\95\OCT\CJ9510.

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

On the Cover

Delphi

By Antony Mosley

Database Apps: Part I

Getting Started: The Database Desktop and Application Expert

Are you new to Delphi? Don't worry — everybody is. Are you new to programming? Never fear — Delphi is the perfect tool for you to get started. You'll be surprised at the applications you can develop with only good drag-and-drop experience.

The application we'll build enables you to generate mailing labels and is composed of a simple database and report. First, we'll build a database table to store the names and addresses of a mailing list. Next, we'll use Delphi to organize the project and do a lot of the window and menu design work. Then we'll create a form for entering and editing names in the database. In the next installment of this series, we will use ReportSmith to create the labels and then put all the elements together in a compiled application. After completing this tutorial, you might want to experiment with various enhancements. This is good — tweak and twist the sample application as much as you like.

Create a Table

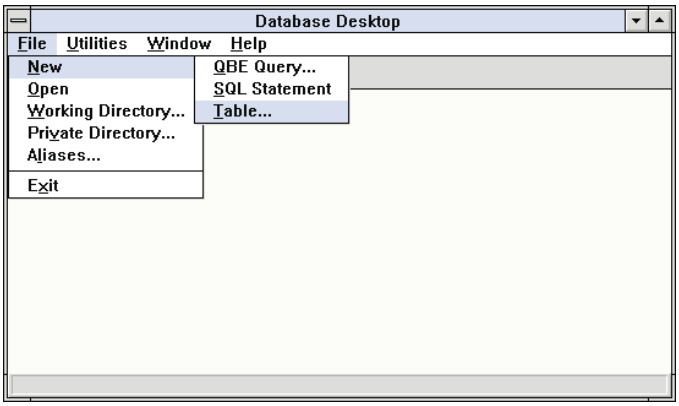
To begin, go to the Windows File Manager and create the directory C:\LABELS. This will be our working directory. Then start Delphi.

If you have worked with other database tools, this process should be a breeze. To build the table, we'll use the Database Desktop that ships with Delphi:

- From Delphi's menu select **Tools | Database Desktop** (or select the Database Desktop icon from the Delphi program group in Windows).
- Select **File | New | Table** from the Database Desktop menu (see Figure 1). You will be prompted to select a table type. Select **dBASE for Windows** from the drop-down list, then click **OK**.
- The Create dBASE Table dialog box will be displayed. Begin typing at the first line under **Field Name**, and press Tab to move to the next field. Enter the database structure as shown in Figure 2.
- Click the **Save As** button to display the Save As dialog box. Enter `C:\LABELS\LABELS.DBF` for the file name, then click **OK**.

Create an Alias

Next we'll create an alias for the \LABELS directory. Among other things, an alias is a shortcut to a database. [For a complete introduction to aliases and their use with Delphi, refer to Cary Jensen's article "A Programmer's Compass" in the Premiere issue of *Delphi Informant*.] As you work with Delphi, you'll find that accessing tables by an alias — rather than a hard-wired directory path — provides far more flexibility.



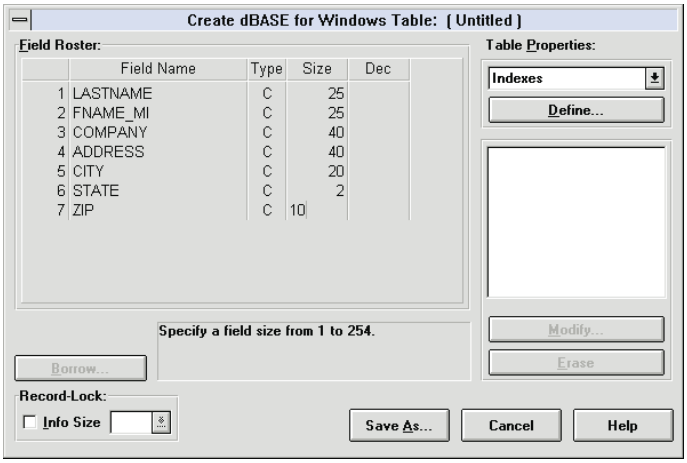


Figure 1 (Top): The Database Desktop handles table creation, queries, and other database-specific tasks. **Figure 2 (Bottom):** Using the Create dBASE Table dialog box to create the table for our sample application.

Here are the steps to create the alias:

- From the Database Desktop, select **File | Aliases** to display the Alias Manager dialog box.
- Click on the **New** button, and enter `Labels` for the **Database Alias**.
- Accept **Standard** (the default) for the **Driver Type**, then change the **Path** to `C:\LABELS`. (You can also click the **Browse** button to select the directory using a browser.) The dialog box should look as it does in Figure 3. Click the **OK** button. You will be prompted to save the alias in the IDAPI.CFG file. Click the **OK** button.

Now let's make `C:\LABELS` the working directory. This will keep us from having to use the browser or enter the complete path name whenever we need a table from the \LABELS directory. To do this, select **File | Working Directory** to display the Set Working Directory dialog box. Then select **Labels** from the **Aliases** drop-down list and click on **OK**.

Enter Some Data

Now take a few minutes to enter about five names so we'll have some live data to work with while designing the sample application. To do this, select **File | Open | Table | LABELS.DBF** to open your table. Then, press F9 or select **View | Edit Data** from the menu to enter Edit mode. Use your imagination (or your phone book) and enter five names and addresses to use

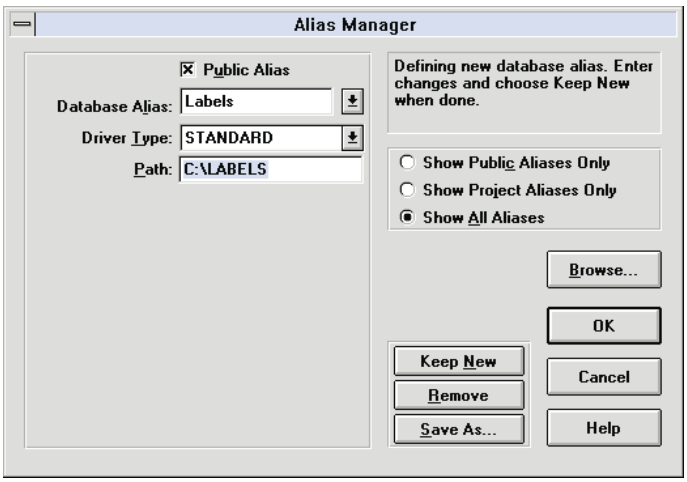


Figure 3: Creating an alias at the Alias Manager dialog box.

for your application. When you're finished, close the Database Desktop window by double-clicking its control box (or by pressing Alt F4). Your changes will be saved automatically.

Configuring Delphi

Delphi is capable of generating many of an application's elements, so we might as well take advantage of this feature. Let's do this by configuring Delphi to generate much of the sample project for us. (A Delphi *project* holds all the files for a complete application.)

From Delphi's main menu, select **Options | Gallery**. The Gallery Options dialog box will be displayed. Select the Form Templates tab (see Figure 4) and click on the Blank Form icon. Then click on the **Default Main Form** button. This will cause Delphi to automatically begin any new project with a blank form as the main form.

Now click on the Project Experts tab, and select the Application Expert icon. Unless you have installed others, it will be the only one available. Click on the **Default Project** button to tell Delphi that you want expert assistance while developing your application. Now Delphi will guide you through most of the design work for a new application. (It's important to keep in mind that every process that Delphi automates can also be performed man-

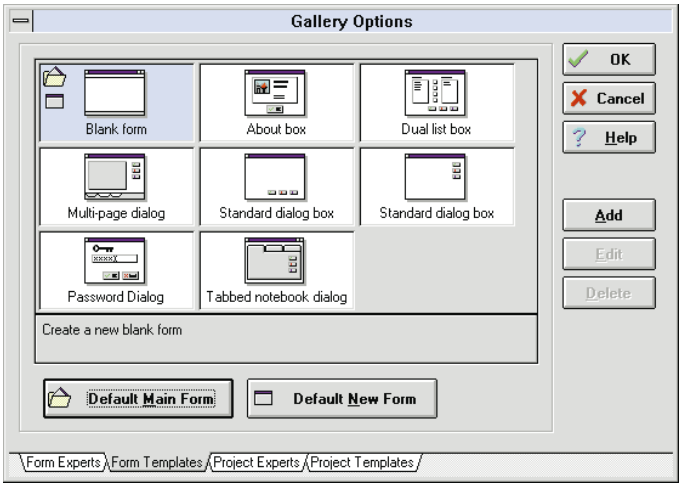


Figure 4: The Form Templates page of the Gallery Options dialog box allows you to select a form template as the default for a main form.

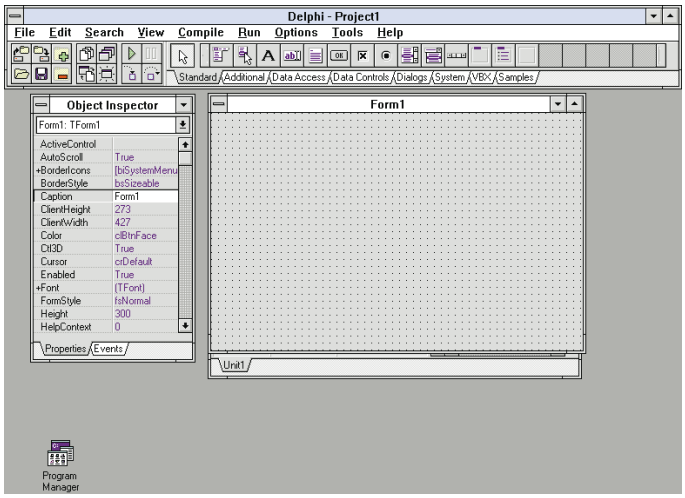


Figure 5: Delphi as it appears at start up by default.

ually. When you begin using Delphi regularly you can decide which features you would like to have automated and which features you would like to handle manually. Who knows — one day you may even decide to code an entire project.)

Click the **OK** button to accept the new Gallery options. Delphi will probably appear as it does in Figure 5. If there is another project open, you may want to close or save it. In any case, Delphi will close any loaded project automatically when you start a new one.

The Application Expert

Now let's put the new Gallery options to use. Select **File | New Project** from the menu. The first screen of the Application Expert will be displayed (see Figure 6). The choices you make at this screen determine if the application you're creating will have a menu bar, and if so, which of up to four standard Windows menu options it will contain. Select all the standard Windows menus: **File Menu**, **Edit Menu**, **Window Menu**, and **Help Menu** (as shown in Figure 6).

Then press the **Next** button to advance to the next screen of the Application Expert (see Figure 7). In this step you specify the file types to be used by the application's File Open and File Save dialog boxes. (Note that this screen of the Application Expert would not be displayed if the **File Menu** option had not been checked on the preceding screen.) Click on the **Add** button to display the Extension Filter dialog box (again, see Figure 7). Enter `Form` as the **Description** and `.DFM` as the **Extension**. Click **OK** and then click the **Add** button again. This time enter `Labels` as the **Description** and `.RPT` as the **Extension**, and click **OK**. The new application will now filter out all files except for .DFM and .RPT files when it presents the File Open or File Save dialog box to the user. Press the **Next** button to continue.

The next screen of the Application Expert allows you to define a speedbar for your application (see Figure 8). Notice that the menu items you selected at the first screen of the Application Expert are listed in the **Menus** list box. Each menu item command has an associated speed button listed in the **Available commands** list box.

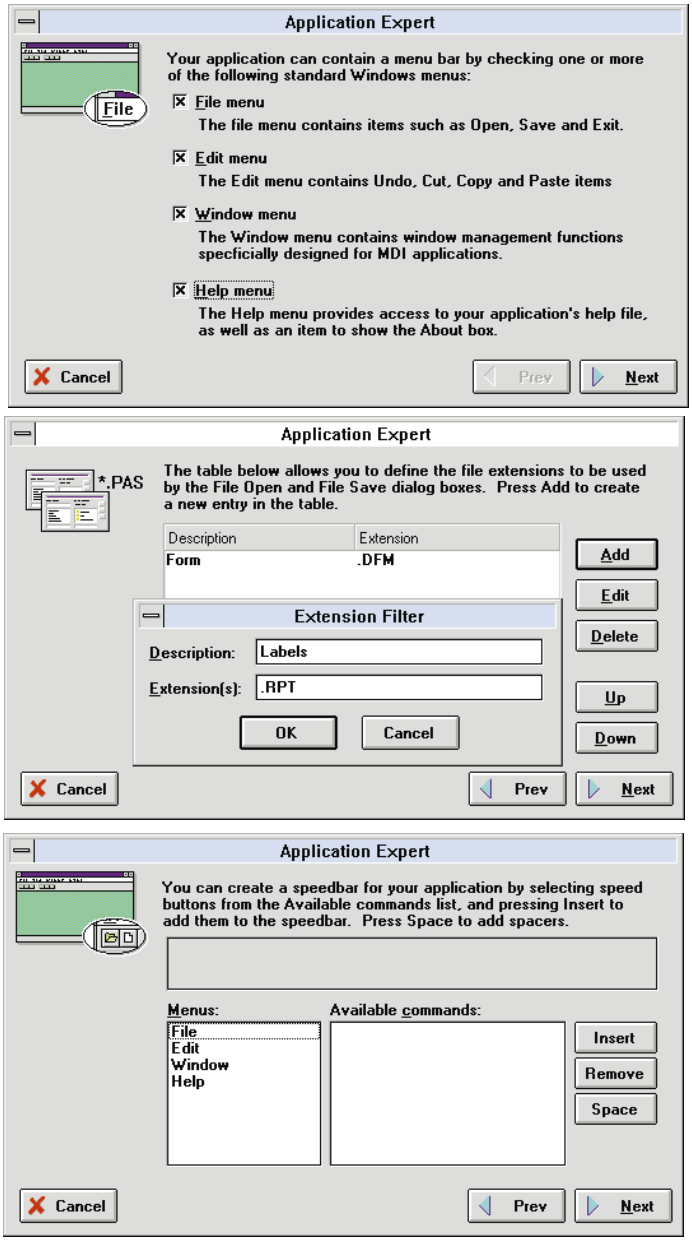


Figure 6 (Top): The first screen of the Application Expert determines which menu options will appear on the new application. **Figure 7 (Middle):** The second screen of the Application Expert (shown here with its Extension Filter dialog box displayed) determines the file types that will be recognized by the new application's File Open and File Save dialog boxes. **Figure 8 (Bottom):** The third screen of the Application Expert allows you to create a speedbar for the new application.

Let's start by selecting the **File** menu. From the **Available commands** list box, select the **New** menu item and click the **Insert** button. The "New" icon will appear in the window above the two list boxes. Click on the **Space** button to add a space after this speedbar item (or simply press Spacebar). To quicken the pace you can now add the **Open**, **Save**, **Print**, and **Exit** icons to your speedbar by double-clicking each one. Now add a space between each icon by clicking on each one and clicking the **Space** button.

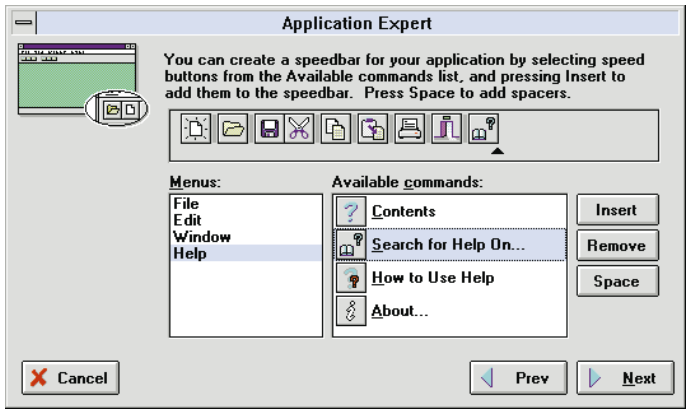
Next, let's add the **Edit** menu items to the speedbar. These should precede the **Print** icon. Click on the **Print** icon on the

speedbar to move the pointer there. In the **Menus** list box, select **Edit** and then add the **Cut**, **Copy**, and **Paste** icons from the **Available commands** list box.

Now let's add a **Help** icon. Place the pointer after the **Exit** icon on the speedbar and select **Help** from the **Menus** list box. Now double-click on the **Search for Help On** icon. When you're done, the screen should be like Figure 9. Press the **Next** button. The last dialog box of the Application Expert prompts you for basic information about the new application (see Figure 10). Enter `LabelPro` as the application's name. Then enter `C:\LABELS` to indicate the directory in which you'll store the application. This is the directory that we created earlier, and that contains the table file, LABELS.DBF. If this directory did not exist, the Application Expert would create it for you.

In the **Options** group box, select **Create a status line** and **Enable hints**. This will direct the Application Expert to place a Status bar at the bottom of the main form, and to display help hints for the various components and menu items. Finally, press the **Create** button.

You have just created a Windows application (see Figure 11), and Delphi has done most of the work for you. The main form is (appropriately) titled MainForm, and contains the menu and speedbar specified using the Application Expert. There are also five components floating about on the form below the speedbar (which we'll discuss shortly). You should also take a look at the window behind the form by clicking on the tab labeled



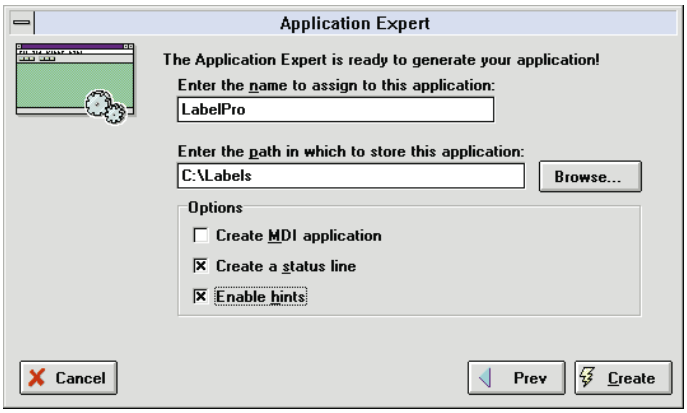


Figure 9 (Top): The completed speedbar screen for our sample application. **Figure 10 (Bottom):** The final screen of the Delphi Application Expert.

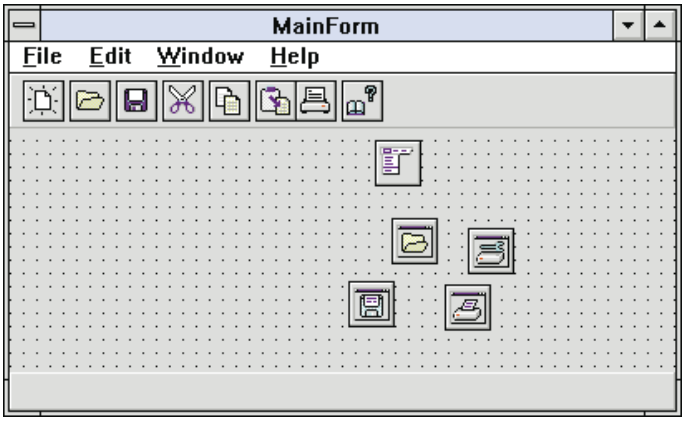


Figure 11: The sample application at design time.

"Main". This window displays the Object Pascal code that Delphi generated to create the form.

The Object Inspector

The Object Inspector is to the left of the main form (see Figure 12). This special dialog box allows you to manipulate the properties of the form and every component on the form. It consists of two tabbed pages: Properties and Events. By default, the Properties page appears on top when you start Delphi. [For a comprehensive introduction to the Object Inspector, see Douglas Horn's article "Delphi: A Visual Tour" in the Premiere issue of *Delphi Informant*.] If you click on a different object on the form, its properties will appear in the Object Inspector's Value column, and the name of the object will appear in the Object Selector. For example, select the Menu component to display its properties in the Object Inspector.

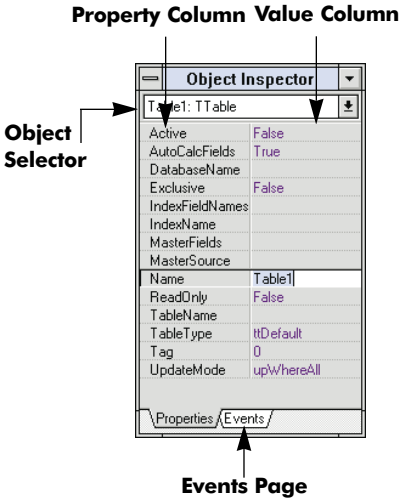


Figure 12: The Object Inspector allows you to modify the properties of the form and every component it contains.

With the Menu component still selected, right-click on it and a pop-up menu with more options will appear. If you select **Menu Designer**, for example, you will enable menu editing features. You can also right-click on the other components on the form to access design features specific to those components.

The five components arranged haphazardly below the speedbar on the form are *non-visual* components. That is, these components will not appear at run-time. However, they *are* visible during design time so their properties can be modified.

Let's use the Object Inspector to change the form's name from `MainForm` to `LabelPro`. Click on a blank area on the form to select it. The form's properties will appear in the Object Inspector. Select the *Caption* property and enter `LabelPro` in the Value column as the form's new name. You'll see the name of the

form change as you type it. To test the application, select **Run | Run** from the Delphi menu (or press F9). The form appears as it will for your users, with an active menu and working speedbar buttons (see Figure 13). Try using some of the menu items by selecting them with the mouse. Resize the window to full screen if you'd like. After you've experimented with your form, close it by clicking on the Exit button on the speedbar. You will immediately return to design mode.

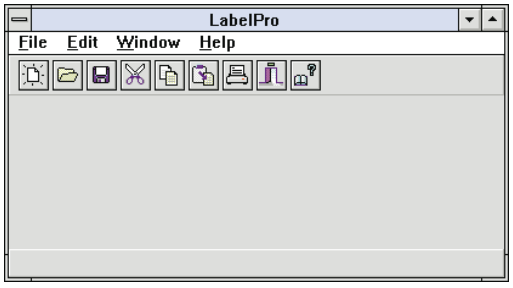


Figure 13: The sample application at run-time.

Making the Form Data Aware

Now let's add data access capabilities to this form. That is, we want to be able to add and edit records in the table (`LABELS.DBF`) we created earlier.

Select the Data Access page on the Component Palette and select the Table component. Now click on a blank space on the form, and the Table component will appear in that space. You can also place a component by simply double-clicking on it to place it in the center of the form. Next, select the DataSource component (located to the left of the Table component on the Data Access page) and place it on your form. Next, from the Data Controls page, double-click the DBGrid component to place it on your form. These — Table, DataSource, and DBGrid — are the three data-aware components we need to complete the sample application. Let's put them to work.

Activate Your Objects

Using the Object Inspector, we'll give life to the DBGrid component. This is done by configuring the Table and DataSource components with the Object Inspector. Select the Table component by clicking on it. Notice that `Table1: TTable` appears in the Object Selector. This indicates the component has focus. All the controls in the Object Inspector are now ready for the Table component.

In the Object Inspector select the *DatabaseName* property and select `LABELS` from the drop-down list in the Value column. This drop-down list contains the alias names we first saw in the Database Desktop. Now set the Table component's *TableName* property to `LABELS.DBF`. This is the table we created in Database Desktop. If there were other tables in the directory specified by the alias, they would also appear in this drop-down list. Set the Table's *Active* property to *True* by double-clicking in the Value column. This activates the connection to the table.

Now click on the DataSource component on your form. The Object Inspector changes again to work with this control. Set the *DataSet* property to `Table1`. This connects the form to the table you specified using the Table component. Set the *Enabled* property to `True`, and the *Name* property to `DataSource1`.

Select the DBGrid. Drag its handles to enlarge it, and move it closer to the bottom left of your form. Then, with the DBGrid still selected, use the Object Inspector to set DBGrid's *DataSource* property to `DataSource1`. The DBGrid is now active with data from the LABELS.DBF table.

A Final Touch

We're going to need a control to move around in the table displayed by the DBGrid. From the Data Controls page of the Component Palette, select the DBNavigator component. Place it on your form next to the speedbar (see Figure 14). In the Object Inspector, set DBNavigator's *DataSource* property to `DataSource1`. If necessary, you can resize your form to allow more room for the DBNavigator and DBGrid. Now test the application by selecting **Run | Run** from the menu (see Figure 15). Play around with the DBGrid. For example, you can resize the columns by dragging their partitions to the desired width.

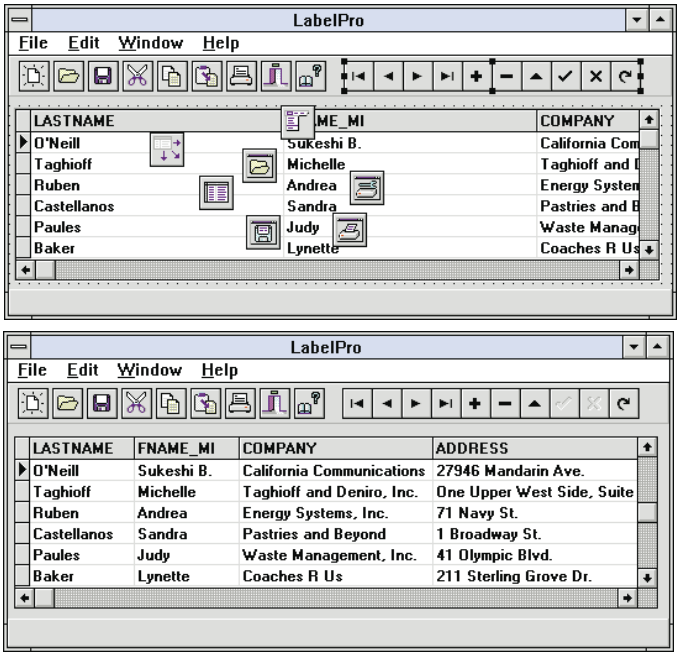


Figure 14 (Top): Adding data-aware components to the sample application. **Figure 15 (Bottom):** The sample application as it should appear after following this article's step-by-step directions.

Conclusion

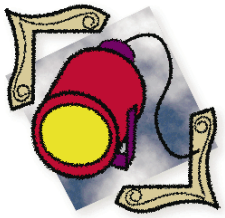
As you can see, creating a simple database application in Delphi isn't difficult, especially with the help of Delphi's Gallery and Application Expert. We were able to create a menu, speedbar, and manipulate data — without writing a single line of code.

We're not done yet, however. In the next article of this series we'll continue with the project. We'll also learn how to use ReportSmith to generate labels. Δ

The demonstration project referenced in this article is available on the Delphi Informant Works CD located in INFORM\95\OCT\AM9510.

Antony Mosley is a technical writer and program developer at Advanced Systems, Inc., a custom software developer and computer systems integrator. He can be reached on CompuServe at 73123,1273 or by mail at: ASI, 3040 Williams Drive, Suite 102, Fairfax, VA 22031.

table of CONTENTS



By Richard Wagner

Delphi32: A First Look

The Standard Bearer for Win32 Development?

The enthusiasm that has surrounded Delphi since its initial release in February of this year has been nothing short of remarkable. Not only has it won over many long-time Visual Basic, PowerBuilder, and C++ developers, but it has been extremely well-received by the press and many large corporations. It's no wonder that when Delphi32 was rolled out at the Borland Developer's Conference in San Diego this August, that the room was packed with excited onlookers awaiting a first glimpse at Delphi32 — the 32-bit release of Delphi for Windows 95 and Windows NT. I didn't take a poll at that initial viewing, but from the excitement in the air, I suspect none were disappointed with what they saw.

In many ways, Delphi 1.0 took Windows 3.x architecture nearly as far as it could go. Delphi32 is aiming to pick up where its 16-bit sibling left off, as evidenced by a feature set designed to maximize the Win32 environment: 32-bit architecture, multithreading, 32-bit optimizing compiler, OCX/OLE support, rich data types, Win95 UI enhancements, and more.

In this article, we'll take an in-depth look at Delphi32 by exploring many of these new features. We'll close by looking at how the 16-bit version of Delphi (referred to as Delphi16 throughout this article) and Delphi32 can coexist.

Two Steps to 32-Bitness

At the Borland Developer's Conference, Delphi product manager Zack Urlocker emphasized that one of the most important capabilities of Delphi32 is to provide full capability with existing Delphi16 code. To convert your 16-bit applications to 32-bit simply follow these two steps:

- Open your project using **File | Open**.
- Select **Project | Compile** from the menu (see Figure 1).

In other words, simply recompile your project to take advantage of the 32-bit architecture. However, as you would expect, there are certain cases when existing code must be modified so it can work in the Win32 environment. Any calls made to Windows API functions that were altered in Win32 must be changed. Additionally, as you will read later, there have been several data type enhancements. The result is that any 16-bit inline assembler code and low-level code that depends on the physical size of integers will need modification.

Win32 Facelift

Borland has historically gone their own way with UI standards over the years (think of BWCC.DLL), but that attitude is now shifting. The company is paying keen

Key Enhancements in Delphi32

- 32-bit optimizing compiler
- Database engine improvements
- OCX support
- Multithreading support
- New data types (long *String*, *variant*, and *WideChar*)
- Ease in porting 16-bit code to 32-bit environment
- OLE Automation
- Win95 common controls
- Closer integration with C++
- Enhanced IDE (debugger)

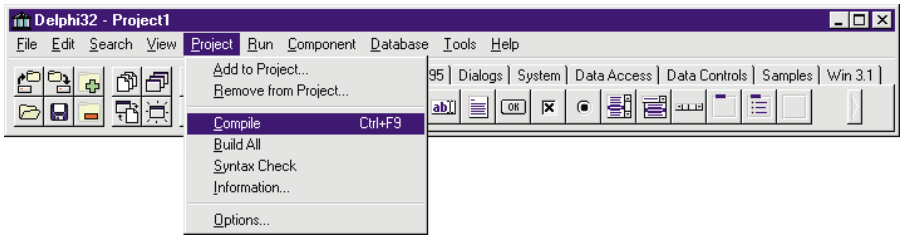


Figure 1: Selecting **Project | Compile** from Delphi32's menu is all that is needed to convert most 16-bit applications into 32-bit .EXEs.

attention to Windows standards these days and Delphi32 definitely reflects the change in philosophy.

The Delphi32 IDE (integrated development environment) has been transformed to look and feel just like Windows 95 software should (see Figure 2). You'll notice two examples immediately: 1) The "Borland-style" tabs on the Object Inspector and Component Palette are now standard Windows 95 tabs. 2) Gone are the bitmap-style **OK**, **Cancel**, and **Help** buttons. More importantly for you, however, Delphi32 provides new Windows 95 common UI controls on the Component Palette for you to use.

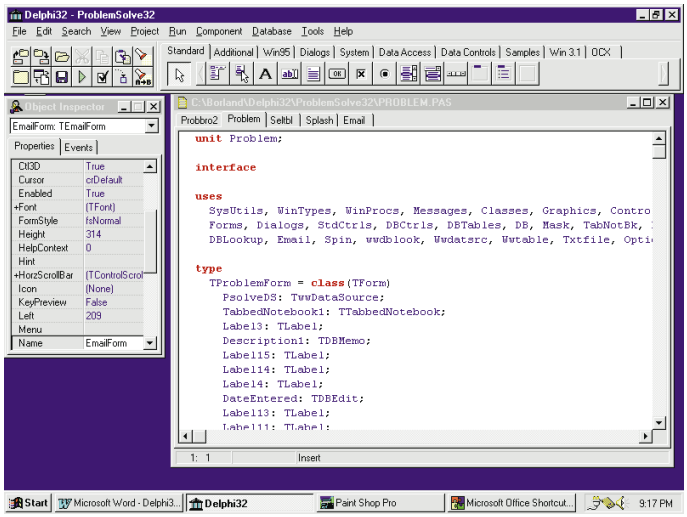


Figure 2: Delphi32's new user interface conforms to the strict Windows 95 logo standards.

Figure 3 illustrates the use of several of these controls, which are discussed below:

- *TPageControl*. PageControl is used to create Win95-style tabbed dialog boxes. With this component, you can add pages more easily than with its Delphi16 predecessor (TabbedNotebook) by right-clicking and choosing **New Page** from the menu. Another ease-of-use feature is the ability to activate a page by clicking its tab with your mouse rather than having to change the *ActivePage* property in the Object Inspector.
- *TTreeView*. TreeView is a Win95 version of the Outline component in Delphi16. The TreeView component is used quite often in Windows 95. Look at the Windows Explorer or Delphi32's Database Explorer (see Figure 11 later in the article) for other examples of a TreeView.
- *TProgressBar*. The ProgressBar is a Win95 "percentage meter" that enables you to show the percentage remaining in a lengthy process.
- *TTrackBar*. The TrackBar is a "slider like" component used to adjust values that fall within a continuous range.
- *TRichEdit*. The RichEdit box goes beyond the basic editing capabilities of the text box to support character properties (font, color, etc.) and paragraph properties (alignment, tabs, numbering, etc.).
- *TTabControl*. The TabControl allows you to create a set of tabs. If you want the tabs associated with "pages", then use the PageControl component.
- *TUpDown*. Used in conjunction with an edit box, the UpDown control is typically used to create a circular loop of input values.
- *THeaderControl*. Enhancing the capabilities of the 16-bit Header component, the HeaderControl is used to display headings above columnar data. You can divide the header into multiple sections when you need to place a heading above multiple columns of text or numbers.

Most of the new Win95 controls are enhancements of 16-bit controls as shown in the table in Figure 4. And, as you work within Windows 95, you will begin to see how extensively these common controls are used throughout the operating system's UI.

Beyond the 64K Barrier

The 64K harness that Windows programmers have always had to wear has been shed forever with Windows 95. As a result, you can declare data structures to be as big as you like in Delphi32, limited only by operating system memory.

Delphi32 has several new type enhancements to take advantage of the new Win32 environment. Because of the platform change,

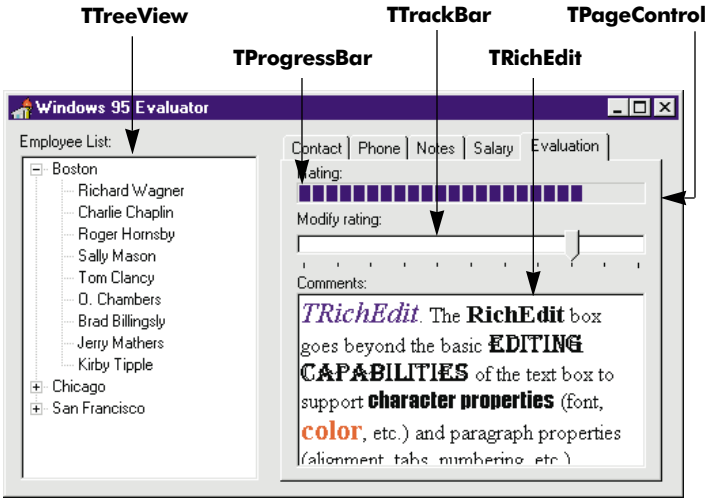


Figure 3: New user interface controls help you create a Windows 95 "look and feel" within your applications.

Delphi16 Control	Delphi32 Control
TOutline	*TTreeView*
TGauge	*TProgressBar*
TTabbedNotebook	*TPageControl*
TEdit	*TRichEdit*
TTabSetT	*TabControl*
TSpinButton	*TUpDown*
THeader	*THeaderControl*

Figure 4: Evolution of Delphi UI controls.

Integer and Cardinal values are now 32-bit values as compared to being 16-bit values in Delphi16. (You can still use a 16-bit value by declaring a *Smallint* or *Word* variable.) Other type enhancements include new string, variant, and character types.

Three New Types

Probably one of the things you became frustrated with and eventually accepted in Delphi16 (as well as earlier versions of Pascal) was the 255-character limit of the *String* type. Say good-bye to that limitation forever. A new long *String* type will enable you to create strings of virtually unlimited lengths. (You will be forced to contain yourself to 3GB or so.)

By default, you will work with the new long *String* type, but you can continue to use the 255-character *String* type by adjusting the new compiler directive `$H`. You can also continue to use a shorter string by declaring the string of type *ShortString* that is equivalent to the Delphi16 *String* type.

Another major benefit to the new *String* type is that long strings are terminated by a null character. Therefore, you can use a long string anywhere you used to use a null-terminated string in Delphi16. If you regularly create DLLs in Delphi, you will definitely appreciate this improvement; you no longer need to constantly convert strings using *StrPCopy* and *StrPas*.

Object Pascal loosens its collar slightly with the addition of a *variant* type. The new *variant* type provides increased flexibility by allowing you to dynamically change the type of a variable. The *variant* type can be useful in many everyday scenarios, but becomes critical for such tasks as OLE automation. The *variant* type can represent a string, integer, or floating-point value and is housed in a 16-byte structure that contains type information and a value.

Delphi32 also sports new character types to support the Unicode standard. (Note that Windows 95 does not fully support Unicode, but Windows NT does.) The equivalent of the *Char* type in Delphi16, *ANSIChar* is the new name for 8-bit characters. If you are creating applications for US and European users, chances are that this character type is all you will need to work with. However, if you need to create applications that will be used by Kanji or Arabic users, you can use the new *WideChar* type, which is a 16-bit Unicode character. You can continue to use *Char* in Delphi32 — it treats this type as equivalent to *ANSIChar*.

Multithreading

Multithreading is one of the underlying strengths of the Windows 95 32-bit environment (see sidebar "Windows 95 Multithreading" on page 29). In Delphi32, you have the ability to create Multithreaded applications. At press time, little information was available on writing Multithreaded applications, but a quick look at some of the functions will show you the capability you can expect.

CreateProcess creates a *process* (the term used to describe an instance of a running program) and a *thread kernel object,* which is used to manage the primary thread of the process. (Because you are in the Win32 environment, think "process" not "application". For example, you should use *CreateProcess* rather than *WinExec* to launch an application in Windows 95.) You can create a new thread in your application using *CreateThread* and set its priority level using *SetThreadPriority*. The *ExitThread* function terminates the thread.

Whenever you have multiple threads within a process, you may need to synchronize their execution. Use *WaitForSingleObject* and *WaitForMultipleObjects* to suspend a thread until an object (another thread, process, and so on) is available. You can also use *EnterCriticalSection* function to force all other threads within the same process to be suspended until the active thread calls *LeaveCriticalThread*.

Optimized Code without the Work

So far, we have discussed new features that you can optionally employ in your applications to exploit the Win32 environment. However, one of most powerful new features of Delphi32 is something you get by default: its new 32-bit optimizing native code compiler. This is actually the same back-end compiler that Borland C++ uses. Its benefits are twofold: faster performance, and tighter code sharing between Delphi and C++.

Faster performance. So you thought performance of your Delphi16 code was great? Wait until you see Delphi32. Borland has benchmarked Delphi32 applications and have found that they are some 300 to 400 percent faster than equivalent Delphi16 applications. Using the Sieve, Whetstone, File Write, and File Read benchmark tests, Delphi32 also was found to be 15 times faster than 16-bit Visual Basic 3.0 and an unbelievable 815 times faster than PowerBuilder 3.0. Figure 5 lists these results. (The higher numbers denote greater performance.)

Tighter code sharing with C++. In Delphi16, you can share C++ code only by using a dynamic link library (DLL) and calling functions within the library at run-time. In some cases, this is advantageous, but you may often want to access those same functions within the same .EXE, not an external file.

Using Delphi32, you can take advantage of Delphi's use of the C++ back-end compiler to do just that. You can compile C++ .OBJ object files into your Delphi application by using the `$L` compiler directive, or you can create .OBJ files with

Test (loops/sec)	PowerBuilder 3.0 (16-bit)	Visual Basic 3.0 (16-bit)	Delphi 1.0 (16-bit)	Delphi32 (32-bit)
Sieve	0.22	11.95	52.77	179.37
Whetstone	0.04	1.41	4.70	15.53
File write	0.05	0.42	0.74	2.89
File read	0.05	0.33	1.75	5.28

Figure 5: Delphi32 benchmark tests. The results are from an overview of the 32-bit Delphi Compiler for Windows 95 and NT (Zack Urlocker, Borland International).

Delphi32 for inclusion in a C++ application (see Figure 6). In other words, you can create a single .EXE using both C++ and Delphi, developing on the platform that makes the most sense. .OBJ support also allows you to utilize an investment in C++ class libraries that you may have already made.

Compiler optimizations. In the past, the task of optimizing compiled code was considered an art form and involved routinely tweaking compiler directives to achieve maximum performance. Delphi32 allows you to achieve optimizations that are "guaranteed bulletproof" without having to do a thing (see Figure 7). The table in Figure 8 details the types of optimizations performed.

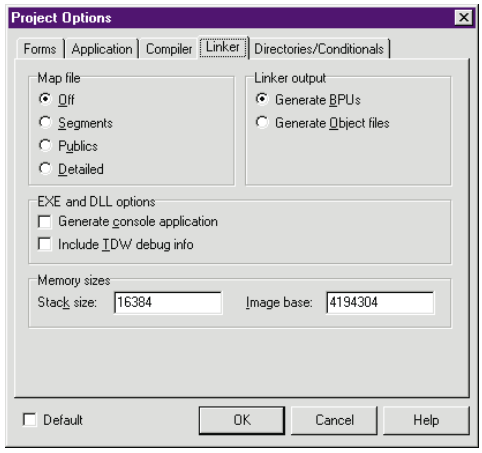


Figure 6: Delphi32's Linker options page in the Project Options dialog box.

Optimizing linker. During the compiling process, a new linker helps produce smaller, more efficient applications by eliminating unused functions, and unused static and virtual methods.

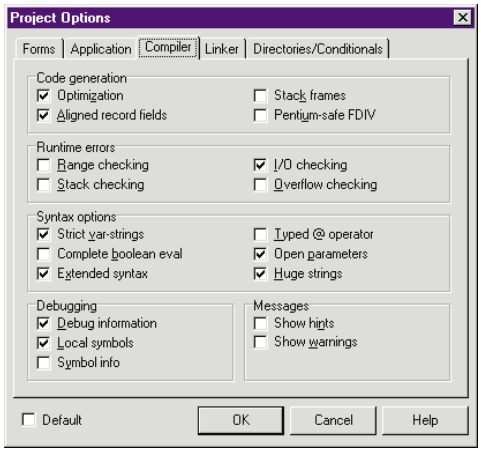


Figure 7: Checking the **Optimization** box in the new Compiler page turns on the optimizing features.

The linker also employs *unit caching*. With this functionality, when you recompile an application, any units or forms that have not been changed since the original compile will be linked in memory instead of from disk. Unit caching can help speed linking by 20 to 50 percent. Additionally, better unit version checking reduces the need to recompile units.

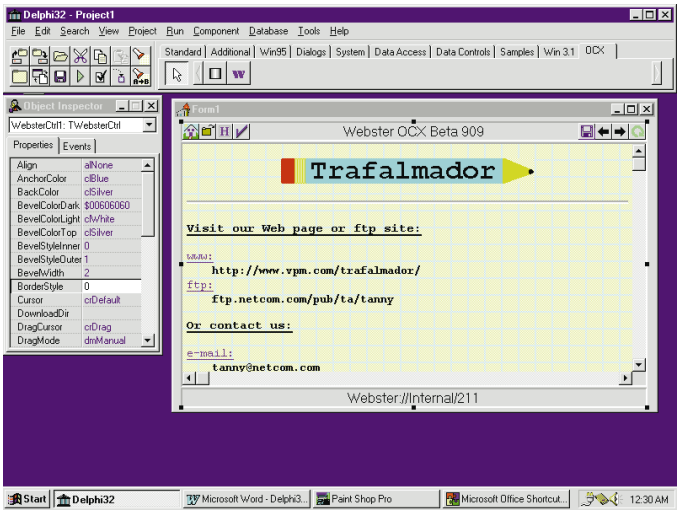
OLE: Key to Win32 Extensibility

We have heard much hype about OLE since Windows 3.1 was released, but the real-world benefits of OLE were fairly suspect until now. This has definitely changed, as the practical uses of OCXes and OLE automation are real and tangible. Delphi32 will provide full support for these technologies which are proving to be the keys to Win32 programming extensibility. You may find it helpful to think of OCX controls as the Win32 equivalent of VBXes. But this is really not true. VBXes were developed for the Visual Basic developer in mind, while OCXes are truly application-independent controls written based on OLE technology. Delphi32 will provide full support for OCX controls. For example, the field test version of Delphi includes an OCX control that is a fully functional Web browser (see Figure 9).

You can work with OCX controls much in the same manner you worked with VBXes in Delphi16. As far as OCX creation, you can certainly create OCXes in Delphi32, but you must do so at the API level, which is no small order. OLE automation allows you to control OLE servers from within another application. For example, you can use OLE automation to create Microsoft Project projects or generate Excel charts from within Delphi32. If you used DDE Execute commands in the past, you will find many of the same basic principles applicable to OLE automation. The simple example in Figure 10 shows that you can use Microsoft Word for Windows as an OLE server to create a new document, paste Clipboard contents into it, and then save it.

Type	Definition
Register optimizations	Frequently used variables are placed into CPU registers to shorten the time it takes to access them. Additionally, the scope of variables is analyzed to determine when registers can be reused.
Call-stack overhead elimination	Parameters are passed into CPU registers when possible rather than being pushed onto the stack.
Common subexpression elimination	Repeated expressions in complex mathematical calculations are eliminated, so that no common subexpression runs more than once. This allows you to construct complex math algorithms that are easy to read, and let the compiler worry about optimizing the operation.
Loop induction variables	Loop induction variables are used to decrease the amount of time it takes to access arrays or strings within loops (e.g. a **for** loop).

Figure 8: Compiler optimizations.




```
implementation

uses OleAuto;

procedure TForm1.Button1Click(Sender: TObject);
var
  { New Variant type }
  MSWord: Variant;
begin
  MSWord := CreateOleObject('Word.Basic');
  { The following commands are actually
    WordBasic macro commands }
  MSWord.FileNew;
  MSWord.EditPaste;
  MSWord.FileSaveAs('MyDoc');
end;
```

Figure 9 (Top): A sample OCX control allows you to surf the Web within your Delphi application. **Figure 10 (Bottom):** Using Word for Windows as an OLE server in Delphi32.

Win32 Database Engine

If you are a database developer, another major enhancement that you will benefit from is the new Borland Database Engine (BDE) 3.0. The new 32-bit BDE has been completely overhauled to provide much greater database support than in the previous 16-bit implementations. Key new improvements include the following:

Win32 compliance. BDE 3.0 supports Win95 logo requirements. The engine itself is 32-bit (as are the companion SQL Link drivers), and supports long filenames. The new BDE uses the Windows Registry to store configuration information, eliminating the need for a separate .CFG (configuration) file.

Multithreading support. BDE 3.0 is thread-safe, supporting concurrent application threads accessing the same database tables. In addition, as a practical example of multithreading with BDE 3.0, queries that used to monopolize CPU time until their completion in Delphi16 are now threaded so that you can perform other tasks at the same time.

Client Data Repository. BDE 3.0 now includes a facility to store persistent data. Known as the *Client Data Repository*, it provides a Common Object Manager to contain object (tables, records, fields, etc.) and relationship classes. You can also use the repository to define extended field attributes (font, picture, color, etc.), so that whenever a field is represented on a form, its UI object will contain those properties. Additionally, the repository can maintain a local version of a back-end server's schema, resulting in quicker access to SQL databases. Additional features of the repository include logical database definition (specific tables, not all tables within an alias or directory) and type casting.

Delayed updates. The new BDE allows you to cache updates to a database locally and send them in batch mode to any SQL or local database. Delayed updates allow you to minimize the length of time it takes to lock resources on the back-end database. Developers familiar with Paradox for DOS will find this a more powerful implementation of its CoEdit feature, which was limited to a single record.

New query engine. BDE 3.0 has an entirely new SQL query engine. In the past, the BDE has always been partially crippled because the query engine was based on QBE. The result was that Local SQL was limited, because many SQL commands could not be translated by QBE. However, the new SQL-based query engine conforms to the SQL-92 standard for local and SQL server tables.

Views. Views, which are essentially named SQL queries, will be supported in BDE 3.0. Views will function as a

Windows 95 Multithreading

Multithreading is a long way from the pseudo "multitasking" environment of Windows 3.x. In Windows 3.x, applications can be multitasked to allow multiple applications to be open at once. Windows 3.x activates an application when it receives a message in the message queue. Then, the application process continues until it's completed. While multitasking is better than singletasking in the DOS world, it still often leaves you staring at an hourglass cursor while a process runs to its completion.

Providing a preemptive multitasking environment, Windows 95 leaves this Windows 3.x architecture behind. The result is that every application process can use one or more threads for executing the program. Windows 95 can manage these multiple threads concurrently. In other words, Windows 95 can invoke a thread and suspend it at any time during execution to call up another thread.

A thread is executed in Windows 95 based upon the priority given to it by the process. (A thread's priority often changes during its life.) A priority is a number between 0 and 31 (although 0 is for system use only) with 31 being the highest priority. When threads are waiting to be executed, the *primary scheduler* looks at their priority level and calls the one with the highest priority first. When two threads have the same priority level, the first thread is invoked for a time slice, then suspended while the second thread is called for a time slice, and so on. To ensure that lower priority threads are not "starved" by these higher priority threads, a second scheduler (known as the *time slice scheduler*) tweaks the priority levels of the lower threads so that they will be executed as well.

table to the user and can be updated. Views on views will also be supported.

Simple transactions. Developers using Paradox and dBASE tables will finally be able to use transactions in their applications. Employing *simple transactions*, the BDE will provide commit and rollback capability on these local databases through client-based transaction logs. Note the use of the word "simple". This version will not support crash recovery or other advanced features.

Database Explorer. Not part of the BDE, but related to Delphi database access, is the new Database Explorer (see Figure 11), which replaces the limited Database Desktop in Delphi16. The Database Explorer provides a window for that hard-to-reach database information in both SQL and local databases. The Database Explorer allows you to view and modify aliases, metadata objects (tables, triggers, views, stored procedures), and user and security information.

Improved Debugger

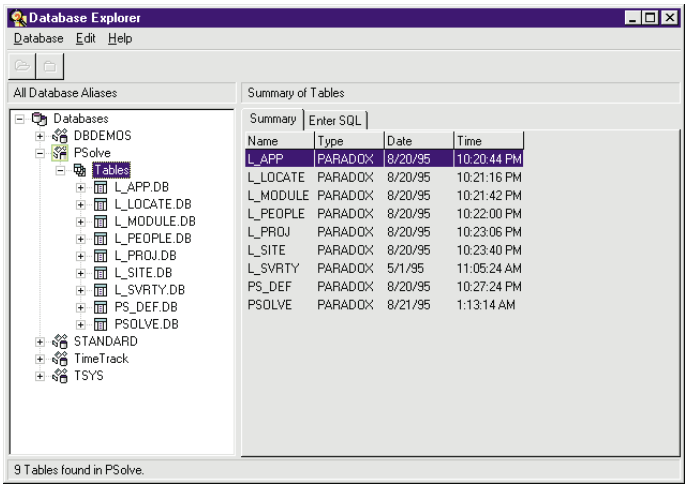
Debugging your Delphi applications will be much easier with several debugging enhancements: multi-error passes, more descriptive messages, and hints and warnings. Delphi32's compiler provides multi-error functionality, such that it will not die on the first error it encounters. Instead, it continues through the rest of the code to provide a complete list of compiler errors (see Figure 12). As a result, you can debug your application more quickly than before. The debugger also prompts you with hints and warnings alerting you to potential problems that may exist in your code, such as uninitialized pointers, unused variables, or unused function return values.

Mixing Delphi16 and Delphi32

By now, most of you probably cannot wait to get your hands on Delphi32. But at the same time, you realize that as much as you want to use Delphi32 exclusively, you still will have to continue to develop some 16-bit applications for users who continue working in Windows 3.x. If that is the case, be sure to keep Delphi 1.0 on your hard disk, because you are going to need it. Delphi32 is designed from the ground up for a 32-bit environment. Therefore, it will not be able to generate 16-bit code.

Nevertheless, Delphi16 and Delphi32 will happily coexist on your system. Besides their program files being stored in separate directories, the two versions of the BDE use completely separate DLLs, so there are no versioning issues that will arise when switching between the two products.

If you do not use any of the new Win32 features, you can recompile Delphi32 code with little or no modification in Delphi16. And while you can share unit (.PAS) and form (.DFM) files, you cannot share project (.DPR) files between Delphi16 and Delphi32. Therefore, get in the habit of saving the 16-bit projects you open in Delphi32 to a new name.



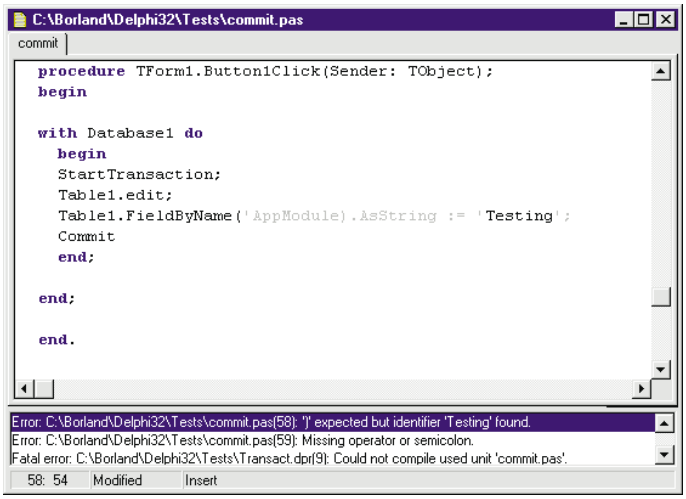


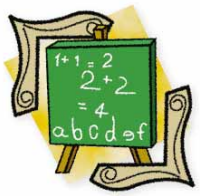
Figure 11 (Top): The Database Explorer makes database development and maintenance much easier. **Figure 12 (Bottom):** Delphi32 displays far more detailed debugging information than its predecessor.

Conclusion

Delphi32 is a convincing upgrade to the 16-bit product and is technologically far beyond its chief competitors — Visual Basic and PowerBuilder. If you are moving to Windows 95, then you will definitely want to add Delphi32 to your tool belt.

Is Delphi32 perfect? Not yet. Among other things, I'm still wondering about the enhanced reporting capabilities that will be in the final product. But on the whole, Delphi32 is proving that it will be a standard bearer for Win32 application development. Δ

Richard Wagner is a technical architect for IT Solutions, a leading Delphi consulting firm in the Boston area. He is author of several Paradox, Windows, and CompuServe/Internet books and is also a member of Team Borland on CompuServe. Richard can be reached on CompuServe at 71333,2031 or via the Internet at rwagner@cis.compuserve.com.



By Charles Calvert

Strings: Part III

Delphi Functions, Debuggers, Text Files, and Beyond

During the past two months, we've covered several string-related topics, including: searching for a sub-string within a string, parsing lengthy strings, stripping blanks from the ends of strings, and the Object Pascal functions for manipulating strings. In this last installment of "Strings", we'll discuss the custom *StripFirstWord* function in depth. We'll also discuss how to parse the contents of a text file and then convert the data into fundamental Delphi types.

Putting *StripFirstWord* to Work

Last month's article [in the September 1995 *Delphi Informant*] introduced the custom Object Pascal function, *StripFirstWord*. The following program, called TESTSTR, gives you a chance to work with the *StripFirstWord* function. This program takes any sentence you enter and separates it into a series of individual words that are displayed in a list box. To avoid any problems that may arise from accidentally prepending spaces before a string, the TESTSTR program makes use of the custom *StripFrontChars* function shown in Figure 1.

```
{-----------------------------------------------------
       Name: StripFrontChars function
 Declaration: StripFrontChars(S: string; Ch: Char): String;
       Unit: StrBox
       Code: S
       Date: 03/02/94
Description: Strips any occurrences of character Ch
             that precede a string.
-----------------------------------------------------}
function StripFrontChars(S: string; Ch: Char): string;
var
  S1: string;
begin
  while (S[1] = Ch) and
        (Length(S) > 0) do
    S := Copy(S,2,Length(S) - 1);
  StripFrontChars := S;
end;
```

Figure 1: The *StripFrontChars* function does exactly what its name implies. It strips the characters from the front (or beginning) of a string.

This routine is quite similar in functionality to the *StripBlanks* function, except that it starts at the opposite end of the string and lets you specify the particular character to cut. [For more information about the *StripBlanks* function, see Charles Calvert's article "Strings: Part II" in the September 1995 issue of *Delphi Informant*.] If you pass it a string and *#32*, it will make sure there are no spaces preceding the string.

StripFrontChars works its magic by first checking to see if the initial character in the string has the same value as *Ch*. If it does, it finds the second character in the string and copies it and the remainder of the string back over the first character of the string. *StripFrontchars* thereby accomplishes a task similar to that undertaken by the second *Move* statement in *StripFirstWord*.

The TESTSTR program uses an Edit control, a button labeled **Parse**, and a listbox. (The form for the program is shown in Figure 2, and its code is shown in Listing One on page 35.)

TESTSTR uses the *OnCreate* event to specify a string and pass it to the *BParseClick* function. This function uses *StripFirstWord* to divide the sentence into individual words and display each word in a listbox. One of the interesting aspects of TESTSTR is that it shows how you can place routines in a unit such as STRBOX and then call them in a neat and easily readable fashion.

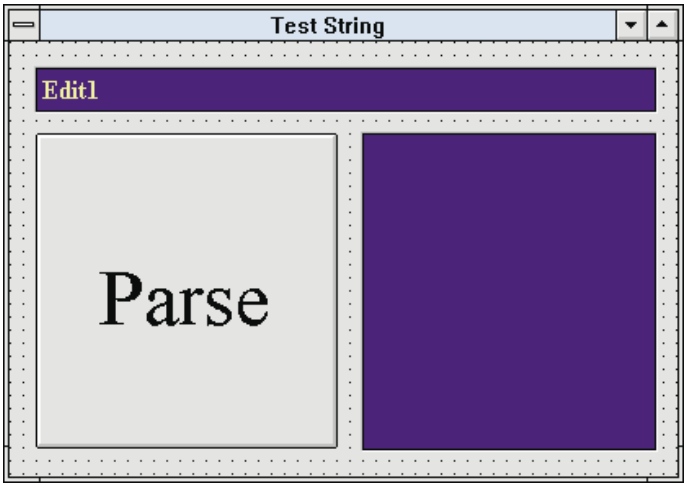


Figure 2: The form for the TESTSTR program.

At this point, you have seen a number of routines for manipulating strings. There are hundreds of different functions you could write to help you perform certain string-oriented tasks. The ones I have shown here should help you get started in creating a custom library to refer to when you need a quick solution for a problem involving a string. The final version of the STRBOX program contains various additional string manipulation routines that I have built over the years. [It is available for download. See end of article for details.]

Limiting the Length of Strings

At times it seems foolish to allocate an entire 250-byte block to deal with a very short string. For instance, you might have a string that held the first name of the current user. Furthermore, your input routines may limit the length of the name the user can enter. For instance, many programs allow you only 30 characters to enter a name. Therefore, the string that holds the user's name need never be longer than 30 bytes. Any more bytes would be wasted every time the string is used.

In situations like this, you can limit the length of the string you declare:

```
Name: string[30];
```

This syntax tells the compiler that it needs to set aside only 31 "slots" for this particular string. The first holds the length byte, and the remaining thirty hold the string. By declaring a string this way, you save 256-31, or 225 bytes.

Delphi enables you to declare strings of any length between 1 and 255. For instance, all of the following are valid string declarations:

```
S:  string[1];
S1: string[255];
S2: string[100];
S3: string[25];
```

Strings of certain lengths are declared so often that you might want to create special types for them:

```
type
  TStr20: string[20];
  TStr25: string[25];
  TStr30: string[30];
  TStr10: string[10];
  TStr80: string[80];
```

Given these declarations, you can write code like this:

```
var
  S1: TStr20;
  S2: TStr30;
```

The syntax shown here can help you save memory without having to perform much extra work. In fact, the types shown here are so useful that I have added them to the STRBOX unit so you can access them easily at any time. I have also added a few other string declarations that can prove useful when you are working with files. For example:

```
DirStr  = string[67];
PathStr = string[79];
NameStr = string[8];
ExtStr  = string[4];
```

Unfortunately, when limiting the length of strings using this technique, you are required to know the length of a string at design time. Working with *PChars* and the *GetMem* function, it is possible to set the length of a string at run-time. This means you can make strings that are exactly long enough to hold the characters stored in them. You can also declare a pointer to a standard string, predeclared by Delphi as a PString, and then allocate the necessary amount of memory needed for it.

Working with Text Files

A text file provides a place for you to store strings. The .PAS and .DPR source files for your programs are text files. So are the .OPT and .DSK files that accompany them, as well as the WIN.INI and SYSTEM.INI files used to configure Windows 3.1. Even the AUTOEXEC.BAT and CONFIG.SYS files are nothing more than text files.

Delphi provides an extremely simple method for reading and writing these text files. To get started, you need only declare a variable of type *Text*:

```
var
  F: Text;
```

Variable F now represents a text file, and it can be used to specify where you want data to be read from or written to. After declaring the file variable, the next step is to associate it with a particular filename:

```
var
  F: Text;
begin
  AssignFile(F,'MYFILE.TXT');
  ...
```

It's traditional to assign the extension .TXT to a text file. Of course, many files don't follow this convention. For instance, .INI files don't use the .TXT extension, and neither do files labeled READ.ME. However, most text files do have a .TXT extension, and this is the accepted way to treat them.

After the assignment statement, the next step is to open the file with the *Reset*, *Rewrite*, or *Append* routine. The *Reset* procedure opens an existing file:

```
var
  F: Text;
begin
  AssignFile(F,'MYFILE.TXT');
  Reset(F);
  ...
end;
```

The *ReWrite* procedure creates a file or overwrites an existing file. The *Append* routine opens an existing file. It does not overwrite its current contents, but enables you to append new strings to the end of a file:

```
var
  F: Text;
begin
  AssignFile(F,'MYFILE.TXT');
  Append(F);
  ...
end;
```

There is no simple way to open a text file and insert a string into it at a particular location. You can create a new file, overwrite an existing file, or append a string onto an existing file. You can't easily add a new string in the fourth line of a twenty-line text file. I say you can't do it easily, but there are ways. These methods usually involve opening the file in either binary or text mode, reading its entire contents into an array or other complex data structure, inserting a string, and then writing the file back out to disk.

Once you have opened a text file, you can write a string to it with the *Write* or *WriteLn* procedure:

```
var
  F: Text;
begin
  AssignFile(F,'MYFILE.TXT');
  ReWrite(F);
  WriteLn(F,'Call me Ishmael.');
  CloseFile(F);
end;
```

Notice that the *WriteLn* statement begins by referencing the file variable as its first parameter.

The method shown here ends with a *CloseFile* statement that must be referenced with a qualifier representing SYSTEM.PAS. Bugs can be introduced into your program if you forget to call *CloseFile*. More specifically, there is an internal buffer associated with each text file that cannot be flushed if you fail to call *CloseFile*. If the buffer isn't flushed, a portion of the file will remain in memory rather than being written to disk. Consequently, it appears that your program is not writing properly to the file, and programmers can spend a long time looking for a memory corruption problem when the source of the trouble is simply failing to call *CloseFile*.

The SYSTEM file is buried deep in the Delphi run-time library (RTL). Every Delphi program you create will automatically have the system file linked into it, even if you don't explicitly reference it in your **uses** clause. Most of the SYSTEM unit contains assembly language code for elemental routines such as *WriteLn*, *Assign*, *ReWrite*, *Reset*, and a few other functions that have survived since the earliest versions of the Turbo Pascal compiler. (To learn more about this unit, you can search on "SYSTEM" in Delphi's on-line help.) The availability of the source code depends on which version of Delphi you purchased. It comes with the Client/Server version, but not the desktop version. If it exists on your system, it would be stored somewhere beneath the \SOURCE sub-directory. You can also buy the RTL separately from Borland. If you are a serious programmer and you don't have it, I recommend buying it.

To read a string from a text file, you can use the *Read* or *ReadLn* statement:

```
var
  F: Text;
  S: String;
begin
  AssignFile(F,'MYFILE.TXT');
  Reset(F);
  ReadLn(F,S);
  WriteLn(S);
  CloseFile(F);
end;
```

Notice that this code uses the *Reset* procedure to open an existing file, then uses *ReadLn* to retrieve the first string from this file. You can also read and write numbers from a text file. For instance, the following code is entirely legal:

```
var
  F: Text;
  S: String;
  i: Integer;
begin
  AssignFile(F,'MYFILE.TXT');
  ReWrite(F);
  S := 'The secret of the universe: ';
  i := 42;
  WriteLn(F,S,i);
  CloseFile(F);
end;
```

This code writes the following line into a text file:

```
The secret of the universe: 42
```

If you had a text file with the following contents:

```
10 101 1001
20 202 2002
```

you could read the first line from this file with the following code:

```
var
  F: Text;
  i, j, k: Integer;
begin
  AssignFile(F,'MYFILE.TXT');
  Reset(F);
  ReadLn(F,i,j,k);
  CloseFile(F);
end;
```

The code shown here would read the numbers 10, 101, and 1001 from the file. If you wanted to read both lines from the file, you could write:

```
var
  F: Text;
  i, j, k, a, b, c: Integer;
begin
  AssignFile(F,'MYFILE.TXT');
  Reset(F);
  ReadLn(F,i,j,k);
  ReadLn(F,a,b,c);
  CloseFile(F);
end;
```

You can use a function called *EOF* to determine if you are at the end of a text file. For instance, if you had a file that contained several hundred lines of numbers as those shown in the small file listed above, you could read the entire file with code shown in Figure 3.

The EASYFILE program demonstrates how to use the *TTextRec* structure to determine a file's name, and if a file is open for input, open for output, or closed. Specifically you can typecast a variable of type *Text* so that you can test its state, as shown here:

```
var
  F: Text;
begin
  if TTextRec(F).Mode := fmClosed then
    OpenTheFile;
end;
```

TTextRec is in the on-line help, the Mode constants are declared in SYSUTILS.PAS as follows:

```
fmClosed = $D7B0;
fmInput  = $D7B1;
fmOutput = $D7B2;
fmInOut  = $D7B3;
```

Conclusion

In this article you've learned the basics about text files. One of the most important points to remember is that you can open a file one of three ways:

- *ReWrite* opens a new file or overwrites an existing file.
- *Reset* opens an existing file for reading.
- *Append* opens an existing file and enables you to append text to the end of it. If the file does not exist, an error condition results.

Visual programming in Delphi is a complete package. That is, you can create an entire application and distribute it, without writing a single line of code. Does this mean that a lay person can turn on a computer, fire up Delphi, drop a couple of components on a form, and create an effective application? The answer to this question, clearly, is "No."

The robust nature of Delphi stems from both its components and Object Pascal code. Developers must be able to effectively use their coding skills to exploit the capabilities of any programming environment. Understanding strings as well as the nature of text files will help enable programmers to build Delphi applications that serve their clients' needs. Δ

This article was adapted from material for Charles Calvert's *Delphi Unleashed* (SAMS, 1995).

The demonstration programs — TESTSTR and EASYFILE, as well the unit files STRBOX and MATHBOX — referenced in this article are available on the Delphi Informant Works CD located in INFORM\95\OCT\CC9510.

```
var
  F: Text;
  i, j, k, Sum: Integer;
begin
  AssignFile(F,'MYFILE.TXT');
  Reset(F);
  while not EOF(F) do
    begin
      ReadLn(F, i, j, k);
      Sum := I + j + k;
      WriteLn(F,'i + j + k := ', Sum);
    end;
  CloseFile(F);
end;
```

Figure 3: Using the *EOF* function to find the end of a file.

Charlie Calvert works at Borland International as a Developer Relations Manager for Languages. He is the author of *Delphi Programming Unleashed, Teach Yourself Windows Programming in 21 Days,* and *Turbo Pascal Programming 101*. He and his wife Marjorie live in Santa Cruz, CA.

Listing One — The TESTSTR program

```
unit Main;

interface

uses
  WinTypes, WinProcs, Classes, Graphics, StdCtrls,
  Controls, Forms;

type
  TForm1 = class(TForm)
    BParse: TButton;
    Edit1: TEdit;
    ListBox1: TListBox;
    procedure BParseClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    TestStr: string;
  end;

var
  Form1: TForm1;

implementation

uses
  StrBox;

{$R *.DFM}

procedure TForm1.BParseClick(Sender: TObject);
var
  S: string;
begin
  ListBox1.Clear;
  TestStr := Edit1.Text;
  repeat
    TestStr := StripFrontChars(TestStr, #32);
    S := RemoveFirstWord(TestStr);
    if S <> '' then
      ListBox1.Items.Add(S);
  until S = '';
  if TestStr <> #32 then
    ListBox1.Items.Add(TestStr);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  TestStr := 'In the long sleepless watches of the night';
  Edit1.Text := TestStr;
end;

end.
```

End Listing One

At Your Fingertips

BY DAVID RIPPY

DELPHI / OBJECT PASCAL

If you don't know where you're going, any path will get you there.

— Unknown

How can I program Enter to act like Tab when pressed?

For applications that require a great deal of data-entry, users will often request that Enter behave like Tab and automatically advance the cursor to the next field when pressed. At first glance, this may seem like a lot of programming, but with a little help from the Windows API, we can achieve this behavior in one line of code.

The sample form in Figure 1 contains six edit fields. Our goal is to provide the user with the ability to maneuver between the fields by simply pressing Enter. First, select each of the edit fields, and double-click on the *OnKeyPress* event in the Object Inspector. Now add the code snippet found in Figure 2 to the *OnKeyPress* event handler. Because we have selected every field object, this method will trigger regardless of the field we are positioned on.

The code first interrogates the procedure's *Key* parameter to see if the key pressed was VK_RETURN, which is the Windows Virtual Key constant for Enter. Since the value of the constant is an integer, we must first convert it to its associated Character value with the *Chr* function. Alternatively, you could use "#13" instead of performing a type conversion, but using the virtual key constant makes the code easier to read.

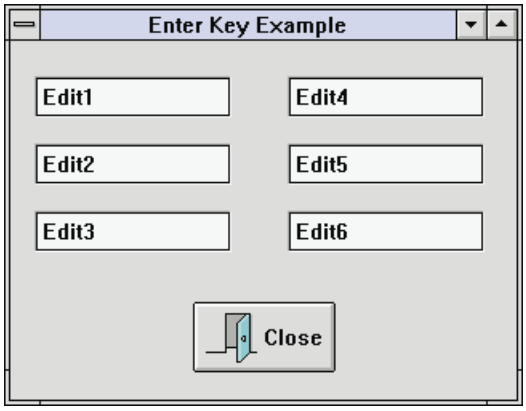


Figure 1: Pressing Enter will maneuver the cursor between these six edit fields. When you move the focus to the **Close** button and press Enter, you will exit the form.

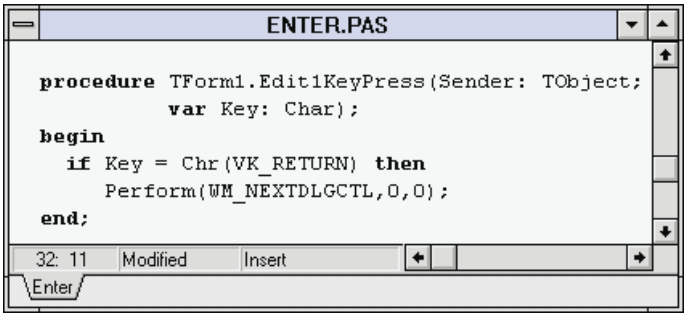


Figure 2: This code is attached to the *OnKeyPress* event handler for the edit fields.

If Enter was pressed, the *Perform* procedure is executed to send the Windows message WM_NEXTDLGCTL back to the application. The WM_NEXTDLGCTL message (Windows Message, Next Dialog Control) instructs the application to set focus on the next dialog box control on the form. In this case, the next control is the next Edit field whose *TabStop* property is set to *True*. That's it!

While this is a relatively simple example, it illustrates how well Delphi integrates with the Windows API. It's a powerful combination. — *Brendan Delumpa, Prescription Solutions*

How can I reduce the size of my Delphi executable?

You've just completed quite an elegant application, but wound up with a really hefty executable. Here's how can you slim down that overweight .EXE.

Choose **Project | Options** from Delphi's menu to invoke the Project Options dialog box. Now click on the Linker tab to display the page of linker options (see Figure 3). Notice a

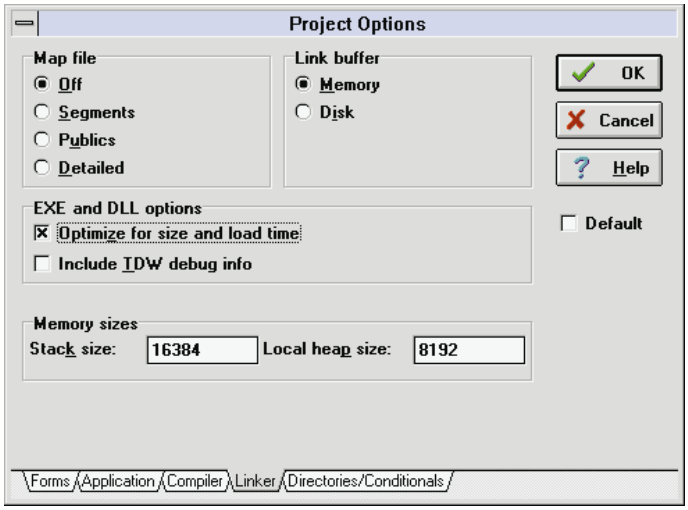


Figure 3: The Linker page of the Project Options dialog box.

group titled **EXE and DLL options**. The first checkbox, **Optimize for size and load time**, will compress the size of your executable, resulting in a quicker load time.

However, the program's link time remains the same. What *is* different is that Delphi's IDE runs the final executable through the W8LOSS.EXE program before it finishes compiling and linking. Interestingly, you can run W8LOSS at the DOS prompt on a program that was compiled with the debug information, and it will strip that information from the .EXE.

Although it's not essential to check the **Optimize for size and load time** option during design time, you should *always* check it when producing the final .EXE.

The second checkbox, **Include TDW debug info**, inserts information used by the Turbo Debugger for Windows into the executable. [This discussion does not address the debugger built into Delphi's IDE. Instead, the reference is being made to the separate Turbo Debugger program that ships with Delphi.] You must check this option if you want to use the Turbo Debugger, so leave it checked during development. Although the debug information does not affect the speed of the executable, it does significantly increase the file's size. Therefore, when it's time to create your final .EXE, you should definitely uncheck this option.

Figure 4 shows the effect of compiling a sample application with the various **Linker** options checked or not. As you can see, these options can make quite a difference. — *Russ Acker, Ensemble Corporation*

Options	Yes	No
Include TDW debug info?	3,407,450 bytes	2,782,720 bytes
Optimized for size?	2,632,192 bytes	3,256,922 bytes

Figure 4: As you can see, the difference in selecting these options will have a dramatic affect on the size of your executable file.

How can I suppress the dialog boxes that appear when using packages such as Crystal Reports?

When creating multi-media applications, the last thing you want the user to see is a Windows-style dialog box, similar to Figure 5. Not only do these dialog boxes increase the odds of the user crashing the system, it also appears out of place on the form. Unfortunately, many third-party products such as Crystal Reports do not give you the option of suppressing these kinds of dialog boxes. However, with Delphi we can solve this problem in zero lines of code. How's that for efficiency!

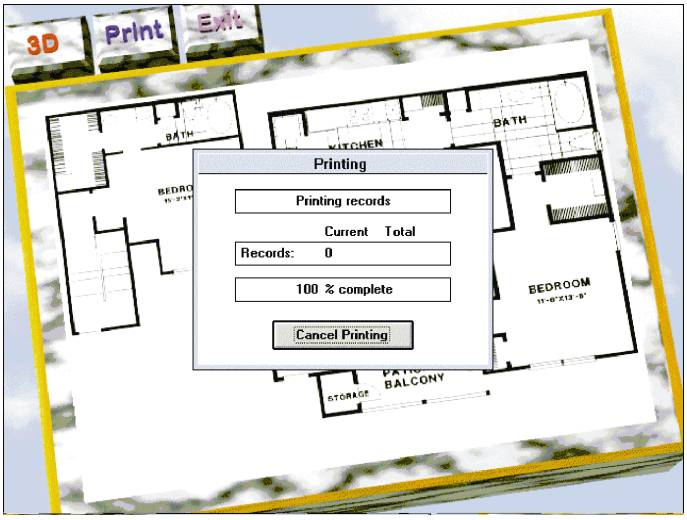


Figure 5: The Printing dialog box looks out of place on this form.

By simply changing the *FormStyle* property of the Form to *fsStayOnTop* as seen in Figure 6, we cause the form to always "cover" the Printing dialog box. The Printing dialog box is still created, but it no longer interferes with our otherwise beautiful application. Δ
— *David Wilson*

The demonstration project referenced in this article is available on the Delphi Informant Works CD located in INFORM\95\OCT\DR9510.

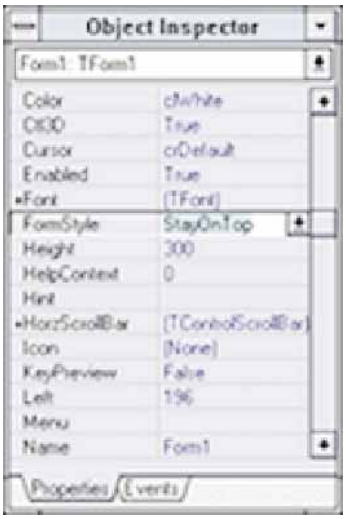


Figure 6: Select Form1 in the Object selector and then set the form's *FormStyle* property to *fsStayOnTop*.

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

table of CONTENTS

By Vince Kellen

Why Do Programmers Love Delphi?

One Developer's Opinion

I've been doing some research lately. Nine out of ten programmers prefer Delphi — and the tenth one hasn't seen the product yet.

It seems that everywhere I go, everyone is asking about Delphi. Clearly I've underestimated Delphi's appeal. Everyone loves it. The programming staff at Kallista is no different. Each staff member has asked me for a Delphi project. If I don't have one, the programmer sulks, whines, and walks away. Then I threaten the programmer with a Paradox 3.5 for DOS maintenance project. That stops the whining, if not the sulking.

It's affected my family as well. A relative of mine recently attended some corporate training in mainframe programming. The instructor said that for programmers today, Delphi is where it's at. He (the relative) called and asked me a bunch of questions about Delphi. He now prefers Delphi and he hasn't seen it!

What's Going On?

I also prefer Delphi, although I don't understand all the reasons why. Of course I can go on about everything everyone knows about Delphi: OOP, .EXEs, source code, great UI, blah, blah, blah. But I'm convinced there's a deeper, more subtle reason why programmers prefer Delphi.

So now, when I host Delphi SIG meetings, speak at conferences, or talk to clients, I ask all the Delphians out there what they like about Delphi. The responses are predictable:
"It's object-oriented."
"It's a compiler."
"It's fast."
"It's got great components."
"It produces .EXEs."
"It's got source code."
"It's got database components built in."
"It's Pascal."
"It's not a Microsoft product."

Yawn. These can hardly be the reasons so many programmers' hearts palpitate when they use Delphi. Or why so many swoon when they talk about it. They really do swoon. I've seen it. I've even heard of some Delphians fainting in the middle of a heavy Delphi conversation. Honest.

No, the reason must be something else.

Meeting the Challenge

To understand the reason, you must ask "Why do programmers like to program?" For many people, programming is not just a job or a hobby, it's an adventure. The love of programming lies in the challenge of it. After all, why do people no longer program in old languages? Because it's boring; there's no challenge in it any more. As soon as a programmer masters an environment, he or she invariably wants to learn another one.

For any activity to sustain life-long enjoyment, that activity must be continually challenging. Mihaly Csikszentmihalyi, in his book, *Flow: The Psychology of Optimal Experience,* describes the concept of *flow*, which many programmers routinely experience. Flow refers to a state of mind where enjoyment rules. Where time seems to stand still, yet fly. Where there are no distractions. Where the mind is so consumed entirely in the task at hand that it is in another world. For programmers, that world is cyberspace and many of our spouses know all too well what cyberspace is. For most of them, it's as good as the neighborhood bar or pool hall. Or an affair. Csikszentmihalyi affirms that to maintain flow year after year, the activity must remain challenging. And if the programmer is improving, the challenge needs to increase as well.

In this regard, Delphi fits nicely. People new to programming can use it and begin the challenge of learning basic programming concepts. For example, an employee of one of our clients — without any programming background — has started using Delphi to create small applications with little code. And one of the developers at Kallista is teaching his 11-year-old daughter how to use Delphi.

For programmers new to object-oriented programming, Delphi presents a new challenge. There's plenty to learn for programmers new to database programming as well. Experienced programmers will find creating components interesting, challenging, and rewarding. Programming wizards can read the VCL source code and try to find weaknesses, inadequacies, or problems. Windows wizards can dig deeper into Delphi to understand how it interacts with the Windows API and the Windows event model. They can exploit that knowledge to build more sophisticated components and applications. For all of these people, there is enjoyment in the challenge. In other words, Delphi presents a continual challenge for all sorts of programmers. But simply providing programmers with a continual challenge does not make Delphi a favorite. After all, even many Paradox for Windows programmers prefer Delphi, and they've been challenged for quite some time now.

No, the answer must lie somewhere else.

Into the Fray

Programmers prefer to be close to, or in the midst of, the strife. Sooner or later, programmers resent having other programmers shield them from battle scenes. While Delphi does shield programmers from the battles of the Borland Database Engine and the Windows API, with the VCL source code at hand, Object Pascal programmers feel very close to the battle.

In fact, there's something nice about using the same tool that Borland programmers used to build Delphi. It's like wearing King Arthur's battle gear. Or wearing Ted Williams' uniform at a softball game. Or holding a hockey stick that Wayne Gretzky used. It's as if you have become The Great One, gliding down the ice, past the onrushing defenders, and scoring. It connects you with the game, and the challenge is in the game.

Having the source code and being close to the battle creates all sorts of illusions. Besides running faster than other environments, Delphi feels — well — "thinner". There's less magic between you and the CPU. When you see a form run it *appears* to run faster because you've read the code that makes it run. Borland isn't saying "Never mind that man behind the curtain." Instead, they have opened the curtain.

I've posed this scenario to many Delphians. They shrug and say "I don't know." They're skeptical. Clearly the answer to my basic question lies elsewhere.

It's the Tools

Perhaps it's the tabbed Component Palette. This tool is so much better than its equivalent toolbox in Visual Basic that I'm tempted to ascribe all programmer's affection for Delphi to it. It neatly and compactly presents several dozen components. No more "icon block" where you stare at the overstuffed VB toolbox window wondering "where oh where is that component?" Sometimes I think this saves me an hour a day.

Or perhaps it's the Object Inspector with the tabbed properties and events pages — in sorted order, no less! What a novel concept. With lists of properties in alphabetic order, it's much easier to find the property you want to change. Sometimes I think this saves another half hour a day. And when you can find a property quickly, you can spend more time thinking about your program.

I have posed these two significant improvements in the typical programming environment as the reason for programmer's rapid acceptance of Delphi. Those who listened to me shrugged their shoulders, unsure if that was the reason, or concerned that I was out chasing windmills again. Evidently, the answer must lie somewhere else. My quest continued.

Must Be the Language

Perhaps it's Object Pascal. It's ironic that many early Delphi naysayers criticized Borland for using Pascal. Borland should have used C or some form of Basic, they argued. On the other hand, I've had Delphians claim that Borland's use of Pascal was a bold and courageous move. I'm not so sure. Each time I speak about Delphi, I ask how many people in the crowd have programmed in Pascal. About half raise their hands. Clearly Borland was on target with Pascal.

And the language *is* nice. The syntax is clear and elegant. And except for a few areas, it's easy to teach anyone with C or Visual Basic experience the Pascal language. Its syntax for handling object-oriented programming is delightfully straightforward without sacrificing power.

Is this what attracts thousands? Many Delphi fans answer in the affirmative. But this time *I'm* skeptical. Tens of thousands didn't rush to the computer store to buy the latest upgrade to Borland Pascal with Objects. The answer must be behind another veil.

It's the Environment

Maybe it's having data in tables visible in design mode. How many times have you accessed a table and spent hours hammering out code, only to realize you've selected the wrong table? Or had to task switch to view the table to make sure it was correct?

By my reckoning, that's saved me another half hour a day. After all, the more information visible to the programmer, the faster the programmer can work. Interruptions are a major impediment to programmer productivity, even if the interruption is simply to view records in a table. If the programmer's mind is taken off the algorithm at hand, flow is diminished and time evaporates.

Maybe it's having the Local InterBase engine (for the Client/Server edition folks) built right in. Adding a client/server dimension to the product and making that dimension immediately accessible opens up new areas of programming for those who haven't had the exposure. And InterBase has great support for the ANSI SQL-92 standard so the skills learned here are transferable to other environments. Perhaps so many bought Delphi to get up to speed in the client/server game — or ahead of the game. I just don't know.

May Be Habit Forming

I talked with a member of the Borland Paradox team. He asked me what I thought of Delphi. I said that application development seems to go faster in Delphi than in Paradox. He expressed concern. After all, it is his product — Paradox — that I slighted. He asked why. I was stumped, I wasn't exactly sure. In fact, it was this conversation that started me on my quest. I told him I thought the rapid program/compile/test cycles in Delphi were faster than in Paradox. Or that having table data visible in design mode made things move faster. Or the complied EXEs, and so on. Blah, blah, blah.

What I forgot to tell him was that perhaps it was because the Delphi environment is just more addictive than other environments. That's it! Maybe the Delphi environment is *addictive* in some way. Maybe it's so popular because it's like a drug, except it's legal and a heck of a lot cheaper.

I'm still not sure exactly why Delphi is so popular. Sometimes I think the answer to that question is staring me right in the eyes. There has to be some reason, don't you think? Δ

Vince Kellen is Vice President of Kallista, Inc., a leading provider of database application consulting, add-on products, and training. Vince is co-author, with Bill Todd, of *Creating Paradox for Windows Applications* [New Riders Publishing , 1995]. His latest book, also with Bill Todd, is *Delphi: A Developer's Guide* [M&T Books, 1995]. Vince is also a member of the IEEE and ACM. He can be reached at Kallista at (312) 663-0101, or on CompuServe at 70511,3511.

By Kenn Nesbitt

See-Through Images

A Windows 95 Color Scheme Work-Around

One of the things you will notice as you begin playing around with Delphi16 (i.e. Delphi 1.0) in Windows 95 is that many of the new color schemes replace the default ButtonFace color. For example, although the Windows Standard color scheme has the same light gray buttons we are so familiar with, the Eggplant color scheme has green buttons and the Brick color scheme has gold buttons.

If your programs count on the fact that *clBtnFace* will always be light gray (i.e. *clSilver*), you may end up with very ugly programs when a user changes color schemes. Figure 1 shows an example of the problem. The gray rectangle would not appear if the form's background color were gray.

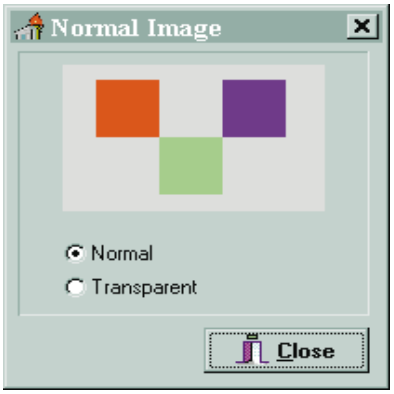
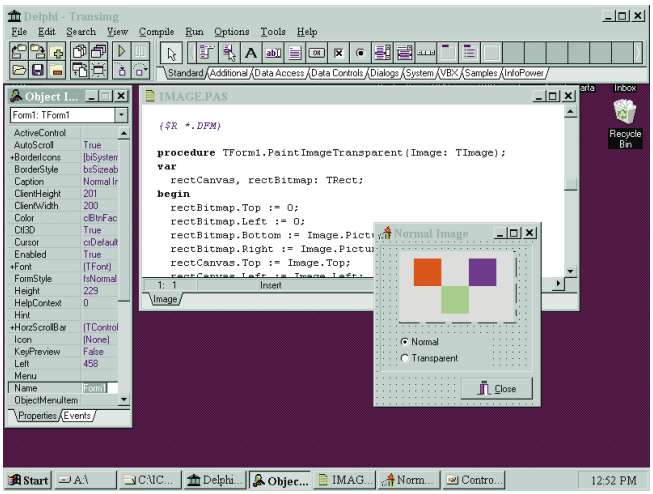


Figure 1: This is how a standard *TImage* control may appear when the user changes the default button face color.

The code example in Figure 2 shows an easy way to create transparent images on a form. In other words, images whose background color is the same as the color of the form.



Delphi16 running on Windows 95 with the Eggplant color scheme selected.

Here's how it works. First, you place *TImage* controls on a form and load bitmaps into them, just as you normally would. Next, change the *Visible* property of the control to *False*. This prevents the image from initially displaying.

Finally, add a procedure to the form to tell it to copy the contents of a *TImage* to the canvas of the form, replacing the specified color in the bitmap (in this case *clSilver*) with the background color of the form itself. In the form's *Paint* procedure, you will need to call this new procedure for each *TImage* control that needs a transparent background color.

```
procedure TForm1.PaintImageTransparent(Image: TImage);
var
  rectCanvas, rectBitmap: TRect;
begin
  { Get the size of the bitmap }
  rectBitmap.Top     := 0;
  rectBitmap.Left    := 0;
  rectBitmap.Bottom  := Image.Picture.Bitmap.Height;
  rectBitmap.Right   := Image.Picture.Bitmap.Width;

  { Get the location of the TImage on the form canvas }
  rectCanvas.Top    := Image.Top;
  rectCanvas.Left   := Image.Left;
  rectCanvas.Bottom := rectCanvas.Top + rectBitmap.Bottom;
  rectCanvas.Right  := rectCanvas.Left + rectBitmap.Right;

  { Set the canvas' brush color to the
    background color of the form }
  Canvas.Brush.Color := Color;

  { Copy the contents of the TImage to the form's
    canvas replacing the color clSilver with the
    canvas' brush color }
  Canvas.BrushCopy(rectCanvas,Image.Picture.Bitmap,
                   rectBitmap,clSilver);
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  { Every time the form paints itself, redraw this image }
  PaintImageTransparent(Image1);
end;
```

Figure 2: The custom *PaintImageTransparent* procedure and the *Paint* procedure that calls it.

The result is a form that looks like the one shown in Figure 3. As you can see, using this approach will cause your applications to display correctly, even when users start trying out the new Windows 95 color schemes. Δ

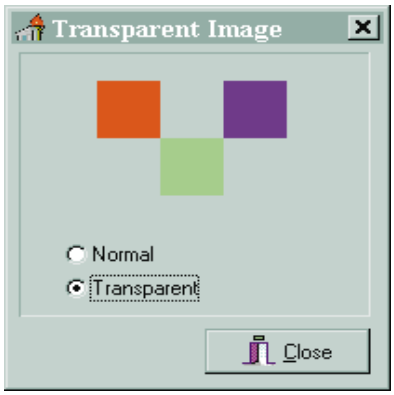


Figure 3: The gray color of the image is now transparent.

The demonstration project referenced in this article is available on the Delphi Informant Works CD located in INFORM\95\OCT\KN9510.

Kenn Nesbitt is an independent Windows and database consultant, formerly with Microsoft Consulting Services. He is a Contributing Writer to *Data Based Advisor*, the Visual Basic columnist for *Access/VB Advisor* magazine, a regular contributor to the German magazine *Office and Database*, and co-author of *Power Shortcuts: Paradox for Windows*. You can e-mail Kenn at kennn@netcom.com, or CompuServe 76100,57.

BY DOUGLAS HORN

Component Create

Custom Components Are Closer, but Still More Than a Click Away

Component Create version 1.0 by Potomac Document Software helps automate the process of designing Delphi custom components. Component Create helps walk users through the process of creating and coding various component parameters. While it still leaves plenty of work to the developer, Component Create does provide a solid framework to build on. Component Create is one of the first Delphi-specific tools to hit the market. Unfortunately, it is not as thoroughly integrated with Delphi as users might wish.

Creating Components

The usual way to create a custom component is to use the Delphi Component Expert by selecting **File | New Component**. The Component Expert requests a Class Name, Ancestor Type, and Palette Page for the new component, then creates a very rough framework for the new component. It adds empty **private**, **protected**, **public**, and **published** statements as Delphi does with any new object, but beyond these, the Component Expert adds fewer than ten lines of useful code (see Figure 1). From this point, users are on their own.

```
unit Unit1;

interface

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

type
  Tcomp = class(TCustomCheckBox)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [Tcomp]);
end;

end.
```

Figure 1: The Pascal code generated by Delphi's native Component Expert automates few component tasks, leaving the component writer with the proverbial blank page.

Component Create is a much more "expert" Expert. To begin with, where the Component Expert lists only the names of various component classes to use as the ancestor — or parent — class, Component Create includes somewhat more descriptive explanations, although they could still be improved (see Figure 2). Component Create also provides a hook into Delphi Help's component class index so that users can get more details. And even after the parent class is selected, Component Create allows users to easily choose a new one.

Once the user selects a parent class for the new component, Component Create presents its main interface: a five-page notebook that allows users to access any of the component's parameters. Each of the notebook's pages — Properties, Methods, Events, and Variables — allow the user to add, delete, and modify the corresponding parameters. In addition, the Main Page controls general information such as parent class, unit name, and the palette onto which the new component will be installed. In other words, the Main Page handles all the same information as the Component Expert, as well as various fields for descriptions, copyright statements, and units to include. (All standard units are added automatically. Advanced users may wish to remove unnecessary units, which can lead to bloated component code.)

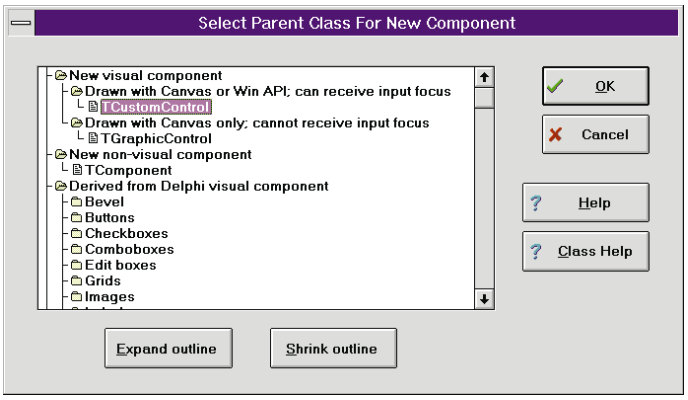


Figure 2: Component Create's Select Parent Class dialog box offers descriptions of various component classes to help users make the most appropriate choice.

Component Create allows users to add or delete component parameters at any time. This eliminates the need to, say, define all properties before moving on to methods. And because a list of the parameters is always displayed on each page, it is difficult to accidentally forget an important variable or property.

The property, method, and variable lists that Component Create displays do not, however, show inherited properties. Inherited events can be shown and overridden, but otherwise, if the user wishes to use or override one of the inherited methods or variables, the parameter must first be added to the list (see Figure 3)

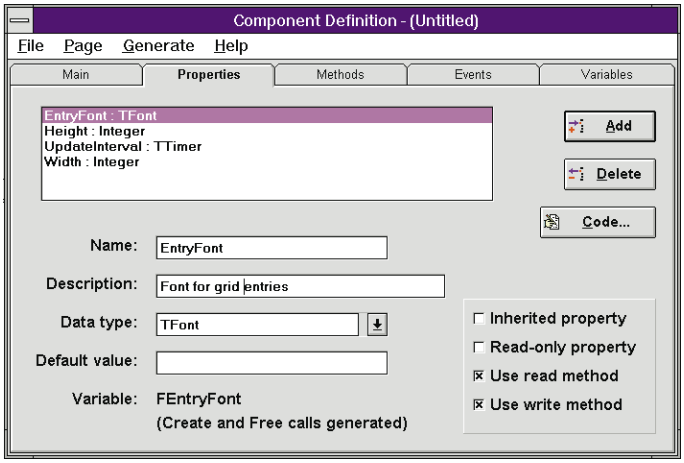


Figure 3: Component Create's main interface is a tabbed-notebook with Main, Properties, Methods, Events, and Variables Pages. Though the component will include all parameters of its parent class, only those which have been added to the componeAnt will be listed on these pages.

Users can set the default value of inherited properties, but cannot alter their behavior. This is not a limitation of Component Create, but of the component creation process itself. Component Create just keeps users from trying to do the impossible. The general rule is: If you need to alter or remove an inherited property, start further up the inheritance tree.

Every parameter a user can add also requests a description. These descriptions are used to comment the component code. This is a nice feature, as it keeps the completed code easy to follow.

Excessive commenting can be disabled simply by leaving the description field blank.

Once the user adds a property or variable, the data type is also required. This can be set to any possible type. One of Component Create's most convenient features is that it automatically handles initialization and destruction of object types such as *TBitmap* or *TFont*. This is a real headache-saver, as failing to properly destroy any aspect of these objects can bring on a Windows general protection fault.

Once a property, method, or event has been added and defined, the code pertaining to it can be edited via Component Create's code editor. This is where the real work begins. Though Component Create adds code comments to suggest what types of functions a given event handler should include (see Figure 4), it does not help generate that code.

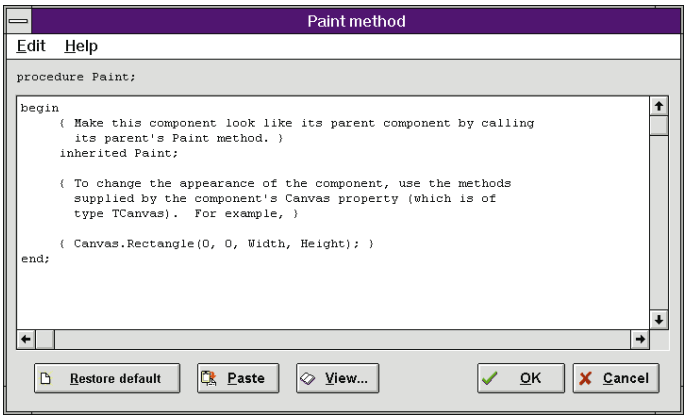


Figure 4: Component Create's code editor provides comments to help users understand where to add their event handlers and what types of functions need to be addressed.

Of course, generating code is the developer's responsibility, but potential Component Create users should be aware that this is the boundary of the program's assistance. The full extent of Component Create's assistance is setting up an outline for the new component's code based on the properties, methods, and events that the user specifies. Anyone who thinks that Component Create is going to hold their hand through the entire process of creating a new component probably owns a lot of Florida swampland and perhaps even the Brooklyn Bridge.

Odds are, Component Create users are going to end up editing a lot of code. To do this, they must use Component Create's code editor. Compared to Delphi's robust code editor, Component Create's leaves quite a lot to be desired. First, keywords and comments are not automatically formatted as they are in Delphi's code editor. Also, Component Create's code editor does not allow users to set preferences, such as typeface and size.

On the other hand, Component Create's code editor does provide extended commenting in many circumstances to help explain what types of programming tasks should be addressed by various event handlers. The code editor also allows users to open existing .PAS files to cut and paste blocks of code into the component definition.

One final characteristic of the code editor is that it only displays code from the section currently being edited. Some users will find this keeps them focused and protects them from being overwhelmed by the volume of code a component may require. Others, however, will become frustrated with the inability to jump to various sections of code as new ideas strike them.

Completing Components

Component Create does an admirable job of automating key stages of the component creation process. This is especially true at the early stages of component definition. Unfortunately, towards the end of the process, Component Create tends to lose some of its steam. While users can continue to work in Component Create almost to the stage of installing the completed component on Delphi's Component Palette, few will choose to do so. The bulk of Component Create users are more likely to define all component parameters (properties, methods, events, and variables), let Component Create generate a component outline, and then switch to back to Delphi for the real programming.

The main problem here is Component Create's code editor. As mentioned above, it does not allow users to easily skip from one block of code to another. Also, some procedures, such as *AutoInitialize* and *AutoDestroy*, it hides completely. More frustrating still, the code editor makes no provision for syntax checking. So anyone who actually tries to develop a component from start to finish in Component Create is likely to have their component file opened in Delphi anyway while trying to install it to the Component Palette — most likely because they forgot a semicolon somewhere along the line.

Performing the last half of the component coding from Delphi saves a lot of time and frustration by taking advantage of Delphi's superior code editor. As Component Create does not naturally save its files in Delphi format, users must first generate a Pascal .PAS file using Component Create's **Generate | Code(.PAS)** menu command. (Component Create also saves files in its own component definition (.CD) format) This .PAS file can then be opened and edited in the Delphi code editor.

This lack of integration is the only serious disappointment in Component Create. Aside from linking to Delphi's help file, Component Create provides no communication with Delphi whatsoever. New components must be saved to a .PAS file to be either edited or installed onto the Delphi Component Palette. Component create also offers no direct support for changing the bitmap image used to represent the component. Future versions would definitely benefit from tighter integration with Delphi. For example, using the Delphi code editor would eliminate the shortcomings of the current editor. And the users would certainly appreciate a way to check for errors as they went along.

But even developers who only use Component Create to generate component outlines will find that they save time and effort. Though it should be simple, the process of defining and outlining all the various parameters of a component can often be more frustrating than writing the actual event handlers themselves.

Other Considerations

In general Component Create is a well put-together program. It installs easily, allowing the user to select what directory and Windows program group to use. The default program group is Delphi, which keeps Component Create in the same (albeit crowded) group as all the other Delphi utilities.

Component Create's documentation is a bit sparse, and its robotic tone brings to mind manuals created using an automated documentation generator. At thirty-two small pages, Component Create's manual is half the length of the manual for a typical VCR. (But then it's a lot easier to program Delphi components than it is to program a VCR.) The on-line help file is an exact clone of the paper manual.

Needless to say, then, that Component Create's documentation does not delve into the hidden secrets of creating Delphi components. Besides a straight explanation of Component Create's controls, it does touch on a few simple component development issues, such as creating components with sub-components. On the whole, however, this is not the program to buy to learn how to create components. While Component Create will be beneficial to those with any level of component writing experience, those who have a strong grasp of component writing techniques will receive the most satisfaction.

Component Create does include five pre-written component definitions (.CD files) for users to study. These offer some insight into Component Create basics, but their use is limited. Again, the sample components included with Delphi (C:\DELPHI\SOURCE\SAMPLES) are better learning aids.

Conclusion

But Component Create never promises to teach users how to program components, and this is a bit unfair to expect of it. It does deliver on what it promises — extending automatic component code generation to provide component developers with a fuller framework on which to operate. Savvy users will find Component Create most adept at creating component outlines that help to ease the busywork and assure that nothing simple is left out. This is valuable stuff, but from that point, users are better off to switch to Delphi to handle the *real* programming. Δ

INFORMANT FACT FILE

Component Create version 1.0 is a code generator that helps Delphi developers create their own components. Though it can handle many phases of component generation, Component Create is most useful in creating outlines that can be fleshed out in Delphi's code editor. The program would be improved by tighter integration with Delphi, but will nevertheless be a useful tool to many component writers.

Potomac Document Software, Inc.
P.O. Box 33146
Washington, DC 20033-0146
Phone: (800) 628-5524
Fax: (202) 244-9065
Price: US$179

Douglas Horn is a freelance writer and computer consultant in Seattle, WA. He specializes in multilingual applications, particularly those using Japanese and other Asian languages. He can be reached via CompuServe at 71242,2371.

table of CONTENTS

BY BILL TODD

[re]Structure

Complete Programmatic Control of Paradox Tables

If you use Paradox tables in your Delphi applications, you've probably noticed that Delphi's ability to create Paradox tables or change their structure is very limited. You can create a table, and define the fields and indices to include in the table's structure using the *CreateTable* method of the *TTable* class. However, there is no way to specify any of the table's other properties, such as validity checks, table lookups, referential integrity, or passwords.

What You Couldn't Do

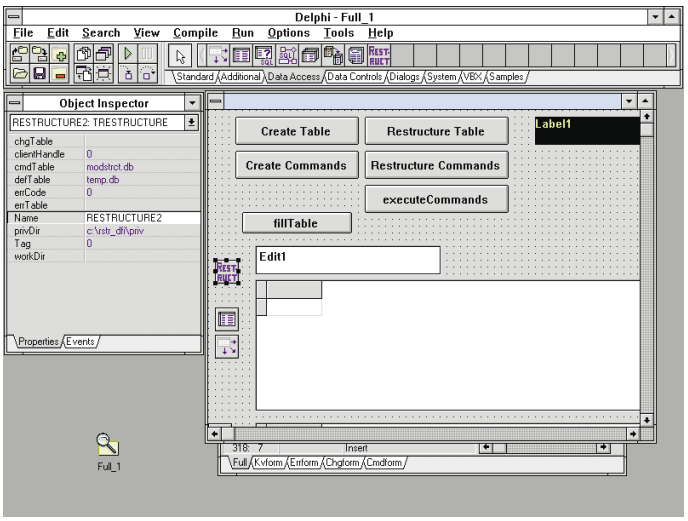
The same limitations apply to changing the structure of a table. You can add and drop fields and indices, but that is all. You can't move a field to a different location in the table's structure, nor can you add, delete, or change any of the other properties unique to Paradox tables.

These limitations can be very annoying in many situations. For example, suppose you need to send users an update to your application. And as part of the update process you need to change the structure of one or more tables without disturbing the data they contain.

You Now Can

[re]Structure from TrayMar Software removes these limitations. Using [re]Structure you can:

- Add, drop, move, or change fields
- Add, drop, and modify valchecks
- Add, drop, and modify table lookups
- Add, drop, and modify primary and secondary indexes
- Add, drop, and modify referential integrity
- Add, drop, and modify master and auxiliary passwords
- Modify the table level (version)
- Modify the table's language driver



[re]Structure comes complete with a Delphi demonstration project.

All the options for these operations are fully supported. For example, you can specify Help and Fill or Fill All Corresponding when you add or change a table lookup. Also, all the master and auxiliary password options are available when you add or change passwords. You can even specify the Pack option when you restructure a table.

Installation

The installation process consists of two steps. First you must copy the RESTRUCT.DLL file to your Windows directory (i.e. the directory that will be your working directory when your program is running, or to a directory on your DOS path). Although you can call the RESTRUCT.DLL functions directly, it's easier to use the Restructure component by adding RESTRUCT.PAS to the Visual Component Library. By default the component appears on the Data Access page of the Component Palette.

Using the DLL

For all of its power, RESTRUCT.DLL is surprisingly easy to use. You simply fill a command table with all of the operations you want performed and call the *DoRestruct* method. This "command table" approach has a number of advantages. First, you can include different operations on different tables in a single command table. For example, you could create two tables and define a referential integrity relationship between them in a single command file. You can also create as many command tables as you need and use them at will since you can pass the command table name as a parameter to the *DoRestruct* method.

Another powerful feature of the command table is that you can leave the name of the table an operation should be performed on blank and specify it dynamically before you call *DoRestruct*. This means you can create a command table and use it on any table you wish. If errors occur during any operation, *DoRestruct* creates an error table in the user's private directory that describes them.

To make it easy to build command tables, the Restructure component includes a method that enumerates information about an existing table to a command table. The Operation field in the command table is filled with a dummy value that does nothing if the command table is executed. All you have to do is change the Operation field in the command table for those fields you want to change in the target table.

Safe and Effective

The author of RESTRUCT.DLL was a member of the Paradox for Windows development team for all versions from 1.0 through 5.0, and he wrote the DLL using documented Borland Database Engine (BDE) functions. Since all table manipulations are done with BDE calls, you can be confident that [re]Structure is safe to use.

If you are tired of tedious work-arounds that result in poor performance when you need to restructure tables, [re]Structure is the tool you've been waiting for. Δ

INFORMANT FACT FILE

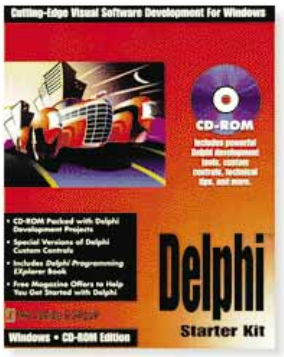
[re]Structure, from TrayMar Software, removes the limitations associated with creating or changing the structure of Paradox tables. It allows developers to add, drop or modify valchecks, table lookups, primary and secondary indexes, referential integrity, and master and auxiliary passwords. It also allows for adding, dropping, moving or changing a field, and modifying the table level and language driver.

TrayMar Software
1254 Middlefield Road
Palo Alto, CA 94301-3346
Phone: (415) 323-0910
CompuServe: 75564,1042
Price: US$149.95

Bill Todd is President of The Database Group, Inc., a Paradox consulting firm based near Phoenix. Bill is co-author of *Creating Paradox for Windows Applications* (New Riders Publishing, 1995), *Paradox for Windows Power Programming* (Que Corporation, 1994), and *Delphi: A Developer's Guide* (M&T Books, 1995). He is also a member of Team Borland supporting Paradox on CompuServe and a speaker at all Borland database conferences. Bill can be reached at (602) 802-0178, or on CompuServe at 71333,2146.

TEXTFILE

For a Few Dollars More: The Delphi Starter Kit



Several months after publishing their excellent book, *Delphi Programming EXplorer*, the Coriolis Group released the *Delphi Starter Kit*. The *Kit* is the same book, with a CD-ROM instead of a diskette. The CD-ROM contains all the diskette's source code and utility programs. The rest of the CD-ROM is filled with a collection of Delphi-related material of wildly-varying — and generally disappointing — quality. Buyers may want to think twice before paying a few dollars more for the *Kit*.

When I reviewed *Delphi Programming EXplorer* [in the August 1995 issue of *Delphi Informant*] I was genuinely pleased with the job that Jeff Duntemann, Jim Mischel, and Don Taylor had done. The book is excellent, presenting Delphi in a lively and highly readable manner. Source code for the book's examples is included on a 3.5" diskette, along with versions of HelpGen's help authoring utility and EarthTrek's Delphi Conversion Assistant for Visual Basic source code conversion.

I hoped that the new *Starter Kit* would build on the book's strong start. The *Kit* certainly looks promising. The box lists "powerful Delphi development tools", "special versions of Delphi custom controls", "technical tips", and a "CD-ROM Packed with Delphi Development Projects". Instead, I found promotional offers, VBXes, and buggy source code written by amateurs. I even found material that had nothing at all to do with Delphi.

The CD-ROM is organized into several directories. The first one contains the same source code as the book's diskette, and the HelpGen utility. The source code files are not compressed. However, if you want to compile them, you'll have to copy them to your hard disk, since Delphi will complain that it can't write its working files to your CD-ROM drive.

Next is a copy of Adobe's Acrobat reader. You don't need it, since it also comes with Delphi. The "Inform" directory contains a copy of the premiere issue of *Delphi Informant* magazine, in Acrobat format. A subdirectory contains all the source code and resource files for that issue, in their native formats. There is also an Acrobat version of some Delphi-related articles from the June/July '95 issue of the Coriolis Group's *PC Techniques* magazine. Unfortunately, the source code is not given separately. If you try to copy it to the Clipboard, Acrobat ignores the column formatting, giving you pureed text. Using the Acrobat reader is very slow, even on a beefy 486 DX50 with a graphics coprocessor and 2X CD-ROM drive. Thumbing through hard copies of the magazines at a bookstore is definitely more fun.

A directory named "Demos" contains demo versions of four commercial Delphi tools. First is the same EarthTrek Visual Basic Conversion Assistant that came on the original book's diskette. Next is the March '95 demonstration version of the Component Toolbox 2.0, Standard Edition, from Gamesman, Inc. of Canada. This consists of 21 assorted VBX controls, without documentation. The demo program could not find one of its VBXes, and left the cursor the wrong shape.

Then there is a Trial Edition of Eschalon Setup 1.0, a utility for installing applications. If you want to use it, you must register it. Finally, there is a beta version of Shoreline's VisualPros widgets. Their documentation says "Sorry if there doesn't seem to be much meat in this beta release." Some of the widgets didn't work well. Others needed DLLs that they couldn't find.

Rummaging around in the "Controls" directory is like visiting a flea market. It contains 28 subdirectories of various Delphi controls, including some Delphi DLLs, a pre-release version of a report printer, a 3-D rendering package that "requires a little knowledge of Object Pascal to be effective", three different support packages for WINSOCK.DLL, and a crippled version of TurboPower's Orpheus Data Entry VCLs.

The Orpheus set is the most professional of the controls. It includes the source code to the demo, but not to the controls, and the controls run only in the Delphi IDE. The same trial package can be downloaded from their BBS, CompuServe, or the Internet.

The quality of the rest of the controls is much worse. Various comments show that many of these are first efforts at programming in Delphi, and some are first efforts in programming in any language at all. Some come with no source code. Others have source code, but no project files. Some have source code for demo programs, but not for their VCLs. Some don't even have demo programs. Many contain un-needed backup and temporary files. Some projects generated compile-time errors. Some of the programs compiled, but didn't do what they advertised. Some tried, but crashed repeatedly. One even crashed my system.

To sum up, the Coriolis *Group's Delphi Starter Kit* contains the same book and source code as their outstanding *Delphi Programming EXplorer* on a CD-ROM. If there is something else listed here that you already know you want, then the convenience of getting it on a CD-ROM may be worth the few extra dollars. Otherwise, get the diskette version of the book.
— *Tim Feldman*

Delphi Starter Kit
Coriolis Group Books,
7339 East Acoma Drive,
Suite 7, Scottsdale, AZ 85260;
(800) 410-0192, or
(602) 483-0192.
ISBN: 1-883577-55-1
Price: US$44.99
627 pages, CD-ROM

table of CONTENTS