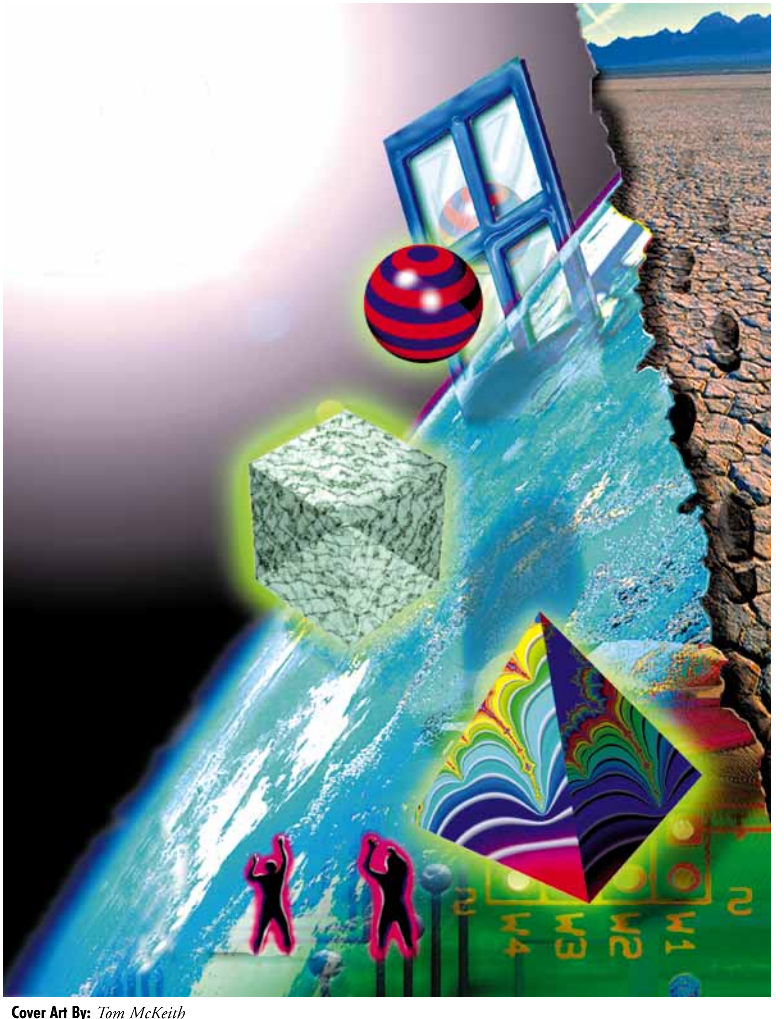
January 1996 - Volume 2, Number 1



Cover Art By: *Tom McKeith*

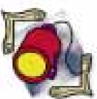
Great Journeys, Single Steps

The State of the Object Art

ON THE COVER

7 **Great Journeys, Single Steps** — Gary Entsminger
Object-oriented programming is a well-accepted approach to software development that's reached its apotheosis with Delphi. But it wasn't always so. Mr Entsminger kicks off the new year with a new series that explores the state of OOP. This first article recaps the history and development of OOP, and compares the object models that dominate the current market: Windows/Visual Basic and Delphi.

FEATURES



13 **Informant Spotlight** — John O'Connell
Most of us have used Windows' DDE (Dynamic Data Exchange) at some point to automate inter-application tasks. When it comes to that first Delphi implementation, however, there's plenty to learn. Which is where Mr O'Connell comes in with his example-laden introduction to the DDEServerConv and DDEServerItem components.

24 **Inside Object Pascal** — Kevin Bluck
The *MessageDlg* function is handy, and meets many needs as it comes. However, it does have its limitations. For example, it can be unwittingly bypassed with a single key stroke. In response, Mr Bluck presents us with a handful of enhanced dialog box routines based on the undocumented Object Pascal function, *CreateMessageDlg*.

31 **DBNavigator** — Cary Jensen, Ph.D.
Dr Jensen takes his popular "DBNavigator" column into the new year with the first of a two-part series. The topic is filtering Paradox data, and Cary begins by describing two techniques based on the Table component. One links related tables using a secondary index, the second demonstrates how to restrict a view to a specific range.

36 **On the Net** — Carol Boosembark
With a plethora of tools and information, the Internet is a resource no Object Pascal programmer should be without. But where to start? Ms Boosembark answers with her guided tour of the Internet from a Delphi developer's perspective. This month the focus is on Borland's World Wide Web site, "Borland Online."

38 **At Your Fingertips** — David Rippy
Start off the year with some great tips! This month Mr Rippy shows us how to perform locates on non-indexed fields, determine the record number in a Paradox table, create a two-line message using the *MessageDlg* function, and move to a specific property quickly in the Delphi Object Inspector.

REVIEWS

40 **Orpheus** — Product review by Cary Jensen, Ph.D.
There are two kinds of third-party Delphi tools: VCLs and — everything else. TurboPower's Orpheus falls into the first category, and according to Dr Jensen, "qualifies as one of the best bargains in the Delphi add-on market."

43 **RoboHELP 95** — Product review by Gary Entsminger
There's no getting around it, building a Windows Help system isn't a lot of fun. However, there are some tools that allay the tedium. Outstanding among them is Blue Sky Software's RoboHELP. Mr Entsminger checks out the Windows 95 incarnation.

47 ***Delphi How-To***
Book review by Larry Clark

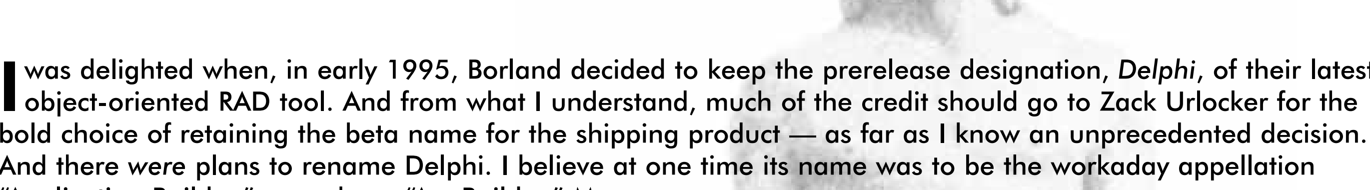
48 ***Secrets of Consulting***
Book review by Jeff Sims

DEPARTMENTS

2 **Editorial**
3 **Delphi Tools**
5 **Newsline**

Symposium

"Over the years, I have made several pilgrimages to Delphi."
— Gore Vidal, *Palimpsest*



I was delighted when, in early 1995, Borland decided to keep the prerelease designation, *Delphi*, of their latest object-oriented RAD tool. And from what I understand, much of the credit should go to Zack Urlocker for the bold choice of retaining the beta name for the shipping product — as far as I know an unprecedented decision. And there *were* plans to rename Delphi. I believe at one time its name was to be the workaday appellation "Application Builder," or perhaps "AppBuilder." Mmmm — sexy.

One wholly pragmatic reason that Delphi is a great choice for a name is that it's short. Six letters and you're done. And no "for Windows" appendage is necessary. There never was, and never will be, a DOS version.

Delphi is also an extraordinarily evocative word, and it turns up in the damnedest places. For example, in his recently published memoirs, *Palimpsest* [Random House, 1995], Gore Vidal tells us the eponymous Greek city has served for him as a sort of spiritual touchstone.

I've been exploring Delphi and its meaning in history. And like the application environment, Delphi the word-city-idea, holds up well under scrutiny. We know it best as the site of the main temple to Apollo, where in Classical Greece, oracles revealed enigmatic prophecy. The Delphic oracle, always female, spoke for Apollo in the first person like a modern "channeler." Eventually, Delphi became the ultimate source of approval and advice.

It was usual to consult the oracle at Delphi before trying to establish a new colony. This was not simply a religious fortification against unknown dangers. Delphi had now attained preeminence among all the Greek holy sites, and as the oracle was constantly being consulted by enquirers from every part of the Greek world — and indeed sometimes by 'barbarians' too — the Delphic priest acquired a great deal of information ... not to mention considerable political influence. — *The Greeks*, H. D. F. Kitto [Penguin Books, 1957]

Any Borlandophile has to smile at the "barbarian" reference. However, the best fit for the barbarian title in this context would be Microsoft. It's well established that Microsoft wastes no opportunity to grill anyone on the topic of Delphi. I know of one ex-Borlander who, once hired by a Microsoft subsidiary, was immediately called to a meeting where the first question was "So what do you know about Delphi?"

Then there is Apollo, the god of rationality, logic, the will to create and build. He is the embodiment of what Camille Paglia calls the "western eye." "Hierarchically tidy and task oriented," Apollo symbolizes order, "elegance," and "cultivated abstraction," ideas of Egyptian origin that reached their zenith in Greece. He also sounds like a good representative for Delphi, the programming environment.

But there is another side to our Greek/European/Western culture. Paglia joins Joseph Campbell and Friedrich Nietzsche in selecting Apollo as the antithesis of Dionysus. And I'm not talking about the popular conception of Dionysus as a drunken wine god, but in his true primeval aspect as the god of abandon, chaos, and madness. Together, they represent the paradoxical nature of our culture — its extraordinary beauty, achievements, repulsiveness, and destructiveness. I doubt that we'll see a software package named Dionysus 1.0.

Apollo also stands at another great cultural nexus, the prehistoric transition to male gods from earlier female gods, in this case Apollo over the goddesses residing at Delphi before him.

Delphi, holiest spot of the ancient Mediterranean, was once dedicated to female deities, as the priestess recalls at the opening of Aeschylus' Eumenides. ... The Delphic oracle was called the Pythia (or Pythoness) after the giant serpent Pytho, slain by invading Apollo. ... The oracle was Apollo's high priestess and spoke for him. Pilgrims, royal and lowly, arrived at Delphi with questions and left with cryptic replies. — Camille Paglia, *Sexual Personae* [Vintage Books, 1990].

All of which begs the question: Why did Borland choose Athena's visage to represent Delphi? I have no idea. And more to the point, what does any of this have to do with Delphi programming? Nothing. But having interacted with many a Delphi developer, I know there are few who will resent this digression into word research. There must always be a life away from computers and programming, and etymology is good for the soul. Again, from Kitto's *The Greeks*:

There is great significance in the religious legend that for three months in the year Apollo left Delphi and Dionysus took his place.

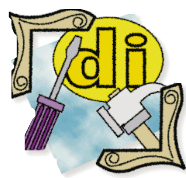
Yeah, Delphi is a great name.

Jerry Coffey, Editor-in-Chief

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

TOOLS

New Products and Solutions



Multi-Edit for Windows Version 7.01

American Cybernetics, Inc. of Tempe, AZ has released *Multi-Edit for Windows*, version 7.01, an upgrade to their programmer's text editor. In addition to the Hex mode editing and ruler, this version features new tools designed to streamline repetitive programming tasks.

The new edit-on-the-fly template system is designed to help developers avoid typing frequently used code. This allows programmers to modify templates as they work, eliminating the need to recompile or exit the editor.

Multi-Edit features Collapsible editing. Developers can view selected segments of code and collapse other sections while scanning through large blocks of code. Multi-Edit also supports the long filename feature in Windows 95 and Windows NT.

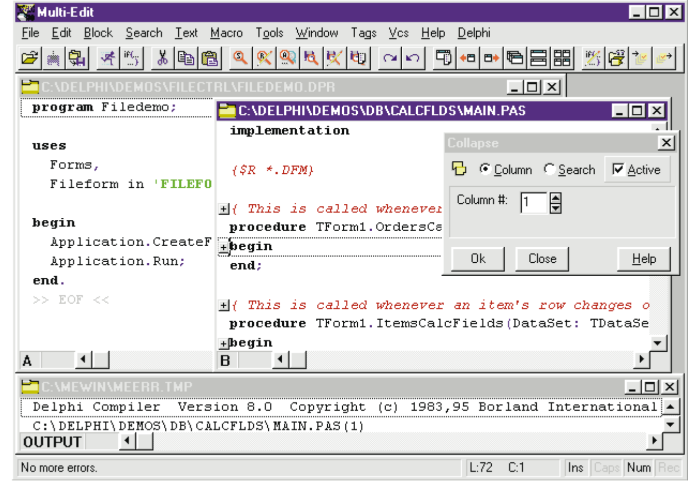
Version 7.01 has background compiling (allowing the programmer to continue editing while a compile is in progress), modeless search/replace dialogs, versions support, and syntax highlighting. In addition, the compiler and color setup have been enhanced.

Registered users of Multi-Edit for Windows, version 7.0 will automatically receive an upgrade free of charge.

Price: US$199.

Contact: American Cybernetics, 1830 W. University Drive, Suite 112, Tempe, AZ 85281

Phone: (602) 968-1945
Fax: (602) 966-1654
E-Mail: Internet: tech@amcyber.com



Starfish Ships Sidekick 95 Deluxe

Starfish Software has released *Sidekick 95 Deluxe*, a multimedia CD-ROM that includes Sidekick 95, Dashboard 95, America Online software, two interactive organizational videos, Sidekick Companions, and electronic user guides.

Sidekick 95 Deluxe is priced at US$79.95. For more information call (800) 765-7839.

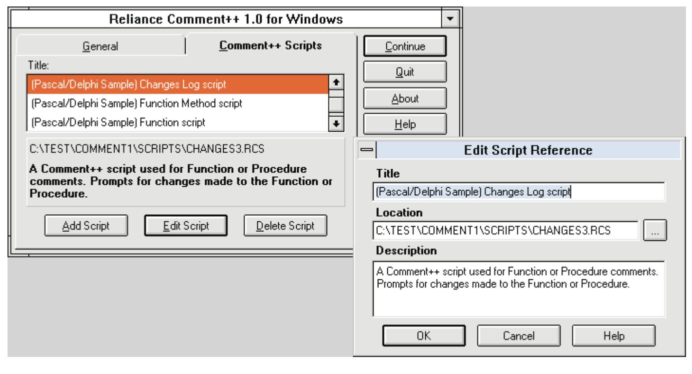
Reliance's Comment++ V1.0 for Windows

Reliance Corporation of Penn Valley, CA has released *Comment++ V1.0 for Windows*. Software developers using Borland's Delphi or C++, Microsoft's Visual C++ or Visual Basic, Premia Corporation's CodeWright, or Symantec's C++ can use Comment++ to automatically insert and maintain comments in their source code.

Comment++ simplifies the task of commenting source code by eliminating the need to recreate comment blocks for every function, procedure, or method in a source code file. Instead, Comment++ uses comment templates stored in scripts that can be invoked and inserted directly into the source code file with one keystroke.

Software developers can use the Comment++ script language to create and layout interactive comments that prompt for information before being inserted into the source code file. The script language, designed specifically for Comment++, allows developers to design custom comments.

Test drives of Comment++ can be downloaded from Reliance Corporation's Web site listed below, or the Borland C++ forum on CompuServe.

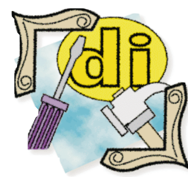


Price: US$29.95 per copy, quotes for site licenses are available upon request.

Contact: Reliance Corporation, 18905 Hummingbird Dr., Penn Valley, CA 95946

Phone: (800) 432-2118 or (916) 432-3285
Fax: (916) 432-4487
E-Mail: CIS: 72223,1240
Web Site: http: //www.reliancecorp.com

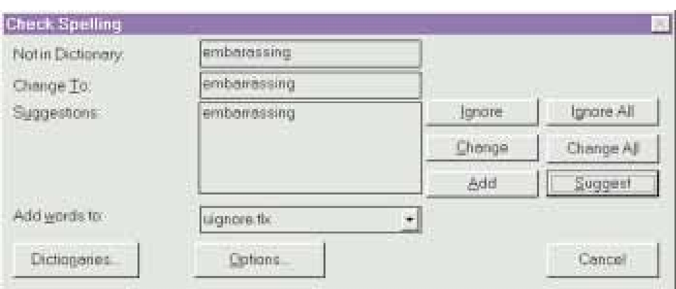
New Products and Solutions



Wintertree Software Ships Sentry Spelling-Checker Engine

Wintertree Software of Nepean, Ontario, Canada has released a 32-bit version of the ***Sentry Spelling-Checker Engine*** (SSCE). SSCE is implemented as a DLL, enabling it to be called from any DLL-capable language and development environment, including Delphi, Paradox, C, C++, Visual Basic, Access, FoxPro, and PowerBuilder.

SSCE has an application program interface that allows other applications to check the spelling of words, locate suggested replacements for misspelled words, and update user dictionaries. Redistributable American- and British-English dictionaries, each containing about 100,000 words, are included with the engine. In addition, French, Italian, German, and Spanish dictionaries are available.



The SSCE can automatically or conditionally replace a misspelled word with its correct spelling. To support this feature, a dictionary containing hundreds of common misspellings and their replacements is included.

SSCE is available as 16- and 32-bit Windows SDKs that allow Microsoft Windows and Windows NT developers to add a spelling checker to their applications. The Windows SDKs also include the SSCE Dialogs DLL. SSCE is also available in ANSI C source code form, enabling the engine to be used on any computer and operating system that supports a C compiler.

Price: US$169, English version of SSCE Windows SDK; US$199, French, Italian, German, and Spanish versions.

Contact: Wintertree Software Inc., 69 Beddington Ave., Nepean, Ontario, Canada, K2J 3N4

Phone: (613) 825-6271
Fax: (613) 825-5521
Web Site: http: //fox.nstn.ca/~wsi/

Delphi Tools CD-ROM Now Shipping

EMS Professional Shareware is now shipping the November 1995 version of its ***Delphi Utility Library***, containing over 250 public domain and shareware tools selected especially for programmers. The library includes a program that allows users to search for a utility by the product name, vendor, type, release date, or description. The Delphi Utility Library costs US$59.50 (CD-ROM) plus shipping and handling, and has a 30-day money-back guarantee. The next update is due to ship in February. For more information contact EMS by phone (301) 924-3594, e-mail Internet: ems@wdn.com, or visit their Web site at http://www.wdn.com/ems.

Inner Media Releases Version 3 of DynaZIP

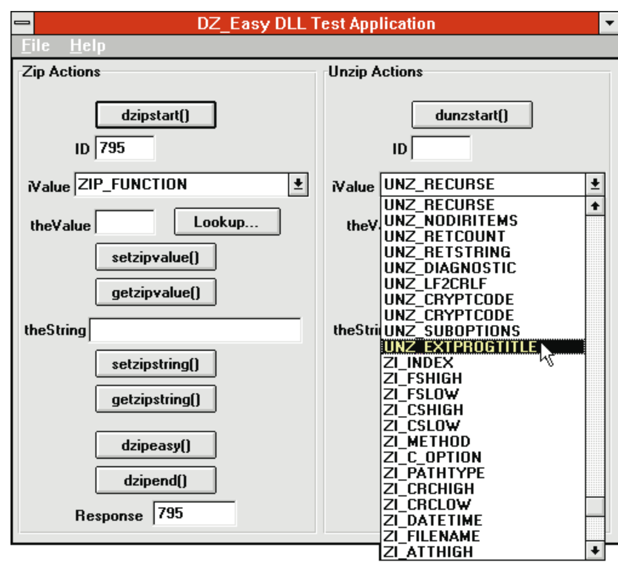
Inner Media, Inc., of Hollis, NH has released ***DynaZIP 3.0*** featuring OCX, VBX, and DLL support. With DynaZIP, developers can add the ability to read, test, create, modify, and write industry standard .ZIP files to their programs — without having to "shell out" to a DOS session. The royalty-free DynaZIP DLLs can read and write files compatible with the latest version of PKZIP from PKWare, Inc. Because DynaZIP has built-in ZIP encoding and decoding logic, PKZIP and PKUNZIP are not used and need not be present on the target machine.

Version 3.0 is available in 16- and 32-bit versions for Windows, Windows 95, and Windows NT. It includes a new DZ_EASY interface, improved overall speed, and increased support for creating Windows-hosted self-extracting .ZIP files.

DynaZIP libraries are compatible with various development languages and platforms, including Delphi, Pascal, Paradox for Windows, and more.

DynaZIP's API provides information about the .ZIP file items, dual progress monitor callback capabilities, and comprehensive status and error reporting.

DynaZIP also includes a Windows .ZIP shell program with full source code in both C and VB, two diagnostic test tools, a Windows help file, coding examples, and printed documentation.

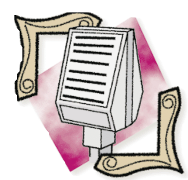


Price: US$249 per developer station, DynaZIP-32 is US$299. Registered owners of DynaZIP and DynaZIP VBX can upgrade to DynaZIP-16 for US$69, and registered owners of DynaZIP NT can upgrade to DynaZIP-32 for US$69.

Contact: Inner Media, Inc., 60 Plain Road, Hollis, NH 03049

Phone: (603) 465-3216, or (800) 962-2949 in the US
Fax: (603) 465-7195
CompuServe: 70444,31

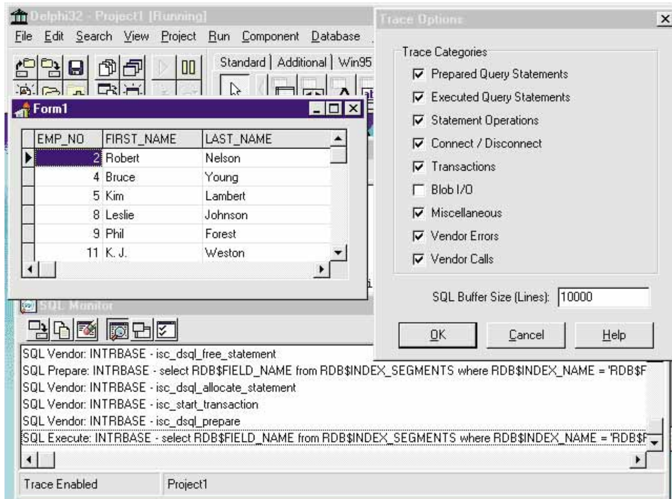
January 1996



Borland Displays Delphi Client/Server Suite 2.0

Las Vegas, NV — Borland International Inc. previewed Delphi Client/Server Suite 2.0, the 32-bit version of Delphi Client/Server for Windows 95 and Windows NT, at Comdex Fall 95.

Delphi Client/Server Suite 2.0 features SQL Explorer, a visual tool that allows team developers to browse and modify server-specific metadata for Oracle, Sybase, InterBase, Informix, and Microsoft SQL Server, including stored procedure definitions, triggers, and index descriptions.



Delphi Client/Server Suite 2.0 also features SQL Monitor, an integrated tool that assists developers with testing, debugging, and performance tuning for SQL queries. Also provided are SQL Links, 32-bit native drivers for Oracle, Sybase, Informix, InterBase, and Microsoft SQL Server, and an unlimited deployment license for these drivers. In addition, this version of Delphi includes Intersolv's PVCS Version Control.

Delphi Client/Server Suite 2.0 will also include InterBase 4.0 NT, a 32-bit version of ReportSmith SQL, and an expanded Open Tools API for integrating third party CASE tools such as LBMS' System Engineer, Popkin's System Architect, SDP Technologies' S-Designor, and CSA's SilverRun.

Delphi Client/Server Suite 2.0 is scheduled for release in the first quarter of 1996, and will be sent as part of the maintenance license agreement for Delphi Client/Server Team Solutions users. Pre-release versions are available through Borland Connections at (800) 353-2211.

Additional information is available on Borland's Web site at http://www.borland.com.

EMail World & Internet Expo Heads to San Jose

EMail World, Web World, & Internet Expo, scheduled for February 19-21, 1996 in San Jose, CA, will showcase advances in Internet, e-mail, and World Wide Web technologies with over 500 exhibits. This year's event has added a new track focusing solely on the issues of electronic publishing. Industry experts to speak at the conference include John Perry Barlow, co-founder of Electronic Frontier Foundation; Marc Andreesen, V.P. Technology, Netscape Communications Corp.; John Patrick, V.P. Internet Applications, IBM; Tim Koogle, President and CEO, Yahoo; and many more.

For more information, call DCI at (508) 470-3880 or visit their Web site at http: //www.DClexpo.com.

Borland to Deliver Tools for Java, Sun's Internet Programming Language

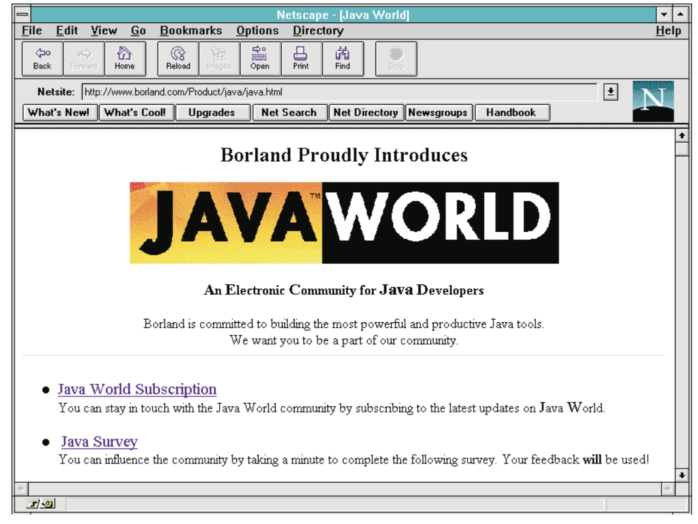
Scotts Valley, CA — Borland International Inc. has announced it will license Sun Microsystems' Java programming language and will develop tools based on the language. This agreement, presented at the Java Conference at Sun Microsystems, Inc., states that Borland will build a rapid application development (RAD) environment for creating Java applications. The product, code-named Latte, will be developed in Java.

According to Borland, Java's platform-independent capability makes it an attractive distributed computing development environment, and there will be a need for rapid application development tools.

Borland is planning to release Latte in stages, with the first commercial release scheduled to ship in the first half of 1996. Borland's product development will focus on the following areas: visual RAD tools to increase productivity for developing Internet; Web applications; object-oriented, component-based architectures; and scaleable, distributed database access.

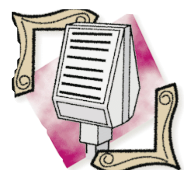
In addition, Netscape will deliver the first commercial Web products enabled by Java. Borland Java tools customers will be able to take advantage of Netscape's clients and servers as a new platform for application development .

To receive additional information about Borland's Internet development tools, access Borland's new Java Web Site at http://www.borland.com/Product/java/java.html.



L I N E

January 1996



Windows 95: Over 10 Million Units Sold

Redmond, WA — Microsoft Corp. announced sales of its Windows 95 operating system has surpassed 10 million units worldwide. This figure includes upgrades and complete versions sold since the product's August 24 release.

Microsoft also reported high customer satisfaction for Windows 95. More than 91 percent of Windows 95 users claimed to be satisfied or very satisfied with the product, as measured by Techscan Inc., an independent market-research firm.

Sales of new applications, hardware products, and PCs with the Designed for Windows 95 logo are also selling well. According to PC Data, a market research firm specializing in the measurement of software retail sales, more than one-quarter of all software sales revenue in the month of September was driven by purchases of Windows 95-based products.

Adoption of Windows 95 within corporations also continues to grow. Over 160 major accounts worldwide have signed purchase agreements and put deployment plans in place, and the majority have installed Windows 95 on at least 10 percent of their PCs in a given geographic area. In the United States in particular, more than half of Microsoft's top 1,000 corporate accounts are in or beyond the pilot testing phase of their deployments for Windows 95.

Currently, Microsoft is not planning to issue a maintenance release for Windows 95. Microsoft will keep Windows 95 users up-to-date by posting online the latest drivers and technology components, such as the Service for NetWare Directory Services, 32-bit DLC protocol stack, ISDN drivers, and Infrared Support.

In addition, Microsoft has added support for Infrared Data Association (IrDA) connectivity to the Windows 95, enabling wireless connectivity between Windows 95-based PCs and peripheral devices. The IrDA support software for Windows 95 can be downloaded from the Internet at no charge, and will be included in future versions of the Windows operating system.

Internationally, the number of language versions and products designed for Windows 95 is increasing. Microsoft was scheduled to release the Japanese-language version of Windows 95 to manufacturing in November 1995. The Japanese-language version is the 17th of the more than 29 localized language editions Microsoft plans to ship for Windows 95.

At press time, more than 400 products developed specifically for Windows 95 have qualified for the Designed for Windows 95 logo, including 136 software products, 181 hardware products, and 130 OEM products.

Kahn Steps Down as Borland Chairman

Scotts Valley, CA — Borland International Inc. announced that Philippe Kahn has stepped down as chairman of the board effective January 1, 1996. Mr. Kahn will continue to serve as a member of the board of directors. Kahn founded Borland International 12 years ago, and has been Chairman of the Board of Directors since then.

"With Delphi a success, a lower cost structure, and the Lotus litigation largely behind it, I feel that Borland is now back on the right track. I am very proud of the team at Borland, and believe that the company is poised for renewed success," Kahn said.

In 1994 Kahn co-founded Starfish Software, and he says he will now focus his energy on this venture.

Acadia Software Begins Operations in Boston Area

Boston, MA — Acadia Software has launched its software consulting company in the Boston, MA area. Formerly the Boston branch of IT Solutions, Acadia Software became independent from its Chicago-based parent in October 1995.

Acadia Software specializes in developing client/server applications, deploying distributed database technology, and providing World Wide Web application solutions. At the core of these technology areas is the principle of object-orientation.

For Windows-based software development, Acadia Software has targeted Borland's Delphi as its primary development tool. According to Richard Wagner, Chief Technical Officer, Acadia selected Delphi because it offers many of the advantages of C++, such as an object-oriented language and the ability to create true binary executables, but it allows developers to create applications in less than half the time. Acadia Software is a Borland Connections partner. For more information, call Acadia Software at (508) 264-4881.

The DCI Data Warehousing Conference

DCI and META Group announced the DCI Data Warehousing Conference is set for February 6-8, 1996 in Orlando, FL. At the Data Warehousing Conference, attendees will get an inside look at the transmutation of 1980's-style decision support systems into GUI-driven client/server executive information systems. Over 50 presentations and keynote sessions, as well as five conference tracks are planned. For more information visit the DCI Web site at http: //www.DCIexpo.com.

Advanced Effective GUI Design

Scheduled for February 1-2, 1996 in San Francisco, CA, Advanced Effective GUI Design is a two-day seminar taught by James Hobart, a senior consultant with Corporate Computing International. This class offers tips, tricks, and helpful hints for creating a graphical user interface. In addition, attendees learn advanced techniques for creating GUI designs by applying rapid application development (RAD) techniques and implementing standards. Advanced Effective GUI Design is priced at US$995. For more information visit the DCI Web site at http: //www.DCIexpo.com.

Delphi / Object Pascal / DDE / Excel

By Gary Entsminger

Great Journeys, Single Steps

State of the Object Art: Part I

It hasn't been a long time, summer of 1988, since I sat with friends (programmers, editors, writers) in a house in the high desert of central Oregon jamming about object-oriented programming (OOP). We were planning to focus an upcoming issue of our magazine, *Micro Cornucopia*, on OOP, and at the time it seemed a risky experiment. We were a small, 20,000 circulation, outfit on a shoe-string budget, and knew that one or two misguided issues could spin us into a free fall.

It was risky because we didn't know that much about OOP yet, and OOP like Windows wasn't the sure bet then that it's become in this decade. Plus, much of what we knew about OOP was rooted in Smalltalk, but our readers were predominantly C and Pascal programmers. Smalltalk alone as our OOP "story," we feared, would go down like flat beer.

Yet Smalltalk had its supporters. It was, and still is, a so-called "pure" object-oriented language, and much can be learned from its object model. In Smalltalk, everything is an object. When you program in Smalltalk, you send messages to objects, which are instances of classes.

In '88, this was strange talk to Pascal and C programmers. To make matters worse, Smalltalk had been developed at a *research center*. Our readers were hackers, in love with everything having to do with computers, and proud of it. So we were betting back in '88 that hackers, being foremost experimenters, would want to "hack" OOP languages like they hacked everything else. So we shaped a magazine around the spirit of exploring and having fun with technology.

I'm reminiscing, not because I'm nostalgic for the good old days (I'm not), but because sometimes looking back helps us look forward. This article kicks off a new *Delphi Informant* series primarily about Delphi's object model and using Delphi's object-oriented techniques. Objects are key to making the most of Delphi, and in particular, knowing how Delphi's object model relates to other object models, especially Windows, will help you develop applications that can manipulate and interact with other Windows objects, including applications.

As you'll learn in this first installment, not all object models are alike. Visual Basic 4 programmers, for example, don't relate to objects or create object-oriented programs the way Delphi programmers do. Although Visual Basic makes clear claim to many things "object": object box, object browser, object data types, object hierarchies, object libraries, object linking and embedding (OLE), object variables, and so on, there's no inheritance facility in the language. Thus, *object hierarchy* in Visual Basic refers to a collection of objects that contain other

objects. Not so in Delphi, where a hierarchy indicates a relationship among objects that inherit properties and methods from ancestral objects. The Visual Basic object hierarchy's counterpart in Delphi is something OOP purists usually refer to as *composition*.

Although *inheritance* is a major feature of almost all OOP languages, not all OOP languages implement it consistently. For example, C++ and Delphi share many OOP characteristics, but they part company at inheritance. In fact, Delphi inheritance more closely resembles Smalltalk's than that of C++.

In short, OOP has matured and prospered dramatically since '88, but approaches to thinking about and organizing objects have remained innovative. An object is always an object (I think), but almost everything else about how we construct and organize applications that use objects is open for discussion.

This series, of course, focuses on Delphi's approach to OOP (you are reading this in the *Delphi Informant*, aren't you?). We'll explore the Delphi object model in general, and you'll learn how to use Delphi applications as the glue or control for groups of objects or applications within the Windows environment. In Windows, everything too is, effectively, an object.

Throughout this series, we'll use OOP, DDE, OLE, DLLs, the Windows API, and anything else that seems appropriate to design applications of interacting objects. These objects might reside within Delphi applications or within the Windows system.

This first article briefly:
- recaps the history of OOP to set the stage for Delphi
- compares two currently dominant object models: Windows/Visual Basic and Delphi
- develops a modest Delphi project that gets information from a set of Microsoft Excel cells using DDE. Microsoft Excel is a Windows *object*. Our Delphi application (also an object) is the glue, using a form (object) and the components (objects) it contains, including a DDE component to interact with a Windows object.

OOP in '88, Our Benchmark

Fortunately for us at *Micro C* in '88, C++ was also beginning to get interesting, and two other OOP languages, Actor and Eiffel, had made recent splashes that looked promising. So we had an issue. Jan Steinman and Barbara Yates at Tektronix took on Smalltalk. Zack Urlocker of the Whitewater Group explained Actor. Bruce Eckel introduced C++.

In '88, OOP was a gold mine if you recognized it. But recognition that OOP would make it so quickly and well into the computing mainstream was blocked by several *reasonable* obstacles.

In '88, DOS, not Windows, was the dominant PC platform. Windows then wasn't even fun to use. Many of our sharpest computing friends were insisting that Windows could never dominate. It was just too clunky to appeal to a *sophisticated* DOS or UNIX user. And the star of the PC language arena was "structured programming" no matter how you diced it: into Basic (and its questionable GoTo), C, or Pascal. Neither of these was remotely event-driven or object-oriented.

There wasn't yet a Smalltalk or even a C++ for Windows, so the Whitewater Group's Actor language, designed exclusively to run within Windows, was a bold leap into a new operating environment and a new language technology (OOP). Microsoft and Borland were still trying to best each other at C, while companies like Tektronix, Digital Equipment, Hewlett-Packard, and Apple were (and had been) investing in the development of Smalltalk-80. Delphi's ancestor, Turbo Pascal with objects, was on the horizon, but not yet an OOP player, and in its first form was quite different from the current Delphi.

The path to OOP and object-based systems clearly wasn't clear to everyone, but two of the writers in *Micro C*'s first OOP issue were well on their way to Delphi. Consider what Zack Urlocker and Bruce Eckel had to say about their favorite object-oriented languages in the November-December '88 issue.

Actor & Delphi

Zack developed a project manager in object-oriented Actor. He wrote:

"The application was easier to develop (and understand) using an object-oriented approach. I spent about two and a half man-weeks conceptualizing and programming, or about two-thirds of the time I would have spent using C, even though I have more experience in C."

That woke up more than a few of us.

Zack went on to say that when designing an object-oriented application you must think of a collection of active objects that operate on themselves. In designing an object-oriented system, you need to focus first on determining which objects will make up the system. In cases where there is no clear logical model for determining these objects, you can determine the objects based on the *user interface*. For example, the objects you'd use for a spreadsheet would include the spreadsheet window and the cells.

Zack's Project Manager consisted of a project window (an object) that could connect to a network, handle menu commands, and so on. He concluded:

"Object-oriented programming encourages the creation of abstract objects with a well-defined public protocol and a private implementation."

The Object Pascal (and Delphi) writing was on the wall. In particular, notice that he suggests using the user interface to determine objects. In Delphi, every application is based on an object-oriented user interface. Each time you create a Delphi

application, you create an instance (i.e. a *variable*) of Delphi's *TApplication* class. Similarly, each time you add a form to an application, you create an instance of Delphi's built-in *TForm* class. When you add a component to a form, you create an instance of Delphi's built-in *TComponent* class, or one of its descendants.

Make sure you're clear about the distinction between class and object. This is a distinction that is more or less consistent among OOP languages:

- A prototype that you use to create specific, related objects is a *class*. A class is a type!
- An instance (or variable) of a class is an *object*. An object is a variable!

A class is a prototype for any number of objects. A class defines the attributes (fields, properties, events, methods) shared by all objects created from the class.

C++ and Delphi

In the November-December issue of *Micro C*, Bruce developed a small CAD system in C++, which he called MicroCad. It managed a linked list of CAD shapes. He designed the application as a framework that our readers could add features to. He saw an immediate OOP benefit — object-oriented systems made it easier for many programmers to add functionality to a single project through objects.

His introduction to C++ emphasized three key features of OOP:

1) abstract data typing or *data encapsulation*
2) type derivation or *inheritance*
3) commonality or *polymorphism*

Happily, these three features are alive and well in Delphi. The first, data encapsulation, is common to all structured and object-oriented programming languages. The following terms used by various languages express data encapsulation: *class*, *form*, *module*, and *unit*. Two main ideas underlie it:

1) data should reside in the same package (or location) as the operations acting on them.
2) encapsulating data with code can hide information that users and other programmers don't need to know about.

In Delphi, a *unit* is the first order of data encapsulation. A unit consists of the constants, declarations, and definitions of classes, objects, functions, procedures, and variables that can be shared within applications, and by more than one application. A unit is an object and can be referenced like any other object.

All units consist of the following fundamental structure:

```
unit <Identifier>;

interface

uses <unit list>  { Optional }

{ Public declarations }

implementation

uses <unit list>  { Optional }

{ Private declarations }

{ Implementations of procedures and functions }

initialization  { Optional }

end.
```

The **interface** part of a unit determines what is visible and accessible to any application (or other unit) using that unit. In the interface, declare the constants, data types, variables, procedures, and functions that other applications or units can use. Note that the bodies (the definitions) of public procedures and functions do not appear in the interface part but, later, in the **implementation** part of the unit.

The **implementation** part of a unit holds the bodies (definitions) of the procedures and functions declared in the interface part of the unit. Declarations made in the implementation part of a unit are private and can be used only within this part of the unit. However, all constants, types, variables, procedures, and functions declared in the interface part are visible in the implementation part.

Optionally, you can use the **initialization** part of a unit to initialize data for the unit.

Each application (object) consists of one or more units.

Class Declarations

You declare classes, the template for objects, in the interface part of a unit, as follows:

```
unit YourUnit;

interface

type
  Obj1 = class(TObject)
  private
    FX: Integer;    { Private data field }
  public            { Public access methods }
    function  GetX: Integer;
    procedure SetX;
end;
```

Here's a translation of the above code. This new class, `Obj1`, is a descendant of the built-in class, *TObject*. That means *Obj1* inherits all the properties, methods (procedures and functions), and events of its ancestor class, *TObject*. Whatever *TObject* contains is now also contained in *Obj1*. In addition, *Obj1* has a new property, *FX*, and two new methods, *GetX* and *SetX*.

Note that by convention, properties (or fields) have an "F" prefix. Also, by convention, fields are declared in the private section of the class. This prevents other unwanted objects from, perhaps inadvertently, manipulating an object's data fields. Each object is responsible for manipulating its own

fields. So, in the *Obj1* class, two public methods are declared to handle manipulation of the *FX* field. Because these methods are public, other objects can access them. The methods, in turn, actually manipulate the field.

Note that a class shares with a unit the ability to hide implementation details and to determine what aspects of its structure is accessible to other parts of an application (other units, other objects, and so on).

A form, too, is a class, which you (or Delphi) can declare in the type section of a unit:

```
type
  Form1 = class(TForm1)
  private
    FX,FY,FZ: Integer;    { Private data fields }
  public                  { Public access to fields }
    State: Boolean;       { An accessible public field }
    function  GetX: Integer;
    procedure SetX(X: Integer);
end;
```

State of the Object Art: Windows, Delphi, Visual Basic, Smalltalk, C++

The current state of "programming art" teems with objects, particularly in the visual programming arena, in languages such as Delphi and Visual Basic. Computing on the PC in general, from the Windows 95 operating system on up through applications such as Paradox, Word, Visual Basic, Visual C++, and Delphi, regardless of application particularity, depends on objects.

From a user's perspective, objects are self-contained packages that perform tasks. The term *object* is the abstraction of anything on a computer screen — a window, a folder, an application, a document, a table, a spreadsheet file. When you design Delphi applications you can use *object* as your own abstraction to make it easier to maintain consistency for users.

Objects, of course, are commonplace in Windows. Notice the status line at the bottom of the Windows 95 folder shown in Figure 1. The folder contains seven icons representing seven objects. Four of these objects are shortcuts that start applications. The others are a text document, a wave file, and a folder that contains its own collection of icons that represent objects.

Yes, although the hierarchical structure isn't visible, the organization of folder within folder is an object hierarchy. Yet, this is not an object hierarchy based on inheritance. Each object in the Windows 95 model is part of a collection of objects.

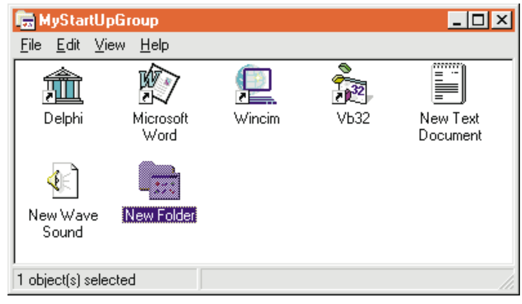


Figure 1: Objects in a Windows 95 folder.

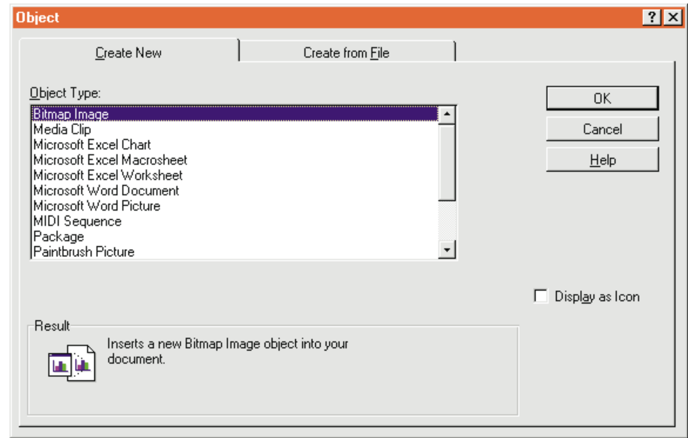


Figure 2: Objects recognized by my Windows 95 configuration.

Figure 2 shows another example of the Windows representation, the objects recognized by my Windows 95 configuration.

Not surprisingly, the Visual Basic 4 programming object model is similar to that of Windows 95. In Visual Basic, you can encapsulate data and operations in a single unit, a class module, and you can build object hierarchies, but one object can't inherit data or behavior from another object. In this regard, Visual Basic has made a major break from the traditional OOP model of Smalltalk, C++, and Object Pascal (Delphi).

Inheritance, a Difference of Opinion

Most object-oriented languages define a *single-rooted, one tree hierarchy*, also called an *object-based hierarchy*. This means that all classes are ultimately inherited from the same root or base class. So all classes in the hierarchy have a common interface. In Object Pascal (Delphi), this common base class is *TObject*. All objects in Delphi are derived from a single ancestor *TObject*. If you don't explicitly derive a new object from an existing one, Delphi automatically derives it for you — from *TObject*. There is one hierarchy for the entire Delphi system. (see Figure 3).

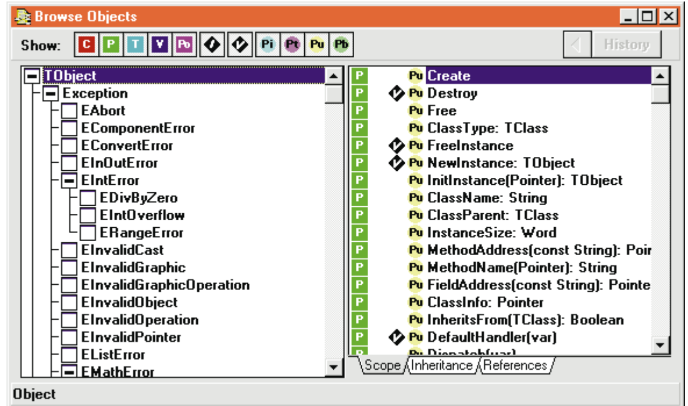


Figure 3: One system, one hierarchy. Delphi's Object Browser provides a graphical representation of its object hierarchy in which all objects share a common base class, *TObject*.

In Smalltalk, all objects are also derived from a single ancestor or base class, called *Object*. As with Delphi, there is one hierarchy for the entire Smalltalk system.

Note that this is not the C++ scenario, where multiple inheritance is possible. In a C++ system, there can be many coexisting (or multiple) hierarchies. From a compiler's perspective, multiple inheritance is hard to perform. From a programmer's perspective, systems based on multiple inheritance are hard to design.

Delphi's Object Model

The Delphi object model based on the *class* is an excellent approach to capturing the gist of object-oriented programming and Windows development. Everything to be known about a class is encapsulated within its class description. Part of this description is in the type declaration for the class in the interface or visible part of a unit, and part is in the class implementation in the implementation or hidden part of a unit.

Yet a class does more than describe the attributes (i.e. the constraints) on a set of related objects. It also establishes relationships among classes. For example, two classes can share some attributes, but handle other attributes differently. Delphi *inheritance* lets you express relationships among classes where underlying base classes spawn more complex derived subclasses. A base class contains all the attributes (fields, properties, methods, and events) that are shared among the classes derived from the base class. Subclasses use existing shared attributes, re-implement shared attributes, or create new attributes.

You typically create a base class to represent the core of your ideas about the subclasses in a system. From the base class, you derive subclasses that express the different ways that core can be realized.

In Delphi, every built-in class and every class you create is part of a class hierarchy. You can examine the hierarchy of classes for any Delphi application as follows:

- Create or open a new project by selecting **File | New Project**, or **File | Open Project** from the Delphi main menu.
- Compile the project by selecting **Compile | Compile** from the Delphi main menu.
- Open the Object Browser by selecting **View | Browser**.

Delphi: Messages and Composition

Delphi has a one-to-one mapping between interface components and classes. Thus, a form is represented by a class. Each time you place a component on a form, such as a command button, Delphi adds a member object to the form class to represent that component. To manipulate that component you send it *messages*, in traditional OOP lingo.

All the visual elements of a Delphi application are objects. Forms are objects. Menus and menu items are objects. Components are objects, and so on.

When an event occurs (i.e. when Windows routes the event to your Delphi application), the element that receives the event calls the appropriate member function (or event procedure) *in the form*. The form owns the message handlers for all events that can occur on itself. Thus, the *form class* is the focal point for everything that goes on in a form. This is OOP at a very fundamental level in Delphi programming.

An event procedure that handles an event also receives the identity of the sender (as an object). Typically, you only need to know that an event occurred, not the identity of the sender. For example, when a user selects a menu item, the application directs the event message to that menu item's event procedure.

Connecting Objects: A DDE Project

OK, that's enough theory for an introduction. Let's conclude by developing a little application that uses OOP to make a DDE connection. This project uses a Delphi application (an object) to access a set of Microsoft Excel cells. The application (DDEEXCEL.DPR) consists of a form that contains five components (specialized objects).

Begin by creating a new project by selecting **File | New Project** from the Delphi main menu. Delphi will automatically create a form and its corresponding unit. Save the form as DDEFORM.PAS, and save the project as DDEEXCEL.DPR.

Use the Delphi Component palette to add the following five components to the form:

- from the System page: DDEClientConv and DDEClientItem (these handle the DDE connection)
- from the Standard page: a label and two buttons (the buttons will allow the user to control the application; the label will receive the contents of the Excel cells)

Figure 4 shows this form at design time.

Figure 4: The example DDEform application at design time.

Use the Object Inspector to modify three of the DDEClientConv component's properties as follows:

- *ConnectMode*: `ddeAutomatic`
- *DDEService*: `Excel`
- *DDETopic*: `sheet1`

Then modify the DDEClientItem component's properties (see Figure 5):

- *DDEItem*: `r1c1…r3c1`.
- *DDEConv*: `DDEClientConv1`

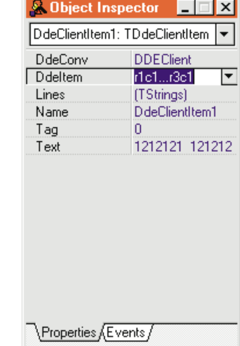


Figure 5: Using the Object Inspector to modify the DDEClientConv component's properties.

Note that once the DDE connection is established, the DDEClientItem component's *Text* property will contain the value of the item specified in *DDEItem*.

Create a *ButtonClick* event procedure for the **Connect** button to set the DDE link and update the caption property of the label:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with DDEClientConv1 do    { With object do }
    if not SetLink(DDEService, DDETopic) then
      MessageDlg('Link not established.',
                 mtInformation, [mbOK], 0)
    else
      Label1.Caption := DDEClientItem1.Text;
end;
```

Create a *ButtonClick* event procedure for the **Exit** button to close the application:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
```

That's all there is to it. Our little Delphi application (an object) can make its connection to another application (an object), obtain data on demand, and display the results. Of course, this is just a beginning. Once we have the data, we can massage the data in virtually any way we want. Figure 6 shows this application interacting with Excel at run time. Note that Excel must already be running to make this DDE connection.

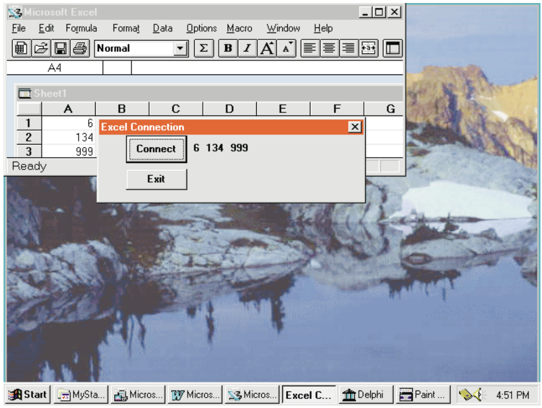


Figure 6: The example application interacting with Excel at run time.

The complete code for this application is shown in Figure 7. Notice that the project's main and only form is a class, derived from the Delphi *TForm* class. The form contains (or is *composed* of) the components (objects) we added and the two procedures (or object methods) we created.

Wrap Up

In a Delphi object-based system scenario, Delphi applications become the glue or links for groups of interacting applications. A Delphi application "shell" connects to Excel if it needs to manipulate data, to Word if it needs to edit, and to other applications as it needs to. For example, using OLE Automation you could create an Excel spreadsheet object, and then manipulate the spreadsheet a procedure at a time by calling Excel (the object) methods.

Another interesting aspect of this part of the object model is that Delphi can access the individual procedures in these connected applications. For example, a Delphi application could use a spell checker in another application to check spelling in a database table running in Paradox.

```
unit Ddeform;

interface

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

{ Excel Connection --
  Establishes a DDE conversation with Excel
  Reads the first three cells of sheet1
  Reports results in a label
  Uses TDDEClientConv and TDDEClientItem objects
  to handle the DDE connection. }

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    DDEClient: TDdeClientConv;
    DdeClientItem1: TDdeClientItem;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  with DDEClientConv1 do
    if not SetLink(DDEService,DDETopic) then
      MessageDlg('Link not established.',
                 mtInformation, [mbOK], 0)
    else
      Label1.Caption := DDEClientItem1.Text;
  end;

end.
```

Figure 7: The complete source listing for Ddeform.pas.

Using OLE Automation, a Delphi 2 (but not a Delphi 1) application can create an object that can be manipulated by other OLE objects. And in turn, a Delphi object/application (using an OLE container) can manipulate other OLE objects. Delphi does allow programmers to create OLE containers. OLE Automation is a 32-bit feature.

So where does this object-based world lead? To paradise? Maybe, maybe not. In the next installment of this series, we'll look more closely at pros and cons and at how we can use object-based design in Delphi to create object managers for handling suites of tasks. ∆

The example project referenced in this article is available on the Delphi Informant Works CD located in INFORM\96\JAN\DI9601GE.

Gary Entsminger's new book, *The Way of Delphi*, an intermediate and advanced guide to object-oriented Delphi development, is forthcoming from Prentice-Hall. He is currently working on a new book and trying to make the most of 32-bit systems and the Web.

By John O'Connell

The Dynamics of Delphi DDE

Putting the DDEClientConv and DDEClientItem Components to Work

Dynamic Data Exchange (DDE) is the feature of Windows that provides the mechanism for inter-program communication between different Windows applications. As the name implies, client and server applications can exchange data during a DDE conversation. You could say that DDE is a "client/server" type of link between the application requesting data (the client) from the application providing that data (the server).

A DDE server can advertise its availability via data pasted to the Windows Clipboard (although some applications store their DDE server information in the Windows Registry). A DDE client begins the DDE conversation by specifying the data it wants through three-character string parameters: the service, topic, and item.

The *service* is generally the name of the server application's executable file unless otherwise defined in the Registry. The *topic* is the "container" for the requested data that is the *item*. For instance, the cell A4 in an Excel spreadsheet named MYSHEET.XLS is identified by the following DDE conversation parameters:

```
"Excel","MYSHEET.XLS","A4"
```

An application can be both a DDE client and server. Generally, if the server application isn't loaded, then the conversation cannot be initiated. However, some clients will attempt to launch the server application.

Basically, DDE works by using a combination of shared global memory and Windows DDE messages that are sent between the client and server. For instance, the server can signal to a DDE client that the requested data is available via a specified global memory handle. These DDE messages signal several items, including:

- the start of a DDE conversation.
- the acknowledgment/negative acknowledgment (ACK/NAK) of a previously sent DDE message.
- the availability of a server's data to the client.
- the actual data transfer from server to client.
- the termination of the DDE conversation.

Besides requesting data from a DDE server, the client can also send data to the server by "poking" data into the server's data item — data then flows from client to server. The client

can also execute a server's "macro" functions which, if supported, instruct the server to perform a certain task by calling that server's macro functions.

By convention, individual server macro function calls are enclosed in brackets ([]) so that multiple function calls, each separately enclosed in brackets, can be executed by one DDE execute message. For instance, the following DDE macro execution string that was sent to Microsoft Word for Windows as a DDE server opens the documents named LETTER.DOC and INVOICE.DOC:

```
[FileOpen("LETTER.DOC")][FileOpen("INVOICE.DOC")]
```

When executing a macro, only the application and topic parameters must be specified — the DDE item is not required.

Conversation Types

There are three types of DDE conversations:

- A hot link — the DDE data is continually updated between the server and client.
- A cold link — the data is only transferred when requested by the client.
- A warm link — the server informs the client that the data has changed so the client can decide if it wants to request the changed data.

As mentioned, the flow of data takes place through shared global memory and is controlled by DDE messages sent between the client and server.

Although DDE makes life easier for end-users of Windows applications, implementing DDE in applications makes life correspondingly difficult for the poor ol' developer with just the Windows API at hand.

But with Delphi, including DDE client/server capability in your applications is made much easier thanks to the various DDE components that are conveniently encapsulated within the Visual Component Library (VCL). In this article I'll discuss the various DDE client and server components, and how to use them with Delphi to build DDE client and server applications.

DDE Servers

Delphi's VCL provides the *TDDEServerConv* and *TDDEServerItem* components for creating server applications. The *TDDEServerConv* defines the topic parameter of a DDE conversation. Its *OnOpen* and *OnClose* event handlers are triggered in response to DDE conversation initiation, and termination by a DDE client, respectively. The *OnExecuteMacro* event handler is triggered by DDE macro execute requests.

The *TDDEServerItem* component defines the item parameter of a DDE conversation and has properties that enable programmatic access to the data contained in the item. The *Text* property provides access to up to 255 characters of item data. The *Lines* property provides access to more than 255 characters of item data which is represented with a *TStrings* object that encapsulates multiple lines of 255 characters each. The first string in the *Lines* property contains the same data as the *Text* property:

```
Lines[0]
```

The *OnChange* and *OnPokeData* server item event handlers are triggered when the item data is changed by the server and client, respectively. (Recall that a client can poke new data into the server item of the DDE conversation.)

You are not required to have a *TDDEServerConv* component in your form to create a server application — the *TDDEServerItem* will do. In such a case, the *ServerConv* property is undefined and the conversation's topic parameter is the parent form's caption. However, in situations when the form's caption will not remain constant, you must use a *TDDEServerConv* component and set the *TDDEServerItem*'s *ServerConv* property to that server conversation component. I prefer to use a *TDDEServerConv* component for convenience and greater control.

The *CopyToClipBoard* method of a *TDDEServerItem* copies its *Text* and *Lines* property to the Clipboard, along with the DDE link information. This information can then be interrogated by a DDE client application to find a DDE link's service, topic, and item parameters.

Appraising Property Value

Suppose we've built our application with DDE server capability. How do we associate the value property of a control with a DDE item? Obviously, we must update the DDE item's data as the contents of say, a *TEdit* (or a *TMemo*) changes while the user edits the control's contents. We can simply assign the *TEdit*'s *Text* property to the *TDDEServerItem*'s *Text* property. However, to associate a *TMemo's* contents we must assign the *TMemo*'s *Lines* property to the *TDDEServerItem*'s *Lines* property. It's that simple.

But where should such assignments occur in the code? To update the *TDDEServerItem's* data as the user edits the contents of a *TEdit* or *TMemo*, simply assign the *Text* or *Lines* properties within the *OnChange* event handler for the *TEdit* or *TMemo*:

```
DDEServer1.Text := Edit1.Text;

...

DDEServerItem1.Lines := Memo1.Lines;
```

The SRVRDEMO program demonstrates the DDE server concepts introduced so far. SRVRDEMO is a DDE server for the records in any table that can be accessed by the Borland Database Engine (BDE). This server allows a DDE client to open a specified table, retrieve a particular

record, move to a specific record, and retrieve a field in the current record — all from the currently opened table. This is achieved using DDE macro execution from any DDE client application (an example DDE client is presented later in the article). Providing DDE macro execution in a DDE server application requires an event handler for the *TDDEServerConv*'s *OnExecuteMacro* event.

The *OnExecuteMacro* event of a *TDDEServerConv* has the event sender and macro text (a *TStrings* object) as its parameters. These must be interpreted and parsed by your code. The *OnExecuteMacro* event handler demonstrates this (see Figure 1).

```
procedure TFrmDDESrv.ParseMacro(Sender: TObject;
                                Msg: TStrings);
begin
  { Only want single-line macros }
  if (Msg.Count > 1) then
  begin
    MessageDlg('Invalid macro command',
               mtError,[mbOK],0);
    Exit;
  end;

  if CompareText(Msg[0],'OpenTable') = 0 then
    OpenTable  { A user-defined method }
  else
  if CompareText(Msg[0],'GotoRecord') = 0 then
    GotoRecord { A user-defined method }
  else
  if CompareText(Msg[0],'GetRecord') = 0 then
    GetRecord  { A user-defined method }
  else
  if CompareText(Msg[0],'GetField') = 0 then
    GetField   { A user-defined method }
  else
    MessageDlg('Unrecognised macro command',
               mtError,[mbOK],0);
end;
```

Figure 1: The *OnExecuteMacro* event handler.

Although this example is trivial, it does show how you can implement the server macro functions *OpenTable*, *GotoRecord*, *GetRecord*, and *GetField*. A more complete example of implementing macro capability is demonstrated in the SRVRDEMO example. We'll discuss this later in greater detail.

DDE Clients

The *TDDEClientConv* and *TDDEClientItem* components provide the building blocks of a DDE client application. A *TDDEClientConv* component defines the topic parameter of a DDE conversation and has event handlers that are triggered when the client conversation is successfully opened (*OnOpen*) and closed (*OnClose*).

The *DDEService* and *DDETopic* properties define the conversation's service and topic that uniquely identify the DDE server application. The *ServiceApplication* property is related to *DDEService*. In some cases, a DDE service is identified by the name of the application's .EXE file rather than a predefined service name. Therefore, *ServiceApplication* must be specified instead of *DDEService*.

The *FormatChars* property specifies the formatting of certain non-printable characters contained in the data received from the server. Typically, non-printable characters (e.g. carriage returns, linefeeds, and tabs) can affect how the data is formatted. Setting *FormatChars* to *False* generally causes spaces to replace non-printable characters and can cause truncation of data longer than 255 characters. Setting *FormatChars* to *True* cures this problem as the *Lines* property of the associated *TDDEClientItem* may be used to hold the received data, instead of the *Text* property that is limited to 255 characters. *FormatChars* enables newline characters to be correctly handled.

The *ConnectMode* property determines when the DDE conversation is initiated. When *ConnectMode* is set to *ddeAutomatic*, the conversation is initiated at run time. Obviously, for this to succeed the *DDEService* and *DDETopic* properties must be specified and must identify a DDE server application.

When *ConnectMode* is set to *ddeManual*, the DDE conversation is initiated only after the *OpenLink* method is called. *OpenLink* returns a Boolean result to indicate if the DDE link was successfully initiated. If the server application is not already running, it will be started automatically by the *TDDEClientConv*. This saves a lot of hassle when checking if the application is already loaded as you must do when using just the Windows API.

This leads to more *TDDEClientConv* methods. *SetLink* allows the service and topic to be specified:

```
if DDEClientConv1.SetLink('WinWord','Myletter') then
  ShowMessage('We have contact!');
```

If *ConnectMode* is set to *ddeAutomatic*, *SetLink* attempts to initiate the conversation with the specified service. This method returns a Boolean value to indicate success or failure.

The *CloseLink* method terminates the DDE conversation. The *RequestData* method requests data from the server. *RequestData* takes a DDE item name as a string and returns the requested data as a null-terminated PChar string that is automatically allocated memory by *RequestData*.

Therefore, when you've finished processing this PChar you must free the memory using *StrDispose*. Otherwise, memory leakage will occur as more memory is allocated with each call to *RequestData*, but is not released when the application terminates. One strategy is to copy the PChar string into a Pascal string (using the *StrPas* function) such as a *TEdit*'s *Text* property and then free the PChar immediately. This works only if the PChar string contains up to 255 characters — any more than this and an alternate strategy is required. You can copy the PChar into a *TMemo*'s *Lines* property using this code:

```
procedure TForm1.Button1Click(Sender: TObject)
var
  DDEData: PChar;
begin
  DDEData := DDEClientConv1.RequestData('TheItem');

  if (DDEData <> nil) then
    Memo1.Lines.SetText(DDEData);

  StrDispose(DDEData);
end;
```

The *SetText* method applies to *TStrings* and *TStringList* objects and copies the contents of PChar to the text buffer of the object's data. The *Lines* property of a *TMemo* and *TDDEServerItem* is a *TStrings* object. Note that *RequestData* doesn't alter the data seen by a *TDDEClientItem* object. This could be done with the following code:

```
DDEClientItem1.Lines.SetText(
  DDEClientConv1.RequestData('TheItem'));
```

Obviously however, this code would cause memory leakage because the PChar returned by *RequestData* can never be freed. The correct way would be to separate the calls to *RequestData* and *SetText*, and then use *StrDispose* to free the PChar returned by *RequestData*.

As mentioned, a DDE client can poke data to a server's DDE item. (The *TClientConv* methods *PokeData* and *PokeDataLines* enable data to be sent to the server item specified as one of the two arguments to these methods. The second argument is the data to be sent.) For *PokeData*, the second argument is a PChar, and for *PokeDataLines* the second argument is a *TStrings* object containing a list of text strings. Each text string has a maximum length of 255 characters.

With *PokeData* you're not limited to poking data of less than 255 characters to a server item. However, when you're poking the contents of a *TMemo* (or any other *TStrings*/*TStringList* object for that matter), *PokeDataLines* is more convenient to use. The following code fragments demonstrate the use of *PokeData* and *PokeDataLines*:

```
procedure SendItToTheServer;
var
  SrvrData: PChar;
begin
  SrvrData := StrAlloc(Ord(Edit1.Text[0]) + 1);
  DDEServerConv1.PokeData("MyItem",
                          StrPCopy(SrvrData,Edit1.Text));
  StrDispose(SrvrData);
end;

procedure LetTheServerHaveIt;
begin
  DDEServerConv1.PokeDataLines("MyItem",Memo1.Lines);
end;
```

Note that the memory used by *SrvrData* was dynamically allocated. Therefore, it must be freed using *StrDispose*. The expression:

```
Ord(Edit1.Text[0])
```

returns the length of *Edit1.Text*. The first byte of a string type is a length byte (this explains why strings are limited to 255 characters), that is converted to its integer representation using the standard *Ord* function.

Finally, the *TDDEClientConv* methods *ExecuteMacro* and *ExecuteMacroLines* allow macro commands to be sent to the server for execution and return a Boolean success code result. These methods take two parameters. The first is the actual text of the macro command(s) that is a PChar for the *ExecuteMacro* method, or a *TStrings* for the *ExecuteMacroLines* method. The second is Boolean, that if *True*, causes subsequent calls to these methods to fail and return *False* if the server is processing previously issued macro function requests. If you specify the second argument as *False*, then the server application must have some way of buffering any macro execute strings that are awaiting execution.

If DDE data is already pasted to the Clipboard, the client application can retrieve the service, topic, and item using the *GetPasteLinkInfo* function. (Yes, I mean *function*, not *method* because *GetPasteLinkInfo* isn't a member of any class. It's a function defined in the DDEMAN unit where Delphi's DDE components are defined.) *GetPasteLinkInfo* takes three strings as variable parameters into which this method writes the service, topic, and item of the DDE data currently on the Clipboard. Then it returns a Boolean success return code of *False* if the service, topic, and item cannot be obtained from the Clipboard data.

Finally we proceed to the *TDDEClientItem* component that encapsulates a DDE conversation's item. This object has no methods, but has the *DDEConv* and *DDEItem* properties to define the DDE topic with the *Lines* and *Text* properties to hold the DDE data. The only event handler is *OnChange* that is triggered when the server changes the DDE data in a hot link. However, *OnChange* is *not* triggered after a call to the *TDDEClientConv.RequestData* method.

The SRVRDEMO Application

Let's take a look at the example SRVRDEMO project in more detail, relating to our discussion. SRVRDEMO acts as a DDE server application accessible by the BDE (see Figure 2). It's not intended to be an interactive database access application. Instead, SRVRDEMO is controlled by macro functions issued by a DDE client. These functions allow the client to open a table, fetch a particular record in that table, move to a specified record number, and retrieve a particular field value from the current record. SRVRDEMO also displays the alias and name of the currently open table.

The Memo component labeled **Server Events** displays a log of the events triggered by a client, such as *OnOpen*, *OnClose*, and *OnExecuteMacro*. The memo component labeled **Server Item** displays the data returned by the most recent macro execute issued by the client. Typing into this memo automatically updates any client's data as is usual in a hot link.

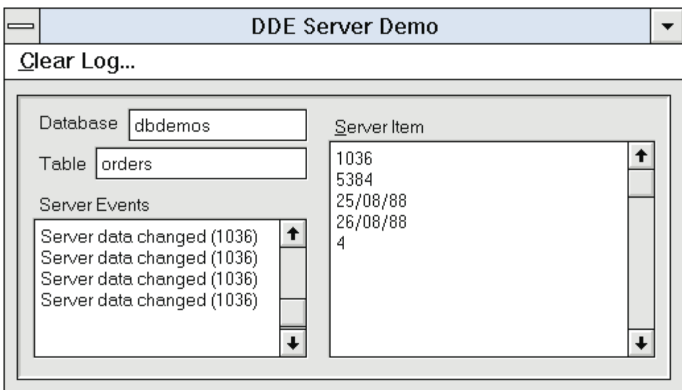


Figure 2: The example DDE server application (SRVRDEMO) at run time.

For DDE client applications lacking data-access capability, SRVRDEMO can be developed into a useful tool for accessing data from various data sources.

Implemented Macros

Let's look at the implementation of the supported server macros — *OpenTable*, *GetRecord*, *GotoRecord*, and *GetField* (see Listing One on page 20):

- The *OpenTable* macro function takes a full table specification (e.g. `:Alias:TableName`) as its sole parameter.
- *GetRecord* takes an integer parameter that identifies which record in the table will be retrieved and then stored in the *Lines* property of the *TDDEServerItem* component named *DDEGetRecord*.
- *GotoRecord* takes an integer parameter, identifying which record in the table will be updated.
- *GetField* takes an integer parameter to specify which field will be retrieved from the current record and then stored in the *Text* property of *DDEGetRecord*.

These macro functions are called in much the same way as any function — that is, their parameters are enclosed in brackets. Because the *Msg* parameter on the *OnExecuteMacro* event handler holds just the strings containing the macro request (as sent by a DDE client), the macro name and arguments must be separated or parsed by code in the *OnExecuteMacro* event handler.

The *ParseMacroArgs* event handler in SRVRDEMO does just that. The calls to the methods *GetFnName* and *GetArgs* extract the name and parameters of the macro function string passed as a parameter to the handler. The extracted macro name is compared with a number of macro names to determine which one to execute. If the macro name isn't found, an error message informs the user that the specified macro wasn't recognized.

Notice that the *Msg* parameter is checked to see if more than a single string line has been passed as a macro call. If so, then the macro execute is rejected with an error message stating that the specified macro is invalid — SRVRDEMO doesn't support multiple line macros or macros longer than 255 characters.

The *CompareText* function used in the **if..then** expressions is defined in the SysUtils unit. It simply performs a non-case-sensitive comparison between the two string arguments. If both strings match, 0 is returned.

Let's look at the other event handlers for the *DDEOpenTable* and *DDEGetRecord* conversation and item components:

- *OnOpen* and *OnClose* simply log the events to the *memEventLog TMemo* component. (We've already covered the *OnExecuteMacro* event handler.)
- The *OnChange* handler of *DDEGetRecord* is triggered when the item's data is changed and logs the event to *memEventLog* with a little information about the new data.
- The *OnPokeData* handler, again, logs the event to *memEventLog* in a fashion similar to *OnChange*. However, notice the flag variable *FInPoke* is set to *True* for the duration of the event before being reset to *False*. This flag can be checked in the *OnChange* event handler for the *memItem TMemo* that displays the DDE item's current data. If this check wasn't made, it's possible that a client poking data to the server item could conflict with a change in the contents of the DDE item due to a user input or a piece of code in the application. Therefore, to avoid conflict, the code in the *OnChange* handler for the *memItem TMemo* checks if *FInPoke* is *False* before assigning the contents of *memItem* to the *Lines* property of the *DDEGetRecord TDDEServerItem* component.

The CLNTDEMO Application

This application holds a DDE client conversation with SRVRDEMO (see Listing Two on page 22) and uses macro execute functions to control the server (see Figure 3). The actual DDE server data is displayed in a *TMemo* component *memServer*. The *edRecNo TEdit* component is used to enter macro function parameters used with the four macro functions that are selectable from the **Execute options** radio buttons.

The *FormatChars* property of the *ClientConv* component can be toggled using the **FormatChars** checkbox. The **Open** and **Close** buttons initiate and terminate the DDE conversation, respectively. The **Request** button calls *RequestData*

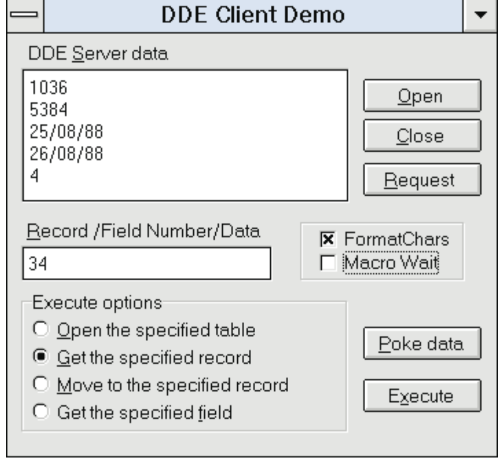


Figure 3: The example DDE client application (DDECLNT) at run time.

to obtain the latest server data, the **Poke data** button pokes the contents of *edRecNo.Text* to the server item, and the **Execute** button executes the selected macro function.

The *OnClick* event handler for the **Open** button sets up the service, topic, and item parameters of the DDE client conversation:

```
ClientConv.ConnectMode := ddeManual;
ClientConv.SetLink('DDEART','DDEOpenTable');
ClientItem.DDEItem := 'DDEGetRecord';

if not ClientConv.OpenLink then
  ShowMessage('Unable to open the link');
```

Notice that the *ConnectMode* property is set to manual and therefore the *OpenLink* method must be used to initiate the DDE link. If *ConnectMode* was set to *ddeAutomatic*, the link would be initiated as soon as *ClientItem.DDEItem* was assigned after calling *SetLink*.

The event handler for the **Close** button calls the *CloseLink* method of *ClientConv* while the **Poke data** button's handler calls the *PokeData* method and passes the dynamically allocated PChar variable *Buf* as a parameter.

The **Execute** button's handler calls the *ExecuteMacro* method with a dynamically allocated PChar passed as the first parameter, and the state of the **Macro Wait** checkbox as the second parameter. If this second parameter is *False*, subsequent calls to *ExecuteMacro*, *ExecuteMacroLines*, *PokeData*, or *PokeDataLines* will be sent to the DDE server regardless of whether the server is busy executing the requested macro. If this second parameter was passed as *True*, then subsequent calls to those methods fail and return *False* if the server is busy. In this case, note the size parameter used with *StrAlloc* to reserve the memory for the null-terminated string variable named *Buf*:

```
Buf := StrAlloc(Length(Cmd)+1);
```

When allocating memory for a PChar, you must allow for the terminating null character that indicates the end of the string. Therefore, to allocate sufficient space for the string variable *Cmd*, we pass the length of *Cmd* plus 1.

The *OnChange* handler for the *ClientItem* component of type *TDDEClientItem* assigns the changed contents of the DDE item to the *memSrvr TMemo*:

```
memSrvr.Lines := ClientItem.Lines;
```

SRVRDEMO and CLNTDEMO Working Together

The CLNTDEMO application was designed for use with SRVRDEMO. However, you can use any DDE-capable application as a DDE client of SRVRDEMO. The DDECLI sample application included with Delphi is ideal, but using CLNTDEMO involves much less typing. Using SRVRDEMO and CLNTDEMO together can teach us a few things to watch out for, especially when writing DDE client applications.

For example, setting the *FormatChars* property can have major effects on the way data is received from the server. DDE server data that is longer than 255 characters will be truncated when *FormatChars* is *False*. Newline characters in the DDE server data are replaced with spaces when *FormatChars* is *False*. To see this, check/uncheck the **FormatChars** checkbox.

With *FormatChars* set to *False* you'll notice that the DDE data appears on one line of the memo with spaces between each field value (each of these appears on separate lines of the memo when *FormatChars* is *True*). If the data is longer than 255 characters, you'll notice truncation of the data. Setting *FormatChars* to *True* solves this problem and will work for any DDE data containing formatting characters such as line breaks. (I've tried this with a bookmarked paragraph of text from Word and it works just fine.)

You may have noticed that toggling *FormatChars* causes the data to be re-requested from the server automatically. Pressing the **Request** button to obtain the DDE data results in the field values being displayed one-per-line in CLNTDEMO's memo. The setting for *FormatChars* has no affect in this case and the PChar returned by the *RequestData* method is used to set the memo's contents.

Another thing to watch for is the second (Boolean) parameter of the *ExecuteMacro* and *ExecuteMacroLines* methods — the state of the **Macro Wait** checkbox. If this is specified as *True*, then the data resulting from the macro execution is not displayed automatically by CLNTDEMO.

However, if this "wait" parameter is specified as *False*, then there's no problem and the data is displayed as it changes at SRVRDEMO. These situations are comparable to a DDE cold-link and a DDE hot-link. In the first case where *Wait* is *True* the data isn't immediately received by the DDE client until it's requested, just like in a cold-link. If another application requests data from SRVRDEMO, then CLNTDEMO receives the updated data immediately.

At this point you may think everything discussed in this section of the article conveniently uses DDE between two Delphi applications as an example. But what about non-Delphi DDE server and client applications? I tested both demonstration programs with Word without any problems, and what I've stated here about the *FormatChars* property and the *Wait* parameter of the *ExecuteMacro*/*ExecuteMacroLines* all holds true. The WordBASIC program code shown in Figure 4 uses SRVRDEMO as a DDE server.

Brackets Are Optional

Earlier, I mentioned that macro commands are usually enclosed in brackets to facilitate executing more than one macro command at the server using a single DDE execute instruction. The server can then distill this multiple macro

```
Sub MAIN
  Chan = DDEInitiate("DDEART","DDEOpenTable")

  If Chan <> 0 Then
    MsgBox "About the get the 3rd record in the " +
           "ORDERS table ..."

    DDEExecute Chan,"OpenTable(:dbdemos:orders.db)"
    DDEExecute Chan,"GetRecord(3)"

    TheData$ = DDERequest$(Chan,"DDEGetRecord")

    MsgBox "The 3rd record in the ORDERS table is: " +
           Chr$(10) + TheData$

    MsgBox "About to retrieve the ORDERNO field of " +
           "the 10th record ..."

    DDEExecute Chan,"GotoRecord(10)"
    DDEExecute Chan,"GetField(1)"

    TheData$ = DDERequest$(Chan,"DDEGetRecord")

    MsgBox "The ORDERNO field of the 10th record is: " +
           Chr$(10) + TheData$

    DDETerminateAll
  Else
    MsgBox "Unable to open DDE conversation"
  End If
End Sub
```

Figure 4: This WordBASIC macro uses SRVRDEMO as a DDE server.

request into the various macro functions and execute them in order.

However, it's not necessary to support brackets in your *OnExecuteMacro* event handler because their use is merely a convenience. To allow your server application to handle multiple macro functions in a single *OnExectuteMacro*, you could have the client send macro functions in one line. This eliminates having to deal with the brackets in an *OnExecuteMacro* handler.

Returning to the WordBASIC macro example, we could send the *OpenTable* and *GetRecord* macro functions to SRVRDEMO using the following code:

```
Chan = DDEInitiate("DDEART", "DDEOpenTable")
DDEExecute Chan,"OpenTable(:dbdemos:orders.db)" +
           chr$(10) + "GetRecord(3)"
DDETerminateAll
```

The *Msg* parameter in SRVRDEMO's *OnExecuteMacro* event handler would then contain the two strings in `Msg[0]` and `Msg[1]`, respectively:

```
"OpenTable(:dbdemos:orders.db)"
```

and:

```
"GetRecord(3)"
```

While the current version of SRVRDEMO will reject this because it doesn't like multi-line macro functions, you may try changing this to implement managing multiple macro function handling in the *OnExecuteMacro* event handler.

And what about the situation where *ExecuteMacro* or *ExecuteMacroLines* is called with the *Wait* parameter passed as *False*? You must then buffer any pending macro requests to avoid the possibility that one macro is still executing when another is received. I'll leave such buffering as an exercise for you.

Conclusion

A final word about DDE client applications: make sure that your application cleans up before it closes by terminating any open DDE client conversations. You should close all DDE links from a form in its *OnDestroy* handler. ∆

The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\JAN\DI9601JO.

John O'Connell is a software consultant (and born-again Pascal programmer) based in London, specializing in the design and development of Windows database applications. Besides using Delphi for software development, he also writes applications using Paradox for Windows and C. John has worked with Borland UK technical support on a regular free-lance basis and can be reached at (UK) 01-81-680-6883, or on CompuServe at 73064,74.

Begin Listing One — The SRVRDEMO Project

```
program Srvrdemo;

uses
  Forms,
  Getrec in 'GETREC.PAS' {FrmDDESrv};

{$R *.RES}

begin
  Application.Title := 'DDE Server Demo';
  Application.CreateForm(TFrmDDESrv, FrmDDESrv);
  Application.Run;
end.

unit Getrec;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, DdeMan, DB,
  DBTables, StdCtrls, ExtCtrls, Menus;

type
  TFrmDDESrv = class(TForm)
    tblSource: TTable;
    DDEOpenTable: TDdeServerConv;
    DDEGetRecord: TDdeServerItem;
    Panel1: TPanel;
    edAlias: TEdit;
    Label2: TLabel;
    Label1: TLabel;
    edTableName: TEdit;
    memEventLog: TMemo;
    MainMenu1: TMainMenu;
    Clearlist1: TMenuItem;
    Label3: TLabel;
    memItem: TMemo;
    Label4: TLabel;
    procedure ParseMacroArgs(Sender: TObject;
                             Msg: TStrings);
    procedure FormDestroy(Sender: TObject);
    procedure DDEOpenTableClose(Sender: TObject);
    procedure DDEOpenTableOpen(Sender: TObject);
    procedure DDEGetRecordPokeData(Sender: TObject);
    procedure DDEGetRecordChange(Sender: TObject);
    procedure Clearlist1Click(Sender: TObject);
    procedure memItemChange(Sender: TObject);
  private
    { Private declarations }
    AliasParam, TableNameParam : string[128];
    FInPoke: Boolean;
    procedure OpenTable(const AliasSpec,
                        TableSpec: string);
    procedure GotoRecord(const NRecNum: LongInt);
    procedure GetField(const NField: byte);
    procedure GetRec(const NRecNum: LongInt);

    function GetArgs(const BufStr: string): string;
    function GetFnName(const BufStr: string): string;
  public
    { Public declarations }
  end;

var
  FrmDDESrv: TFrmDDESrv;

implementation

{$R *.DFM}

procedure TFrmDDESrv.ParseMacroArgs(Sender: TObject;
                                    Msg: TStrings);
var
  TmpAlias,
  TmpName,
  FnName:       string[128];
  ParamText:    string;
  InAlias:      Boolean;
  i, j:         byte;
  FnArg:        LongInt;
begin
  if (Msg.Count = 1) then
    begin
      ParamText  := GetArgs(Msg[0]);
      FnName     := GetFnName(Msg[0]);
      InAlias    := False;

      if CompareText('OpenTable', FnName) = 0 then
        begin
          TmpName   := '';
          TmpAlias  := '';
          j         := Length(ParamText);
          { Scan the string to extract the alias
            and table name }
          for i := 1 to j do
          begin
            if (ParamText[i] = ':') then
              begin
                InAlias := not InAlias;
                continue;
              end;
            if InAlias then
              begin
                TmpAlias := TmpAlias + ParamText[i];
                continue;
              end
            else
              TmpName := TmpName + ParamText[i];
          end;
          OpenTable(TmpAlias, TmpName);
        end
      else
        if CompareText('GetRecord', FnName) = 0 then
          begin
           try
              FnArg := StrToInt(ParamText);
            except
              MessageDlg(ParamText +
                         ' is not a valid integer',
                         mtError, [mbOK], 0);
              Exit;
            end;
            GetRec(FnArg);
          end
        else
          if CompareText('GetField', FnName) = 0 then
            begin
              try
                FnArg := StrToInt(ParamText);
              except
                MessageDlg(ParamText +
                           ' is not a valid integer',
                           mtError, [mbOK], 0);
                Exit;
              end;

              GetField(FnArg);
            end
```

```
          else
            if CompareText('GotoRecord',FnName) = 0 then
              begin
                try
                  FnArg := StrToInt(ParamText);
                except
                  MessageDlg(ParamText +
                            ' is not a valid integer',
                            mtError, [mbOK], 0);
                Exit;
              end;

            GotoRecord(FnArg);
          end
        else
          MessageDlg('Unrecognised macro function ' +
                     FnName, mtError, [mbOK], 0);
      end
    else
      MessageDlg('Invalid macro function call',
                 mtError, [mbOk], 0);
end;

procedure TFrmDDESrv.OpenTable(const AliasSpec,
                                     TableSpec: string);
begin
  edAlias.Text     := AliasSpec;
  edTableName.Text := TableSpec;

  tblSource.Close;
  tblSource.DatabaseName  := AliasSpec;
  tblSource.TableName     := TableSpec;

  try
    tblSource.Open;
  except
    On E: Exception do MessageDlg(E.Message,mtError,
                                  [mbOK], 0);
  end;
end;

procedure TFrmDDESrv.GetField(const NField: byte);
begin
  if (tblSource.Active) and
     (NField <= tblSource.FieldCount) then
    memItem.Text := tblSource.Fields[Pred(NField)].Text

  else
    MessageDlg('Unable to get field value',
               mtError, [mbOK], 0);
end;

procedure TFrmDDESrv.GotoRecord(const NRecNum: LongInt);
begin
  if (tblSource.Active) and
     (NRecNum <= tblSource.RecordCount) then
    begin
      tblSource.First;
      tblSource.Refresh;
      tblSource.MoveBy(Pred(NRecNum));
    end
  else
    MessageDlg('Unable to move to specified record',
               mtError, [mbOk], 0);
end;

procedure TFrmDDESrv.GetRec(const NRecNum: LongInt);
var
  i, j: byte;
begin
  if (tblSource.Active) and

     (NRecNum <= tblSource.RecordCount) then
    begin
      j := Pred(tblSource.FieldCount);
      DDEGetRecord.Lines.Clear; { Clear the DDE item data }

      tblSource.First;
      tblSource.Refresh;
      tblSource.MoveBy(Pred(NRecNum));

      memItem.Clear;

      for i := 0 to j do
        { Write each field in the record to the memo }
        memItem.Lines.Add(tblSource.Fields[i].Text);
      end
    else
      MessageDlg('Unable to get table record',
                 mtError, [mbOK], 0);
end;

function TFrmDDESrv.GetArgs(const BufStr: string): string;
var
  ArgStart,
  ArgEnd: byte;
begin
  ArgStart := 0;
  ArgEnd   := 0;

  { See if any brackets exist and
    that they're properly closed }
  ArgStart := Pos('(', BufStr);

  if (ArgStart > 0) then
    ArgEnd := Pos(')', BufStr);

  if (ArgStart > ArgEnd) then
    raise Exception.Create(
      'Unable to extract function argument');

  Result := Copy(BufStr, Succ(ArgStart),
                 ArgEnd - Succ(ArgStart));
end;

function TFrmDDESrv.GetFnName(
                         const BufStr: string): string;
var
  OpenBracketPos: byte;
begin
  OpenBracketPos := Pos('(', BufStr);

  if (OpenBracketPos <= 0) then
    raise Exception.Create(
      'Unable to extract function name');

  Result := Copy(BufStr, 1, Pred(OpenBracketPos));
end;

procedure TFrmDDESrv.FormDestroy(Sender: TObject);
begin
  tblSource.Close;
end;

procedure TFrmDDESrv.DDEOpenTableClose(Sender: TObject);
begin
  memEventLog.Lines.Add('Server close');
end;
```

```
procedure TFrmDDESrv.DDEOpenTableOpen(Sender: TObject);
begin
  memEventLog.Lines.Add('Server open');
end;

procedure TFrmDDESrv.DDEGetRecordPokeData(Sender: TObject);
begin
  { This flag avoids potential conflicts }
  FInPoke := True;
  if (DDEGetRecord.Lines.Count > 0) then
    begin
      memEventLog.Lines.Add('Server data poked (' +
                            DDEGetRecord.Lines[0] + ')');
      memItem.Lines := DDEGetRecord.Lines;
    end;

  FInPoke := False;
end;

procedure TFrmDDESrv.DDEGetRecordChange(Sender: TObject);
begin
  if (DDEGetRecord.Lines.Count > 0) then
     memEventLog.Lines.Add('Server data changed (' +
                           DDEGetRecord.Lines[0] + ')')
  else
     memEventLog.Lines.Add('Server data changed to null');
end;

procedure TFrmDDESrv.Clearlist1Click(Sender: TObject);
begin
  memEventLog.Lines.Clear;
end;

procedure TFrmDDESrv.memItemChange(Sender: TObject);
begin
  { Avoid conflict with poked data }
  if not FInPoke then
    DDEGetRecord.Lines := memItem.Lines;
end;

end.
```

End Listing One

Begin Listing Two — The CLNTDEMO Project

```
program Clntdemo;

uses
  Forms,
  Ddeclnt in 'DDECLNT.PAS' {Form1};

{$R *.RES}

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


unit Ddeclnt;

interface

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

const
  FnList: array [0..3] of string[10] =
    ('OpenTable', 'GetRecord', 'GotoRecord', 'GetField');

type
  TForm1 = class(TForm)
    ClientConv: TDdeClientConv;
    ClientItem: TDdeClientItem;

    Button1: TButton;
    Button2: TButton;
    memSrvr: TMemo;
    Button3: TButton;
    Button4: TButton;
    Label1: TLabel;
    edRecNo: TEdit;
    Label2: TLabel;
    rdgExec: TRadioGroup;
    Button5: TButton;
    GroupBox1: TGroupBox;
    CheckBox1: TCheckBox;
    CheckBox2: TCheckBox;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure ClientItemChange(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure CheckBox1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  ClientConv.FormatChars := CheckBox1.Checked;
  ClientConv.ConnectMode := ddeManual;
  ClientConv.SetLink('SRVRDEMO', 'DDEOpenTable');
  ClientItem.DDEItem := 'DDEGetRecord';

  if not ClientConv.OpenLink then
    ShowMessage('Unable to open the link');
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  ClientConv.CloseLink;
end;

procedure TForm1.ClientItemChange(Sender: TObject);
begin
  MessageBeep($FFFF);
  memSrvr.Lines := ClientItem.Lines
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  Buf: PChar;
  i:   byte;
```

```
begin
  i := Length(edRecNo.Text);
  if (i <= 0) then
    Exit;

  { Allocate memory for the null-terminated string }

  Buf := StrAlloc(Succ(i));
  if not ClientConv.PokeData(
     'DDEGetRecord',StrPCopy(Buf,edRecNo.Text)) then

    ShowMessage('PokeData failed');

    { Free memory used by Buf }

    StrDispose(Buf);
end;

procedure TForm1.Button4Click(Sender: TObject);
var
  Buf: PChar;
    i: byte;
  Cmd: string;
begin
  if (Length(edRecNo.Text) <= 0) then
    Exit;

  Cmd := FnList[rdgExec.ItemIndex]+'('+edRecNo.Text+')';
  Buf := StrAlloc(Succ(Length(Cmd)));

  if not ClientConv.ExecuteMacro(StrPCopy(Buf, Cmd),
                              CheckBox2.Checked) then


    ShowMessage('Unable to execute macro');
  StrDispose(Buf);  { Free memory used by Buf }
end;

procedure TForm1.Button5Click(Sender: TObject);
var
  Buf: PChar;
begin
  Buf := ClientConv.RequestData('DDEGetRecord');
  if (Buf <> nil) then
    memSrvr.SetTextBuf(Buf); { Assign the data to memo }

  {This must always be done after we've
   finished with the requested DDE data }
  StrDispose(Buf);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  ClientConv.CloseLink;   { Tidy up }
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  ClientConv.FormatChars := CheckBox1.Checked;
end;

end.
```

End Listing Two

By Kevin J. Bluck

Building a Better MessageDlg

The Undocumented CreateMessageDialog Function

One of the primary benefits of the object-oriented programming model is enhanced code reusability and customizability. You don't need objects to reuse code, however. Anybody who has ever used a run-time library function has capitalized on the principles of code reuse. Why bother inventing a routine to calculate a cosine when a programmer at Borland has already gone to the trouble?

Most Delphi programmers are familiar with the *MessageDlg* function (and perhaps its lesser-known brother *MessageDlgPos*) and use it frequently to communicate with their users. It's a slick little routine that displays a handsomely formatted modal dialog box — neatly sized to fit the text it contains. *MessageDlg* allows you to specify a glyph and a number of different pushbuttons to suit your immediate needs. Available from a single-line call, it saves you from cluttering your project with various special-use dialog boxes that otherwise must be created visually to facilitate simple communication with the user. *MessageDlg* is very handy indeed.

However, the *MessageDlg* function has a few shortcomings. Many programmers would like to specify a caption for their dialog boxes, but *MessageDlg* provides no means to do this. Furthermore, *MessageDlg* is silent. Many users of Delphi applications are heads-down data entry clerks, who probably won't see a dialog box appear and must be jolted from their industrious typing trances by a sound. This means the programmer must always precede the *MessageDlg* call with a call to the **MessageBeep** Windows API procedure, or another sound-generating procedure to provide the audio cue.

Furthermore, the dialog box can always be closed by pressing Enter. It's altogether too easy for users wrapped up in data entry to enter an erroneous value, receive the error dialog box, and close it by pressing Enter. They never know an error occurred. Ideally, *MessageDlg* would let you specify a caption, sound off automatically, and have a variant that beeps at all further keystrokes — forcing the user to close it by some method other than pressing Enter.

Making these changes would be simple if the dialog box generated by *MessageDlg* was created visually. Changing a couple of properties and events would solve the problem. However, *MessageDlg* dynamically creates and destroys the *TForm* object entirely within its code. Because a global variable or .DFM file is not available for handling these tasks, the trick is to obtain access to a reference to the dialog box's *TForm* object.

Reusing Existing Code

Fortunately, Borland provides exactly what we need: a routine that dynamically creates a message dialog box. It's in the Dialogs unit with the *MessageDlg* function, but is undocumented. Its declaration is:

```
function CreateMessageDialog(
  const Msg: string;
  AType: TMsgDlgType;
  AButtons: TMsgDlgButtons): TForm;
```

The *CreateMessageDialog* function does the dirty work of creating a *TForm* object, setting its properties, creating the appropriate glyph for the dialog box type, setting the text and sizing the dialog box to fit, and creating the specified buttons. *MessageDlg* merely calls this function to create the dialog box, shows it modally, frees it, and returns the modal result. A simple call to *CreateMessageDialog* returns a *TForm* reference to the fully constructed dialog box — ready for customization.

First, we must duplicate the identifiers used to specify the dialog box type and buttons so they are available throughout this unit. (Having to include the Dialogs unit just to access these identifiers is highly annoying.) The custom types are defined in Figure 1.

Adding the Caption and Beep

To provide the new functionality, we must declare functions to supplant *MessageDlg* and *MessageDlgPos*. First, however, we'll write a custom variant of the *CreateMessageDialog* function to be called by the new functions. The entire set of *MsgDlg* and *MsgDlgPos* parameters are passed to this function, which creates and customizes the dialog box:

```
function CreateMsgDlg(const Msg: string;
                      const Caption: string;
                      AType: TNewMsgDlgType;
                      Buttons: TNewMsgDlgButtons;
                      HelpCtx: LongInt;
                      X: Integer;
                      Y: Integer): TForm;
```

First, the *CreateMsgDlg* function uses the existing VCL method to create the default *MessageDlg TForm*. The *Result* variable is, of course, a *TForm* type since that is the function's return type. No other local variable is necessary.

Since the *CreateMessageDialog* function handles creating the *TForm*, it's unnecessary to call a constructor, although the *TForm* object must be destroyed later. Note that the *AType* and *Buttons* parameters must be cast into the equivalent types defined in the Dialogs unit, or a compiler "Type Mismatch" error will occur:

```
Result := CreateMessageDialog(Msg,TMsgDlgType(AType),
                              MsgDlgButtons(Buttons));
```

Next, *CreateMsgDlg* accomplishes the first of our primary objectives: it sets the dialog box's caption. (It already has the default caption that was provided by the VCL's *CreateMessageDlg* function.) If the *Caption* parameter is an empty string, the caption is unchanged. Otherwise, it sets the *TForm* object's *Caption* property to the parameter's value:

```
if Caption <> EmptyStr then
  Result.Caption := Caption;
```

Now the function sets the dialog box's position — which is initially centered — on the screen. If the *X* or *Y* parameters are -1, the function leaves the dialog box centered along that axis. Otherwise, it sets the *TForm* object's *Left* property to *X*, and *Top* property to *Y*:

```
if X > -1 then
  Result.Left := X;
if Y > -1 then
  Result.Top := Y;
```

Next, we'll set the *TForm* object's *HelpContext* property directly from the *HelpCtx* parameter and make sure that it's scaled properly for the current screen resolution. The dialog box is designed for a standard VGA resolution of 96 pixels per inch. Calling the *TForm's ScaleBy* method, with the actual pixels per inch of the current screen and the normal value of 96, guarantees that the dialog box appears the same size, regardless of the user's screen resolution:

```
Result.HelpContext := HelpCtx;
Result.ScaleBy(Screen.PixelsPerInch, 96);
```

Finally, we'll call the API procedure **MessageBeep** to generate the desired beep, according to the dialog box's type. Using the MB_ICON constants causes **MessageBeep** to play the system sounds associated with the dialog box type in the Control Panel (if the user has the necessary hardware and system sound enabled):

```
case AType of
  mtWarning:      MessageBeep(MB_ICONEXCLAMATION);
  mtError:        MessageBeep(MB_ICONSTOP);
  mtInformation:  MessageBeep(MB_ICONINFORMATION);
  mtConfirmation: MessageBeep(MB_ICONQUESTION);
end;
```

The dialog box provided by the VCL's *CreateMessageDialog* function is now customized for our needs. Now it's time to implement the *MsgDlg* and *MsgDlgPos* functions that display the dialog box.

MsgDlgPos simply passes its parameters to *CreateMsgDlg* to create the dialog box, shows the dialog box modally, assigns the modal result as the return value, and frees the dialog box object (see Figure 2).

```
type
  TNewMsgDlgType = (mtWarning, mtError, mtInformation,
                    mtConfirmation, mtCustom);
  TNewMsgDlgBtn  = (mbYes, mbNo, mbOK, mbCancel, mbAbort,
                    mbRetry, mbIgnore, mbAll, mbHelp);
  TNewMsgDlgButtons = set of TNewMsgDlgBtn;

const
  mbYesNoCancel      = [mbYes, mbNo, mbCancel];
  mbOKCancel         = [mbOK, mbCancel];
  mbAbortRetryIgnore = [mbAbort, mbRetry, mbIgnore];
```

Figure 1: The custom identifiers.

```
function MsgDlgPos(const Msg: string;
                   const Caption: string;
                   AType: TNewMsgDlgType;
                   Buttons: TNewMsgDlgButtons;
                   HelpCtx: LongInt;
                   X: Integer;
                   Y: Integer): TModalResult;
var
  { Handle to the dynamically created dialog box. }
  Dlg: TForm;
begin
  Result := 0;
  try
    Dlg := CreateMsgDlg(Msg, Caption, AType, Buttons,
                        HelpCtx, X, Y);
    Result := Dlg.ShowModal;
  finally
    Dlg.Free;
  end;
end;
```

```
function MsgDlg const Msg: string;
                const Caption: string;
                AType: TNewMsgDlgType;
                Buttons: TNewMsgDlgButtons;
                HelpCtx: LongInt): TModalResult;
begin
  Result := MsgDlgPos(Msg, Caption, AType,
                      Buttons, HelpCtx, -1, -1);
end;
```

Figure 2 (Top): The *MsgDlgPos* function.
Figure 3 (Bottom): Causing *MsgDlgPos* to display the dialog box centered on-screen.

MsgDlg is even simpler. It calls *MsgDlgPos* with default *X* and *Y* arguments of -1, causing *MsgDlgPos* to display the dialog box centered on the screen (see Figure 3).

Making the Dialog Box Persistent

So far, we've accomplished two of our objectives with no great fuss. Now, let's implement the persistent version, which won't close using Enter and beeps at keystrokes. This promises to be more complicated.

Preventing the dialog box from allowing Enter to click on the default button might seem straightforward, but proves to be fairly complex. First, we must set the default button's *Default* property to *False*. The problem is that we have no idea which button is the default, nor any variable to reference it. Furthermore, in this case, it's not enough to disable the default button. Since the only focusable controls in the dialog box are the buttons, by definition, one of them always has focus. As some experimentation will prove, it's impossible to prevent a focused button from being clicked by Enter using any of the key handling events, either for the button or form. Apparently, the keystroke invokes the button click before any event handlers are called.

The trick is to ensure that none of the pushbutton controls begin with focus. Since they are the only focusable controls on the form as it's created normally, we must provide a custom control to invisibly trap the focus away from the buttons. We'll accomplish this in the same way the *CreateMessageDlg* function created the button controls — by dynamically creating a component at run time and placing it on the *TForm* object.

The logical choice for our invisible control is a simple *TEdit*. The *TEdit* control should be invisible to users so that they are unaware of what's occurring behind the scenes. The obvious way to do this is to set the *Visible* property to *False.* However, that defeats our purpose for placing the control by making it incapable of receiving focus. Instead, we'll manipulate other properties and allow the *TEdit* to blend with the form's background to be effectively invisible. Figure 4 contains the code fragment that accomplishes the dynamic creation and camouflage of the *TEdit*.

```
TrapFocus                := TEdit.Create(Result);
TrapFocus.Parent         := Result;
TrapFocus.Name           := 'TrapFocus';
TrapFocus.AutoSelect     := False;
TrapFocus.BorderStyle    := bsNone;
TrapFocus.Color          := Result.Color;
TrapFocus.Ctl3D          := False;
TrapFocus.ReadOnly       := True;
TrapFocus.TabOrder       := 0;
TrapFocus.Text           := EmptyStr;
TrapFocus.Width          := 0;
```

Figure 4: Dynamically creating and camouflaging a *TEdit* component.

TrapFocus is a variable of *TEdit* type. Recall that *Result* is the function's return value, which in the case of *CreatePersistentMsgDlg*, is of type *TForm*. First, the *TEdit* object is created in memory by calling the **Create** constructor with the dialog box *TForm* object as its owner. Setting the *TEdit's* owner is important because the owner is responsible for de-allocating its owned objects when it's destroyed. Our code won't have the chance to manually destroy the *TEdit*, so we'll ensure that the *TForm* object does it for us.

Next, we must set the *TEdit* control's *Parent* property to the *TForm* object as well. A windowed control such as *TEdit* must have a parent window assigned to it immediately after construction and before doing anything else with the object. Otherwise, a General Protection Fault will probably occur. While the *Parent* property represents the Windows concept of Parent and Child windows, the *Owner* property controls ownership of the Pascal objects representing those Windows entities. During normal visual form construction in the Delphi IDE, the *Parent* property is set automatically for windowed controls when they are dropped on the form. However, in this case *Parent* must be set manually.

The remainder of the property assignments are aimed at making the control invisible and ensuring that it has the focus when the dialog box is shown. Note that the color of the *TEdit* object is set to the form color because it's dangerous to assume the form will be any particular color. *ReadOnly* is set to *True* so the control won't accept input, and *TabOrder* is set to 0 to guarantee it's the first control in tab order. The frame is turned off, and the width is set to 0 so that the blinking cursor will be invisible.

Eliminating Enter

Unfortunately, we still must eliminate the user's ability to close the form with Enter. One of the pushbuttons will probably have its *Default* property set to *True*, which activates the button when Enter is pressed (even when the button doesn't have focus). We must check each *TBitButton* object and verify that the *Default* property is *False*. With variables referencing these objects, it would be a simple matter. However, without them, we only have the reference to the *TForm* object.

Fortunately, we still have a way to access these controls. You'll recall that we set the *TrapFocus TEdit* object's *Parent* property to the *TForm* object when we created it. This was also done for the *TBitButton* objects when they were created. One of the effects of assigning the *Parent* property is that the parent control inserts a reference to the child control in its *Controls* property. The *Controls* property is an array of *TControl* objects that are children of the parent. The *ControlCount* property returns how many controls are children of the parent.

Using these properties with a **for** loop, we can iterate through the *TForm's Controls* property, determine which controls are *TButton* objects (or descendants of *TButton*, such as *TBitButton*) using the run-time type information (RTTI) operator **is**, and set their *Default* properties to *False*. This code fragment accomplishes this task:

```
for ControlCtr := 0 to (Result.ControlCount - 1) do begin
  if (Result.Controls[ControlCtr] is TButton) then begin
TButton(Result.Controls[ControlCtr]).Default := False;
  end;
end;
```

Note that *Controls* is a zero-based array, so its range of indexes runs from 0 to *ComponentCount* - 1. In addition, its elements are *TControl* objects, so they must be cast to *TButton* before the *Default* property is accessible.

An Earful

Beeping at keystrokes is neatly accomplished by setting the *TForm* object's *KeyPreview* property to *True* and assigning a handler to the *OnKeyDown* event. Normally, this would be a straightforward operation. However, we are not creating a *TForm* descendant as we normally do when visually designing a form. This message dialog box is actually a *TForm*, so the code for the handler must come from somewhere else.

An event handler is actually a property whose data type is a pointer to a procedure. This procedure pointer is declared as a specific type of procedure, and therefore, only procedures with a given list of parameters can be assigned to a given event. The *OnKeyDown* event points to a procedure of type *TKeyEvent* that is declared as:

```
procedure(Sender: TObject;
          var Key: Word;
          Shift: TShiftState) of object;
```

The qualifier **of object** means the procedure must be a member method of a class. It makes a difference because class methods have an "invisible" parameter passed to them — a reference to the object instance that's calling the method. This is known by programmers as the *Self* variable. Procedures that are not members of a class are therefore not type-compatible with procedures that are members.

To obtain a procedure suitable for assignment to the *OnKeyPress* event handler, we'll declare a "throwaway" class descended from *TComponent* that includes only a member procedure of type *TKeyEvent*:

```
type
  TKeyDownHandler = class(TComponent)
  public
    procedure KeyBeep(Sender: TObject;
                      var Key: Word;
                      Shift: TShiftState);
  end;
```

The message handler procedure is quite simple. It beeps at all keystrokes and nullifies them by setting the *Key* parameter to 0, except for Esc. Note that the user can still navigate between the form's focusable controls using Tab. This is because Windows handles the Tab keystroke before it's submitted to the form window for action:

```
procedure TKeyDownHandler.KeyBeep(Sender: TObject;
                                  var Key: Word;
                                  Shift: ShiftState);
begin
  if Key <> VK_ESCAPE then begin
    MessageBeep(MB_ICONEXCLAMATION);
    Key := 0;
  end;
end;
```

Since a reference to a specific instance of the class is passed to class member procedures, an instance of the class must be instantiated before we can safely assign the member procedure to the *TForm*'s event. If an instance does not exist, the dreaded General Protection Fault will probably occur when the handler is called. By setting the *KeyDownHandler* object's *Owner* to the *TForm*, we ensure that the memory allocated to the object is freed when the *TForm* is freed. We also set the *TForm's KeyPreview* property to *True* to make sure the form receives the keystroke before the focused control:

```
KeyDownHandler    := TKeyDownHandler.Create(Result);
Result.OnKeyDown  := KeyDownHandler.KeyBeep;
Result.KeyPreview := True;
```

The *PersistentMsgDlgPos* and *PersistentMsgDlg* functions are direct echoes of the *MsgDlgPos* and *MsgDlg* functions. The *PersistentMsgDlgPos* creates the dialog box's *TForm* object using the *CreatePersistentMsgDlg* function, shows the dialog box modally, frees it, and returns the modal result. *PersistentMsgDlg* merely calls *PersistentMsgDlgPos* using the -1 position parameters to center the dialog box on the screen.

For the complete listing of the MsgDlgs unit, see Listing Three on page 28. A project that demonstrates the unit's functions (see Figure 5) is available on diskette or for download.

Conclusion

Delphi's extensibility is one of its greatest advantages, and you aren't limited to merely extending component objects. As we've seen, it's possible to extend or replace run-time functions and procedures as well. It's also possible to access components and forms at run time, enabling the developer to modify objects not known to exist at design time. Creatively using these concepts will enable you to produce seemingly magical effects in your applications. ∆

Figure 5: The demonstration project makes use of the new dialog box functions.

The MsgDlgs unit and demonstration application referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\JAN\DI9601KB.

Kevin J. Bluck is the Senior Programmer at Lender Service Bureau of America, a real estate servicing company in Sacramento, CA, where he develops applications using Delphi and InterBase 4.0. He can be reached on CompuServe at 74552,201, by e-mail at KevinBluck@aol.com, or by mail at LSBOA, 555 University Ave. #125, Sacramento, CA 95825.

Begin Listing Three — The MsgDlgs Unit

```
unit MsgDlgs;

interface

uses Forms;

type
  TNewMsgDlgType = (mtWarning, mtError, mtInformation,
                    mtConfirmation, mtCustom);
  TNewMsgDlgBtn = (mbYes, mbNo, mbOK, mbCancel, mbAbort,
                   mbRetry, mbIgnore, mbAll, mbHelp);
  TNewMsgDlgButtons = set of TNewMsgDlgBtn;

const
  mbYesNoCancel      = [mbYes, mbNo, mbCancel];
  mbOKCancel         = [mbOK, mbCancel];
  mbAbortRetryIgnore = [mbAbort, mbRetry, mbIgnore];

{ Dynamically creates an improved message dialog box. }
function CreateMsgDlg(const Msg: string;
                      const Caption: string;
                      AType: TNewMsgDlgType;
                      Buttons: TNewMsgDlgButtons;
                      HelpCtx: Longint;
                      X, Y: Integer): TForm;

{ Similar to MessageDlg. Allows caption to be specified.
  Beeps automatically. }
function MsgDlg(const Msg: string;
                const Caption: string;
                AType: TNewMsgDlgType;
                Buttons: TNewMsgDlgButtons;
                HelpCtx: Longint): TModalResult;

{ Similar to MessageDlgPos. Allows caption to be specified.
  Beeps automatically. }
function MsgDlgPos(const Msg: string;
                   const Caption: string;
                   AType: TNewMsgDlgType;
                   Buttons: TNewMsgDlgButtons;
                   HelpCtx: Longint;
                   X, Y: Integer): TModalResult;

{ Dynamically creates a persistent message dialog. }
function CreatePersistentMsgDlg(
  const Msg: string;
  const Caption: string;
  AType: TNewMsgDlgType;
  Buttons: TNewMsgDlgButtons;
  HelpCtx: Longint; X, Y: Integer): TForm;

{ Similar to MsgDlg. Enter does not close and beeps
  at all keystrokes. For head-down data entry where
  user may not be paying attention to the screen. }
function PersistentMsgDlg(
  const Msg: string; const Caption: string;
  AType: TNewMsgDlgType; Buttons: TNewMsgDlgButtons;
  HelpCtx: Longint): TModalResult;

{ Similar to MsgDlgPos. Enter does not close and beeps
  at all keystrokes. For head-down data entry where user
  may not be paying attention to the screen. }
function PersistentMsgDlgPos(
  const Msg: string; const Caption: string;
  AType: TNewMsgDlgType; Buttons: TNewMsgDlgButtons;
  HelpCtx: Longint; X, Y: Integer): TModalResult;

implementation

uses
  SysUtils, Classes, Graphics, StdCtrls, Dialogs,
   WinProcs, WinTypes;

{ ***** Message Dialog Methods ***** }

{ Dynamically creates an improved message dialog box. }
function CreateMsgDlg(const Msg: string;
                      const Caption: string;
                      AType: TNewMsgDlgType;
                      Buttons: TNewMsgDlgButtons;
                      HelpCtx: Longint;
                      X, Y: Integer): TForm;

begin
  { Use the existing VCL method to dynamically create a
    MessageDlg. }
  Result := CreateMessageDialog(Msg,
                             TMsgDlgType(AType),
                             TMsgDlgButtons(Buttons));
  { If user specified a caption, set the dialog box's
    caption. }
  if Caption <> EmptyStr then begin
    Result.Caption := Caption;
  end; { if }

  { Set the dialog's position on screen. }
  if X > -1 then begin
    Result.Left := X;
  end; { if }
  if Y > -1 then begin
    Result.Top := Y;
  end; { if }

  { Set the help context and ensure correct video
    scaling. }
  Result.HelpContext := HelpCtx;
  Result.ScaleBy(Screen.PixelsPerInch, 96);

  { Beep. Show the dialog modally and return the
    ModalResult. }

  case AType of
    mtWarning:      MessageBeep(MB_ICONEXCLAMATION);
```

```
    mtError:        MessageBeep(MB_ICONSTOP);
    mtInformation:  MessageBeep(MB_ICONINFORMATION);
    mtConfirmation: MessageBeep(MB_ICONQUESTION);
  end;
end;

{ Similar to MessageDlgPos. Allows caption to be
  specified. }
function MsgDlgPos(const Msg: string;
                   const Caption: string;
                   AType: TNewMsgDlgType;
                   Buttons: TNewMsgDlgButtons;
                   HelpCtx: Longint;
                   X, Y: Integer): TModalResult;
var
  Dlg: TForm;  { Handle to the dynamically created
                 dialog box. }
begin
  Result := 0;
  try
    { Create message dialog box and show it, returning
     the modal result. }
    Dlg := CreateMsgDlg(Msg, Caption, AType, Buttons,
                        HelpCtx, X, Y);
    Result := Dlg.ShowModal;
  { Ensure dialog memory is freed. }
  finally
    Dlg.Free;
  end; { finally }
end;

{Similar to MessageDlg. Allows caption to be specified.}

function MsgDlg( const Msg: string;
                 const Caption: string;
                 AType: TNewMsgDlgType;
                 Buttons: TNewMsgDlgButtons;
                 HelpCtx: Longint): TModalResult;

begin
  { Calls MsgDlgPos with default position arguments. }
  Result := MsgDlgPos(Msg, Caption, AType, Buttons,
                      HelpCtx, -1, -1);
end;

{ ***** Persistent Message Dialog methods ***** }

{ Provides an object to provide a method of type
  TKeyEvent (an Of Object method type) to assign to
  the persistent message dialog.}
type
  TKeyDownHandler = class(TComponent)
  public
    procedure KeyBeep(Sender: TObject;
                      var Key: Word;
                      Shift: TShiftState);
  end;

{ Method of type TKeyEvent for assigning to the
  persistent message dialog's OnKeyDown event.
  Beeps at all keystrokes except the Escape key.}
procedure TKeyDownHandler.KeyBeep(Sender: TObject;
                                  var Key: Word;
                                  Shift: TShiftState);
begin
  if Key <> VK_ESCAPE then begin
      MessageBeep(MB_ICONEXCLAMATION);
      Key := 0;
  end; { if }
end;

{ Dynamically creates a persistent message dialog box. }
function CreatePersistentMsgDlg(
  const Msg: string; const Caption: string;
  AType: TNewMsgDlgType; Buttons: TNewMsgDlgButtons;

  HelpCtx: Longint; X, Y: Integer): TForm;
var
  { Dummy object to provide an OnKeyDown handler. }
  KeyDownHandler: TKeyDownHandler;
  { A TEdit control to trap focus away from the
    pushbuttons.}
  TrapFocus: TEdit;
  { Counter to iterate through dialog's controls. }
  ControlCtr: Integer;
begin
  { Dynamically create an improved MsgDlg. }
  Result := CreateMsgDlg(Msg, Caption, AType, Buttons,
                         HelpCtx, X, Y);

    { Create an invisible TEdit control that will
      trap focus away from the pushbuttons so that
      Return and Space will not click them.}
    TrapFocus := TEdit.Create(Result);
    TrapFocus.Parent := Result;
    TrapFocus.Name := 'TrapFocus';
    TrapFocus.AutoSelect := False;
    TrapFocus.BorderStyle := bsNone;
    TrapFocus.Color := Result.Color;
    TrapFocus.Ctl3D := False;
    TrapFocus.ReadOnly := True;
    TrapFocus.TabOrder := 0;
    TrapFocus.Text := EmptyStr;
    TrapFocus.Width := 0;
    { Create an instance of the KeyDownHandler object
      with the dialog box as owner to ensure that the
      KeyDownHandler object will be freed along with the
      dialog box. Set the dialog's OnKeyDown handler to
      point to the KeyDownHandler object's KeyBeep
      method. Also set the form to preview keystrokes.}
    KeyDownHandler    := TKeyDownHandler.Create(Result);
    Result.OnKeyDown  := KeyDownHandler.KeyBeep;
    Result.KeyPreview := True;

    { Iterate through form's controls. Ensure every
      pushbutton's Default property is set to False
      to prevent the Return key from closing the
      dialog box.}
    for ControlCtr :=
      0 to (Result.ControlCount - 1) do begin
        if (
          Result.Controls[ControlCtr] is TButton) then begin
            TButton(
              Result.Controls[ControlCtr]).Default :=
                False;
        end; { if }
    end; { for }
end;

{ Similar to MsgDlgPos. Enter does not close and beeps
  at all keystrokes. For head-down data entry where user
  may not be paying attention to the screen.}
function PersistentMsgDlgPos(
  const Msg: string; const Caption: string;
  AType: TNewMsgDlgType; Buttons: TNewMsgDlgButtons;
  HelpCtx: Longint; X, Y: Integer): TModalResult;
var
  { Handle to the dynamically created dialog. }
  Dlg: TForm;
begin
  Result := 0;
  try
    { Dynamically create the persistent message
      dialog box. }
    Dlg := CreatePersistentMsgDlg(Msg, Caption,
                                  AType, Buttons,
                                  HelpCtx, X, Y);
    { Show dialog box and return modal result. }
    Result := Dlg.ShowModal;
```

```
  { Ensure dialog box's allocated memory is freed. }
  finally
    Dlg.Free;
  end; { finally }
end;

{ Similar to MsgDlg. Enter does not close and beeps
  at all keystrokes. For head-down data entry where
  user may not be paying attention to the screen.}
function PersistentMsgDlg(
  const Msg: string; const Caption: string;
  AType: TNewMsgDlgType; Buttons: TNewMsgDlgButtons;
  HelpCtx: Longint): TModalResult;

begin
  { Calls PersistentMsgDlgPos with default position arguments.}
  Result := PersistentMsgDlgPos(Msg, Caption,
                                AType, Buttons,
                                HelpCtx, -1, -1);
end;
end.
```

End Listing Three

By Cary Jensen, Ph.D.

Filtering Tables: Part I

Displaying Selected Information in a Delphi Database Application

When providing information to the user, there are times when you may want to display only some of the records from a table. For example, while you might have a table that contains one record for each item purchased by each customer, you may only want to display those items bought by a particular customer on a given date.

This month's "DBNavigator" is the first of two articles that examine how you can display a subset of a table's records. In this installment, two techniques that employ the Table component are considered: linked tables and table ranges. Next month, we'll discuss a technique that employs the Query component.

Let's begin by considering why it's sometimes necessary to display less than all of a table's records.

Less Is More

Most of the databases that you create with Delphi are *relational*. In a relational database, the data is stored in more than one table, a table being a file that holds data in a structured format. For example, while a sales database may include a main table whose fields identify the individual items purchased, there are often many support tables. For instance, there are probably tables that hold information about the invoice under which the individual items were purchased; current customers who can buy items; employees who can write up sales; product vendors; available product lines; products on order; and so forth.

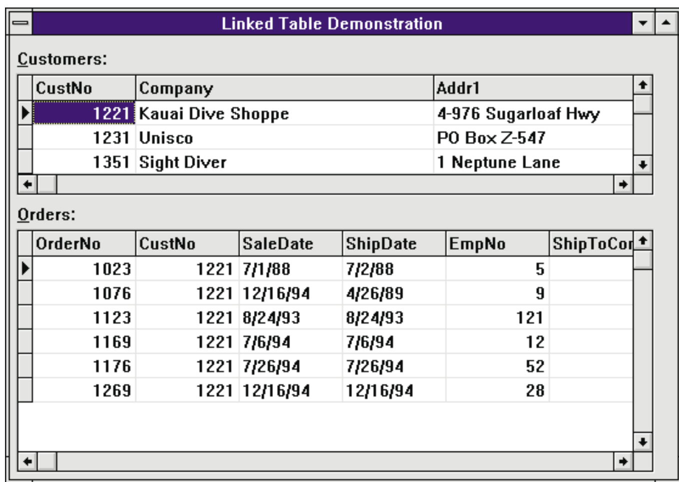
While many tables are used, they are not independent. For example, the main sales table that contains the items purchased usually includes an invoice number, or some other value that is assigned to all items purchased by the same customer during an order. This same invoice number can also be found in the invoice table, which holds the specifics of the sales transaction, including a customer ID that identifies the customer making the purchase; an employee ID identifying the employee responsible for the sale; the date of the sale; etc. Furthermore, the customer number found in the invoice table corresponds to a customer number in the customer table. This table holds information unique to the customer, such as their mailing address, billing address, line of credit, and any other pertinent information.

In other words, the tables of the database are "related." This relation is based on the correspondence of data between tables. For example, the main sales table is related to the invoice table based on the invoice number field; the invoice table is related to the customer table based on the customer field; the main sales table is related to the product table based on a product number field; and so forth.

In the preceding example, the tables are related by a single field, but this is not always the case. In some instances tables are related by two or more fields. For example, in a course enrollment database, two or more instances of a course offered in a specific term can be distinguished by three pieces of information: the term identifier (semester/year), course number, and section number. A table designed to hold student course credits would need to include all three of these fields, and therefore can be linked to the course table by means of these three fields.

Displaying Linked Tables

Using a Table component, there are two basic techniques that you can use to limit the records. The first involves using linked tables. When two tables are linked, the records available within one table — the *detail* — are limited to those that match the current record in another table — the *master* — based on the linked fields. An example of a form using linked tables is shown in Figure 1. Notice that only those records matching the Customer number in the Customer table are displayed in the Orders table.



Linked Table Demonstration

Customers:

CustNo	Company	Addr1
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy
1231	Unisco	PO Box Z-547
1351	Sight Diver	1 Neptune Lane

Orders:

OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToCo
1023	1221	7/1/88	7/2/88	5	
1076	1221	12/16/94	4/26/89	9	
1123	1221	8/24/93	8/24/93	121	
1169	1221	7/6/94	7/6/94	12	
1176	1221	7/26/94	7/26/94	52	
1269	1221	12/16/94	12/16/94	28	

Figure 1: A form that uses linked tables.

The easiest way to create linked tables is to use the Database Form Expert, shown in Figure 2. There are two ways to access it. First, you can select **Help | Database Form Expert** from Delphi's main menu. Second, if you have enabled the Gallery for use with forms, select the **Database Form** expert from the Experts page of the Form Gallery after selecting **File | New Form**. (You can enable the Gallery for use with new forms by checking the **Use on New Form** option in the **Gallery** group on the Preferences page of the Environment Options dialog box.)

Once the Database Form Expert is loaded, enable **Create a master/detail form** in the **Form Options** group on the first page of this dialog box. The expert will then guide you through the process of selecting the master and detail tables, and identifying the fields upon which the link is based.

While the Database Form Expert simplifies the process of creating a form with linked tables, you can best understand how linked tables work by creating an example from scratch. The form in Figure 1 was built using linked tables. Use the following steps to produce this form.

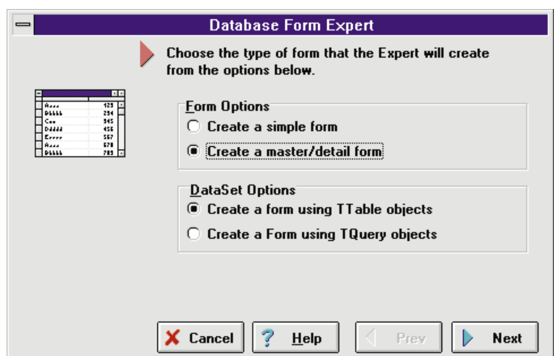


Figure 2: The first screen of the Database Form Expert.

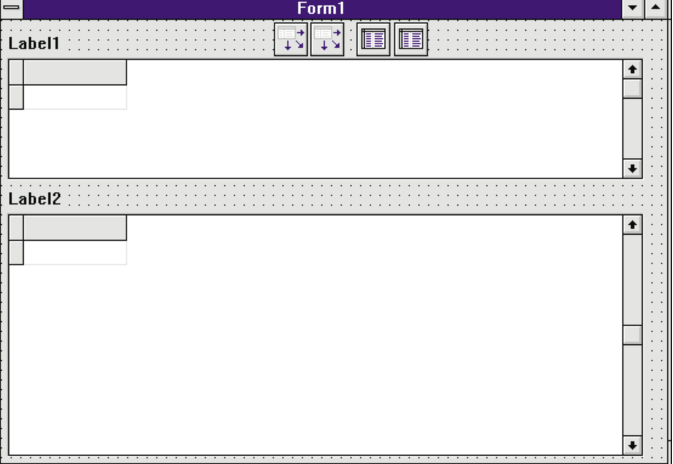


Figure 3: The basic components for creating a linked form.

Begin by creating a new project. On the new form place two DataSource components, two Table components, two DBGrids, and two Label components. When you are done, your form should resemble Figure 3.

Set the *Caption* property of *Label1* to `&Customer:`, and its *FocusControl* property to *DBGrid1*. Set the *Caption* property of *Label2* to `&Orders:`, and its *FocusControl* to *DBGrid2*. Establishing this relationship allows Delphi to connect the hot key shortcut defined in the label (i.e. `&Customer:`) with its associated grid component. Therefore, when the user presses Alt C, *Label1* will "catch" the hot key and transfer focus to the component defined in its *FocusControl* property — in this case *Grid1*.

Set the *Caption* property of *Form1* to `Linked Table Demonstration`.

Now, set the *DataSet* property of *DataSource1* to *Table1* and set the *DataSet* property of *DataSource2* to *Table2*. Next, set the *DataSource* property of *DBGrid1* to *DataSource1*, and the *DataSource* property of *DBGrid2* to *DataSource2*.

With *Table1*, set *DatabaseName* to `DBDEMOS` (the alias defined by the Delphi installation program), *TableName* to `CUSTOMER.DB`, and *Active* to *True*. With *Table2*, set *DatabaseName* to `DBDEMOS`, *TableName* to `ORDERS.DB`, and *Active* to *True*.

At this point you have a form that resembles the one in Figure 4. Note that although this form does show data from both the

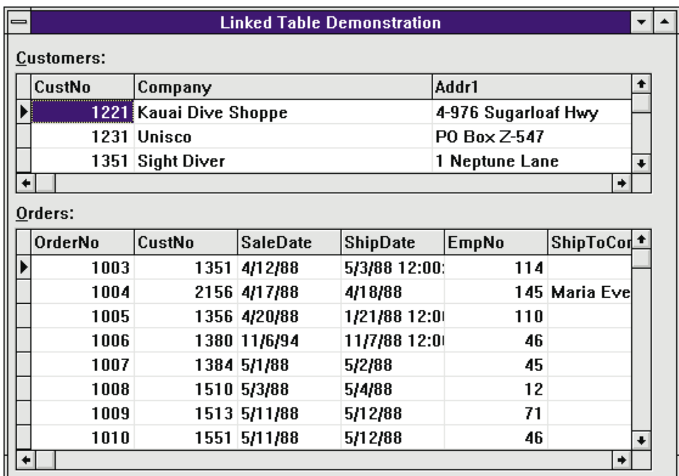


Figure 4: The two tables in this figure are not linked. Notice that the Orders table records are not limited to the currently displayed record in the Customer table.

Customer and Orders tables, all the records of both tables are visible (or would be if you scrolled the tables). To limit the display of the records in the Orders table to those associated only with the currently selected customer, it's necessary to link the Orders table to the Customer table.

Vital Links

Linking *Table2* to *Table1* requires three steps. First, you must ensure that the current table index contains the field or fields that are associated with the link. In this example, the Orders table must be linked to the Customer table based on the CustNo field. Since the default index (the primary index) of the Orders table is not based on the CustNo field, you must select a secondary index for the Orders table that is based on this field. To do this, set the *IndexName* property of *Table2* to `CustNo`.

(It's important to note that the CustNo secondary index exists because it was created by whoever built the sample tables — Paradox tables in this case — for Delphi. Usually, you are required to actually create an appropriate secondary index for a table before you can link it. While this can be done using Object Pascal code, it's more convenient to create this index interactively using the Database Desktop, a separate application that is installed with Delphi. From the Database Desktop, select **Utilities | Restructure**. From the Restructure dialog box select **Secondary Indexes** from the **Table Properties** combobox, and then click the **Define** button. Use the displayed dialog box to create a secondary index.)

The second step in creating linked tables is to set the *MasterSource* property of your detail table to the DataSource associated with the master table. In this example, set *Table2*'s *MasterSource* property to *DataSource1*.

Finally, you must define the linked fields — the fields upon which the detail and master tables are linked. The easiest way to do this is with the Field Link Designer. This is the property editor for the detail table's *MasterFields* property. With *Table2* selected, select the *MasterFields* property in the Object

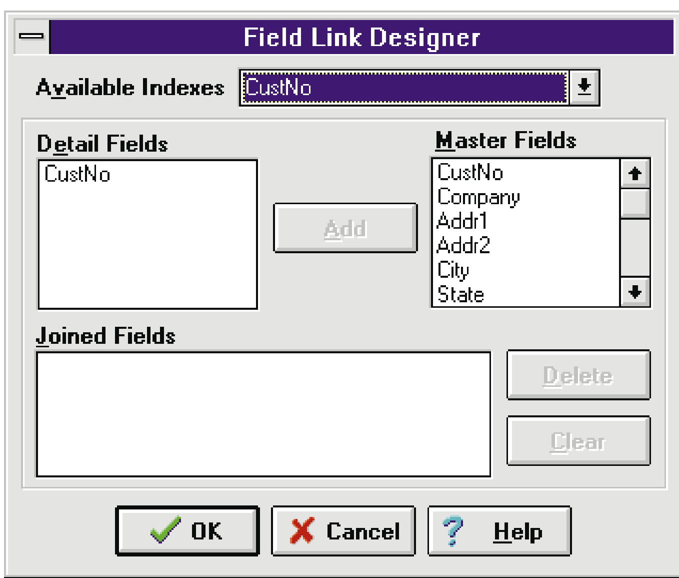


Figure 5: Use the Field Link Designer to select the field pairs that form the link between the detail and master tables.

Inspector and click the **...** button to display the Field Link Designer (see Figure 5).

In this dialog box, select the first field of the current detail table index from the **Detail Fields** list, select the master table's corresponding field from the **Master Fields** list, and then click the **Add** button. The linked field pairs are now displayed in the **Joined Fields** list, and the selected fields are removed from their respective lists, as shown in Figure 6. If the detail table index consists of more than one field, repeat this process for each of the index's remaining fields. (Notice the Field Link Designer also contains an **Available Indexes** combobox for setting or changing the detail table index that is the basis for the link.)

Once you accept the Field Link Designer, you'll notice that *DBGrid2* will display records from only one customer — the customer currently selected in *DBGrid1*. The form is now complete. Run it to display the form shown in Figure 1.

A few final comments about linked tables are in order. First, the master table in this example employs a DBGrid. While this produced a useful interface, you can also display the master table in a single-record interface by using DBEdit components and other single-record data-aware controls. In fact, if you create a linked master-detail form using the Database Form Expert, this is precisely the type of form you'll produce.

It's important to note that you can also benefit from linked tables in cases when no data-aware controls are used. Specifically, you may want to produce linked tables for background operations where the user does not view or interact with the tables being manipulated.

To do this, your form only requires two Table components — one for the master table and one for the detail table. In addition, you have to add one DataSource. It must point to the Table component for the master table, and the detail table's *MasterSource*

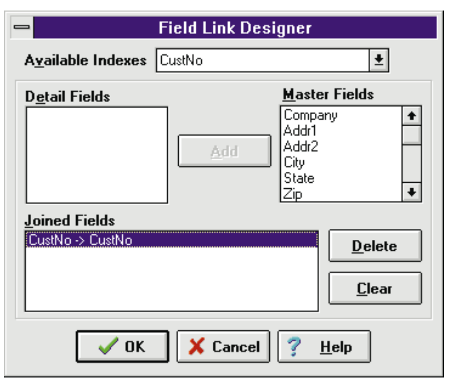


Figure 6: The Field Link Designer showing the CustNo field link between the detail and master tables.

property will point to the DataSource. This is, in fact, the only situation I am aware of where a DataSource component is necessary in the absence of data-aware controls. Typically, the purpose of a DataSource component is to provide an interface between a DataSet descendant and data-aware controls.

Finally, while these linked tables were defined during design time, it's also possible to produce linked tables at run time. Similar to the design time process, this requires assigning values to the *MasterSource*, *IndexName* (or alternatively *IndexFields*), and *MasterFields* properties. The main difference is that you do not use the Field Link Designer to assign a value to the *MasterFields* property at run time. (See the online help reference for the *MasterFields* property for an example of assigning values to it at run time.)

Using Ranges

While linked tables are useful, there are times when you may want to restrict the display of records in a table to a subset of records. However, the database does not include a master table for this purpose. For example, imagine that you want to display records only for those customers living in a particular city. If you do not have a table that includes all possible cities of residence, linked tables are not an option.

The Table component encapsulates a feature of the Borland Database Engine (BDE) that permits you to define ranges of records for display. A range, when applied, restricts the display of data to those records that lie within the range.

When applying a range, you define the lowest and highest value in a field, or set of fields. For example, let's say you want to display only those customer records where the credit limit is $1,000 or less. To do so, you identify the beginning of the range (the lowest value) as 0, and the end of the range (the highest value) as 1,000. When applied, only those records where the credit limit lies inclusively within the specified range are displayed.

Ranges can also be used to display exact matches. To display records for a customer whose mailing address is a particular city, New York for example, you can set the beginning and end of the range to the same value.

There is one requirement for using ranges: the table you want to apply the range to must be indexed on the field (or fields) used for the range. Consequently, to apply a range based on a customer's credit limit, there must be at least one index where credit limit is the first field in the index. Likewise, a city-based range requires at least one index where city is the first index field.

Ranges are not limited, however, to single fields. You can also create a range that defines starting and ending values for two or more fields. Of course, this requires that the table you are applying the range to has an index that includes those fields as the first fields in the index.

For example, to see the list of all students enrolled in a particular course during a specific term, regardless of section number, you can assign the same term value as both the starting and ending range on the term field, and the same course number as the starting and ending range on the course number field. Provided there is an index that has term and course number as the index's first fields, this range is acceptable. Specifically, you can use an index that is based on term, course number, and section number, even though no range on section number is defined.

You have two options when it comes to applying a range. The first and easiest to use is the *SetRange* method. It has the following syntax:

procedure SetRange(const *StartValues*, *EndValues*: array of const)

Both arrays that are passed as parameters must have the same number of elements. The value in the first element of the first array corresponds to the beginning (or lowest value) for the range on the first field of the index. The value in the second element, if provided, identifies the lowest value for the range on the second field of the index, and so on. The elements of the second array identify the ending, or highest values of the range for each field, with the first element corresponding to the first field in the index; the second, if provided, for the second field in the index, and so on.

The following demonstrates how *SetRange* can be used. Assume that *Table1* is a component defined for a table named CLIENTS.DB. Furthermore, assume this table has an index named CityIndex that is a single field index on the City field of CLIENTS.DB. This statement:

```
Table1.IndexName := 'CityOrder';
Table1.SetRange(['New York'],['New York']);
```

sets the *IndexName* property to *CityIndex*, and then sets a range to display only those clients whose records contain New York in the City field.

To set a range based on a multi-field index, include more than one set of starting and ending values in the array parameters. For example, if you have a table named Invoices and it uses an index based on the fields CustNo and InvoiceDate, the following statement will display all records for customer C1573 for the dates 12/1/95 through 2/1/96:

```
Table1.SetRange(['C1573','12/1/95'],['C1573','2/1/96']);
```

Using ApplyRange

There is an alternative, albeit a more involved one, to using *SetRange*. You can use the *SetRangeStart*, *SetRangeEnd*, and *ApplyRange* methods to select a range. While these statements also require an index (either primary or secondary), it permits fields to be explicitly assigned their starting and ending range values without using an array. The following example defines the same range as that demonstrated in the preceding listing:

```
Table1.SetRangeStart;
Table1.FieldByName('CustNo').AsString      := 'C1573';
Table1.FieldByName('InvoiceDate').AsString := '12/1/95';
Table1.SetRangeEnd;
Table1.FieldByName('CustNo').AsString      := 'C1573';
Table1.FieldByName('InvoiceDate').AsString := '2/1/96';
Table1.ApplyRange;
```

Removing a range is much easier than applying one. To remove a range, use the *CancelRange* method. Here is its syntax:

***TableName*.CancelRange;**

Range Example

Use the following steps to create a project that demonstrates setting a range for a table.

Create a new project and on the new form, place two Labels, two Edits, a DataSource, Table, DBGrid, and Button. Your form should now resemble Figure 7.

Set the *Caption* property of *Label1* to `&Start Range:`, and its *FocusControl* property to *Edit1*. Next, set the *Caption* property of *Label2* to `&End Range:`, and its *FocusControl* property to *Edit2*.

Set the *Caption* property of *Form1* to `Range Demonstration`. Select *Edit1* and erase its *Text* property. Do the same for *Edit2*.

Select *DataSource1* and set its *DataSet* property to *Table1*. Select *Table1* and set its *DatabaseName* property to `DBDEMOS`, its *TableName* property to `CUSTOMER.DB`, and its *Active* property to *True*. Since the range in this example will be defined for the Company field, set *Table1*'s *IndexName* property to `ByCompany`.

Select the DBGrid and set its *DataSource* property to *DataSource1*. Finally, select *Button1* and set its *Caption* property to `&Apply Range`. You have now set all necessary properties.

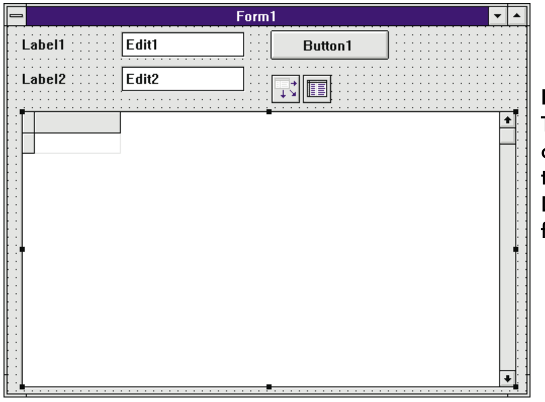


Figure 7: The basic components of the Range Demonstration form.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Button1.Caption = '&Apply Range' then
    begin
      Table1.SetRange([Edit1.Text],[Edit2.Text]);
      Button1.Caption := '&Drop Range';
    end
  else
    begin
      Table1.CancelRange;
      Table1.Refresh;
      Button1.Caption := '&Apply Range';
    end;
end;
```

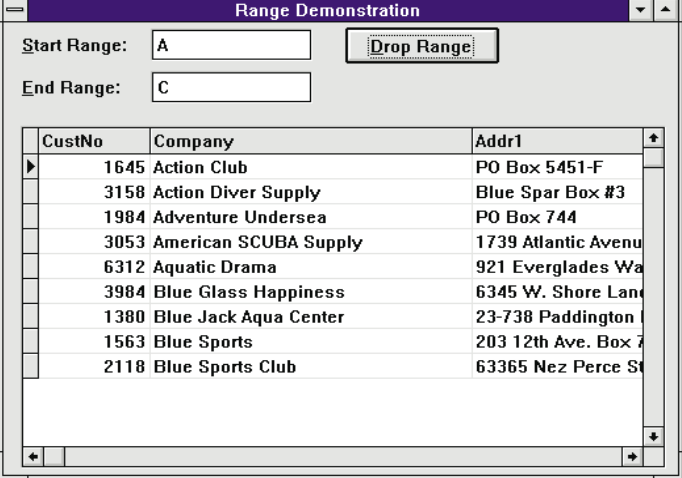


Figure 8 (Top): The *OnClick* event handler for *Button1*. **Figure 9 (Bottom):** A range limiting display of records to those companies whose names begin with A through C has been applied to the table.

Finally, define the event handler for *Button1*'s *OnClick* event property. To do this, double-click *Button1* to create an event handler. Now enter the code shown in Figure 8.

This completes the project. Press F9 to run it. Enter the letter A in the **Start Range** field, and the letter C in the **End Range** field. (Note that the range is case-sensitive only if a case-sensitive index is used. In this case, the index is not case-sensitive.) Click on **Apply Range** to apply the range. Now, only those companies whose names begin with A and B will be displayed, as shown in Figure 9. To remove the range, click on the **Drop Range** button.

Conclusion

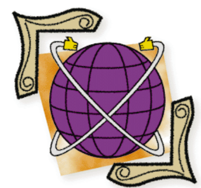
Using linked tables and ranges, you can control which records are displayed in a table. While using these features requires some planning — for example, the creation of secondary indexes — they provide your applications with a wide range of display options and useful features. ∆

The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\JAN\DI9601CJ.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, and is Contributing Editor of *Paradox Informant* and *Delphi Informant*. Cary is this year's Chairperson of the Paradox Advisory Board for the upcoming Borland Developers Conference. He has a Ph.D. in Human Factors Psychology, specializing in human-computer interaction. You can reach Jensen Data Systems at (713) 359-3311, or through CompuServe at 76307,1533.

ON THE NET

BY CAROL BOOSEMBARK



Borland Online

For the experienced Web walker, Borland Online is a welcome site. This Web page is easy to understand, and finding information is a breeze. If you're new to the World Wide Web, Borland Online is a great place to start learning about what can be found on the Internet.

This month we'll visit Borland Online and discuss a few of its features. To begin, point your browser to http://www.borland.com (see Figure 1). This first page gives you a variety of options including a text version of the Web page, Borland news and information, special programs and services, technical and product information, a listing of Borland Worldwide offices, a table of contents, and much more.

Figure 1: Borland Online at http://www.borland.com.

If you're looking for information about the 7th Annual Borland Developers Conference, visit http://www.borland.com-/ProgServ/events/bdc96/call4p.htm (see Figure 2). There you'll find the event will be held in southern California, between July 27 and August 14, 1996. (The site and specific dates will be available soon.) You can contact Christine Sherman, Chairman of the Conference Advisory Board, directly from this page. (You can also reach her at csherman@wpo.borland.com, or via CompuServe at 76067,507).

Additionally, this page has links to a listing of Advisory Board contacts, a conference deliverables schedule, author information, as well as information about presentations, conference tracks, and more.

For the application developer, Borland Online offers The Legal Toolbox. Hosted by Bob Kohn, Senior Vice President & General Counsel for Borland, this page features eight topics: Copyrights, Patents, Trademarks, Trade Secrets, Contracts, Insurance, Articles, and Lawsuits. There's a complete transcript of the US Court of Appeals decision in the Lotus v. Borland case located on the Lawsuits page. And the most entertaining page, by far, is at http://www.borland.com/-ProgServ/US/legal/jokes.html. It's a must see.

Turning to recent additions, Borland Online premiered Java World, a Web site for Java developers last November. Located at http://www.borland.com/Product/java/-java.html, Java World features a Java World Subscription (e-mail messages for keeping current on updates to Java World), a Java

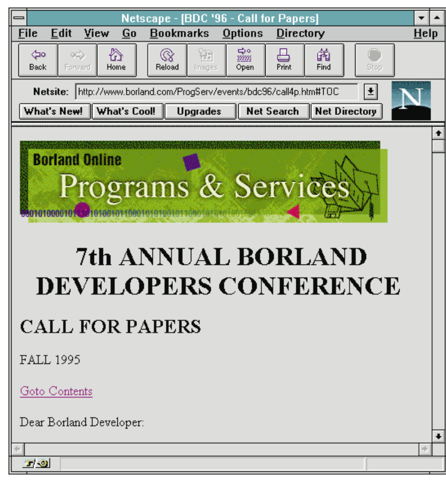


Figure 2: Here's your chance to get involved in the next Borland Conference.

Survey, related Java press releases, and Java on the Net, a listing of over 20 Java sites. (For related Borland information see "Borland to Deliver Tools for Java, Sun's Internet Programming Language" in Newsline, page 5.)

There are many other Borland-related Web sites. In the coming installments, we'll cover these, from Borland's latest ventures to the many "unofficial" Borland Web sites created by independent developers and vendors. If you have a great Web site to recommend, feel free to send me e-mail at 75702.1274@compuserve.com. ∆

Due to the flexible nature of the Internet, all Web sites mentioned may have changed. If a Web page address has changed, a new link is usually left at the old address to guide you.

Carol Boosembark is Products Editor for *Delphi Informant*.

Web Site of the Month

If you're looking for a specific software component, your best bet is to visit Imagicom Software Component Resource at http://www.xmission.com-/~imagicom/ (see Figure A). This Web site categorizes software components, making research quick and easy. Currently on Imagicom, you can search by component type (edits, listboxes, grids, image manipulation, etc.), file type (DLL, OCX, VBX, VCL, etc.), or vendor.

Once you've selected a search type, Imagicom produces a Web page listing the appropriate components (see Figure B). Each component name is listed with its company name, a link to its company information, a description that may include compatible platforms, and pricing information. If available, a demonstration version of the product, a custom e-mail page for the vendor, and a custom order page for the product are linked to the description. Additionally, some components have full page advertisement links for online viewing.

At press time, Imagicom Software Component Resource had a total of 42,497 visitors to its Web page. Its most popular index searches are conducted on:

- VCL Delphi (695 Total)
- Uncategorized Delphi Add-Ons (508 Total)
- OCX (733 Total)
- Communication and Telephony (930 Total)
- VBX (2975 Total)

In addition to its indexing abilities, Imagicom provides a comprehensive listing of product vendors and direct e-mail links. To view the list, select the "Vendor List" hypertext link from the main menu, or point your browser to http://www.xmission.com/~imagicom/index/vendors.html.

Imagicom also produces monthly announcements about its newest features and products. To get on the mailing list, just select the "Mailing List" button from their home page. For more information about Imagicom you can e-mail the company at imagicom@xmission.com.

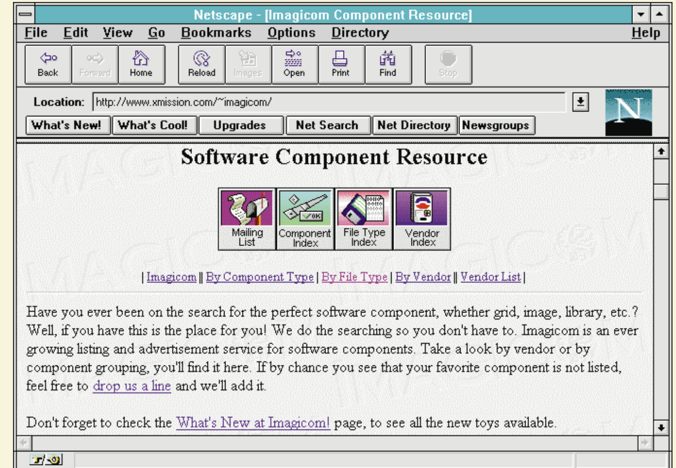


Figure A: The Imagicom Software Component Resource Web page.

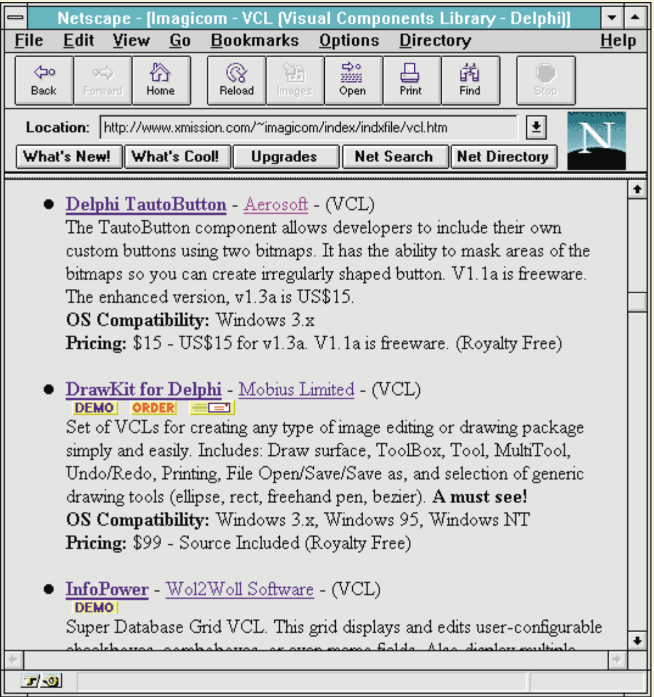


Figure B: A listing of VCL controls.

AT YOUR FINGERTIPS

BY DAVID RIPPY

DELPHI / OBJECT PASCAL

You'll learn more about a road by traveling it than by consulting all the maps in the world.
— Unknown

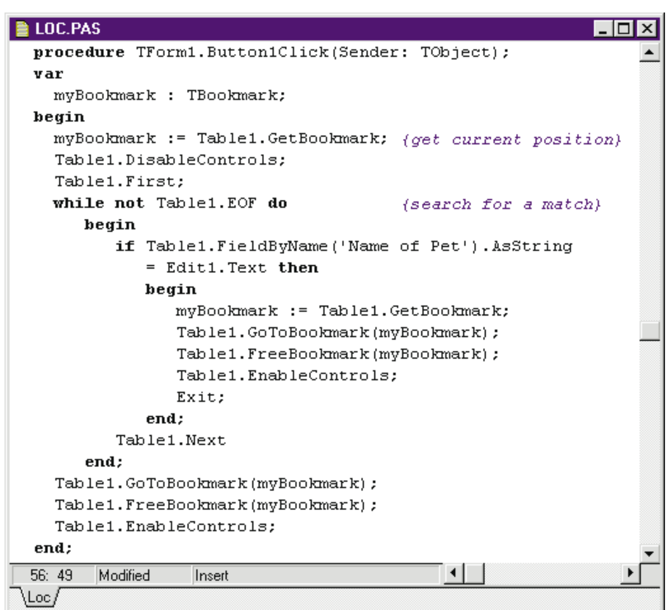
How can I perform a locate on a non-indexed field?

Searching a table for a specific value in an indexed field is easy with Delphi — just use the *FindKey* or *FindNearest* methods. However, there isn't a method for finding a value in a non-indexed field. To do so, you'll need to write a little code.

Examine the form in Figure 1. The Edit component (*Edit1*) allows the user to enter a value to search for in the non-indexed column, **Name of Pet**, in *Table1*. When the user presses the **Search** button, the form executes the code in Figure 2.

Last Name	First Name	Favorite Color	Name of Pet
Bell	Eric	Yellow	Jackson
Breaux	Bruce	Aqua	Killer
Dawsey	Noel	Blue	Midgey
Goodman	Tony	White	Greedy
Jones	Jason	Red	Puggly
Rippy	Chris	Purple	Chewie

Figure 1: This form performs a search on the non-indexed field, **Name of Pet**.



LOC.PAS

```
procedure TForm1.Button1Click(Sender: TObject);
var
  myBookmark : TBookmark;
begin
  myBookmark := Table1.GetBookmark; {get current position}
  Table1.DisableControls;
  Table1.First;
  while not Table1.EOF do             {search for a match}
    begin
      if Table1.FieldByName('Name of Pet').AsString
        = Edit1.Text then
        begin
          myBookmark := Table1.GetBookmark;
          Table1.GoToBookmark(myBookmark);
          Table1.FreeBookmark(myBookmark);
          Table1.EnableControls;
          Exit;
        end;
      Table1.Next
    end;
  Table1.GoToBookmark(myBookmark);
  Table1.FreeBookmark(myBookmark);
  Table1.EnableControls;
end;
```

56: 49 Modified Insert

Loc

Figure 2: This code is attached to the *OnClick* method of the *Button1* component.

First, the *GetBookmark* method sets a bookmark on the record the user is positioned on. Next, the table is searched sequentially starting from the first record. If a match is found, the bookmark is set on the first record that matches the search criteria, and the table cursor is moved to that record with the *GotoBookmark* method. If a match is not found, the cursor returns to the record that the user was originally positioned on.

Notice the use of the *DisableControls* method. This accelerates our search process by disconnecting the DataSet from the DataSource component. If we had not called this method, the DBGrid would be updated every time the *Next* method was called. For fun, comment out the call to *DisableControls* and see what happens. — *D.R.*

How can I determine the record number in a Paradox table?

This is a common request with a less-than-obvious solution. Delphi does not inherently provide a method for determining the record number, but the Borland Database Engine (BDE) can help.

The form in Figure 3 displays a DBGrid component and a Label component. The Label (*Label1*) displays the record number of the current record in the DBGrid. As the user nav-

igates the records in the DBGrid, the record number updates accordingly.

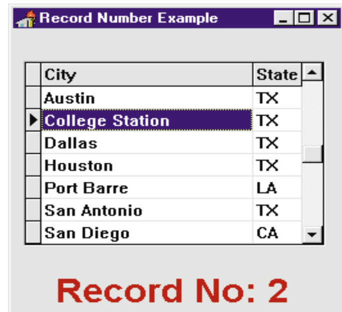
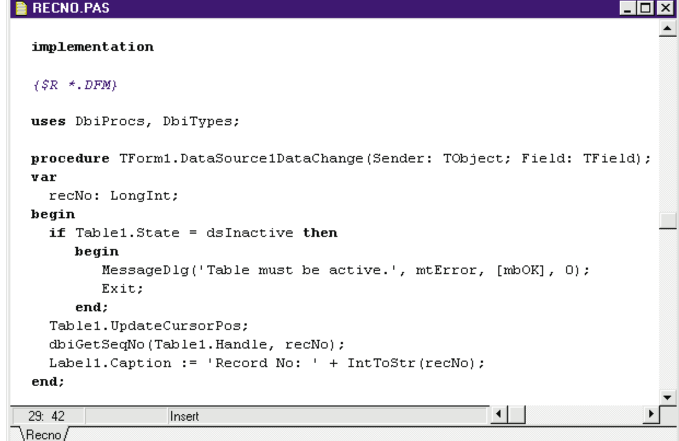


Figure 3: The **Record No** Label component is updated as the user navigates the DBGrid.

Examine the code in Figure 4. The first thing you'll notice is that two units, `DbiProcs` and `DbiTypes`, have been added in a **uses** clause. These units provide the program with the information necessary to make calls to the BDE application programming interface. You must include these header files in your **uses** clause for this example to work.

The code in Figure 4 is attached to the DataSource component's *OnDataChange* event handler. It's placed here so the record number label is updated when the user navigates through the DBGrid. First, the code checks to ensure *Table1* is active. If it's inactive, no attempt is made to update *Label1*.

Next, the *UpdateCursor* method is called to ensure the Table is synchronized with the BDE cursor. The BDE *dbiGetSeqNo* method is then called to retrieve the record number of the current record in the Paradox table. Finally, *Label1* is updated with the new record number.


RECNO.PAS

```
implementation

{$R *.DFM}

uses DbiProcs, DbiTypes;

procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
var
  recNo: LongInt;
begin
  if Table1.State = dsInactive then
    begin
      MessageDlg('Table must be active.', mtError, [mbOK], 0);
      Exit;
    end;
  Table1.UpdateCursorPos;
  dbiGetSeqNo(Table1.Handle, recNo);
  Label1.Caption := 'Record No: ' + IntToStr(recNo);
end;
```

29: 42 Insert

Recno

Figure 4: Notice the **uses** clause that's been added to include `DbiProcs` and `DbiTypes`.

If you need more information on the available BDE calls, Borland offers the *Borland Database Engine User's Guide* as a complete reference of the BDE. To order call Borland Customer Service at (510) 354-3828.
— *D.R.*

How can I create a two-line message using the MessageDlg function?

If you've ever used the *MessageDlg* function to display a long message, you probably ended up with a dialog box similar to the one

Figure 5: This *MessageDlg*-generated dialog box is too wide.

in Figure 5. The dialog box would be more attractive if the message, "There is nothing uglier than a really long dialog box," were split into two lines. [For more information about *MessageDlg*, see Kevin Bluck's article "Building a Better MessageDlg" on page 24.]

To divide a message into two lines, insert a carriage return and line feed into the string where you want to break it. For instance, to split the example message into two lines, use the following Object Pascal statement:

```
MessageDlg('There is nothing uglier' + #13#10 +
           'than a really long dialog box.',
           mtInformation, [mbOK], 0)
```

Figure 6: That's better!

Figure 6 shows the same dialog box with the message on two lines. This technique is also useful with other functions that generate a dialog box (e.g. *ShowMessage*, *InputBox*, and *InputQuery*) when you have a large amount of text to display, but don't want a wide dialog box. — *Russ Acker, Ensemble Corporation*

How can I quickly move to a property in the Object Inspector?

While positioned in the Object Inspector, press Tab. This moves the cursor from the Value column (right side) to the Property column (left side). You can then type the first letter(s) of the property you want, and the cursor will jump to that property in the list. — *David Faulkner, Silver Software Inc.* ∆

The sample projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\JAN\DI9601DR.

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

NEW & USED

BY CARY JENSEN, PH.D.

Orpheus

A Grab Bag of Powerful Custom Components Complete with Source Code

Orpheus, from TurboPower Software Company, is a large collection of Delphi components that qualifies as one of the best bargains in the Delphi add-on market.

Among the components included in Orpheus are: a text editor with 16MB capacity, data-aware edit fields with strong data-integrity support, a timer that can control multiple timer events while using only one Windows timer resource, a set of flexible spinner controls, a Windows 95-like tabbed notebook, and much more.

Overall, the components available in Orpheus are well designed from an object-oriented standpoint. Most of Orpheus' components descend from the *TOvcBase* class, which is a direct descendant of Delphi's *TCustomComponent*. Through *TOvcBase*, Orpheus components inherit a property that enables them to use the *TOvcController* component. (Later, we'll discuss *TOvcController* in more depth.)

As mentioned, Orpheus consists of a remarkably diverse collection of components. So many, in fact, that it's impossible to adequately describe all of them in this review. In this limited space, I hope to provide you with a feel for what types of components you'll find in this product.

Inside Orpheus

The components of Orpheus can be roughly divided into six categories: edit fields, array editors, text editors, file viewers, a sophisticated grid control, and a whole collection of useful components that I will simply call "gadgets."

The edit fields consists of both simple edit fields as well as data-aware controls. The data-aware controls use the Delphi DataSource component to work with table data, and therefore, are similar to Delphi's data-aware controls.

However, all these controls make available validation features that you will find invaluable. For example, there are edit controls for general text, as well as for numbers and pictures.

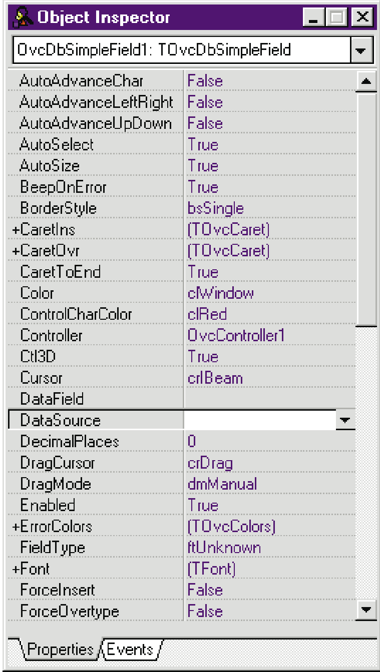


Figure 1: Several of the *TOvcDbSimpleField* component's rich set of properties, as seen through the Delphi Object Inspector.

Furthermore, the text and number fields support complex picture properties that allow you to control the format and content of a user's input. For example, the Object Inspector in Figure 1 displays the properties for the data-aware *TOvcDbSimpleField* component. Clearly, you can see the richness of this component.

The array editors let users edit data stored in arrays, linked-lists, or tables using a listbox-like control. Because the array editors are not actually Windows listboxes, they are not

bound by Windows' 32K limit. This added capacity comes at a price, however, because you must write the code to paint each cell of the array editor. Fortunately, this is not too difficult since Orpheus provides an easy-to-use *OnGetItem* event property for the purpose.

Orpheus supplies three text editor components: *TOvcTextEditor*, *TOvcTextFileEditor*, and *TOvcDBTextEditor*. They allow users to edit files up to 16MB in length, and provide common editor features such as word wrap, bookmarks, and search-and-replace. The basic text editor, *TOvcTextEditor*, can allow users to edit data, while *TOvcTextFileEditor* adds I/O capability, including reading directly from, and writing to, text files. *TOvcDBTextEditor* is a data-aware version of the text editor, and enables users to edit the contents of a text field from a specified DataSource component.

The file view components enable you to add file browsing capability to your applications. The two basic viewer components that you'll probably use include *TOvcTextFileViewer,* which is used to view ASCII files, and *TOvcFileViewer*, which can be used for viewing either text or binary files. Figure 2 shows an example project that ships with Orpheus that uses the *TOvcFileViewer* component.

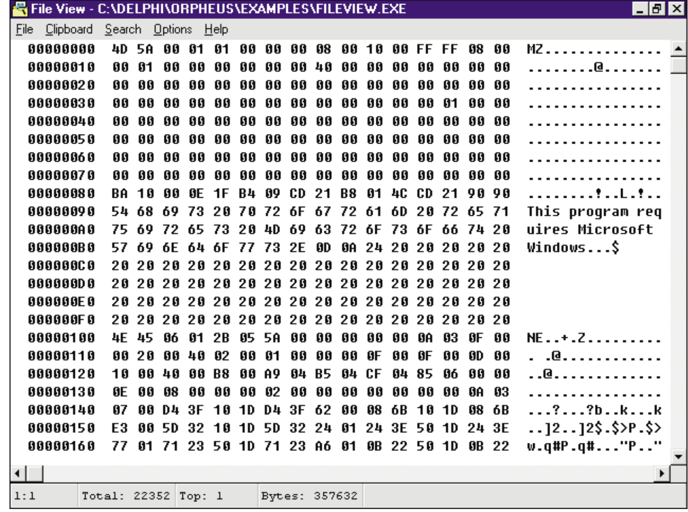
The Orpheus Table component, *TOvcTable*, is a sophisticated grid control that can display data from tables. While this component provides far more features than Delphi's DBGrid, it's also substantially more difficult to use. On the plus side, *TOvcTable* allows you to create a grid control that displays table data using simple fields, checkboxes, combo-boxes, graphics, and memos. Consequently, you have a great deal more flexibility in how you can display table data than using DBGrid. On the downside, *TOvcTable* does not have a *DataSource* property. Instead, you must write event handlers to read and write the data from a *TDataSet* component.

The final group that I (probably unfairly) call gadgets, is a varied collection of components that you can implement in many of your applications. Among these are a tabbed notebook that permits you to display tabs either vertically or horizontally, a set of spinner controls, a calendar component, a progress meter, a rotated text component, a data transfer component (for moving data easily between forms), and a timer pool (a component that manages multiple timer events while using only one Windows timer). Most of these components are sophisticated, and I regret not being able to give them adequate coverage here.

The Controller

Most of Orpheus' components make use of *TOvcController*. It serves as a central processing component for key strokes and exceptions. The first time you place a controller-enabled component on a form, a *TOvcController* component is created.

While the controller provides the Orpheus components with access to exception handling facilities, it provides you, as the developer, with the ability to define keystroke-to-command mapping for all the Orpheus components. This can be done at design time through the Command Table Editor, shown in Figure 3.



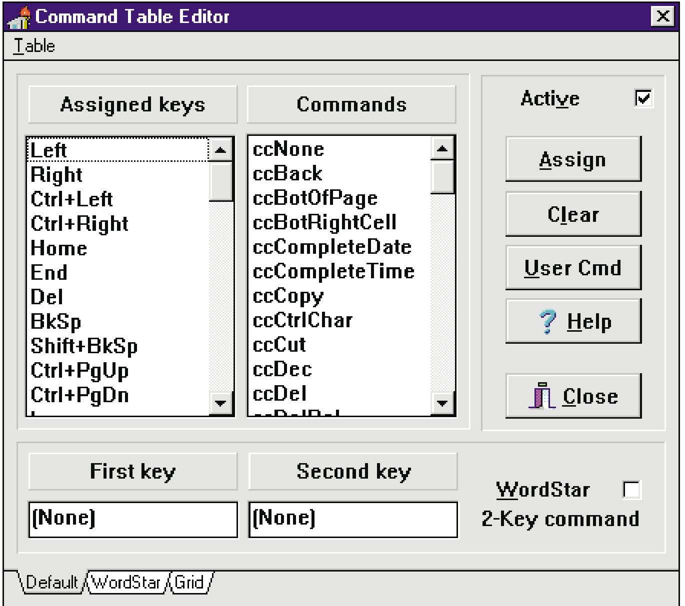


Figure 2 (Top): The example project, FILEVIEW.EXE, uses the *TOvcFileViewer* component to display the contents of a file in hex.
Figure 3 (Bottom): The Orpheus Command Table Editor.

This dialog box is the property editor for the *EntryCommand* property of the *TOvcController* component. With the Command Table Editor, you can map up to two key strokes or key-stroke combinations to a given command. Additional flexibility is provided by permitting you to load alternative keystroke-to-command tables for use by the controller at run time. This allows you to easily customize the user interface to meet your user's needs.

Documentation

The Orpheus documentation is 600 pages, and covers both installation and component use. Each component is described in detail, including an object tree that shows the ancestry of each component, property, method, and event lists, as well as a detailed description of each developer method.

The writing is clear and understandable, and the examples are well thought out and complete. It's without a doubt, the best documentation that I've seen for a third-party product.

Orpheus also ships with a collection of example projects that demonstrate how to use the various components. These examples are especially useful in helping you understand some of the more complicated components, such as the *TOvcTable* component and array editors.

Installation

Orpheus includes an installation program that loads the necessary units and support files, including source code, help files, and example projects.

To make the help available from within the IDE, you must manually copy these help files to the appropriate directory, and run the Delphi HelpInst utility. This is not complicated, however, and the process is described in the Orpheus documentation.

Installing all the Orpheus components, help, and example files requires about 10MB of disk space.

Conclusion

Orpheus is a large collection of powerful Delphi components that includes something for every developer. The objects themselves are well designed, and backed by outstanding documentation. The heavily documented source code that is included serves as an excellent example of how to build Delphi components. In short, even if you have a use for only one or two of Orpheus' many components, you will find this product well worth the money. ∆

Cary Jensen is Contributing Editor to *Delphi Informant.*

Orpheus is a large collection of mainly data-related components. It comes with source code and a royalty-free license to distribute applications based on these components. Considering the 60-day money-back guarantee and the reasonable price, it would be hard to go wrong with Orpheus.

TurboPower Software Company
P.O. Box 49009
Colorado Springs, CO 80949
Voice: (719) 260-9136 or
BBS: (719) 260-9726
E-Mail: CIS: 76004,2611
Web Site: http://www.tpower.com/
US or Canadian orders: (800) 333-4160
Price: US$199

BY GARY ENTSMINGER

RoboHELP 95

A Better Way to Create Help Systems

RoboHELP 95 is a tool for creating Windows-compatible Help systems. From RoboHELP's point of view, help can be the Help system you call from an application's main menu, or it can be a stand-alone look-up system that allows navigation through virtually any kind of information — catalogs, classification systems, histories, employee handbooks, training manuals, tutorials, etc.

Used with Microsoft Word for Windows, RoboHELP develops help projects that are compiled with the Windows Help compiler. RoboHELP is the bridge between a Word help document and the Windows Help compiler. If you've ever tried to move from a Word document to the Help compiler, you'll appreciate the RoboHELP bridge.

RoboHELP creates and organizes the help project. The Windows Help compiler expects the Word document to contain codes indicating how your Help system operates. These codes contain help topics, graphics, hotspots, jumps, macros, and other links. Creating those codes from scratch is a lot of work, so RoboHELP assists you with a visual tool palette.

In a nutshell, RoboHELP does the dirty, low-level work of presenting your text as a Windows Help system to the Windows Help compiler. It handles the details, allowing for more time to focus on creating, formatting, and fine tuning the information in your Help system. After all, the information is what counts, right?

The Tool Palette

Creating information is mostly editing, importing, and formatting text and graphics, and you can manage this best with a word processor. To make Word and RoboHELP accessible from one screen, RoboHELP's tool palette floats on top of Word (see Figure 1).

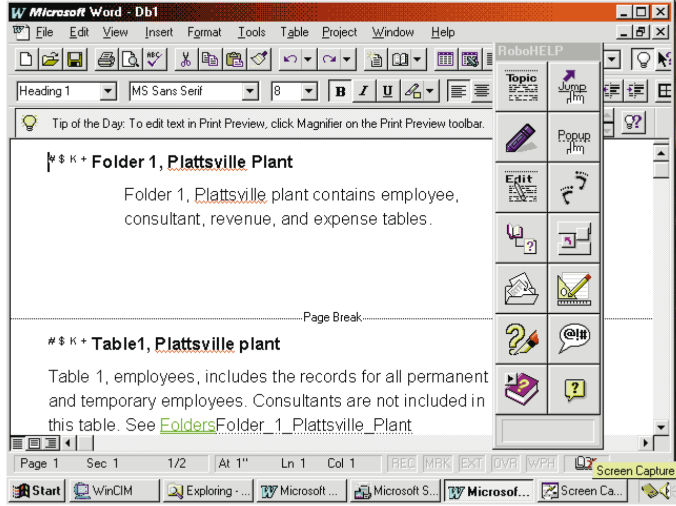


Figure 1: RoboHELP's tool palette floats on top of Word.

In the RoboHELP system, the two tasks of creating help information and a help project are well integrated. If your screen is big enough, you could even have a Delphi project displayed alongside the Word help document you're creating for that project, as well as the RoboHELP visual palette.

As you can see in Figure 1, RoboHELP is relatively unobtrusive. In addition, it modifies the Word menu during a RoboHELP/Word session, allowing you to select RoboHELP commands from Word menus (see Figure 2). This is a clear, intuitive approach.

Using a Help System

Consider how you typically use a Help system. Let's say you're working in Delphi and select **Help | Contents** from the menu. Although you have requested an action from Delphi, it passes the request to the Windows help viewer, which, in turn displays the Delphi Help system contents window. This window, like the others in the Delphi Help

system, is displayed by the Windows help viewer. Thus they have an appearance consistent with other Windows Help systems.

Consistent interfaces enable users to learn one system and apply that knowledge to others. Because Help systems created with RoboHELP access the Windows help viewer, they maintain this consistency (see Figure 3).

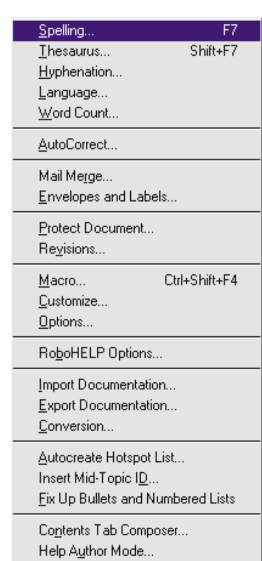


Figure 2: RoboHELP's menu options are integrated with Word's.

Without RoboHELP, creating a Help system is a fairly dark business, and I'll spare you the details. Instead I'll show you how to develop a Help system using RoboHELP and Word. I'm using Windows 95, Word 7, and RoboHELP 95, but you can also use Windows 3.x, RoboHELP 3, and Word 6 to produce similar results. The Windows 95 help interface has a new look and more features, but generally acts like a Windows 3.x Help system. Note that RoboHELP 95 is required to create Help systems for Windows 95 and Windows NT, and RoboHELP 3 is used to develop within Windows 3.x. You cannot run a help file compiled by the Windows 95 Help compiler in Windows 3.x.

Building a Help Project

There's really not a lot of work to it. To create a Help system you must:

- define a set of help topics (not necessarily all at once)
- create or import the actual text for the topics
- create hotspots and jumps to allow the user to navigate the Help system
- optionally, import graphics and other personalizing stamps into the Help system

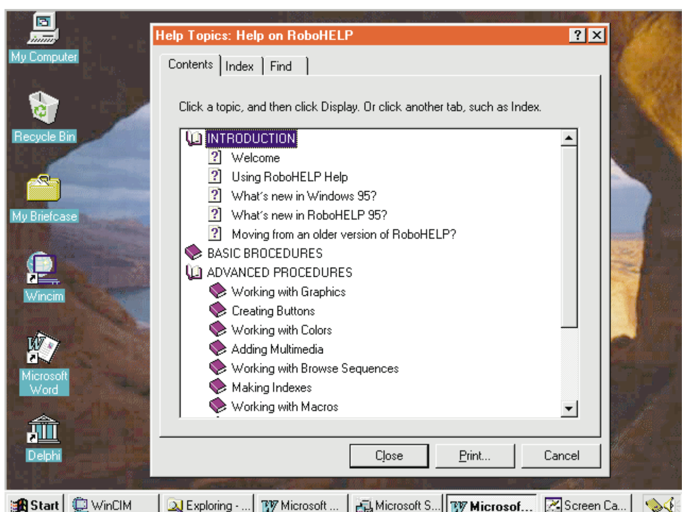


Figure 3: The Contents page of the RoboHELP Help system.

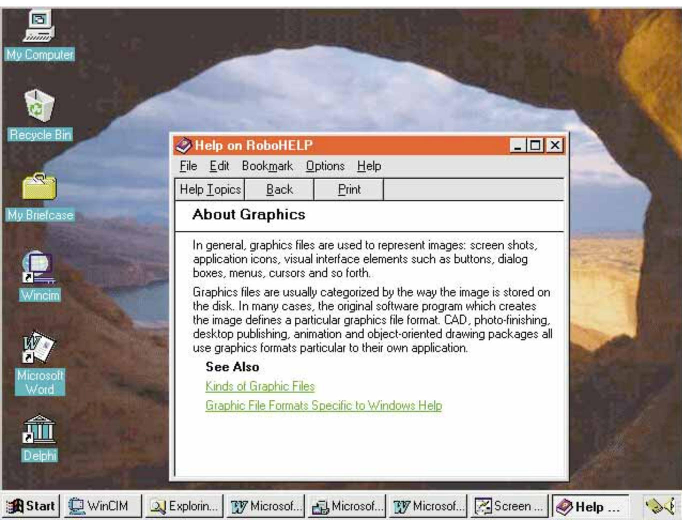


Figure 4: Windows Help system hotspots are green and underlined.

In addition, you need some knowledge of *hotspots* and *jumps*. Hotspots are the green (by default) underlined text in help windows (see Figure 4). Jumps are destination topics, indicating where the system should go when a user selects a hotspot. Each jump points to a help topic to display, making the Help system interactive.

A Few Steps

Here we go: RoboHELP Help system 1, Take 1. You've already installed RoboHELP (that was a snap). Now start it up and begin creating a RoboHELP project.

The first thing that RoboHELP does is fire up a copy of Word. Even if Word is open, RoboHELP loads a new copy. This allows the RoboHELP/Word interaction to be independent of whatever else you might be doing in Word.

RoboHELP then displays its main menu and the Create New Help Project dialog box (see Figure 5). Notice the **Title** and **File Name** edit boxes, and the default information placed in these boxes by RoboHELP. Simply add a title (such as "Delphi sample Help system") and rename the project. For our example, name the project db1.hpj.

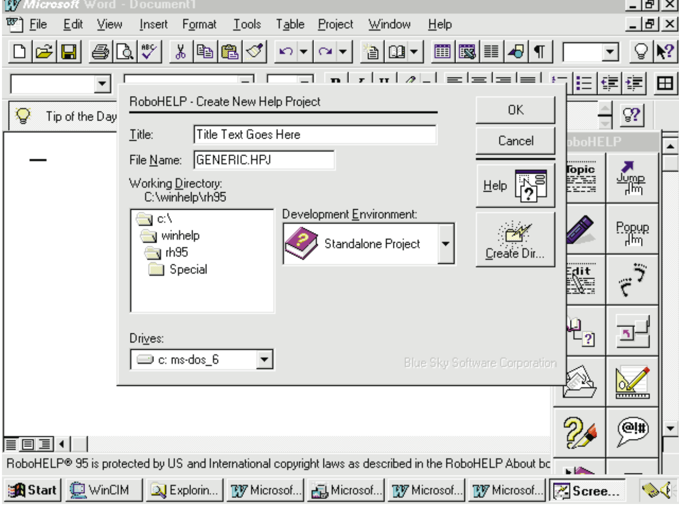
RoboHELP creates the project files and displays a message in the Word document (see Figure 6).

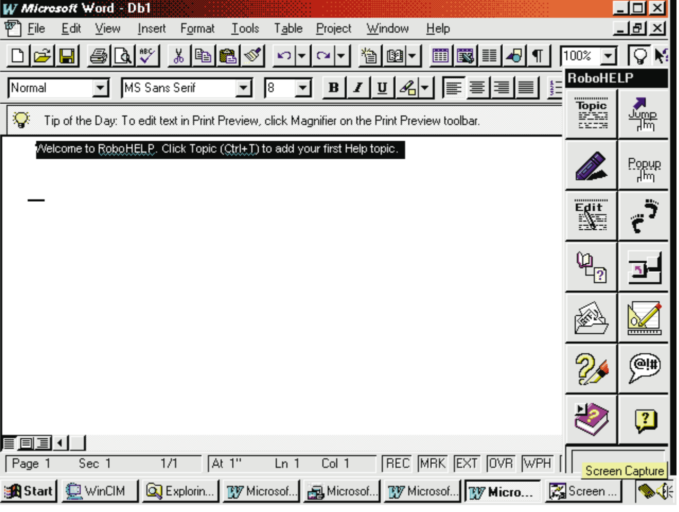
This is the pattern. Each time you select items from the visual palette, RoboHELP assists by making suggestions and displaying prompts.

Creating Topics and Jumps

Next, create a topic by following RoboHELP's suggestion to click the Topic tool. The New Topic dialog box opens. Add the topic, "Folder 1, Plattsville Plant," and press **OK**.

RoboHELP creates the codes for the new topic and indicates where the text for the topic should go (see Figure 7).





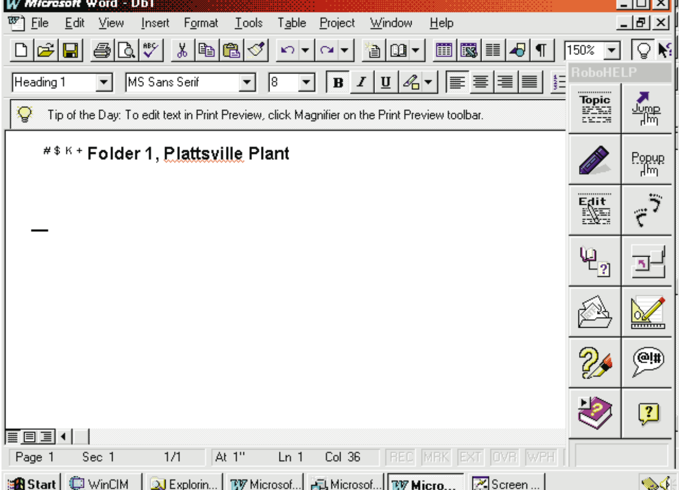


Figure 5 (Top): The RoboHELP Create New Help Project dialog box. **Figure 6 (Middle):** Getting started. RoboHELP creates the project files and displays a message in the Word document. **Figure 7 (Bottom):** RoboHELP creates the codes for a new help topic.

Add some help text, "Folder 1, Plattsville plant contains employee, consultant, revenue, and expense tables."

Now create a second topic, "Table1, Plattsville plant," and press **OK**. RoboHELP again creates the codes for the new topic and indicates where the topic's text should go. It creates the new topic on a new page. This means that the new topic won't be shown at the same time as the first topic, which is on the first page.

Add more help text for the Table1, employees topic, "Table 1, employees, includes the records for all permanent and temporary employees. Consultants are not included in this table."

To create a jump, select the Jump tool and use the Jump dialog box to create the hotspot text and the help topic to display when the user clicks on the hotspot text. RoboHELP creates the codes for the new jump (see Figure 8).

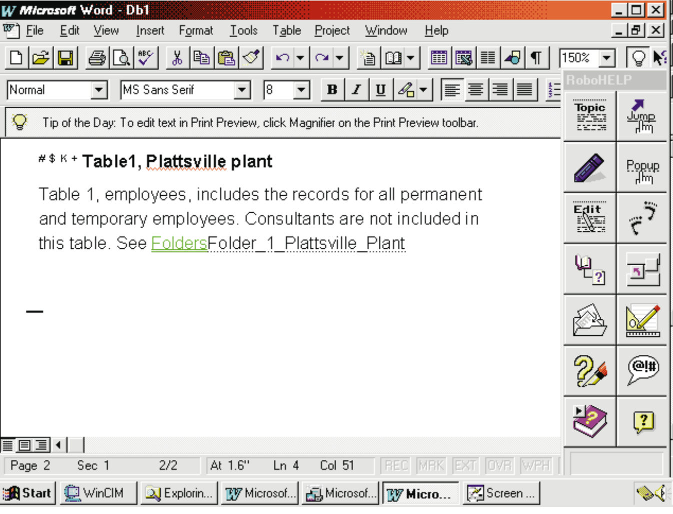


Figure 8: RoboHELP creates the codes for a new jump.

It's as simple as that. Use similar tools and dialog boxes to add graphics. As your system develops, you can fine tune and modify your work within Word and/or RoboHELP.

When you want to compile your help creations, you *make* the project by selecting the Make tool from the visual palette. This is as straightforward as compiling an Object Pascal, Paradox for Windows, or Visual Basic project (see Figure 9).

Conclusion

The RoboHELP and Word combination for creating Help systems is a good one. You enjoy the ease of accomplishing word processing tasks with a sophisticated word processor, and RoboHELP handles the project and conversion details.

Assuming you are familiar with Word, you already know most of the requirements to create a Help system with RoboHELP. You use Word to add and manipulate graphics, and to import information from other sources.

RoboHELP allows you to access the Windows Help engine and customize the appearance of help windows. You can add glossary buttons and even context-sensitive help by using the custom control Hypertext Help button included with

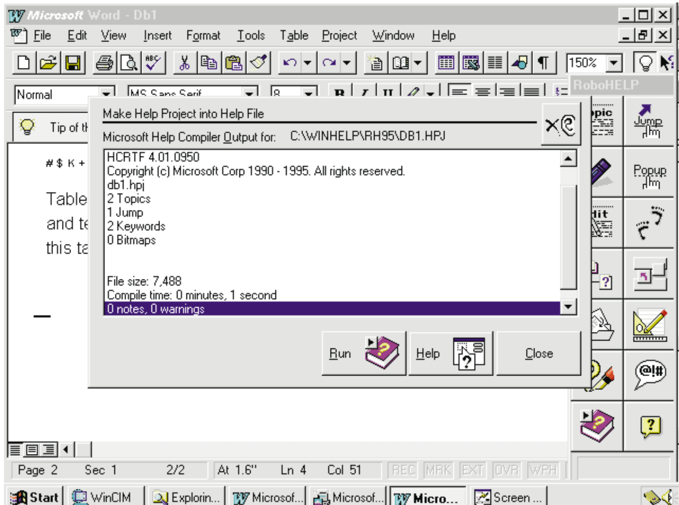


Figure 9: Making a help project.

RoboHELP. You can use RoboHELP to import existing help source projects, graphics, text, and .RTF files. In addition, RoboHELP 95 supports new Windows 95 help features, such as the Contents Tab. Two RoboHELP tools — Screen Capture and Contents Tab Composer — are especially useful.

The RoboHELP manual is informative and readable, and I breezed through several of the examples. When at one point I did need technical support in a hurry, I received it.

Finally, although you can create context-sensitive Help systems with RoboHELP, that's another ball of wax, requiring access to Borland Resource Workshop, the Microsoft SDK Dialog Box Editor, or the equivalent. Although creating a basic Help system is relatively easy, tackling a context-sensitive Help system is for the lion-hearted.

Other RoboHELP support tools are available from Blue Sky, the makers of RoboHELP. These tools can help you create more sophisticated Help systems that include OLE and video connections. These, along with help debugging tools, are packaged as RoboHELP Office 95. ∆

INFORMANT FACT FILE

RoboHELP 95 is a tool for building Windows-compatible Help systems. It can be a stand-alone look-up system, or it can call help from an application's main menu. Developers can also customize the help windows, and add glossary buttons and context-sensitive help.

Blue Sky Software Corp.
7777 Fay Avenue, Suite 201
La Jolla, CA 92037
Phone: (619) 551-2485 or (800) 459-2356
Fax: (619) 551-2486
CompuServe: 73473,3636
Web Site: http: //www.blue-sky.com
Price: RoboHELP 95 and RoboHELP 3.0, US$499. Upgrades from RoboHELP 3.0, US$149. RoboHelp Office is US$599.

Gary Entsminger's new book, *The Way of Delphi*, an intermediate and advanced guide to object-oriented Delphi development, is forthcoming from Prentice-Hall. He is currently working on a new book and trying to make the most of 32-bit systems. He can be reached on CompuServe at 71141,3006.

TextStream

Delphi How-To Solves Programmers' Problems

If you've already plowed your way through one or more of the traditional Delphi books, you may be ready for *Borland Delphi How-To: The Definitive Delphi Problem-Solver*, by Gary Frerking, Nathan Wallace, and Wayne Niddery. Like other books in the Waite Group Press' "How-To" series, this is not a tutorial, but a collection of techniques for solving specific programming problems.

Each section poses a question, beginning with the words "How Do I ..." and followed by a topic such as "Create a system modal form," "Scroll portions of a dialog," or "Run another application from my application." The text of each section begins with a brief problem statement, followed by a short description of the technique to be used. A longer section labeled "Steps" then walks you through the creation of a complete program that illustrates the technique.

Each step in the sequence is an atomic action, such as setting a group of properties or keying a (usually short) procedure. I particularly like the presentation of code fragments within the steps. After a sentence indicating what you're about to do, the code appears in a distinguishing font. Lines that Delphi generates automatically appear against a normal white background, while lines that must be keyed appear against a light gray screen.

The Steps section is followed by a discussion of "How It Works," which provides commentary on the code. In some cases, other subheadings are added to address key points raised by the example. Concluding remarks appear under the heading "Comments."

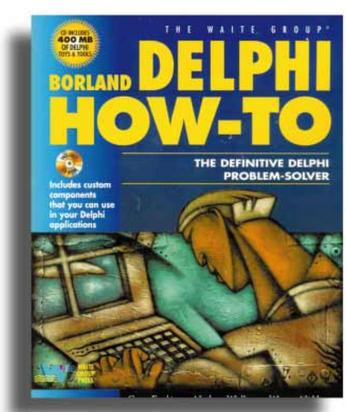
Topics vary from the fairly trivial to those that are lengthy and complex. An indicator at the start of each topic classifies the material that follows as easy (e.g. how to center a form within its parent), intermediate (e.g. how to use drag-and-drop in applications), or advanced (e.g. how to create a user-customizable toolbar). While the indicators help, the material seems directed toward too broad an audience.

Programmers who need the easy topics may not follow the advanced topics, and those who use the advanced material certainly won't need the easy topics.

Some topics seem to be carry-overs from the Waite Group's earlier books on Visual Basic. While the ability to compare techniques is interesting, it isn't clear that Delphi programmers necessarily have the same concerns as their VB counterparts. A few other topics seem too long for this format. The detailed examples are helpful, but it's easy to lose the essence of the technique when the example spans 20 or more pages dense with code.

Topics are grouped into chapters according to subject matter: Forms, Graphics, Multimedia, Database, etc. Within each chapter, topics are ordered in an apparent attempt to create a logical presentation sequence, rather than by complexity.

The Waite Group claims that *Delphi How-To* can be read sequentially. Although the topic ordering makes that approach possible, most readers are likely to jump around among topics of immediate interest.



The accompanying CD-ROM provides the code for each of the 113 examples in the book. Since the code is uncompressed and contains an executable file, it is possible to run the examples directly from the CD. This lets you test the program's effect without either keying the steps or loading the source code.

If you want to retrieve the source code for one or more projects (or even all of them), an installation program provides a high degree of control over which projects get installed and where.

The CD also contains three bonus programs not described in the book; demonstration versions of three sets of commercial component (CIUPKS's User Friendly VCL,

"*Delphi How-To*"
(continued on page 48)

There Really Are *Secrets of Consulting*

Gerald M. Weinberg's *The Secrets of Consulting* is subtitled *A Guide to Giving & Getting Advice Successfully*. If the definition of consulting truly is "the art of influencing people at their request" (as provided in his preface), Mr Weinberg succeeds at giving and getting advice. After all, he wrote his book and I read it — and nobody forced me.

You may not care for his opinion that "some software consultants ... are retained strictly as supplementary programming labor ... [for] grunt work ... turning out computer code," and that "The last thing their 'clients' want is to be influenced." If this description sounds a little too familiar, then you might also be offended when you read that Weinberg considers calling such "temporary workers" consultants is merely a ploy for paying them a few dollars less.

Published by Dorset House, *Secrets* will not help you write or debug code. Nor will it help you design a database. If you can get past this, however, there is much to be learned from Mr Weinberg. While there are several aspects to his book I don't particularly like (and I will share them with you), I encourage everyone to read this book. As the subtitle promises, it is a helpful guide to both giving and getting advice.

Here are my criticisms. I don't like the way the book is arranged. He mixes up headings, sub-headings, "the secrets," and a variety of adages, and delivers them unpredictably. I wish he would be more consistent. The illustrations that help introduce each chapter also annoy me — they're a little too Ralph-Steadmanesque.

I also don't like all the laws, rules, and principles; there are over 100 of them! Couldn't he make consulting a little less involved? And while a clever handle helps my memory (e.g. The Law of the Jiggle: Less is more), this soon becomes trite and tiresome. I just wish he'd quit trying to be so cute. In spite of this, I still enjoyed one aspect of Weinberg's writing style: while some consultants take themselves too seriously, he does not.

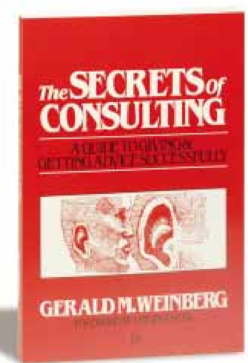
This leads me to what makes *Secrets* worthwhile: the content. Weinberg clearly understands the client/consultant relationship. Many times I read his clear explanations of how to handle situations, while reflecting on similar circumstances that I unfortunately misunderstood.

This isn't just a book to be read once; it's a reference. I'll confidently refer to it when I'm troubled by a project or prospect that I can't convert into a client. I have already avoided several problems by following his advice.

For instance, are you familiar with *the orange juice test*? And, how do you price your services? There's a whole chapter on pricing. Although a taboo, Weinberg sorts through the issues and quickly explains ten underlying laws, including the "Second Law of Pricing: The higher the price, the more the client loves the consultant," and "The less they pay you, the less they respect you."

Trust is also examined in detail. Much of the chapter devoted to this topic seems obvious, but is nonetheless important. He includes such pearls as "Never promise anything," yet "Always keep your promise." And when it comes to contracts, there are three important rules: "First, get it in writing. Second, get it in writing. Third, get it in writing."

The Secrets of Consulting contains 14 chapters, a preface, a forward, an index, a listing of the laws, rules, and principles, and an excellent section of



"ideas that might get you learning even more secrets of consulting." These include suggested books to read, workshops to attend, and people to learn from.

I recommend this book.

— *Jeff Sims*

The Secrets of Consulting by Gerald M. Weinberg, Dorset House Publishing, 353 West 12th Street, New York, NY 10014, (212) 620-4053.

ISBN:0-932633-01-3
Price: US$32.50
248 pages

"Delphi How-To" (cont.)

TurboPower's Orpheus, and Async's Professional Toolkit); and three shareware/freeware components.

Borland Delphi How-To is not a tutorial, although you'll probably pick up a few interesting tidbits from its well presented examples. The approach is pragmatic, and the presentation style is strictly business without being excessively dry. You probably will never use everything in the book, but odds are that you'll find a few examples that are just what you need, either now or on some future project.

And just one ready-made solution to a nagging problem will easily justify the book's cost.

— *Larry Clark*

Borland Delphi How-To: The Definitive Delphi Problem-Solver by Gary Frerking, Nathan Wallace, and Wayne Niddery, Waite Group Press, 200 Tamal Plaza, Corte Madera, CA 94925; phone: (415) 924-2575; fax: (415) 924-2576.

ISBN: 1-57169-019-0
Price: US$39.95
851 pages, CD-ROM