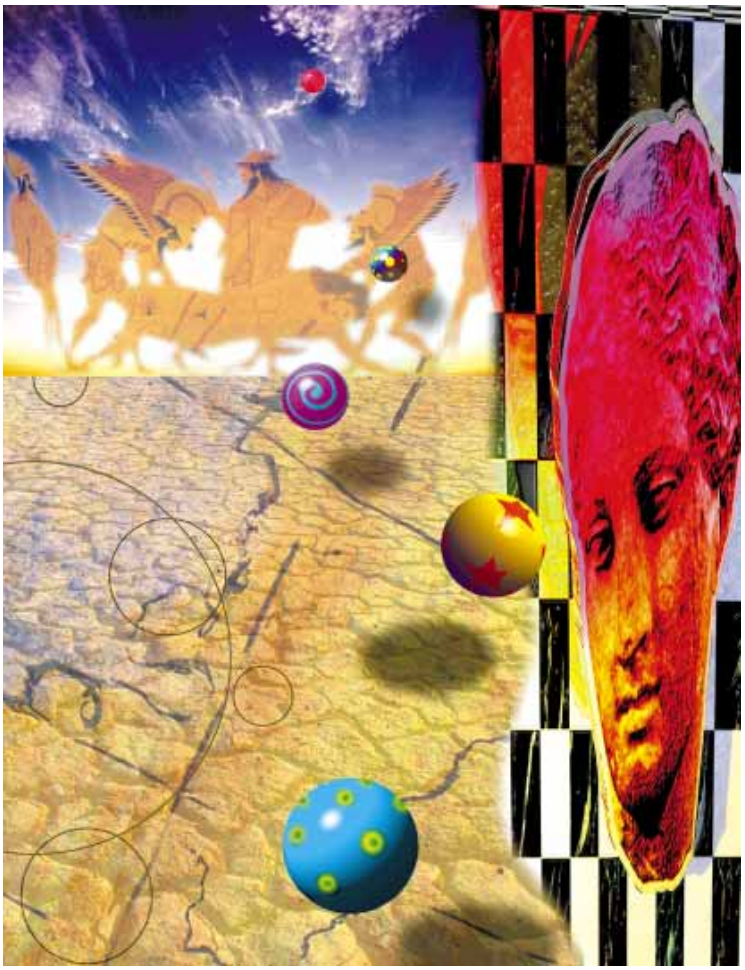


Cover Art By: *Tom McKeith*

Sharing Components

Techniques for Delphi 1.0 and 2.0

A Word from the Editor

You may have noticed I did not write the editorial (or "Symposium") column of this or the March issue of *Delphi Informant*. Instead, the work of some prominent "guests" has been presented. Last month's "Symposium" was used to kick off Richard Wagner's "File | New" column which now appears on the last page. In this month's "Symposium," Borland's Zack Urlocker reveals the origin of the name "Delphi". I hasten to note that Informant Communications Group is not affiliated with Borland International and that Borland has no control of the editorial content of any of our magazines. Conspiracy buffs should instead revisit the Zapruder film. — *Jerry Coffey, Editor-in-Chief*

ON THE COVER

FEATURES

REVIEW

DEPARTMENTS

Giving Birth

One of the most fun things I've done in my career is to help build the "1.0" version of Delphi. Okay, 2.0 was pretty fun too, but, as they say, there's nothing quite like the first time.

To be honest, when we started building Delphi 1.0, it was hard slogging. A lot of the tools out there, like PowerBuilder, SQL Windows, and Visual Basic, were pretty good. Long before we even had a prototype up and running, I was on the road talking to developers to understand the problems they were facing. To be honest, most of them were pretty happy. I remember particularly thinking that the VB corporate users by and large were just the happiest bunch of developers I'd ever met. Heck, they were even having fun!

I remember one particular meeting with about a dozen developers from a major airline. They told a not uncommon story of how a senior VP had decreed that they would use Visual Basic after he prototyped an application on a weekend. The developer told me that he didn't want to have any association with — *ugh* — Basic, but after trying it out for a while, he changed his mind.

Another time, I was meeting with a development team at a Wall Street foreign exchange. They showed me this tremendously impressive application for monitoring currencies — written in SQL Windows. It was beautiful. I thought, "Oh no, another satisfied customer. Time to move on." So after a demonstration I asked how they liked the application. His answer: "It's a dog." The response time was simply too long to even be considered for production use. After all, Wall Street practically defines the phrase "Time is money."

I saw these scenarios repeated in meeting after meeting, city after city. On the surface, customers seemed pleased with the productivity gains of "Rapid Application Development" tools. But as I delved further, I found the love affair often came to a bitter end when they tried to move from prototype to production. I found lots of spaghetti code out there, and DLLs written in C to make up for performance bottlenecks in applications written in PowerBuilder, SQL Windows, and VB.

It was some nine months into the two years of the development of Delphi 1.0 before we showed it to potential customers. There were two reasons for that. First of all, Delphi was an underground project that was truly secret. Heck, for the first year we had more code names than beta testers! Secondly, and perhaps more importantly, I wanted to make sure we understood customers problems rather than simply gauging a reaction to a demonstration. That way we could ensure we were building the right product, rather than fine tuning the wrong one.

When we finally started showing Delphi to customers, the reaction was dramatic. After all, we were solving the problems they had told us about. Our mantra: "Performance, Reuse, RAD, and Scalability." This helped us to not only define the product, but to communicate the benefits to the customer.

Oddly enough, one of the problems we faced was how to name the product. As Jerry Coffey pointed out in this column ["Symposium," *Delphi Informant*, January 1996], we were experimenting with all kinds of names. Although we had early on decided that "Pascal" should not be included in the name since it wasn't really meaningful except to long-time fans, we really hadn't made much progress on the name until a few months before its release.

Leading contenders included "Visual AppBuilder" (luckily, it was taken) "Application Architect" (too much like a CASE tool), "Client Builder" (sounds like a sales prospecting package), "Object Vision" (ahh, we used that already, didn't we?) and just about every combination of the words "Visual", "Power", "SQL", "Application", "Object", and "Builder" ("Visual Power SQL Application Object Builder" anyone?)

As we were in the late stages of selecting a name, whenever I'd do a presentation, whether to potential customers, sales reps, or third party vendors, I'd always ask them what they thought of the proposed names like "AppBuilder." Invariably, the response was lukewarm. Then they'd ask, "Why don't you just call it Delphi?" So in the end, we did.

Danny Thorpe, then on the QA team, now currently part of the R&D group, had came up with the original code name "Delphi" since it was to be a client/server tool connecting to the likes of Oracle among others. We had to come up with a code name for our first beta test and everyone felt that Delphi was acceptable. We had many later code names for internal use, external use, different countries, and at one point, I must admit, I randomly made up a new code name for every presentation, so that if there ever was a leak, we'd know from where it came.

Delphi 2.0 has come a long way since then. Our original goal was to address a few of the usability issues and also migrate to a 32-bit compiler and take advantage of platform features like OLE automation, OCXes, etc. Along the way, we introduced several major innovations like Data Module Objects and Visual Form Inheritance that increased code reuse.

The 32-bit compiler itself was actually started way back when the original 16-bit version of Delphi was started. At the time it seemed like a safe bet that "Chicago" would slip out of 1994 and that Windows 3.1 would still be a viable development platform for Delphi 1.0. We were able to have most of the VCL ported to 32-bits and running with the new compiler prior to the release of Delphi 1.0. So we were quite confident that the architecture would ensure compatibility with most code from Delphi 1.0, assuming it wasn't dependent on 16-bit data assembler, data structures, or unsupported API functions.

Although it's a bit too early to announce plans for the next version of Delphi, we're certainly working on a number of fronts to further reduce the amount of code folks need to write, and make it easier to support very large projects.

Zack Urlocker

Zack Urlocker is Director of Delphi Product Management at Borland International. The views expressed here are his own. He can be reached on CompuServe at 76217,1053.

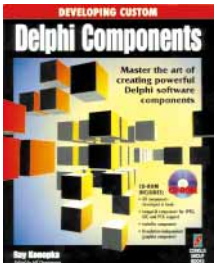
table of CONTENTS

Delphi TOOLS

New Products and Solutions

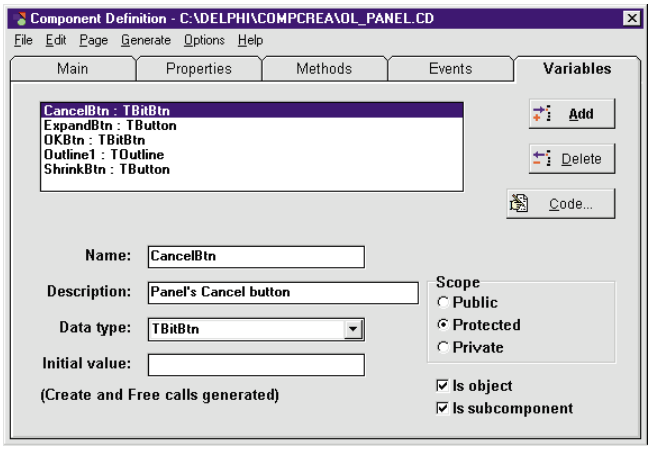
New Delphi Book

Developing Custom Delphi Components
Ray Konopka
Edited by Jeff Duntemann
The Coriolis Group



ISBN: 1-883577-47-0
Price: US$39.99, Canada $54.99 (500 pages; CD-ROM)
Phone: (800) 410-0192

Component Building Tool for Delphi Updated



Potomac Document Software, Inc., of Washington, DC, has released ***Component Create 2.0***, a design tool and code generator that enables Delphi developers to produce new components that can be added to Delphi's Component Palette, and dropped onto forms.

With version 2.0, developers can import container components from form files and create components that "wrap around" forms, such as a Windows common dialog box.

Developers can also make component properties (including inherited properties), invisible in Delphi's Object Inspector, generate palette bitmaps, view Delphi form (.DFM) files converted into text, and select detailed or normal commenting in the generated Object Pascal code.

Component Create users can develop and register custom property editors based on drop-down lists or on a developer's custom Delphi forms. In addition, version 2.0 features an improved code editor that can undo errors and add smart tabbing.

A 32-bit version of Component Create 2.0 is expected by press time and will be available to registered users at no charge.

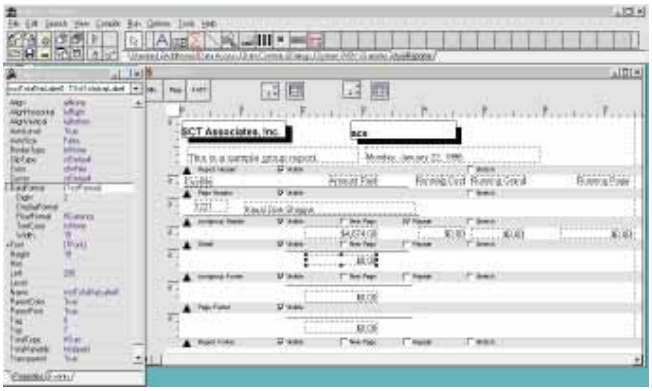
Price: US$179

Contact: Potomac Document Software, Inc., PO Box 33146, Washington, DC 20033-0146
Phone: (800) 628-5524
Fax: (202) 244-9065

SCT Associates Releases New Delphi Reporting Component

SCT Associates, Inc. of Oak Lawn, IL has released ***ACE Reporter version 1.0*** for Delphi, a VCL report component. With Ace Reporter you can create and compile reports into an .EXE file.

Ace Reporter works as a container for report components (not data components), allowing you to place multiple reports that share open tables on one form. For example, on a data entry screen for a customer table, there could be a button that prints the current customer, another button that lists all customers, and a third button that prints an open order report for that customer. Ace Reporter also allows you to minimize each report and modify the form at design time.



Ace Reporter features a Fast button for selecting multiple fields from tables on a form. Once selected, Ace Reporter generates default headings that can be edited, allows you to select bands for the headings and fields, and places them on the report.

Using the Run button in design mode, Ace Reporter tests the non-code portion of the report without building the application. You can run the report from design mode, but features relying on code (expression variables, events, etc.) will need to be tested after building the application.

ACE Reporter comes with an online help file and a Delphi .KWF file (for integrating it into Delphi's IDE), and a free upgrade to the Delphi 2.0-compatible version and source code.

A free trial version of ACE Reporter is available for downloading from the Delphi and Informant CompuServe forums, and SCT's and Informant's Web Sites, file name: ACETRIAL.EXE.

Price: US$245, including a 30-day money-back guarantee.

Contact: SCT Associates, Inc., 9221 S. Kilpatrick Ave., Oak Lawn, IL 60453-1813
Phone: (708) 425-0205
Fax: (708) 422-3877
E-Mail: CIS: 73766,1224
Web Site: http://ourworld.compuserve.com/homepages/sct

Delphi TOOLS

New Products and Solutions

Visual Components Updates to Include OCX Custom Controls

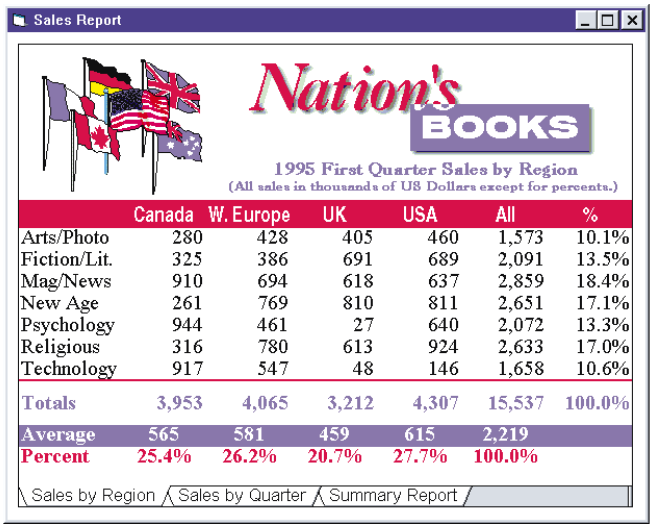
Visual Components, Inc. of Lenexa, KS has upgraded *Formula One, First Impression*, and *VisualSpeller*, as OLE Custom Controls, and announced plans to release a new version of *VisualWriter*, also as an OLE custom control.

These Visual Components upgrades include 16- and 32-bit versions, enabling developers to build applications for Windows 95, Windows NT, as well as Windows 3.1.

Being OLE-enabled, Formula One, First Impression, VisualSpeller, and VisualWriter can be used in Delphi 1.0, Delphi 2.0, Paradox 7, Visual C++ 4.0, Access 95, Visual FoxPro, Visual Basic 4.0, and other environments that support the OCX component standard.

Visual Components also plans to release a new edition of the Visual Developers Suite Deal. It will include new versions of Formula One, First Impression, VisualWriter, and VisualSpeller.

By subscribing to the new Suite Subscription Plan, developers can update and maintain their Suite software for an annual fee. The plan features quarterly updates to all software in the Suite, and includes any bug fixes and online releases. The update CDs will be mailed to subscription holders automatically.



	Canada	W. Europe	UK	USA	All	%
Arts/Photo	280	428	405	460	1,573	10.1%
Fiction/Lit.	325	386	691	689	2,091	13.5%
Mag/News	910	694	618	637	2,859	18.4%
New Age	261	769	810	811	2,651	17.1%
Psychology	944	461	27	640	2,072	13.3%
Religious	316	780	613	924	2,633	17.0%
Technology	917	547	48	146	1,658	10.6%
Totals	3,953	4,065	3,212	4,307	15,537	100.0%
Average	565	581	459	615	2,219	
Percent	25.4%	26.2%	20.7%	27.7%	100.0%	

Price: Visual Developers Suite Deal, US$299 (including online documentation); printed documentation for the Suite, US$75; Visual Developers Suite Deal Plus (includes the Suite and enrollment in the Suite Subscription Plan), US$429; Visual Developers Suite Deal Gold (includes the Suite, enrollment in the Suite Subscription Plan, and a one-year Gold Support contract), US$499. Developers that own the Visual Developer Suite Deal of OCX components may enroll in the Suite Subscription Plan for US$179.

Contact: Visual Components, Inc.
15721 College Blvd.,
Lenexa, KS 66219
Phone: (800) 884-8665, or (913) 599-6500
Fax: (913) 599-6597
BBS: (913) 599-6713
E-Mail: Internet: sales@-visualcomp.com
Web Site: http://www.-visualcomp.com

Starfish Releases EarthTime

Starfish Software, Inc. has posted EarthTime, a plug-in for Netscape Navigator 2.0, for developers to download and evaluate. With the EarthTime plug-in, Netscape Navigator users can check the time anywhere in the world without leaving their browsers. EarthTime is a component of Sidekick 95, and runs on Windows 95 and Windows NT. The EarthTime plug-in enables users to schedule appointments, conference calls, and chat sessions with participants in multiple time zones. It also allows users to simultaneously monitor the time and date in eight cities of their choice. For more information contact Starfish Software at (800) 765-7839, or visit their Web site at http://www.starfishsoftware.com.

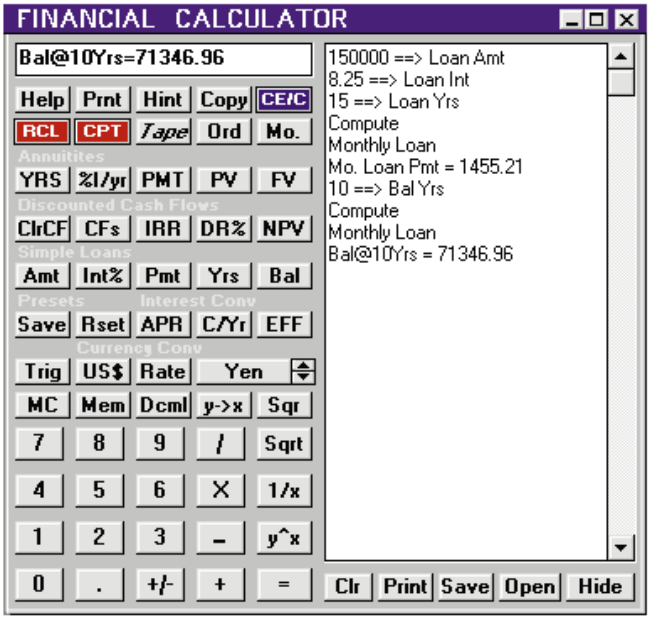
New VCL/DLL Provides a Financial Calculator

Odyssey Technologies, Inc. of Cincinnati, OH has released *VCaLc*, a native financial calculator VCL or DLL.

Developers can drop this component into their applications and allow their users to do the financial calculations used in most companies.

VCaLc can calculate annuity calculations, internal rates of return and net present value calculations, simple loans, interest conversions (effective to APR, APR to effective), and currency conversions.

Users also have a "tape" capability like a desk calculator, and may print the tape and calculations. Standard calculator features, such as addition, subtraction, and trigonometric functions are also provided.



Price: US$49

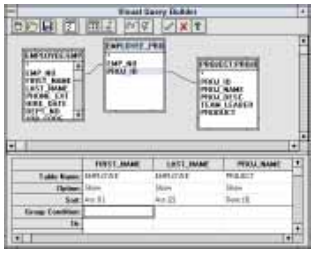
Contact: Odyssey Technologies, Inc.,
PO Box 62733,
Cincinnati, OH 45262-0733
Phone: (800) 293-7893
Fax: (513) 777-8026
E-Mail: Internet: Odyssey@eos.net
Web Site: http://www2.eos.net/-odyssey/
CompuServe: GO DELPHI

Table of CONTENTS

News LINE

April 1996

Visual Query Builder Now Available



Borland is now selling the Visual Query Builder separately for US$79.95 plus tax and shipping. Also sold as part of Delphi 1.0 Client/Server Edition, the tool enables developers to generate SQL statements using a visual metaphor. The Visual Query Builder cannot be deployed without royalties. For details call Borland at (800) 453-3375.

Borland Outlines Phased Internet and Intranet Strategies

Scotts Valley, CA — Borland International Inc. has outlined its short- and long-term strategies for producing development tools for building Internet and intranet applications. Borland recently outlined its three-phase plan for addressing the evolving needs of PC LAN and intranet developers, and displayed new tools for building Java applications. The new tools are the first components of Latte, Borland's native Java visual development environment.

The current phase of Borland's strategy calls for Internet-enabled versions of Borland C++, Delphi, Visual dBASE, and Paradox that will provide customers with the ability to develop new Web-based applications and to extend existing applications with Web interfaces.

Borland's Latte is expected to accelerate the growth of the Internet and Web as a platform for corporate computing solutions. An incremental delivery of Latte is planned. It began with the previously announced add-on components for Borland C++, and will result in Delphi-like visual tools for Java.

Latte will be an integrated visual development toolset for Java programmers developing applications for the Internet and intranets. In addition to compilers and tools, the development suite will offer class libraries, RAD functionality, native database connectivity, data aware controls, and X-platform RDBMS connections. Formal delivery dates for Latte were not available at press time.

In the next phase, Borland will introduce the InterBase InterClient, connectivity software for InterBase written in Java for networked InterBase databases. It will contain both client and server components, and will also eliminate the complexity of remote database access for Java developers.

In the last phase of this strategy, Borland will migrate intranet developers to a three-tier environment that takes advantage of cross-platform and emerging protocol standards with a Borland application server for remote Java and database access.

Code-named Nexus, this product will feature simple clients, low-maintenance client configuration, centralized business rules and validation, and database connectivity. Anticipated delivery dates were not made public, although beta versions are expected in the second half of 1996.

In addition to building client/server and Internet tools around Java, Borland sees long-term growth opportunities in the client/server marketplace by developing tools for building three-tier, distributed applications using the Web and intranets as the operating platforms.

Borland also plans to support Microsoft's Internet initiatives, including the forthcoming Sweeper SDK, which will be supported in Borland's existing Windows development tools, such as Borland C++ and Delphi.

Borland Focuses on Client/Server Market

Scotts Valley, CA — Borland International Inc. recently announced its strategy for expanding in the client/server market. Borland President and CEO Gary Wetsel said they plan to strengthen their position in the PC LAN market by delivering desktop tools that enhance productivity and shorten the development cycle.

According to Paul Gross, Borland's senior vice president of Research and Development, the success of Delphi Client/Server and its InterBase server in the last year has propelled the company's client/server revenue from approximately two percent to more than 15 percent of the company's total revenues. In addition, the company estimates that nine out of 10 Delphi client/server customers are new Borland customers.

According to the company, Borland will continue to focus on the departmental/divisional segment of the client/server market where Delphi is positioned as a database-neutral tool, co-existing with previous enterprise database standards. To support its client/server business, Borland also plans to enhance its service, sales, and support organizations to address the needs of these customers.

Visigenic to Develop ODBC Driver for Borland's InterBase

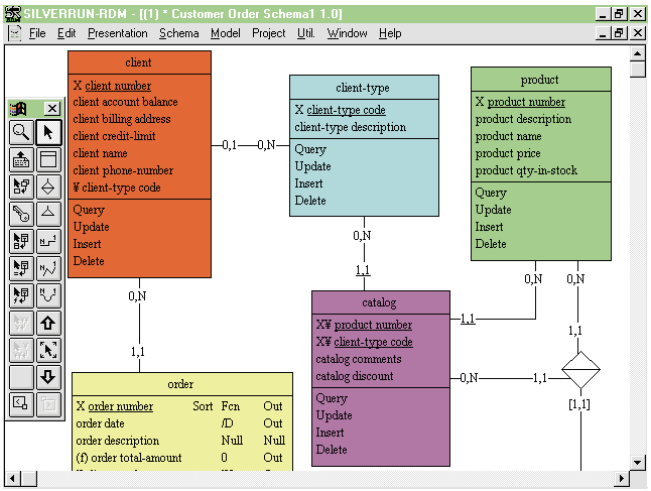
Scotts Valley, CA — Borland International Inc. has announced that Visigenic Software will develop a set of ODBC drivers for Borland's InterBase. This agreement will allow ODBC-enabled applications and development tools to access InterBase 4. InterBase 4 is designed for workgroup and departmental computing environments, and is available for Windows and most UNIX platforms. The Visigenic ODBC Drivers for InterBase will support Windows 95, Windows NT, Sun Solaris, HP-UX, and other UNIX platforms.

For more company information, visit Visigenic's Web site at http://www.visigenic.com.

News LINE

April 1996

Computer Systems Advisers Bring Data Modeling to Delphi 2.0



Woodcliff, NJ — Borland and Computer Systems Advisers (CSA) have integrated CSA's Silverrun business modeling software with Delphi Client/Server Suite 2.0. The integration will enable developers to generate data models supporting heterogeneous database environments.

Silverrun's Relational Data Modeler (RDM) module — one of four tools in the Silverrun Professional Series — is incorporated into Delphi Client/Server Suite 2.0 as an import/export facility. It enables Delphi developers to access RDBMSs such as Oracle, InterBase, Informix, and Sybase.

Silverrun-RDM provides the architecture for launching client/server applications based on graphical models designed with information stored in the RDBMSs. The models will then become incorporated into a Delphi data dictionary.

Silverrun Professional is an integrated business process and data modeling workbench for client/server development on the Windows, Solaris, OS/2, and Macintosh platforms. Silverrun Professional Series includes four modules: Entity Relationship Expert, Relational Data Modeler, Business Process Modeler, and Workgroup Repository Manager.

For more information visit Computer Systems Advisers' Web site at http://www.silverrun.com, or contact them by phone at (201) 391-6500, or e-mail: info@silverrun.com.

Spring Internet World 96 Keynote Line-Up Includes Kahn

The Mecklermedia Corp. has announced that Philippe Kahn, co-founder and chairman of Starfish Software, will be a keynote speaker at their Spring Internet World 96, scheduled for April 29-May 3, 1996 at the San Jose Convention Center in San Jose, CA. Additional keynote speakers include Larry Ellison, chairman and CEO of Oracle; Bill Gates, chairman and CEO of Microsoft; Bill Joy, founder and vice-president of research at Sun Microsystems; and Tim Krauskopf, co-founder and vice president of research at Spyglass. For more information visit http://www.iworld.com/.

Borland's New C++ 5.0 for Windows 95, Windows NT, and Java

Scotts Valley, CA — Borland has released Borland C++ 5.0, an update to its object-oriented C and C++ product.

Borland C++ 5.0 provides developers with the tools necessary to migrate to 32-bit operating systems, including: support for both 16- and 32-bit platforms; a new version of Object-Windows Library; and support for 16- and 32-bit VBX controls.

Borland C++ 5.0 includes a new native 32-bit hosted development environment, which lets developers target multiple platforms, including Windows 95, Windows NT, Windows 3.1, and DOS, from a single integrated development environment.

Version 5.0 also includes a multi-target project manager, which lets developers build 16- and 32-bit applications concurrently. A Windows 95 logo-certified product, Borland C++ 5.0 is compatible with most Windows 95 user-interface standards and features, including OLE, registry, and long filenames. Borland C++ 5.0 also includes the complete 16-bit hosted Borland C++ 4.52, for programmers using Windows 3.1.

Borland C++ 5.0 has a 32-bit debugger with integrated resource editing, including new Windows 95-based controls, multi-threaded and multi-process debugging support, and an expanded dialog editor that provides support for Windows 95-based common controls.

For Internet development, Borland C++ 5.0 includes integrated development tools for Java, including Sun's Java Development Kit (JDK). The JDK works within Borland's integrated development environment (IDE), and allows programmers to develop cross-platform code, which can run on many popular operating systems, including Windows 95, Sun Solaris, Macintosh, and others.

In addition, Borland C++ 5.0 includes the Borland Debugger for Java (the only GUI debugger for Java written in Java), as well as AppExpert for Java-specific applications and applets.

Borland also announced the release of Borland C++ Development Suite, which includes Borland C++ 5.0; CodeGuard 32/16, a new version of Borland's automated bug detection and diagnosis tool; PVCS Version Manager; InstallShield Express; and the new App*Accelerator* for Java, a compiler that increases the

"*Borland's New C++ 5.0 for Windows 95, Windows NT, and Java*"
continued on page 7

News LINE

April 1996

The Coriolis Group Joins ITP Media Group

International Thomson Publishing (ITP) announced The Coriolis Group has joined their ITP Media Group. The ITP Media group now includes Course Technology, boyd & fraser, Ventana Communications Group, and The Coriolis Group. The Coriolis Group will remain based in Scottsdale, AZ and will function as an autonomous publishing operation under The Coriolis Group imprint. The Coriolis Group's Web site can be accessed at http://www.coriolis.com.

Delphi 2.0 Information on the Internet

Scotts Valley, CA — Borland International has released several technical documents that outline Delphi 2.0's new features. These include D2_Q&A.ZIP, a question and answer document that addresses general questions; D2_COMP.ZIP, a detailed look at the inner workings of the Delphi 2.0 compiler; D2_DB.ZIP, a white paper describing the client/server database architecture of Delphi Client/Server Suite 2.0; and DLP2BKS.ZIP, a list of the Delphi 2.0 books currently in production.

All these documents are available online. On CompuServe type "GO DELPHI", or visit the Borland FTP site at ftp.borland.com/pub/techinfo/techdocs/language/delphi/gen/. You will also find articles comparing Delphi 1.0 with Visual Basic and PowerBuilder, and more.

Borland also has a Web site at http://www.borland.com, and a BBS at (408) 431-5096. For Delphi-specific technical information visit the http://www.borland.com/techinfo/delphi/index.html page of Borland Online.

Borland Ships New Paradox 7 Runtime, Client/Server, and Developer Tools

Scotts Valley, CA — Borland International Inc. has announced three new products in the Paradox 7 family of database products: Paradox 7 Runtime, Paradox 7 Client/Server, and Paradox Developer Tools.

Paradox 7 Runtime allows database developers to distribute their single-user or PC LAN Paradox applications without requiring end-users to have a copy of Paradox 7 installed.

Additionally, Paradox 7 Runtime protects a developer's source-code, and doesn't require royalty or license fees for distribution. Paradox 7 Runtime is priced at US$299 and bundles Stirling Technologies' InstallShield product. Owners of previous Runtime versions of Paradox can upgrade for US$249.

Paradox 7 Client/Server features development tools for creating front-ends to existing Oracle, Sybase, SQL Server, InterBase, and Informix database servers. The product includes Paradox 7, a client/server-enabled version of Paradox 7 Runtime, unlimited deployment licenses for Borland's new 32-bit SQL Links, a single-user Local InterBase Server, and Borland's Data Pump Expert. Paradox 7 Client/Server costs US$1,495.

For more information about the Paradox 7 Runtime or Paradox 7 Client/Server, call Borland at (800) 233-2444.

Available at no charge to members of Borland's Paradox Developer Connections Program, Paradox Developer Tools is a series of add-on utilities that make Paradox application development easier and more productive. Many of these tools have been developed in ObjectPAL by the Borland Paradox development team. The first of these utilities to be available is ObjectSpy, an object-based documentation add-in that documents the components of a Paradox application. Future additions to the Paradox Developer Tools program will include the source code for the Experts in Paradox 7, and a new Library Prototyping Expert. For more information about Paradox Developer Tools and the Paradox Developer Connections Program, call Borland at (800) 353-2211.

Delphi Developers Conference Set for May

Orange, CA — The Orange County Delphi Users Group will be sponsoring the Southern California Delphi Developers Conference May 4, 1996 at Chapman College in Orange, CA. The event will include tracks for beginner through advanced Delphi developers.

The conference costs US$99. For more information or to register, call (714) 855-9789, fax (714) 457-9641, or e-mail drdelphi@mannatech.com.

Borland's New C++ 5.0 for Windows 95, Windows NT, and Java (cont.)

performance for Java applications and applets by up to 10 times. App*Accelerator* works with all Java applications and applets regardless of the development tools used to create them.

Borland C++ Development Suite 5.0 is priced at US$499.95, and Borland C++ 5.0 is priced at US$349.95. Current owners of other Borland products and owners of Microsoft Visual Basic, Microsoft Visual C++, or Watcom or Symantec C or C++ products, can buy Borland C++ 5.0 for US$249.95.

Upgrades will be provided on CD-ROM and include complete online documentation. Diskettes and printed documentation are available separately at an additional charge. For more information, call Borland at (800) 645-4559.

News LINE

April 1996

Borland Announces Premier Value Added Partner Program

Borland has announced the Premier Value Added Partner Program, a new program targeting client/server system integrators and consultants that provide corporations and government clients with applications, consulting, and training services for Borland's client/server software products (Delphi Client/Server, InterBase, and ReportSmith).The Premier Value Added Partner Program's annual fee includes software, rebates, technical support, and marketing tools and programs. Each partner must also complete Borland sales and product training. For more information, contact Borland at (408) 431-5117.

Borland Reports a Profit in Third Quarter Fiscal Results

Scotts Valley, CA — Borland International Inc. announced a net income of US$849 thousand, or US$.03 per share, on revenues of US$47.3 million for its third fiscal quarter ending December 31, 1995. These results reflect the third consecutive quarter of profitability since Borland restructured its operations in January, 1995, and focused its strategy on software developers. The net income for the nine months ending December 31, 1995 was US$6.3 million, or US$.20 per share, on revenues of US$152.3 million.

The company incurred a net loss of US$22.9 million, or US$.80 per share, in the same quarter of the previous fiscal year, on revenues of US$48.1 million. Included in this result is a US$10 million gain associated with the sale of Borland's Quattro Pro spreadsheet product line to Novell, Inc.

In the nine months ended December 31, 1994, net income was US$38.8 million, or US$1.21 per share, on revenues of US$198.6 million. Included in the previous year's results is a US$109.9 million non-operating gain on the sale of Quattro Pro, revenue of US$24.5 million from the sale of Paradox licenses to Novell, and a one-time charge for purchased technology of US$16.2 million related to the company's acquisition of ReportSmith. Excluding these transactions, Borland would have reported a pre-tax loss of US$68.6 million in the nine month period of the previous fiscal year.

Java 1.0 Available for Download

Palo Alto, CA — JavaSoft, the newly-formed operating company of Sun Microsystems, Inc., has made the Java 1.0 programming environment available for download at http://java.sun.com.

This release incorporates the Java Applet Viewer for running and testing applets, the Java Compiler, a prototype debugger, and the Java Virtual Machine to run Java-based programs. Also included are class libraries for graphics, audio, animation, and networking.

Java 1.0 is available for Windows 95 and Windows NT on Intel, Solaris, and SPARC platforms. Java 1.0 for Mac OS 7.5 is expected by the end of the first quarter of 1996. In addition, ports to other significant operating systems are underway outside of JavaSoft. For example, IBM has announced plans to build ports for Microsoft Windows 3.1 and OS/2, and OSF has announced plans to build ports for additional versions of UNIX.

Java-based applications are platform-independent; only the Java Virtual Machine needs to be ported to each platform. It acts as an interpreter between an end-user's computer and the Java-based application. An application written in the Java environment can run anywhere, ending the need for porting applications to multiple platforms.

The Java Virtual Machine is currently available through JavaSoft's HotJava and Netscape Navigator 2.0 Web browsers, and will be available in Oracle's PowerBrowser and Spyglass' Mosaic browsers.

InstallShield Provides Software Deployment Toolkit for Borland C++

Schaumburg, IL — InstallShield Corp. and Borland have created a software deployment toolkit, InstallShield Express, for the Borland C++ Development Suite.

Borland will also work with InstallShield to provide software deployment solutions for other Borland products, such as Delphi 2.0, Paradox 7, and Visual dBASE.

InstallShield Express Borland C++ Development Suite 5.0 Edition is a custom version of InstallShield Express that integrates with Borland C++ Development Suite 5.0. It allows programmers to create installations for Windows 95 and Windows NT.

The new product features InstallShield Objects, which automate the installation of ObjectWindows Library 5.0, Visual Database Tools, and other technologies used by the Borland C++ Development Suite.

It also enables developers to specify product components and files, set up program folders and icons, make system file and registry changes, and select InstallShield Objects to add third-party software components.

InstallShield Express Borland C++ Development Suite 5.0 Edition assists in creating installation programs that know how to install shared dynamic link libraries, OCX controls, and the Borland Database Engine on the target system.

Borland C++ Development Suite users may purchase InstallShield Express Professional from InstallShield Corp.

For more information visit Borland's Web site at http://www.borland.com.

table of CONTENTS

By Dana Scott Kaufman

Working in Streams (without Getting Wet)

Using Streams to Read and Write Memo Field Data

Memo fields are common in applications that need to store an indefinite amount of character-based information in individual records in a database. Although memo fields are widely used, Delphi contains no easy way to import and export data from these fields. The DBMemo component is the only ready-made control that can access a memo.

But what if you need to store and manage large amounts of text, but don't include a *TDBMemo* object on any of your Delphi forms? For example, I regularly store program configuration data in string lists for easy manipulation at run time. The program must be able to load or store this information at any time.

Borland provides the means to perform these tasks, but they are poorly documented. This article will explain how to use streams to import and export data from fields in a database. Later, we'll address using these techniques with data stored on a SQL server, and the benefits of using these methods. Code examples are included that show how to read and write a string list to a memo field.

Into the Stream

To accomplish these data import/export feats, we must first become familiar with the concept of *streams*. Streams allow applications to read and write data sequentially to and from a medium that can store binary data. The stream can store the data in memory, on disk, or on other devices.

The main strength of streams is that the actual medium used is irrelevant. All streams work the same, in that the same methods work on memory blocks, individual files, etc. Streams have two properties, size and position, that denote the stream's size in bytes and the current position within the bytes. Streams also provide methods for reading, writing, and copying bytes into and out of the stream.

It may help to think of a stream as you would magnetic audio tape. Sound is recorded sequentially on tape in much the same way that data is stored in a stream. Continuing this metaphor, a tape counter is similar to the stream position property because it indicates the current position on the tape. To hear the sounds, or obtain data from the stream, you must ensure you "rewind" the position property back to where you want to start, usually position 0 for the beginning.

BLOb Streams to the Rescue

The key to manipulating memo fields in Object Pascal is the *TBlobStream* class. BLOb streams provide an easy way to access or modify a memo field by allowing you to read or write to the field as if it were a file or stream.

The *Create* constructor is used to link the field to the BLOb stream. Here is its syntax:

```
constructor Create(Field : TBlobField;
                   Mode : TBlobStreamMode );
```

Create has two parameters. The first is a reference that points to the memo field to be manipulated. The second specifies the read-write *mode* with one of the *TBlobStreamMode* constants: *bmRead*, *bmWrite*, or *bmReadWrite*. As Delphi's online documentation states: use *bmRead* to access an existing memo field, *bmWrite* to clear the contents of the field and assign a new value, and *bmReadWrite* to modify an existing value. After the BLOb stream is created, the data

can be accessed. To do this, we use the stream methods available to all classes derived from the *TPersistent* class. Note that many of these objects contain stream calls that are not documented in Delphi's online help.

MemoToList: Reading Memo Fields into Memory

First, we'll discuss how to read data from a database into memory where it can be readily used. Then we'll go over the steps needed to write data to a memo field in the database.

To read data into memory, we use the *LoadFromStream* procedure. Here is its syntax:

procedure LoadFromStream(*Stream*: TStream);

This will load data from the specified stream into the object. The *MemoToList* procedure creates a *BlobStream* in read mode and then pulls the contents of a memo field into a *TStringList*:

```
procedure MemoToList(SourceTable:  TTable;
                     SourceField:  string;
                     DestList:     TStringList);
var
  BlobStream1: TBlobStream;
begin
  BlobStream := TBlobStream.Create(TMemoField(
    SourceTable.FieldByName(SourceField)),bmRead);
  DestList.LoadFromStream(BlobStream1);
end;
```

As you can see, *MemoToList* takes three parameters. The first is a *TTable* that should be attached to the table and positioned on the record you want to read. The second is a *String* containing the field name from the table to read, and the third is the *TStringList* that will contain the text read back from the memo field. In the *Create* constructor, we use the *TMemoField* to inform the compiler that *SourceField* is actually a memo field.

ListToMemo: Writing Memo Fields to a Table

Now that we can read data from the memo field, we must be able to put data into it. As in the previous example, we have to create a *TBlobStream* that references the field we want to manipulate. This time, we'll use *bmWrite* as the mode. This will clear the field before the data is inserted. We then save the text from the string list to the BLOb stream. The syntax of the *Write* method is:

function Write(const *Buffer* ; *Count* : Longint) : Longint;

Write requires two parameters: *Buffer*, which is a block of memory that contains the string you want to write, and *Count*, which is the number of characters (bytes) to be written. *Write* is a function and returns the number of characters that were written.

Figure 1 shows the source code for the *ListToMemo* procedure. In our example, the text we want to write to the memo is contained in a list. We can get a pointer to the memory that contains the text by using the *GetText* function. This will return the address in memory where the string starts. Because the *Write* function requires the actual text as the *Buffer* parameter, we attach a caret character (^) to the *GetText* call. This instructs Delphi to use the actual string that *GetText* is pointing to.

```
procedure ListToMemo(DestTable:  TTable;
                     DestField:  string;
                     SourceList: TStringList);
var
  BlobStream1: TBlobStream;
begin
  BlobStream1 :=
    TBlobStream.Create(TMemoField(
      DestTable.FieldByName(DestField)),bmWrite);

  try
    BlobStream1.Write(SourceList.GetText^,
                      StrLen(SourceList.GetText));
  finally
    BlobStream1.Free;
  end;

end;
```

Figure 1: The *ListToMemo* procedure.

We also need to pass the size of the string. We do this by using the *StrLen* function. *StrLen* takes a null-terminated string as a parameter and returns the number of characters contained in the string. So, we pass the text from the *TStringList* we retrieve to the *StrLen* function by using the *GetText* function again. The call to *Write* resembles this:

```
BlobStream1.Write(SourceList.GetText^,
  StrLen(SourceList.GetText));
```

That's it. You now have the code to read and write text to and from memo fields in a database. For convenience, I placed the *MemoToList* and *ListToMemo* procedures in a unit called *MemoList*. When I need their functionality in a Delphi form, I include the *MemoList* unit in the form's **uses** clause.

This article is accompanied by the sample Memo I/O application (see Figure 2) that uses these two routines. The form contains two *TMemo* components (*Memo1* and *Memo2*), two *TButtons* (one to save and the other to load a button), and a *TTable*.

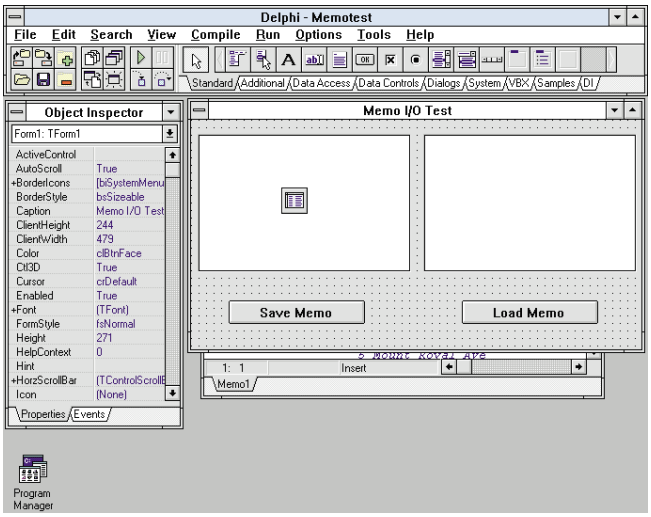


Figure 2: The sample Memo I/O application.

The **Save Memo** button writes the contents of *Memo1* to a memo field in a sample Paradox table using the *ListToMemo* function. Memo controls have a *Lines* property that is a kind of string list, allowing me to cast the *Lines* property to a *TStringList* so I can properly call the *ListToMemo* method. The *OnClick* method for the **Save Memo** button is similar to this:

```
procedure TForm1.SaveButtonClick(Sender: TObject);
begin
  Table1.Open;
  Table1.Edit;
  ListToMemo(Table1,'MemoField',TStringList(Memo1.Lines));
  Table1.Post;
end;
```

Also, notice we need to put the table into edit state before calling the *ListToMemo* function. We can do this by using the table's *Edit*, *Append*, or *Insert* method before calling *ListToMemo*. Finally, we must post the changed record after the *ListToMemo* call. The call to *MemoToList*, the **Load Memo** button's *Click* method, looks similar to the *ListToMemo* call above. Remember the *TTable* must be on the appropriate record that you want to read.

Conclusion

We have discussed getting data into and out of tables. However, we haven't addressed the types of tables that can be used. The sample Memo I/O program uses a Paradox table to store the data. The *ListToMemo* and *MemoToList* methods can also be used on SQL server tables. I have successfully used these routines on InterBase, Microsoft SQL Server, and Sybase tables. There is no reason it shouldn't work on other database server types that are available through the Borland Database Engine.

At first glance, streams can be a complex and confusing issue. To make matters worse, there's little to no documentation on this extremely useful type of object. Once you understand their functionality however, you will likely use them all the time. By using the methods described above, you can add memo fields to all your databases and not have to worry about how to fill them.

ListToMemo and *MemoToList* are extremely handy routines that I use on most of my development projects. The same techniques can be used to store and retrieve graphics, data points, and even other objects — but that's for another article. Δ

The Memotest project is available on the Delphi Informant Works CD located in INFORM\96\APR\DI9604DK.

Dana Kaufman is a Senior Consultant with Apogee Information Systems, Inc., a Massachusetts-based consulting and development firm specializing exclusively in Delphi and Paradox applications. He is a contributing technical editor for QUE on their Delphi 2 products. Dana can be reached at (508) 481-1400 or via the Internet at aisys@ix.netcom.com.

Delphi / Object Pascal

By Craig L. Jones

PQA: Part II

Practical Quality Assurance Techniques for Delphi

As every programmer knows, much can go wrong with even the simplest of programs. Worse yet, the number of possible problems grows geometrically with the size of an application. The good news is that learning just a little quality assurance (QA) savvy can go a long way towards heading off disaster. There's nothing mysterious about QA — just think of everything that can go wrong and test for it. The trick is to eliminate the tedious aspects by writing test drivers that automate as much of the testing as possible.

This is the second of three articles on assuring the quality of Delphi programming projects. This series is primarily directed towards Delphi programmers and assumes no prior knowledge on the subject of QA. Hopefully, programmers who previously gave QA little thought will discover some enthusiasm for applying the techniques presented here. These techniques are quick to implement, easy to maintain, and thereafter automatically reusable.

This installment will expand the QA tool kit (introduced last month) to cover the testing of more complicated procedures and object methods. Facilities will be added for running multiple test drivers consecutively (unattended) and using a comparison program to check the results against an established baseline. [For an introduction to the QA tool kit, see Craig Jones' article "PQA: Part I" in the March 1996 *Delphi Informant*.]

A Quick Review

Last month, some general QA theory was presented, along with how it applies to the different stages of program development. Eight types of testing were outlined:

1) requirements verification
2) design validation
3) unit
4) integration
5) user acceptance
6) alpha
7) beta
8) gamma

In addition, a tool kit was established for performing unit testing — this was probably the most important of the eight.

Unit testing is a matter of focusing on the subprocesses contained within a program and testing each individually. To introduce the subject, a test driver was written to exercise a simple function, showing how to consider four different areas of concern: *path coverage*, *boundary conditions*, *performance*, and *regression*.

The example function tested took one string argument, a book title, and returned another string that represented the title in a sortable form. For example:

"A 3 Tier Solution"

would be converted to:

"THREE TIER SOLUTION, A"

The test driver called the function multiple times, passing it a series of different titles and checking the results with an "assertion tool."

Once an automated test driver is written, it can be executed quickly at any time to ensure that the unit (still) works properly.

Expanding and Encapsulating the Tool Kit

Before proceeding, it would be prudent to better organize the tool kit. For the sake of simplicity, the tools were initially presented as a collec-

Procedure	Description
QAStart	Procedure called to start a new test run, passing it an 8-character string to identify the test.
QALog	Procedure called to directly record an entry in the log.
QAsAssert	Procedure called to assert an equality between two string values.

Figure 1: Summary of the QA tool kit as presented in the first part of this series.

Object	Element	Description
TQA	*fileQALog*	File handle for writing the test results to disk (as an ASCII .TXT file).
	sTestID	8-character test ID, used as the name of the .TXT file.
	FilePath	Specifies the drive and/or sub-directory to contain the test result log files.
	UseForm	Set to *True* if the results will be displayed on-screen using the *TFormQALog* object.
	UseFile	Set to *True* if the results will be written out to disk according to the *FilePath* and *sTestID* fields shown above.
	Start	Procedure called to start a new test run, passing it an 8-character string to identify the test.
	Stop	Procedure called to end a test run.
	Log	Procedure called to directly record an entry in the log.
	sAssert	Procedure called to assert an equality between two string values.
	bAssert	Procedure called to assert an equality between two Boolean values.
	iAssert	Procedure called to assert an equality between two integer values.
	nAssert	Procedure called to assert an equality between two floating point values.
TFormQALog	*memoQALog*	Memo component to display the results of a test run.
	btnQAOK	**OK** button to clear the window (hide the form).

Figure 2: Summary of the new QA tool kit that is organized around two objects: *TQA* (based on *TObject*) and *TFormQALog* (based on *TForm*).

tion of stand-alone functions (see Figure 1). A better way would be to redefine those functions as the methods of an object. For one thing, this allows for defining some data fields that the newer methods need in common. Figure 2 shows this new organization. The associated code is shown in Listing One on page 16.

A secondary object, a general-purpose form, has been defined for displaying the results of any test run. The form simply consists of a Memo component that displays test results, and an **OK** button that closes it. With this generic form, the tester no longer needs to create a specific form for the task (as in last month's article). Also, the various *Assert* methods have been written to report their findings through the *Log* method. *Log*, in turn, has been modified to optionally handle writing those findings to a disk file in addition to, or instead of, the screen.

Encapsulating the Test Driver

When writing test driver code for testing an object's methods, a convenient place to store that test driver is within the object itself as another method. Figure 3 shows the class definition for a sample object, called *TVagueDate* (its code, Listing Two, begins on page 17). This class definition includes a method, *SelfTest*, that uses the new QA tool kit to exercise *TVagueDate's* other methods.

The *TVagueDate* object handles storing and processing incomplete or non-specific dates. Such an object may be found as part of a scheduling or contact management application. *TVagueDate* allows dates to be specified with unknown portions. For example, you may know when a person celebrates a birthday (i.e. month and day), but not know that person's year of birth. Likewise, you may know the month and year an event occurred, but not the specific day.

Figure 4 is a simplified form that is the front-end to a contact management database using *TVagueDate*. The birthday is stored in the database using a raw byte storage field that is 7 bytes long (the size of the combined data fields of the *TVagueDate* class). The form uses a display-only calculated field to display a text representation of the *VagueDate* birthday that is stored in the byte field (via *TVagueDate*'s *AsString* property). In addition, a **Specify** button allows the birthday to be entered or changed, via the Vague Date Entry dialog box (see Figure 5).

Another database field of type *Date* is automatically filled with a fully-specified approximation of the birthday using the *AsDateTime* property of *TVagueDate*. The **Birthday** field is set to read-only on the form so that the user is forced to properly go through the Vague Date Entry dialog box to change it.

By entering records into the database and specifying various birthdays, we can put the *TVagueDate* object through its paces. Such manual testing, however, is tedious, inconsistent, and error prone. Thus, the form also features a **SelfTest** button that calls *TVagueDate*'s *SelfTest* method. The button is invisible when the application is not in test mode (i.e. the compiler directive token *QA_Mode* is not defined, as described in our first article).

The code in Figure 6 shows how *TVagueDate*'s *SelfTest* method uses the *AsString* and *AsDateTime* properties (specifically their read methods, *GetString* and *GetDateTime*). Figure 7 shows the on-screen test results. So far, the only real difference between this test driver and the one in last month's article, is that there are lines of code before each *Assert* call that are needed to set up for the assertion check. Primarily, though, we still are only verifying a function's returned value.

Testing More Complicated Functions

Let's move on to testing some more complicated functions where the function's result value is not the only thing affected. For example, let's say that the code changes the value of an object's property, the value of a global variable, or the contents of a data-

```
TVagueness = (vdOn, vdAbout, vdBefore, vdAfter);
TVagueDate = class(TObject)
  private
    iVagueness: TVagueness;
    iMonth,iDay,iYear: Word;
  public
    procedure InitBlank;
  private
    constructor Create;
    function GetDT: TDateTime;
    function GetString: string;
    procedure SetDT(dtFrom: TDateTime);
    procedure SetMonth(iValue: Word);
    procedure SetDay(iValue: Word);
    procedure SetYear(iValue: Word);
    procedure SetVagueness(iValue: TVagueness);
  public
    property AsString: string read GetString;
    property AsDateTime: TDateTime read GetDT write SetDT;
    property Month: Word read iMonth write SetMonth;
    property Day: Word read iDay write SetDay;
    property Year: Word read iYear write SetYear;
    property Vagueness: TVagueness read
      iVagueness write SetVagueness;
    procedure UpdateDlg;
    procedure GetData(var Buff: TVDBuff);
    procedure SetData(const Buff: TVDBuff);
    function AdjustDay: Boolean;
    {$IFDEF QA_MODE}
    procedure SelfTest;
    {$ENDIF}
end;
```

Figure 3: The class definition for *TVagueDate*. Note the *SelfTest* method that is used as a test driver to exercise the other methods of the object.

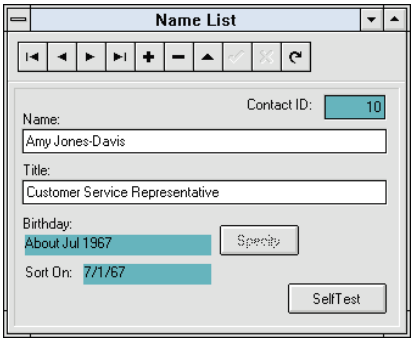


Figure 4: A simplified contact management system that uses the *TVagueDate* object for tracking birthdays. The **SelfTest** button executes the test driver associated with the *TVagueDate* object. The button only appears if the application is compiled with *QA_Mode* defined.

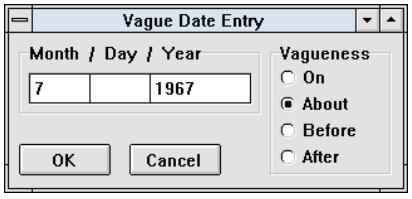


Figure 5: A pop-up form that is provided when calling the *UpdateDlg* method of *TVagueDate*. It allows the user to specify a date with unknown parameters.

base table or a disk file. Such code may be a function that only returns a status flag, or a procedure that returns nothing at all.

If a status flag is returned, it can certainly be tested using *bAssert*, for example, which is used to test a *Boolean* value. As with *sAssert*, the first argument passed to *bAssert* is the actual result of the function being tested, and the second argument is the expected result (*True* or *False*). Another routine, *iAssert*, can be used to test functions that return an integer status flag.

One of *TVagueDate*'s methods, *AdjustDay*, is used to check the combination of values currently defined by an instance of the

```
{$IFDEF QA_MODE}
procedure TVagueDate.SelfTest;
var
  QA: TQA;
begin
  QA := TQA.Create;
with QA do
    begin
      UseForm  := TRUE;
      FilePath := 'C:\QALOGS\';
      UseFile  := TRUE;
      Start('VAGUEDT');
      Log('','TVagueDate Self-Test');
      Log('','--------------------');

      InitBlank;
      Year:=1996; Month:=4; Day:=1;
      sAssert(AsString,'Apr 1st 1996');
      nAssert(AsDateTime, EncodeDate(1996,4,1));
      { Unknown Day }
      Year:=1996; Month:=4; Day:=0;
      sAssert(AsString,'Apr 1996');
      nAssert(AsDateTime, EncodeDate(1996,4,1));

      { Unknown Month & Day }
      Year:=1996; Month:=0; Day:=0;
      sAssert(AsString,'1996');
      nAssert(AsDateTime, EncodeDate(1996,1,1));

      { Unknown Year }
      Year:=0; Month:=4; Day:=2;
      sAssert(AsString,'Apr 2nd');
      nAssert(AsDateTime, EncodeDate(1996,4,2));
      { Additional tests here... }
      Stop;
    end;
  QA.Free;
end;
{$ENDIF}
```

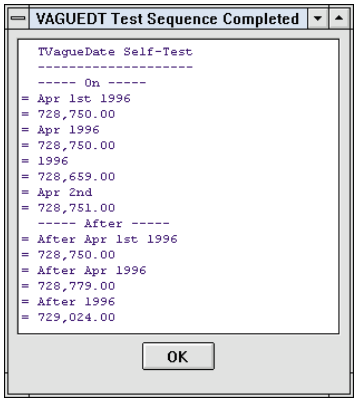


Figure 6 (Top): Excerpted code from *TVagueDate.SelfTest*.

Figure 7 (Left): Screen output of a successful test driver execution. If any of the equality assertion tests failed, then an X would have appeared in place of the equal sign (=).

object and adjust them if necessary. For example, if the day is set to 31, but the month is set to *February*, *AdjustDay* will change the day to either 28 or 29, depending if the year is a leap year. *AdjustDay* is a function that returns a *Boolean* value: *True* if the day had to be adjusted, *False* if not.

As before, to test this routine we'll call it several times, setting it up with a series of sample data designed to exercise all the various combinations. We can check the *Boolean* result flag for an expected value, but that hardly tells us anything. How do we know that the routine actually adjusted anything (or properly refrained from modifying anything)? The answer lies in using our tool kit to make some secondary assertions. In this case, we'll simply check the value of the day field directly to see if it's correct. A portion of the *SelfTest* method (see Figure 8) illustrates such test code.

```
{ Month with 30 days }
InitBlank;
Year:=0; Month:=4; Day:=30;
bAssert(AdjustDay,FALSE);
Day:=31;
bAssert(AdjustDay,TRUE);
iAssert(Day,30);

{ Leap year day }
Year:=1996; Month:=2; Day:=10;
bAssert(AdjustDay,FALSE);
Day:=31;
bAssert(AdjustDay,TRUE);
iAssert(Day,29);

{ Non-leap year }
Year:=1995;
bAssert(AdjustDay,TRUE);
iAssert(Day,28);
```

Figure 8: Additional test code to be included in *TVagueDate.SelfTest*. This code exercises the *AdjustDay* method. Since the *Day* property is changed as a ramification of the *AdjustDay* method, the value of the property is checked in addition to checking the result code returned by the method.

Similarly, if the function being tested involves adding records to a table, then an appropriate secondary assertion might be to check the table's size. Directly searching the table for a specific record — to ensure that it does or does not exist — may be a better method. If the function causes a global variable to change, or otherwise changes the state of the system, then that too could be checked directly (or indirectly).

QA Impact on System Design

As previously stated, a good software engineer knows how a system will be tested before a single line of code is written. This is because there are times when testing issues can have direct bearing on the system's design. Let's say that a test driver must be written for a function that uses the current system date as a factor (e.g. for computing a person's age or calculating a depreciation). We need to call this function several times within the driver and be able to declare the expected results. However, this is impossible if the test driver is to be reusable in the future when the current date will differ. Furthermore, we really ought to test for significant upcoming dates such as month-end, year-end, and the turning of the millennium.

One solution is to define a global variable that represents the current system date (e.g. *SysCtrls.Today*), initializing it once at startup, and then always referring to it within the application instead of directly calling the *Date* function. During normal operation, this variable is set only once at startup and then left alone, but during testing we are free to change it at will. Some other system factors that could be handled in this fashion include: user information (full name, login ID, initials), workstation information (node address, local vs. remote), and printer connection information.

Comparing Log Files

As the size of an application grows, the number and size of the associated test drivers should also grow. After a while, running the test drivers one at a time and visually inspecting the results on the screen will become tiresome. If *TQA.UseFile* is set to *True*, then the results of the various assertion checks will be written to a disk file, according to the specified file path and test name. This makes it possible to run all the test drivers for an entire system consecutively and then inspect the results afterwards, simultaneously.

Furthermore, once the results are verified, the files can be copied to another subdirectory for future reference as a baseline. Thereafter, visually inspecting the results of subsequent test runs would be unnecessary. Instead, the newly generated set of files could be compared to the baseline copies using an ASCII file comparison utility program. A simple comparison utility, FC.EXE, is provided with MS-DOS (including Windows 95). Some comparison utilities are bundled with version control software packages, and others are available from programmer catalogs, as well as through many online services.

Conclusion

With the proper tools and planning, assuring software quality can be relatively painless. We've seen how to systematically develop unit test drivers that are machine-executable, and therefore easily repeatable, and we built a QA tool kit to accommodate the task.

The final installment in this series will discuss other commercially available testing software that can be used with Delphi programs. Δ

The demonstration project referenced in this article is available on the Delphi Informant Works CD located in INFORM\96\APR\DI9604Q2.

Craig Jones is a contract software engineer in Southern California, with over 14 years of programming and consulting experience. He is the programming standards SIG leader for the Orange County Delphi Users Group. He is also a member of Team Borland, supporting Paradox and Delphi on the GEnie network. Mr Jones can be reached at craig.jones@genie.geis.com or on CompuServe at 71333,3515.

Begin Listing One — UT1QA.PAS

```
{ Project:  UT1 - General Application Utilities
  Function: Quality Assurance Tool Kit }
unit Ut1qa;
{$I UT1Incl.PAS}

interface

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

type
  TformQALog = class(TForm)
    memoQALog: TMemo;
    btnQAOK: TButton;
    procedure btnQAOKClick(Sender: TObject);
  end;

  TQA = class
  private
    fileQALog: TextFile;
    sTestID: string[8];
  public
    FilePath: string[79];
    UseForm, UseFile: Boolean;
    procedure Start(sTestName: string);
    procedure Stop;
    procedure Log(sPrefix, sMessage: string);
    procedure sAssert(sActual, sExpected: string);
    procedure bAssert(bActual, bExpected: Boolean);
    procedure iAssert(iActual, iExpected: Integer);
    procedure nAssert(nActual, nExpected: Double);
  end;

var
  formQALog: TformQALog;
  QA: TQA;

implementation

{$R *.DFM}

{ Start test run. sTestName = Identifier for the test. }
procedure TQA.Start(sTestName: string);
begin
  Screen.Cursor := crHourglass;
  sTestID       := sTestName;
  if UseForm then begin
    formQALog.Caption := sTestID+' Test Sequence';
    formQALog.Show;
    with formQALog.memoQALog do begin
      Lines.Clear;
      Font.Color  := clNavy;
      Font.Name   := 'Courier New';
      Font.Size   := 8;
    end;
  end;
  if UseFile then begin
    AssignFile(fileQALog, FilePath + sTestID + '.TXT');
    { Erase the file, if not already empty. }
    Rewrite(fileQALog);


  end;
end;

{ Log message during test run. sMessage = message to log. }
procedure TQA.Log(sPrefix, sMessage: string);
begin
  if UseForm then
    formQALog.memoQALog.Lines.Add(
      Format('%-2.2s',[sPrefix]) + sMessage);
  if UseFile then
    WriteLn(fileQALog, sPrefix+sMessage)
end;

{ Assert that two strings are equal. sActual = the value
  to test. sExpected = what the tested value should be. }
procedure TQA.sAssert(sActual, sExpected: string);
var
  sLine: string;
begin
  if (sActual = sExpected) then
    Log('= ',sActual);
  else begin
    Log('X ',sActual);
    Log('  ',sExpected);
  end;
end;

{ Assert that two Boolean values are equal. bActual =
  value to test. bExpected = what tested value should be.}
procedure TQA.bAssert(bActual, bExpected: Boolean);
var
  sLine, sActual, sExpected: string;
begin
  if bActual then
    sActual := 'true'
  else
    sActual := 'false';
  if bExpected then
    sExpected := 'true'
  else
    sExpected := 'false';
  if (bActual = bExpected) then
    Log('= ',sActual);
  else
    Log('X ',sActual+' -- expected: '+sExpected);
  end;
end;

{ Assert that two integers are equal. iActual = the value
  to test. iExpected = what the tested value should be. }
procedure TQA.iAssert(iActual, iExpected: Integer);
var
  sLine: string;
begin
  if (iActual = iExpected) then
    Log('= ',IntToStr(iActual));
  else
    Log('X ',IntToStr(iActual) +
         ' -- expected: ' + IntToStr(iExpected));
  end;
end;

{ Assert that two real numbers are equal. nActual = the
  value to test. nExpected = what tested value should be. }
procedure TQA.nAssert(nActual, nExpected: Double);
var
  sLine: string;
begin
  if (nActual = nExpected) then
    Log('= ',Format('%n',[nActual]))
  else
    Log('X ',Format('%n',[nActual])+' -- expected: ' +
      Format('%n',[nExpected]));
end;

procedure TQA.Stop;   { End a test run }
begin
  if UseForm then
    formQALog.Caption:=
      sTestID+' Test Sequence Completed';

  if UseFile then
    Close(fileQALog);
  Screen.Cursor := crDefault;
end;

{ Hide the QA results form }
procedure TformQALog.btnQAOKClick(Sender: TObject);
begin
{$IFDEF QA_MODE}
  formQALog.Hide;
{$ENDIF}
end;

end.
```

End Listing One

Begin Listing Two — CM1Vague.PAS

```
{ Project: CM1 - Contact Manager Example Application
  Function: Object to represent vaguely specified dates
  (e.g. "After Apr 1996") }
unit Cm1vague;
{$I UT1Incl.PAS}
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls, ExtCtrls,
  UT1App, UT1QA;

const
  MonthList = 'JanFebMarAprMayJunJulAugSepOctNovDec';

type
  { Holding buffer for TVagueDate data }
  TVDBuff = array[0..6] of Byte;
  TVagueness = (vdOn, vdAbout, vdBefore, vdAfter);


  TVagueDate = class(TObject)
  private
    iVagueness: TVagueness;
    iMonth,iDay,iYear: Word;
  public
    procedure InitBlank;
  private
    function GetDT: TDateTime;
    function GetString: string;
    procedure SetDT(dtFrom: TDateTime);
    procedure SetMonth(iValue: Word);
    procedure SetDay(iValue: Word);
    procedure SetYear(iValue: Word);
    procedure SetVagueness(iValue: TVagueness);
  public
    property AsString: string read GetString;
    property AsDateTime: TDateTime read GetDT write SetDT;
    property Month: Word read iMonth write SetMonth;
    property Day: Word  read iDay  write SetDay;
    property Year: Word read iYear write SetYear;
    property Vagueness: TVagueness read iVagueness write
      SetVagueness;
    procedure UpdateDlg;
    procedure GetData(var Buff: TVDBuff);
    procedure SetData(const Buff: TVDBuff);
    function AdjustDay: Boolean;
  {$IFDEF QA_MODE}
    procedure SelfTest;
  {$ENDIF}
  end;

  TformVagueDate = class(TForm)
    radioVagueness: TRadioGroup;
    btnOK: TButton;
    boxMDY: TGroupBox;
    editMonth: TEdit;
    editDay: TEdit;
    editYear: TEdit;
    btnCancel: TButton;
    procedure btnOKClick(Sender: TObject);
    procedure btnCancelClick(Sender: TObject);
  end;

var
  formVagueDate: TformVagueDate;

implementation

{$R *.DFM}
{ Initialize a blank vague date }
procedure TVagueDate.InitBlank;
begin
  iMonth      := 0;
  iDay        := 0;
  iYear       := 0;
  iVagueness  := vdOn;
end;

{ Copy data to an external buffer. Parameter: 7-byte
  buffer in which to place the data. }
procedure TVagueDate.GetData(var Buff: TVDBuff);
begin
  Move(iVagueness,Buff,SizeOf(TVDBuff));
end;

{ Load the data from an external buffer. Parameter:
  7-byte buffer with the source data. }
procedure TVagueDate.SetData(const Buff: TVDBuff);
begin

  Move(Buff,iVagueness,SizeOf(TVDBuff));
end;

{ Build a string representation of the date }
function TVagueDate.GetString: string;
var
  sVagueness,sMonth,sDay,sYear: string;
begin
  sVagueness  := ''; { Initialize }
  sMonth      := '';

  sDay        := '';
  sYear       := '';
  case iVagueness of   { Vagueness prefix }
    vdAbout:  sVagueness := 'About ';
    vdBefore: sVagueness := 'Before ';
    vdAfter:  sVagueness := 'After ';
  end;
  { Month, e.g. Oct }
  if (iMonth >= 1) and (iMonth <= 12) then
    sMonth := copy(MonthList,iMonth*3-2,3)+' ';
  case iDay of   { Day, e.g. 31st }
    0:        sDay := '';
    1,21,31:  sDay := inttostr(iDay)+'st ';
    2,22:     sDay := inttostr(iDay)+'nd ';
    3,23:     sDay := inttostr(iDay)+'rd ';
    else      sDay := inttostr(iDay)+'th ';
  end;
  if iYear>0 then   { Year, 4 digits }
    fmtstr(sYear,'%4.4d ',[iYear]);
  { Combine them }
  result := sVagueness + sMonth + sDay + sYear;
  while (length(result)>0) and
        (copy(result,length(result),1) = ' ') do
    delete(result,length(result),1);
  end;

{ Build a specific approximation of the date }
function TVagueDate.GetDT: TDateTime;
var
  iM, iD, iY: Word; { Working copies }
  iThisMonth, iThisDay, iThisYear: Word; { Today }
  bConverted: Boolean; { Conversion success flag }
begin { -- Initialize -- }
  iM := iMonth;
  iD := iDay;
  iY := iYear;
  if (iM<1) then begin      { Month }
    case iVagueness of
      vdBefore:  iM := 1;
      vdAfter:   iM := 12;
      else       iM := 1;   { Use July 1st? }
    end;
  end;
  if (iD<1) then begin      { Day }
    case iVagueness of
      vdBefore:  iD := 1;
      vdAfter:   iD := 31;
      else       iD := 1;   { Use the 15th? }
    end;
  end;
  if (iY<1) then begin      { Year }
    DecodeDate(AppCtrl.dtToday,iThisYear,
               iThisMonth,iThisDay);
    iY := iThisYear;
  end;
  { Combine them }
  bConverted := FALSE;
```

```
  while (not bConverted) do begin
    try
      Result := EncodeDate(iY,iM,iD);
      bConverted := TRUE;
    except on EConvertError do begin
      if iD > 28 then begin
        iD := iD -1;
        Continue;
      end;
      Result := EncodeDate(1,1,1);
      bConverted := TRUE;
      end; { EConvertError }
    end; { try }
  end; { while }
end;

{ Load the date from a TDateTime record }
procedure TVagueDate.SetDT(dtFrom: TDateTime);
begin
  DecodeDate(dtFrom,iYear,iMonth,iDay);
  iVagueness := vdOn;
end;


{ Directly set the month }
procedure TVagueDate.SetMonth(iValue: Word);
begin
  if (iValue>=1) and (iValue<=12) then
    iMonth := iValue
  else
    iMonth := 0;
end;

{ Directly set the day }
procedure TVagueDate.SetDay(iValue: Word);
begin
  if (iValue>=1) and (iValue<=31) then
    iDay := iValue
  else
    iDay := 0;
end;

{ Adjust the day to agree with the month and/or year }
function TVagueDate.AdjustDay: Boolean;
var
  bEncoded: Boolean;
  iMaxDay: Word;
begin
  Result := FALSE; { Assume no adjustment necessary }
  if (iDay>28) and (iMonth>0) then begin
    if (iYear>0) then begin
      { Encode full date and adjust until it works }
      bEncoded := FALSE;
      while (not bEncoded) do begin
        try
          EncodeDate(iYear,iMonth,iDay);
          { If we get this far, the encode worked }
          bEncoded := TRUE;
        except on EConvertError do begin
          if iDay > 28 then begin
            iDay := iDay-1;
            Result:=TRUE;

          end;
          if iDay > 28 then
            Continue;   { Try again }
            bEncoded := TRUE;
          end; { EConvertError }
        end; { try }
      end; { while }
    end else begin
      case iMonth of    { Unknown year, so go by month }
        2:         iMaxDay := 29;
        4,6,9,11:  iMaxDay := 30;
        else       iMaxDay := 31;
      end;
      if iDay > iMaxDay then begin
        iDay := iMaxDay;
        Result := TRUE; { Adjustment made }
      end;
    end;
  end;
end;

{ Directly set the year }
procedure TVagueDate.SetYear(iValue: Word);
begin
  if (iValue>=1) and (iValue<=9999) then
    iYear := iValue
  else
    iYear := 0;
end;

{ Directly set the vagueness factor }
procedure TVagueDate.SetVagueness(iValue: TVagueness);
begin
  iVagueness := iValue;
end;

{ Pop up a dialog box to enter vague date info }
procedure TVagueDate.UpdateDlg;

var
  iTemp: Word;
begin
  with formVagueDate do begin
    { Initialize the dialog box fields: Month }
    if (iMonth > 0) then
      editMonth.Text := IntToStr(iMonth)
    else
      editMonth.Text := '';
    if iDay > 0 then    { Day }
      editDay.Text := IntToStr(iDay)
    else
      editDay.Text := '';
    if iYear > 0 then    { Year }
      editYear.Text := IntToStr(iYear)
    else
      editYear.Text := '';
    { Vagueness }
    radioVagueness.ItemIndex := Ord(iVagueness);
    ShowModal;
    { -- Save the changes (if Okayed) -- }
    if ModalResult = mrOK then begin
      try    { Save the month }
        iTemp := StrToInt(editMonth.Text);
      except on EConvertError do iTemp := 0;
      end;
      if (iTemp<1) or (iTemp>12) then
        iTemp := 0;
      iMonth := iTemp;
      try     { Save the day }
        iTemp := StrToInt(editDay.Text);
      except on EConvertError do iTemp := 0;
      end;
      if (iTemp<1) or (iTemp>31) then
        iTemp := 0;
      iDay := iTemp;
      try    { Save the year }
        iTemp := StrToInt(editYear.Text);
      except on EConvertError do iTemp := 0;
      end;
      if (iTemp<1) or (iTemp>9999) then
        iTemp := 0;
      iYear := iTemp;
      { Save the vagueness }
      iVagueness := TVagueness(radioVagueness.ItemIndex);
    end; { Changes okayed }
  end; { with }
end;

{$IFDEF QA_MODE}
procedure TVagueDate.SelfTest; { Test driver }
var
  dtTemp: TDateTime;
begin
  QA := TQA.Create;
  with QA do begin
    UseForm  := TRUE;
    FilePath := 'C:\QALOG\';
```

```
  UseFile  := FALSE;
  Start('VAGUEDT');
  Log('','TVagueDate Self-Test');
  Log('','--------------------');
  { -- Vagueness = "On" -- }
  Log('','----- On -----');
  InitBlank; Vagueness:=vdOn;
  Year:=1996; Month:=4; Day:=1;
  sAssert(AsString,'Apr 1st 1996');
  nAssert(AsDateTime,EncodeDate(1996,4,1));
  { Unknown Day }
  Year:=1996; Month:=4; Day:=0;
  sAssert(AsString,'Apr 1996');
  nAssert(AsDateTime,EncodeDate(1996,4,1));
  { Unknown Month & Day }
  Year:=1996; Month:=0; Day:=0;
  sAssert(AsString,'1996');
  nAssert(AsDateTime,EncodeDate(1996,1,1));
  { Unknown Year }
  Year:=0; Month:=4; Day:=2;
  sAssert(AsString,'Apr 2nd');
  nAssert(AsDateTime,EncodeDate(1996,4,2));
  { -- Vagueness = "After" -- }
  Log('','----- After -----');
  InitBlank; Vagueness:=vdAfter;
  Year:=1996; Month:=4; Day:=1;
  sAssert(AsString,'After Apr 1st 1996');
  nAssert(AsDateTime,EncodeDate(1996,4,1));
  { Unknown Day }
  Year:=1996; Month:=4; Day:=0;
  sAssert(AsString,'After Apr 1996');
  nAssert(AsDateTime,EncodeDate(1996,4,30));
  { Unknown Month & Day }
  Year:=1996; Month:=0; Day:=0;
  sAssert(AsString,'After 1996');
  nAssert(AsDateTime,EncodeDate(1996,12,31));
  { Unknown Year }
  Year:=0; Month:=4; Day:=2;
  sAssert(AsString,'After Apr 2nd');
  nAssert(AsDateTime,EncodeDate(1996,4,2));
  { -- Unusual Years -- }
  Log('','----- Unusual Years -----');
  InitBlank; Vagueness:=vdAbout;
  Year:=1; Month:=4; Day:=1;
  sAssert(AsString,'About Apr 1st 0001');
  nAssert(AsDateTime,EncodeDate(1,4,1));
  Year:=101; Month:=4; Day:=1;
  sAssert(AsString,'About Apr 1st 0101');
  nAssert(AsDateTime,EncodeDate(101,4,1));
  Year:=1801; Month:=4; Day:=1;
  sAssert(AsString,'About Apr 1st 1801');
  nAssert(AsDateTime,EncodeDate(1801,4,1));
  Year:=2000; Month:=4; Day:=1;
  sAssert(AsString,'About Apr 1st 2000');
  nAssert(AsDateTime,EncodeDate(2000,4,1));
  Year:=2001; Month:=4; Day:=1;
  sAssert(AsString,'About Apr 1st 2001');
  nAssert(AsDateTime,EncodeDate(2001,4,1));
  { -- Adjust Day Function -- }
  Log('','----- AdjustDay Function -----');
  { Month with 30 days }
  InitBlank;
  Year:=0; Month:=4; Day:=30;
  bAssert(AdjustDay,FALSE);
  Day:=31;
  bAssert(AdjustDay,TRUE);
  iAssert(Day,30);
  { Leap year day }
  Year:=1996; Month:=2; Day:=10;
  bAssert(AdjustDay,FALSE);
  Day:=31;
  bAssert(AdjustDay,TRUE);
  iAssert(Day,29);
  { Non-leap year }
  Year:=1995;
  bAssert(AdjustDay,TRUE);
  iAssert(Day,28);
  Stop;
 end;
  QA.Free;

end;
{$ENDIF}

{ Close (OK) the dialog box }
procedure TformVagueDate.btnOKClick(Sender: TObject);
begin
  ModalResult := mrOK;
end;

{ Cancel the dialog box }
procedure TformVagueDate.btnCancelClick(Sender:
TObject);
begin
  ModalResult := mrCancel;
end;

end.
```

End Listing Two

Delphi 1.0 / Delphi 2.0 / Object Pascal

By Cary Jensen, Ph.D.

Sharing Components

Sharing Objects Between Forms: Techniques for Delphi 1.0 and 2.0

The properties of an object are sometimes objects themselves. For example, the *DataSet* property of a DataSource component is a property of the type *TDataSet*. This means that you can assign an object of type *TDataSet* (or one of its descendants) to this property.

In most cases, Delphi makes it easy to assign a value to an object property. In the Object Inspector, when you click the down arrow of an object property, Delphi displays a list of all objects defined for that form that are of the specified type (or a descendant of it). With the *DataSet* property of a DataSource component, for example, the drop-down menu for the property contains all *DataSet* descendants defined for the form.

While this list is convenient, it does not necessarily contain all objects that can be assigned to that property. Actually, there may be objects defined within a project that are of the appropriate type, but they do not appear on the displayed list for a particular property.

Specifically, an object that is of the appropriate type, but is declared as part of another form in the project, will not appear in the property's drop-down list, even though it's a valid object for that property. For example, if *Form1* contains a Table component, the *DataSet* property of a DataSource defined on *Form2* can be assigned this Table from *Form1*, even though the Table will never appear in the drop-down list for the DataSource's *DataSet* property.

This article describes how to assign an object defined on one form (or unit) to the object property of an object defined on another form. For Delphi 1.0, this technique requires that you add Object Pascal code. Performing this task is even easier in Delphi 2.0. Let's take them in order. (And please keep in mind that the Delphi 2.0 information in this article is based on a pre-release version of that product. Behavior in the shipping version may differ.)

Sharing Objects between Forms in Delphi 1.0

A DataSet descendant (a Table, Query, or StoredProc component) defined on one form will never appear in the drop-down list for the *DataSet* property of a DataSource defined on another form. This merely means that this property cannot be set at design time.

Fortunately, however, it can be assigned at run time. Doing so allows you to share DataSet and DataSource components across multiple forms, permitting the forms to also share the same Table component (or other DataSet), and consequently, a common cursor to the table.

This valuable technique is simply a specific application of the more general capability of assigning to properties, objects and methods contained in another unit. In other words, although this technique is demonstrated here to show two forms sharing a common cursor, it could just as easily be used to let two forms share any other type of object, or even event handlers. Now that we've got the basic concepts outlined, it's time to build an interactive example. Carry out the following instructions to create two forms that share a common DataSet.

Building *Form1*

Begin by creating a new project. On the displayed form, add the following components: a DataSource, a Table, a MainMenu, and a DBGrid. Set the *DataSet* property of the DataSource to *Table1*. For the Table component, set the *DatabaseName* property to `DBDEMOS`, the *TableName* property to `CUSTOMER.DB`, and the *Active* property to *True*. Set the DBGrid's *DataSource* property

Figure 1: A form under construction.

to *DataSource1* and its *Align* property to *alClient*. Your form should now resemble Figure 1.

Now double-click the MainMenu component and create two main menu items, `&File` and `&View`. Add an `E&xit` option under the `&File` menu, and add an `&Single Record` item under the `&View` menu. Your menu should resemble Figure 2. Close the Menu Designer and enter the following statement in the *OnClick* event handler for the *MenuItem Exit1*:

```
Close;
```

Then enter the following statement in the *OnClick* event handler for *SingleRecord1*:

```
Form2.Show;
```

Now, while we're still working with *Unit1*, we'll need to add *Unit2* to a **uses** statement in *Unit1* (we'll create the second form shortly). In the **implementation** section of *Unit1*, add the following **uses** clause:

```
uses
  Unit2;
```

The completed *Unit1* is shown in Listing Three on page 23.

Building *Form2*

Now, we'll create *Form2* that will be a single record form in this project. A single record form — where individual fields are displayed using DBEdit components — is easier to create with the Database Form Expert.

Select **Help | Database Form Expert** from Delphi's menu to access this tool. Then follow these steps:

- On the first page, set **Form Options** to **Create a simple form**, and **DataSet Options** to **Create a form using TTable objects**. Click the **Next** button.
- On the second page, set the **Drive or Alias Name** combo box to **DBDEMOS**, choose **CUSTOMER.DB** from the **Table Name** list, and click **Next**.
- On the third page, click on the double right arrow (>>) to move all fields from the **Available Fields** list to the **Ordered Selected Fields** list, and click **Next**.

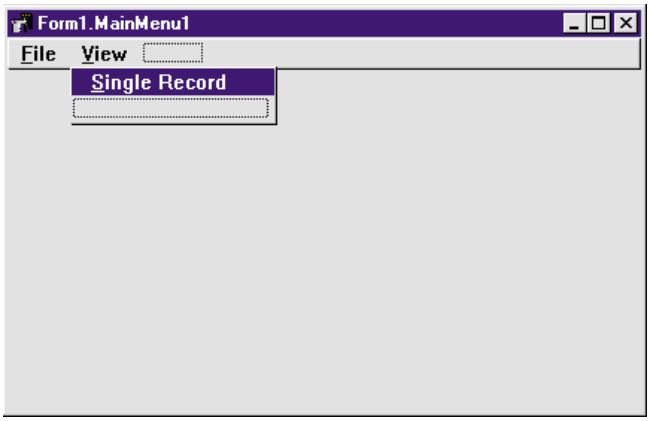


Figure 2 (Top): A menu for *Form1*.

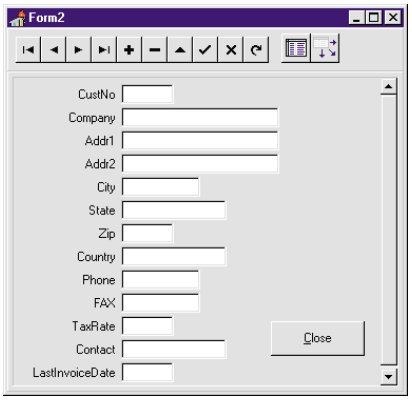


Figure 3 (Left): A single-record form created using the Database Form Expert.

- On the fourth page, set the field layout to **Vertical** and click **Next** to advance.
- On the fifth page, select **Left** to display field labels to the left of the fields, and then click **Next**.
- Finally, on the sixth page, remove the check from the **Generate a main form** check box. Click the **Create** button to build the form shown in Figure 3.

Since the purpose of this demonstration is to use *Table1* from *Form1* in *Form2*, remove the Table component from *Form2*.

The final two steps require code to be placed in *Unit2* to assign *Table1* from *Form1* to the *DataSet* property of *DataSource1* on *Form2*. (If you're working in Delphi 2.0, replace these final two steps with the ones under the section "Sharing Objects Between Forms in Delphi 2.0.") Begin by selecting *Form2* in the Object Inspector and displaying the Events page. Then, double-click the *OnCreate* event property to display *Form2*'s *OnCreate* event handler. It already contains this statement:

```
Table1.Open;
```

Delete this code and replace it with the following line:

```
DataSource1.DataSet := Form1.Table1;
```

Finally, add a **uses** clause to *Unit2* so that it can refer to the objects declared in *Form1*. Within *Unit2*'s **implementation** section, add:

```
uses
  Unit1;
```

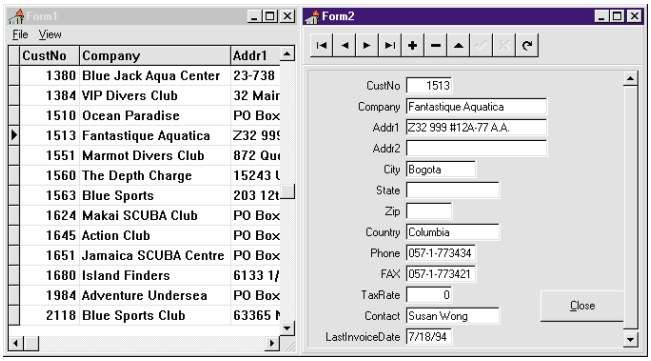


Figure 4: The project Share1.DPR. *Form1* and *Form2* both use the Table component defined in *Form1*. As a result, both forms share a common cursor, which causes them to remain synchronized, regardless of which form you use to navigate the table.

The completed *Unit2* should now resemble Listing Four on page 23.

You're done! Run this project and select **View | Single Record** to display *Form2*. With both forms displayed, notice that as you navigate one form, the other automatically remains synchronized, with both forms always showing the same record (see Figure 4). This is because the forms share the same *DataSet*, *Form1.Table*. Thus, they share a cursor.

Sharing Event Handlers

We've demonstrated that it's possible to assign a value defined in another unit to a component's property. As mentioned, this same technique can be used to assign an event handler defined in another unit to the event property of an object. For example, if you create a button named *Button1* on *Form2*, you can add the following statement to the *OnCreate* event handler for *Form2* to call the *OnClick* event handler for *Exit1* from *Form1* when *Button1* is clicked:

```
Button1.OnClick := Form1.Exit1Click;
```

Instead of assigning the procedure name defined in *Unit1* to the Button's *OnClick* event handler, you can alternatively assign the *OnClick* property of the MenuItem, *Exit1*, to this property. The following statement is functionally identical to the preceding one:

```
Button1.OnClick := Form1.Exit1.OnClick;
```

With this button added to *Form2*, clicking it causes *Form1* to close. As a result, *Form2* closes as well.

Sharing Objects between Forms in Delphi 2.0

While it's not difficult to share objects between forms in Delphi 1.0, it's even easier with Delphi 2.0. This is because Delphi 2.0 includes a feature that permits one unit to use another unit's declarations. Importantly, this feature in Delphi 2.0 permits you to set object properties at run time.

You can demonstrate how easy this is in Delphi 2.0. Create a new project by following all but the last two steps of the preceding example. Then, continue as follows:

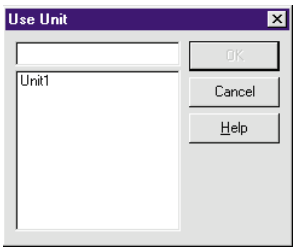


Figure 5: The Use Unit dialog box permits you to choose a unit whose interface declarations are available to the current form.

- With *Form2* selected, select **File | Use Unit** to display the Use Unit dialog box (see Figure 5). Select *Unit1* and then click **OK**.
- Select *DataSource1* on *Form2* and then display the Object Inspector. Open the drop-down list for the *DataSet* property of *DataSource1*. As shown in Figure 6, *Table1*, a *DataSet* descendant defined on *Form1*, is displayed in this list. Select *Form1.Table1* to assign *Table1* from *Form1* to the *DataSet* property of the *DataSource1* object on *Form2*. This project can now be run to produce synchronized forms.

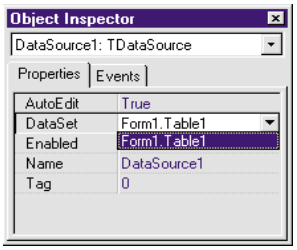
The only limitation to the Use Unit feature in Delphi 2.0 is that it does not allow you to select event handlers defined in another unit at design time. This is, however, much less of a common need than assigning objects to object properties.

Using Data Modules

In addition to a form, which can hold visual as well as non-visual objects, Delphi 2.0 supports a non-visual version of a form, called a *data module*. One of the primary uses for a data module is to hold DataSource and DataSet objects that can be used by multiple forms. The advantage of a data module over a regular form is that it takes fewer resources because it does not have a visual representation.

The use of a data module is demonstrated in the project Sharedm.DPR. This project is similar to the one shown in Figure 4, with one important exception — neither *Form1* nor *Form2* contains any DataSource or DataSet components. Instead, a data module was created by selecting **File | New Data Module**. This data module, named *DataModule1*, is shown in Figure 7.

Both *Form1* and *Form2* must use the unit associated with the data module to use the components placed in the data module. This is accomplished by selecting *Form1*, selecting **File | Use Unit**, and then selecting the data module's unit



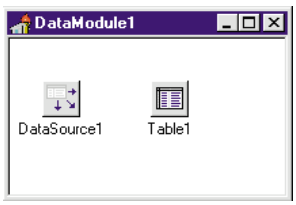


Figure 6 (Left): Because *Unit2* uses *Unit1*, a Table component declared in *Unit1* can be assigned to the *DataSet* property of *DataSource1* on *Form2* at design time. **Figure 7 (Right):** A data module holding the DataSource and DataSet components used by the Sharedm.DPR project.

from the Use Unit dialog box. This process must be repeated for *Form2*.

Conclusion

Using Delphi 1.0, the objects and event handlers defined with one unit can be used by objects on other forms by adding a small amount of code that makes the necessary property assignments at run time.

With Delphi 2.0, the Use Unit dialog box enables you to assign object properties at design time as well, greatly simplifying the process of using objects across forms. Finally, the ability to define data modules — non-visual, form-like objects — permits database developers to separate data associations from the forms used to display the data. Δ

The demonstration forms referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\APR\DI9604CJ.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including the upcoming *Delphi in Depth* [Osborne, MacGraw-Hill, 1996]. He is also Contributing Editor of *Paradox Informant* and *Delphi Informant,* and this year's Chairperson of the Paradox Advisory Board for the upcoming Borland Developers Conference. You can reach Jensen Data Systems at (713) 359-3311, or on CompuServe at 76307,1533.

Begin Listing Three — Unit1.PAS

```
unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, Grids, DBGrids, DB;

type
  TForm1 = class(TForm)
    DBGrid1: TDBGrid;
    DataSource1: TDataSource;
    Table1: TTable;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    View1: TMenuItem;
    SingleRecord1: TMenuItem;
    procedure Exit1Click(Sender: TObject);
    procedure SingleRecord1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses
  Unit2;

procedure TForm1.Exit1Click(Sender: TObject);
begin
  Close;
end;

procedure TForm1.SingleRecord1Click(Sender: TObject);
begin
  Form2.Show;
end;

end.
```

End Listing Three

Begin Listing Four — Unit2.PAS

```
unit Unit2;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, StdCtrls, Forms, DBCtrls,
  DB, DBTables, Mask, ExtCtrls;

type
  TForm2 = class(TForm)
    ScrollBox: TScrollBox;
    Label1: TLabel;
    EditCustNo: TDBEdit;
    Label2: TLabel;
    EditCompany: TDBEdit;
    Label3: TLabel;
    EditAddr: TDBEdit;
    Label4: TLabel;
    EditAddr2: TDBEdit;
    Label5: TLabel;
    EditCity: TDBEdit;
    Label6: TLabel;
    EditState: TDBEdit;
    Label7: TLabel;
    EditZip: TDBEdit;
    Label8: TLabel;
    EditCountry: TDBEdit;
    Label9: TLabel;

   EditPhone: TDBEdit;
    Label10: TLabel;
    EditFAX: TDBEdit;
    Label11: TLabel;
    EditTaxRate: TDBEdit;
    Label12: TLabel;
    EditContact: TDBEdit;
    Label13: TLabel;
    EditLastInvoiceDate: TDBEdit;
    DBNavigator: TDBNavigator;
    Panel1: TPanel;
    DataSource1: TDataSource;
    Panel2: TPanel;
    Table1: TTable;
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form2: TForm2;

implementation

{$R *.DFM}

uses
  Unit1;

procedure TForm2.FormCreate(Sender: TObject);
begin
  DataSource1.DataSet := Form1.Table1;
end;

end.
```

End Listing Four

Delphi / Object Pascal / SQL servers / Paradox

By Bill Todd

Design for Upsizing

Making It Easier to Move Delphi Database Applications from Paradox to Client/Server

As more organizations begin the move to client/server computing, the odds are increasing that your Delphi applications that use local Paradox and dBASE tables for data storage will have to be changed to work with data stored on a SQL database server. There are a number of things that you can do when you write a local table application to make the move to client/server easier.

Some of the first problems you'll encounter are that the rules for naming tables, indexes, and columns are different on SQL servers. Here are some incompatibilities you may face.

Table names. Servers will not accept a table name that contains a period. For example, if you have a Paradox table named CUSTOMER.DB you won't be able to create a table with the same name on a server. The closest you could get would be a table named "Customer".

There are several ways to deal with table name incompatibilities. The first is to add a unit to your project that's sole purpose is to hold constants that are used throughout your program. Define a constant for each table name and index name. The unit would then resemble the Globals unit shown in Figure 1.

```
unit Globals;

interface

const
  { Tables }
  tnCustomer     = 'customer.db';
  tnOrders       = 'orders.db';
  tnInvoice      = 'invoice.db';
  tnItems        = 'items.db';
  { Indices }
  inCustCustNo   = '';
  inCustName     = 'LastFirst';
  inOrderOrderNo = '' ;
  inOrderCustNo  = 'CustNo';

implementation

end.
```

Figure 1: The Globals unit.

You can add this unit to the **uses** clause of every unit where you need to use a table name or index name in your code and use the constant instead of a literal. This means that if the names of tables or indexes change when you move to a server, you need only change the constant declaration in one place, recompile, and you're ready to go.

Using a Globals unit is good practice for any fixed value. By defining a value in a single place and referencing it by the constant name, you not only make your code more readable, but you make changing any constant value easy.

Index names. All indexes on a SQL server must have a name. However, the primary index of a Paradox table does not have a name. To get around this, specify the primary index of a Paradox table in Delphi as a null string:

```
CustTbl.IndexName := ''
```

Then, when the transition to a SQL server is made, you can modify the constant without having to revisit every reference to the primary key.

Column names. Column names in Paradox tables can include spaces and punctuation characters. RDBMS column names are typically restricted to letters, numbers, and the underscore character.

The only way to avoid column name incompatibilities is to use column names in your local tables that will be acceptable to most

servers. This means that column names should start with a letter, include only the letters A through Z, the digits 0 through 9, and the underscore character.

A Property Value Problem

Using constants solves the problem when table or index names are used in code. However, you still need to deal with table and index names that are used as property values for *TTables* and other components. One technique that works in all situations is to assign the values of all table and index properties in the form's *OnCreate* event handler using the constants from the Globals unit described earlier.

However, there's one problem with assigning table names in your form's *OnCreate* event handler. If the *TableName* property is not assigned at design time, you won't be able to see live data or open the Fields editor to instantiate field objects for your tables.

Here's another alternative that you can use with *TTable* components for the table name. Use the name of the table with *no* extension as the value for the component's *TableName* property. Normally, the *TableType* property is set to *ttDefault* and Delphi determines the table's type from the file extension of the *TableName* property.

If you omit the extension from the *TableName* property you must also set the *TableType* property to either *ttParadox* or *ttDbase* so Delphi will know the table's type. This will make the move to a database server easy, provided you can create the tables on the server with the same names you use for your local tables. You do not have to change the *TableType* property back to *ttDefault* when moving your data to a server. Delphi only uses the *TableType* for local tables.

Note, however, that this only works for table names. The only solution for the primary index of a Paradox table is to assign the value in the form's *OnCreate* event handler from a constant. The index name must change when you move to a server; no server will let you create an index with a null name.

A *TQuery* Problem

TQuery components also present a problem if the SQL statement contains a table name that includes a file extension. Fortunately, in queries you can omit the file extension and not worry about the table type. The query will first search for the table with a .DB extension. If the query does not find a Paradox table, it will then search for a file with a .DBF extension (i.e. a dBASE table).

Alternatively, you can also assign those lines of the SQL statement that include table names in the form's *OnCreate* event handler by using the constants from the Globals unit. For example, the following statement will create the FROM clause of a SQL query from a constant by assigning a new string to the fourth line of the query (remember that the first line is 0):

```
CustQry.SQL[3] := 'FROM ' + tnCustomer;
```

TDatabase: Easing Migration to C/S

Using a *TDatabase* component in your application will not only make conversion to client/server easier, but also offers several benefits while you are using local tables. When moving your data to a server, you'll need a Database component to maintain your connection to the server, provide explicit transaction control through its *StartTransaction*, *Commit*, and *Rollback* methods, and specify the transaction isolation level on the server.

To use a Database component:

1) Add it to your main form.
2) Set the *AliasName* property to the alias that contains your data.
3) Set the *DatabaseName* property to the name of the temporary alias you will use in your project.
4) Set the transaction isolation level if you want a level other than the default of *tiReadCommitted*.
5) Set the *Connected* property to *True* either at design time or in your main form's *OnCreate* event handler.
6) Use the temporary alias you assigned to the *DatabaseName* property everywhere else in your project where an alias is required.

With the Database component already in place, you can move to data on a server by simply running the BDE Configuration Utility, and changing the definition of the alias used in the Database component's *AliasName* property. You must do this so that the alias points to the database on the server instead of to a subdirectory on your hard disk or file server.

While using local tables you can specify the path to the directory that contains your tables instead of setting the *AliasName* property. To use a path, set the *DriverName* property to `Standard` and add the path parameter to the Database component's *Params* property.

The path parameter takes the form:

```
PATH=C:\DIR1\DIR2
```

and can be in either upper- or lower-case. Figure 2 shows an example of setting the path parameter at design time using the String list editor. This is a particularly useful technique for a program that you will distribute to many sites because it avoids having to define a permanent alias as part of the installation procedure.

You can obtain the path from the command line, or an .INI file. If you put the .EXE file in the same directory as the tables, you can obtain the path from the *Application.ExeName* property. Just remember to remove the .EXE file's name from the end of the string so you are left with just the path to the directory containing the .EXE file.

Passing either the path to the data files, or the value to assign to the Database component's *AliasName* property on the command line, is also a handy technique because it lets you run the program against more than one database easily.

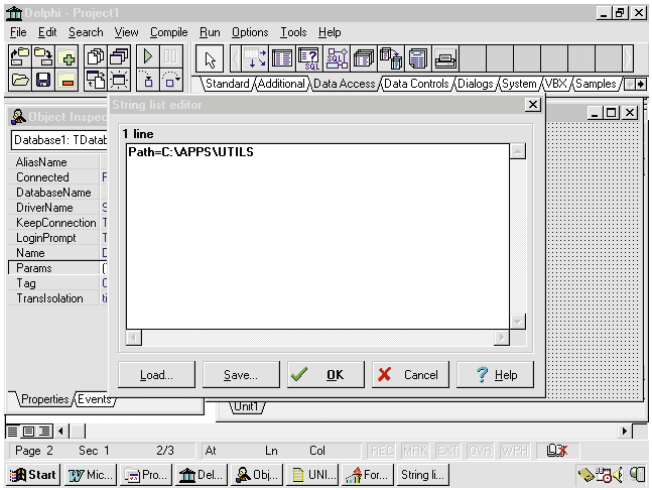


Figure 2: Using the String list editor to set the path parameter at design time.

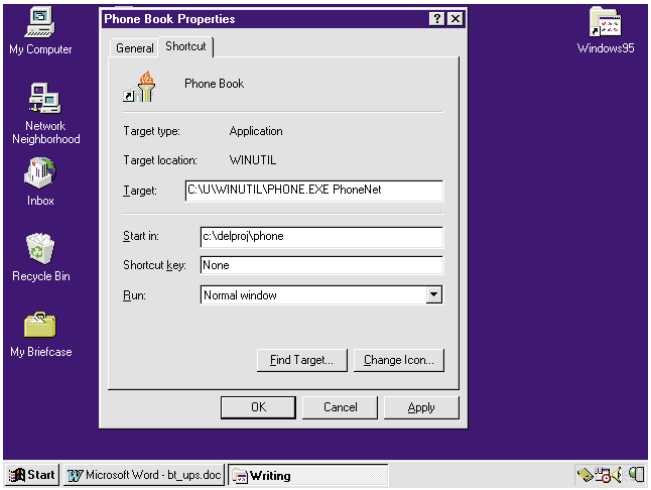


Figure 3: A Windows 95 shortcut — starting PHONE.EXE and passing the PhoneNet alias on the command line.

For example, if you have a production database and a test database, you can provide two icons on the Windows desktop. One will include the name of the production database on the command line, and the other will include the name of the test database.

Another situation where this technique is useful is with a traveler who uses a notebook computer while on the road. By passing the database alias or path on the command line, you enable the user to employ a copy of the database on the local hard drive while traveling, and the production database on the network when he or she is in the office. To get a path from the command line, use the following code in your main form's *OnCreate* handler.

```
with Database1 do
begin
  Connected := False;
  Params.Clear;
  Params.Add('Path=' + ParamStr(1));
  Connected := True;
end;
```

To pass an alias name on the command line, change the code as follows:

```
with Database1 do
begin
  Connected := False;
  AliasName := ParamStr(1);
  Connected := True;
end;
```

Figure 3 shows an example of a Windows 95 shortcut that starts an application named PHONE.EXE and passes an alias name, PhoneNet, on the command line.

Conclusion

You should write your Delphi programs to make conversion to a client/server architecture as easy as possible. It takes little effort and can save many hours of conversion and debugging time later on. Even if you do not convert the program to client/server, using a Globals unit (instead of sprinkling literal values throughout your code), will pay dividends in both readability and maintainability. In addition, using a Database component enables you to change your data's location at run time. Δ

Bill Todd is President of The Database Group, Inc., a Phoenix area consulting and development company. He is co-author of *Delphi: A Developer's Guide* [M&T Books, 1995], *Creating Paradox for Windows Applications* [New Riders Publishing, 1994], and *Paradox for Windows Power Programming*; Technical Editor of *Paradox Informant*; a member of Team Borland; and a speaker at every Borland database conference. He can be reached at (602) 802-0178, or on CompuServe at 71333,2146.

By James Hofmann

The Readers Speak

1996 *Delphi Informant* Reader's Choice Awards

Since its premiere in April of 1995, *Delphi Informant* has endeavored to bring you current information about the products and services available to the Delphi developer community. Each month's issue features Delphi-centric product announcements and industry news in our "Delphi Tools" and "Newsline" columns. *DI* also regularly features book reviews and product reviews in our respective "TextFile" and "New & Used" columns.

This month, however, we turn the tables. It's time to voice your opinion and this is the result: The First Annual *Delphi Informant* Reader's Choice awards.

We asked you to pick your favorites from nearly 100 products in 12 categories. And you responded, sending ballots by fax, e-mail, the World Wide Web, and even snail mail. As expected, some categories were highly competitive, with winners determined by few votes. In others, precedents have been set by establishing clear leaders in the Delphi add-on market.

But enough preamble; let's cut to the chase.

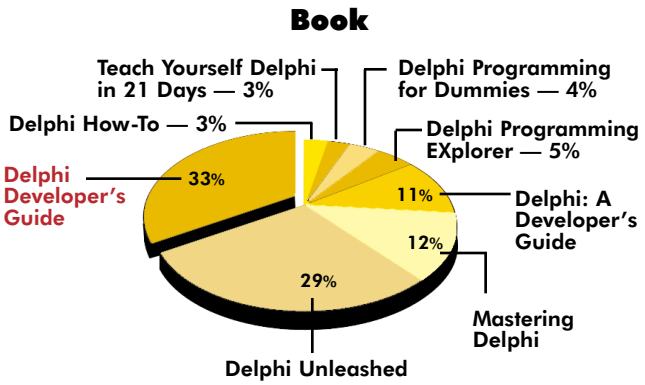
Product of the Year

With the most votes overall, this year's Product of the Year is Woll2Woll Software's InfoPower. Woll2Woll Software began by producing Paradox-specific tools, and leapt into the Delphi tools market early, releasing InfoPower just months after Delphi 1.0 shipped. For more information about InfoPower, see "Best VCL" below, and the sidebar "InfoPower Selected as Product of the Year" on page 29.

The Reader's Choice Awards proved to be highly competitive. In fact, the vote was so close for Product of the Year that we decided to give the second place finisher — Borland's InterBase server — an Honorable Mention award.

Best Delphi Book

Two books clearly dominated the Best Book category. Both from SAMS Publishing, *Delphi Developers Guide,* by Xavier Pacheco and Steve Teixeira, took first, narrowly beating Charles Calvert's *Delphi Unleashed*.



"If you're serious about Delphi, you need this

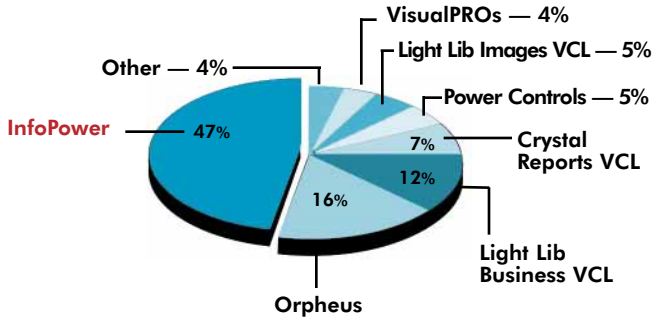
book," said Tim Feldman in his review of *Delphi Developers Guide*, in the November 1995 *DI*. "In 22 chapters, it covers every major area of Windows ... programming using Delphi."

And while perhaps not on the top shelf, two other Delphi books made a very good showing. From Sybex, Marco Cantu's *Mastering Delphi*, and from M&T Books, *Delphi: A Developer's Guide* by Bill Todd and Vince Kellen, finished third and fourth respectively.

Best VCL

Besides being Product of the Year, Woll2Woll Software's InfoPower got the nod as Best VCL. A collection of data-aware components that enhance Delphi's existing VCL components, InfoPower includes an enhanced data grid, database filtering, lookup combo boxes, expanding memo dialog boxes, incremental search components, and more.

VCL



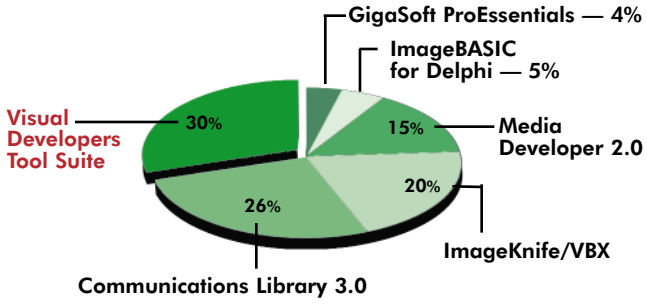
Blow the dust off your August 1995 *DI* for a detailed look at InfoPower's unique features. In his review, Joseph Fung called InfoPower's components "complete and well thought out, significantly enhancing the development process." It appears you agree. In one of the more populated categories, Woll2Woll carried a whopping 47 percent of the votes.

In the second tier of popularity, TurboPower's Orpheus and DFL Software's Light Lib Business VCL, took second and third places respectively.

Best VBX

Given the variety of VBXes, it's no surprise the competition was stiff in this category. Yet when all was said and done, Visual Components' Visual Developers Tool Suite edged out its opponents with 30 percent of the total votes.

VBX

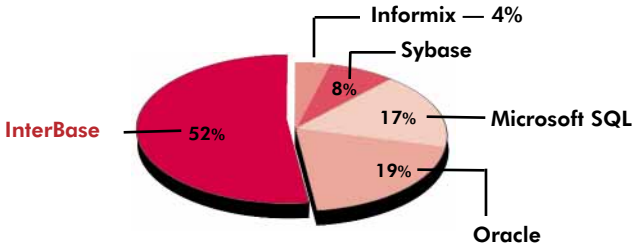


A close second, MicroHelp's Communications Library 3.0 acquired 26 percent of the vote, while Media Architects' ImageKnife/VBX took third with 20 percent.

Best Database Server

In addition to receiving Honorable Mention for its close second-place finish as Product of the Year, Borland International's InterBase secured 52 percent of the vote in the Best Database Server category.

Database Server

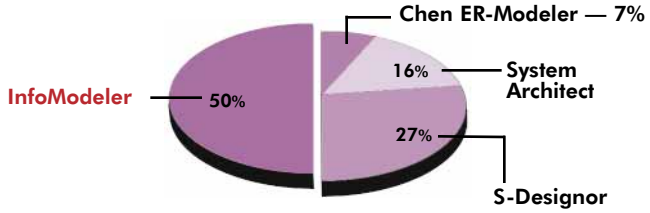


Borland's InterBase 4.0 Workgroup Server has had a great year. In its biggest sale ever, Borland closed a deal with the US Army to use InterBase in its Advanced Field Artillery Tactical Data System (AFATDS). The Army selected InterBase because of its platform-independence, robustness, and other unique features. For more information about InterBase, see the sidebar "InterBase Emerges from the Shadows" on page 30.

Best Database CASE Tool

There's no doubt about your choice as Best Database CASE Tool. Half of you agreed that Asymetrix Corporation's InfoModeler is the best way to automate database creation and maintenance. Asymetrix shipped InfoModeler 1.5 in October of 1994, adding connectivity to several databases including Informix, Ingres, Sybase, and Visual dBASE. Version 1.5 also incorporates model import/export, and two new tools: Fact Assistant and Verbalizer.

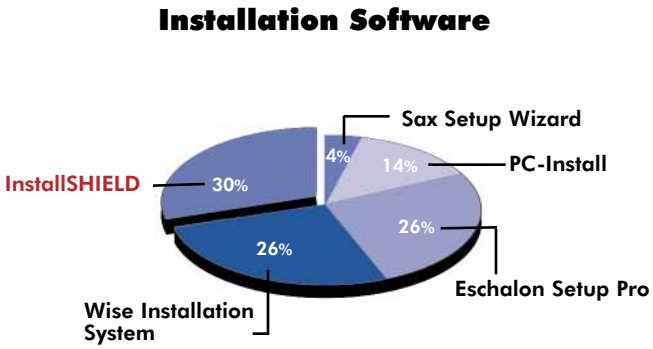
Database CASE Tool



S-Designor from SDP Technologies came in second place, followed by Popkin Software's System Architect, and Chen & Associates' Chen ER-Modeler.

Best Installation Software

To the finish, the Best Installation Software category was one of the tightest races. Three products dominated the category: InstallSHIELD from InstallShield Corporation finished first,



narrowly besting Great Lakes Business Solutions' The Wise Installation System, and Eschalon Development's Eschalon Setup Pro.

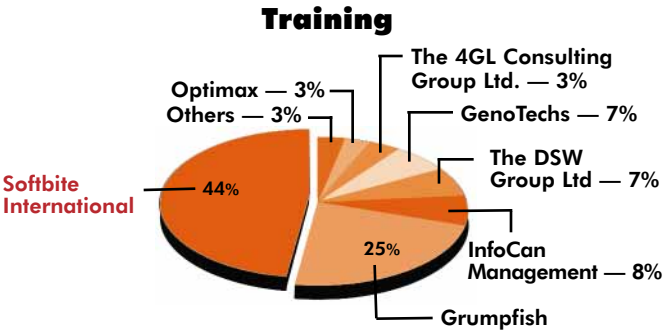
Recently, InstallShield Corporation and Borland International, have collaborated to create a software deployment toolkit, InstallShield Express, for Borland's C++ Development Suite. Borland is also working exclusively with InstallShield Corporation to provide software deployment solutions for all other Borland products, including Delphi 2.0, Visual dBASE, and Paradox 7.

Reviewed by Micah Bleecher in the February 1996 *DI*, The Wise Installation System features full system access, a custom dialog box editor, multimedia support, Windows 95 support, and notably, specific Borland Database Engine (BDE) support, among others.

You obviously consider Eschalon Setup Pro a worthy installation tool. So do AT&T, DOW, Kodak, Electronic Arts, Macromedia, Polaris, Sprint, Xerox, and the Army Corps of Engineers, all of whom have made it their standard installation tool.

Best Training

Established in 1988, Softbite International has become known not only as a leader in software training, but in consulting as well. You chose Softbite as Best Training organization, giving it 44 percent of your votes. Oregon-based Grumpfish Incorporated made a strong second place showing with 25 percent of the votes.



Still expanding, Softbite recently added accounting services. Regarding their 1996 agenda, Softbite founder Kevin Smith said, "We'll increase support for Delphi and stay on the developer side of Paradox 7."

Best Reporting Tool

All questions have been answered about the Best Reporting Tool — Crystal Reports from Crystal Services/Seagate Software is the clear winner. The recently released version, Crystal Reports 4.5, includes a Delphi VCL, a full-featured OLE (OCX) control, a 32-bit Report Engine DLL, the ability to drill down on graphs, new Lotus Notes and Excel 5.0 export formats, and the ability to save report options with the report.

These are some of the features that keep Crystal Reports a leader in the reporting tool market, and the winner of the Best Reporting Tool with 44 percent of the votes. Borland International's ReportSmith finished a healthy second, and Nevrona Designs' ReportPrinter took third place.

InfoPower Selected as Product of the Year

Delphi Informant readers selected Woll2Woll Software's InfoPower as their favorite Delphi add-in product by casting more votes for it than any other product in any category.

Woll2Woll reviewed a beta version of Delphi in late 1994 with hopes of creating a Delphi tool. They wanted to help database developers familiar with products like Paradox for Windows feel more productive with Delphi. In their market research, Woll2Woll found database developers didn't want to compromise the quickness or functionality of their applications, but wanted the speed of true .EXE files.

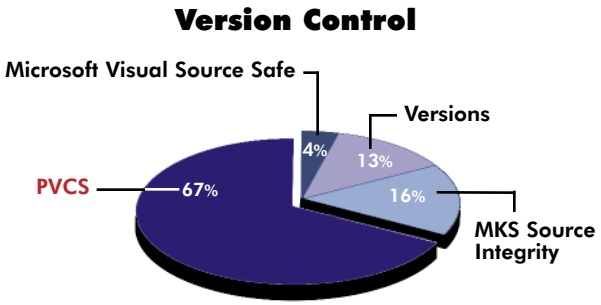
The InfoPower suite includes 15 visual and non-visual data-aware Delphi components, designed for 16- and 32-bit Windows database applications. The suite features an enhanced data-aware grid that supports fixed columns, check-boxes, embedded multi-field lookup combo boxes, dynamic cell coloring, multi-line titles, memo display, and editing. It also includes customizable dialog boxes for locating field values in a table, query, or query-by-example (QBE), where searches can be case-sensitive, pattern based, or incremental.

InfoPower also provides database lookup facilities and customizable dialog boxes that can be attached to any event. Additionally, InfoPower includes a QBE component that operates just as its Paradox counterpart, as well as a Table component that provides data filtering capabilities.

Woll2Woll is currently working on InfoPower 2.0 which they expect to ship in May, 1996. New functionality will include field validation, visual filtering and querying, and the ability to print the contents of a grid. The new version will also feature enhancements to existing InfoPower components, such as multi-record selection and bitmap support in the grid component, and other developer-requested features and functions.

Best Version Control

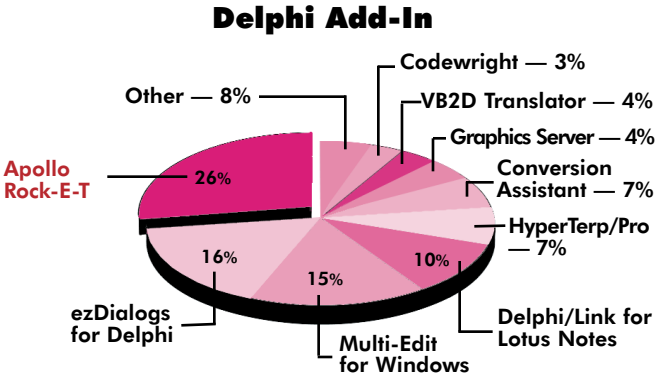
An established software configuration management tool, INTERSOLV's PVCS Version Manager commands the Best Version Control category with 67 percent of the ballots. PVCS Version Manager enables a team of programmers to track file changes during application development. Developers can check out and modify source code, executables, utilities, and documentation files.



INTERSOLV has recently announced that Borland will incorporate PVCS Version Manager into the Borland C++ 5.0 Development Suite.

Best Delphi Add-In

In another hotly-contested category, SuccessWare International and Woll2Woll Software battled to the end for Best Delphi Add-In. With an onslaught of last minute votes, SuccessWare International's Apollo Rock-E-T emerged as your favorite with 26 percent of the votes.

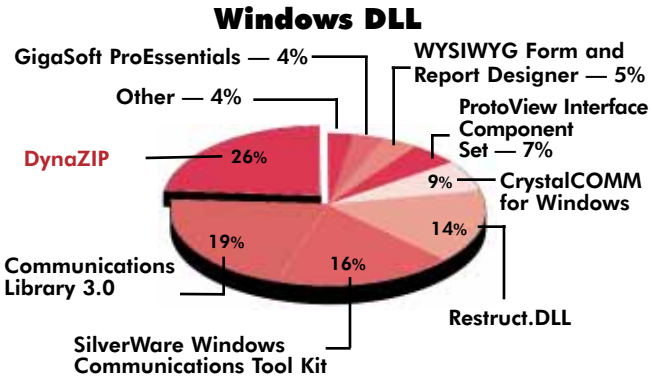


Woll2Woll Software's ezDialogs for Delphi received 16 percent of the votes. And in a near-tie for second place, American Cybernetics' Multi-Edit for Windows took third with 15 percent of the votes.

Of the 12 categories, the race for Best Delphi Add-In was the most crowded with over 13 products receiving votes. Other notables included: Borland/Brainstorm Technologies' Delphi/Link for Lotus Notes, HyperAct's HyperTerp/Pro, and EarthTrek's Conversion Assistant.

Best DLL

Giving it 22 percent of the votes, you named Inner Media's DynaZip as Best DLL. Balloting was tight however, and MicroHelp's Communications Library 3.0 came in a close second with 19 percent of the votes.



Released during the last quarter of 1995, DynaZip features 16- and 32-bit versions for Windows, Windows 95, and Windows NT, plus a new DZ_EASY interface. DynaZip also offers OCX and VBX support.

The third and fourth place finishers — SilverWare's Windows Communications Tool Kit, and TrayMar Software's Restruct.DLL — made strong showings as well.

Best Windows Help Authoring Tool

The Best Help Authoring Tool category had two clear front-runners, but Blue Sky Software's RoboHELP came in first with 46 percent of the votes. For more information about RoboHELP, pull out the January 1996 issue of *DI* and help yourself to Gary Entsminger's review.

InterBase Emerges from the Shadows

InterBase's strong showing in the *Delphi Informant* Reader's Choice Awards is just another example of its recent, rapid rise into prominence. Considered the "Crown Jewel" of the Ashton-Tate acquisition, InterBase nevertheless remained obscure and isolated from the rest of the Borland product line. Its bundling with the phenomenally successful Delphi changed all that.

Borland has finally begun a tighter integration of InterBase with the rest of their products, propelling InterBase forward in sales and market penetration. Delphi 2.0 Client/Server Suite is the latest effort in Borland's "Trojan horse" strategy to entice Delphi developers to embrace InterBase. The new Suite includes not only the new Windows 95/NT Local InterBase server, but also the multi-user InterBase NT Server for prototyping and developing multi-user applications. In addition, a new Event Alerter component has been added to the VCL, allowing developers much tighter integration to the InterBase engine.

Borland isn't content with Delphi/InterBase synergies, however, and has rumored that future versions of their Java tools, code named "Latte," may include InterBase or InterBase connectivity as well. The endorsement by Borland's InterBase of the new JDBC standard by Sun Microsystems, Inc. is further evidence that something formidable is brewing.

With growth of InterBase sales close to 40% in 1996, and new versions available almost every month, it looks like Borland's evolution from desktop tools provider to client/server provider is succeeding.

Contacting the Winners

Best Book
Delphi Developer's Guide
Xavier Pacheco & Steve Teixeira
SAMS Publishing
ISBN: 0-672-30704-9
Phone: (800) 428-5331
Web Site: http://www.mcp.com

Best VCL
InfoPower
Woll2Woll Software
2217 Rhone Drive
Livermore, CA 94550
Phone: (800) 965-2965 or (510) 371-1663
Fax: (510) 371-1664
CompuServe: GO WOLL2WOLL
Web Site: http://woll2woll.com

Best VBX
Visual Developers Tool Suite
Visual Components, Inc.
15721 College Boulevard
Lenexa, KS 66219
Phone: (913) 599-6500
Fax: (913) 599-6597
Web Site: http://visualcomp.com

Best Database Server
InterBase
Borland International Inc.
100 Borland Way
Scotts Valley, CA 95066-3249
Phone: (408) 431-1000
Web Site: http://www.borland.com

Best Database CASE Tool
InfoModeler
Asymetrix Corporation
110 110th Ave. NE, Suite 700
Bellevue, WA 98004-5840
Phone: (800) 448-6543 or (206) 462-0501
Fax: (206) 454-7696
Web Site: http://www.asymetrix.com

Best Installation Software
InstallSHIELD
InstallShield Corporation
1100 Woodfield Road
Suite 108
Shaumburg, IL 60173
Phone: (800) 374-4353 or (847) 240-9111
Fax: (847) 240-9120
E-Mail: Internet: info@installshield.com
Web Site: http://www.installshield.com

Best Training
Softbite International
33 N Addison Road, #206
Addison, IL 60101-8401
Phone: (708) 833-0006
Fax: (708) 833-0584
Web Site: http://www.softbite.com

Best Reporting Tool
Crystal Reports
Crystal Services/Seagate Software
1095 West Pender St., 4th Floor
Vancouver, BC Canada
V6E 2M6
Phone: (800) 663-1244 or (604) 681-3435
Fax: (604) 681-2934
CompuServe: GO REPORTS
Web Site: http://www.seagate.com/software/crystal

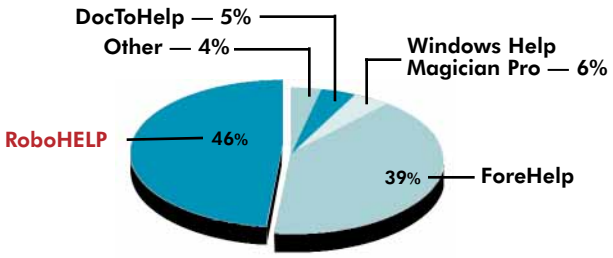
Best Version Control
PVCS
INTERSOLV
1700 NW 167th Place
Beaverton, OR 97006
Phone: (800) 547-7827 or (503) 645-1150
Fax: (503) 645-4576
E-mail: Internet: pvcsinfo@intersolv.com
Web Site: http://www.intersolv.com

Best Delphi Add-In
Apollo Rock-E-T
SuccessWare
27349 Jefferson Avenue, Suite 111
Temecula, CA 92590
Phone: (800) 683-1657 or (909) 699-9657
Fax: (909) 695-5679
CompuServe: GO SWARE

Best DLL
DynaZIP
InnerMedia, Inc.
60 Plain Road
Hollis, NH 03049
Phone: (800) 962-2949 or (603) 465-3216
Fax: (603) 465-7195
E-Mail: CIS: 70444,31

Best Help Authoring Tool
RoboHelp
Blue Sky Software Corp.
7777 Fay Ave., Suite 201
La Jolla, CA 92037
Phone: (800) 677-4946 or (619) 551-2485
Fax: (619) 551-2486
Web Site: http://www.bluesky.com

Help Authoring Tool



ForeFront's ForeHelp dominated the other contenders, placing second with 39 percent.

Thank You

In the final analysis, third-party Delphi products are only productive when they enable developers to "get the job done." This makes you — the greater Delphi community — the only true arbiter. Thanks to you, the Delphi community now has a good idea of the products available — and which are most popular.

We would also like to thank the vendors for spending the time and money to develop such excellent Delphi products. The Delphi third-party marketplace is buzzing, and the excitement will only increase when Delphi 2.0 hits the shelves. We're looking for many new Delphi products in the coming year, including OCXes, Web-enabling components, and many more we cannot even envision. Δ

James Hofmann is Assistant Editor for *Delphi Informant*.

Delphi 1.0 / Object Pascal

By Andrew J. Wozniewicz

DLLs: Part II

Completing the Example Dynamic Link Library

Last month, we introduced dynamic link libraries (DLLs) and how to create them using Delphi. As part of that interactive discussion, we created the sample unit, XSTRING.PAS, to contain five string-handling functions: *FillStr*, *UpCaseFirstStr*, *LTrimStr*, *RTrimStr*, and *StripStr*.

In this installment, we'll create the last three functions needed to fill XSTRING.PAS, then learn to export DLLs. [If you haven't built XSTRING.PAS, refer to "DLLs: Part I" in the March 1996 *Delphi Informant*.]

The *LTrimStr* Function

Often, you'll need to ensure that the string you're working with begins with a meaningful (i.e. non-blank) character. If there are any leading blanks, you may need to remove them.

For example, the standard procedure *Val* converts strings of digits into their numeric representations and requires that the first character of the argument string is a valid, non-blank digit. Before you can pass a string of digits to *Val* for conversion, you must trim the string's leading blanks.

The *LTrimStr* function is designed to perform the "trimming" of the leading blanks on the left side of the passed string argument. Enter the following declaration inside the **interface** section of the XString unit:

```
function LTrimStr(const S: string):
  string; export;
```

The only parameter passed to *LTrimStr*, *S*, is the string to be trimmed. The result returned is the same string, but without leading blanks. If there are no leading blanks in *S*, then *LTrimStr* returns the argument string unchanged.

Here's how *LTrimStr* is implemented in the **implementation** section of the XString unit:

```
function LTrimStr(const S: string):
string;
var
  Index, MaxIndex: Integer;
begin
  Index    := 1;
  MaxIndex := Length(S);
  while (Index <= MaxIndex) and
    (S[Index] = ' ') do
    Inc(Index);
 Result := Copy(S,Index,
                MaxIndex-Index+1);
end;
```

The *LTrimStr* function relies on a **while** loop to determine the position of the first non-blank character in the string. The loop terminates when a non-blank character is found or when the loop reaches the end of the string.

This last consideration is especially important to handle if the function is to be a robust, production-quality subroutine. You must ensure proper behavior with a passed argument string that contains no leading blanks, or is entirely blanks. In the former case, the string is returned unchanged. In the latter case, the result of trimming all leading blanks from a string consisting of only blanks is returned: a null (empty) string.

The loop counter, *Index*, is initialized to point to the beginning of the argument string. To ensure that the loop doesn't run past the end of the string, a guardian value,

MaxIndex, is established. *MaxIndex* is the index of the string's last character, and is equal to its original length.

After the index of the first non-blank character of the argument string is determined, the portion of the argument string *S* containing the leading blanks is removed, and the remainder is returned as the function's result.

The standard string function *Copy* is used to select only the meaningful, non-blank portion of the original argument string *S*, omitting the leading blanks from the function's result, as required.

Here's how *LTrimStr* is called in an application:

```
var
 S1, S2, S3 : string;
begin
  ...
  S1 := LTrimStr('        Teach Yourself Delphi');
  S2 := LTrimStr('    123');
  S3 := LTrimStr('    456    ');
  ...
end;
```

After the assignment statements are executed:

- *S1* contains, "`Teach Yourself Delphi`"
- *S2* contains, "`123`"
- *S3* contains, "`456  `"

Note that *LTrimStr* has no effect on trailing blanks. This is the job for the next trimming function, *RTrimStr*.

The *RTrimStr* Function

In concept, *RTrimStr* is similar to *LTrimStr*. Instead of removing the leading blanks, however, *RTrimStr* removes any trailing blanks, ensuring that the resulting string is as short as possible. A typical use of the *RTrimStr* function is to prepare a string for conversion to a number using the *Val* procedure.

Enter the following declaration inside the **interface** section of the XString unit:

```
function RTrimStr(const S: string): string; export;
```

As you can see, the declaration of *RTrimStr* is nearly identical to that of *LTrimStr*. *RTrimStr* is implemented as follows in the **implementation** section of the XString unit:

```
function RTrimStr(const S: string) : string;
var
  Index: Integer;
begin
  Index := Length(S);
  while (Index > 0) and (S[Index] = ' ') do
    Dec(Index);
  Result := Copy(S,1,Index)
end;
```

The trick to implementing *RTrimStr* that makes it even simpler than *LTrimStr*, lies in the **while** loop. This code determines the position of the last non-blank character of the string argument, runs "backward" from the end of the string, and proceeds toward the beginning.

When the loop counter *Index* — initialized to mark the argument string's last character — indicates a position of a non-blank character in the argument string, the loop has counted all trailing blanks and terminates.

In case the argument string consists of blanks only, the loop condition checks if the iteration has reached the beginning of the string, indicated by an *Index* value of less than 1. This condition also terminates the **while** loop.

After the **while** loop is finished, the *Index* variable holds the position of the argument string's last non-blank character. The *Copy* function is then used to extract the non-blank leading portion of the string and return it to the caller as the function result.

The *RTrimStr* function can be used as follows:

```
var
  S1, S2, S3: string;
begin
  ...
  S1 := RTrimStr('Teach Yourself Delphi       ');
  S2 := RTrimStr('123   ');
  S3 := RTrimStr('   456   ');
  ...
end;
```

After the assignment statements execute:

- *S1* contains "`Teach Yourself Delphi`"
- *S2* contains "`123`"
- *S3* contains "`  456`"

RTrimStr removes the trailing blanks from the argument, but the leading blanks remain.

The *StripStr* Function

The last string-handling subroutine you'll implement inside your DLLFirst library is *StripStr*. It removes all blanks — whether they are leading, trailing, or embedded within the argument string — and returns a result of all non-blank characters. The declaration of the *StripStr* function is similar to that of the others:

```
function StripStr(const S: string): string; export;
```

Since the number of iterations is known, *StripStr* uses a **for** loop for top performance. The loop must run through all the characters in the argument string, removing any blanks as it visits the consecutive character locations. Here's the implementation of the *StripStr* function:

```
function StripStr(const s: string): string;
var
  Index : Integer;
begin
  Result := '';
  for Index := 1 to Length(S) do
    if S[Index] <> ' ' then
      Result := Result + S[Index];
end;
```

The **for** loop visits every character position of the argument string *S*, and determines if the character is a blank. If it is, the current character is appended from the argument string to the function's *Result* string. Thus, the result string is accumulated one character at a time, omitting blank characters in the process.

It's important to note that *Result* is an implicit variable of the same type as the function's return type — **string** in this case — and is automatically available for every Object Pascal function. Delphi automatically initializes *Result* to an empty string. This ensures that the result is defined even if the argument string is empty and the **for** loop is never executed.

Here's an example of *StripStr* in use:

```
var
  S1, S2, S3, S4 : string;
begin
  ...
  S1 := StripStr('     Teach Yourself Delphi     ');
  S2 := StripStr('123   ');
  S3 := StripStr('   456   ');
  S4 := StripStr('   789');
  ...
end;
```

Following execution of the assignment statements:

- *S1* contains "`TeachYourselfDelphi`"
- *S2* contains "`123`"
- *S3* contains "`456`"
- *S4* contains "`789`"

Using the XString Unit

And that does it! You've implementing the XString unit which now contains five useful string-handling functions. To help you verify the code you entered is complete and valid, Listing Five (beginning on page 35) shows the entire unit. Comments were added to make the code more readable and understandable.

Each of the functions implemented in the unit's **implementation** section — *FillStr*, *UpCaseFirstStr*, *LTrimStr*, *RTrimStr*, and *StripStr* — is listed in the **interface** section. In turn, each function is made exportable by placing the **export** directive after the **function** declaration. This makes it possible to include the subroutines in an **exports** clause and make them available outside the DLL.

With the functions implemented, now it's time to complete the DLL project by actually exporting the functions so they are visible and accessible outside DLLFIRST.DLL.

Exporting String Functions

To reiterate, you must list the functions you want to make available to client applications inside an **exports** clause of your library source code file.

First, enter the following code (select **View | Project Source** from Delphi's menu, if necessary, to return the main library file):

```
library DLLFirst;

uses
  WinTypes,
  XString in 'XSTRING.PAS';

exports
  FillStr,
  UpCaseFirstStr,
  LTrimStr,
  RTrimStr,
  StripStr;

begin
end.
```

This code shows you how to make the routines implemented in the XString unit visible to applications using DLLFIRST.DLL. The XString unit is listed in the **uses** clause of the library module to make its interface visible inside the library file so you can reference the names of the subroutines you want to export.

Because our five string-handling routines are listed in the **exports** clause, their names will be available so that applications can use the library's services. Make sure you recompile the DLLFirst project so the executable DLL reflects all your most recent changes.

Looking Ahead

Now that we've successfully implemented a DLL with Delphi, we'll next turn to the application side of the dynamic-linking equation.

The key to accessing externally implemented subroutines (e.g. a DLL function) is to create **import** units. Any application that wants to use the DLL's services must have the appropriate **import** reference in its **uses** clause. The WinProcs unit, for example, imports the subroutines defined in three Windows DLLs that comprise the Windows API: USER, KERNEL, and GDI. To use subroutines implemented in these DLLs, you must include WinProcs in the **uses** clause of your application.

In Part III of this series, we'll create a corresponding **import** unit for the custom DLLFirst library. We'll also discuss the **external** directive, interface with DLLFirst, import subroutines by ordinal number, and build a sample application. See you then. Δ

This article was adapted from material for Teach Yourself Delphi in 21 Days *[SAMS, 1995], by Andrew Wozniewicz.*

Andrew J. Wozniewicz is president and founder of Optimax Development Corporation (http://www.webcom.com/~optimax), a Chicago-based consultancy specializing in Delphi and Windows custom application development, object-oriented analysis, and design. He has been a consultant since 1987, developing primarily in Pascal, C, and C++. A speaker at international conferences, and an early and vocal advocate of component-based development, he has contributed articles to major computer industry publications. Andrew can be contacted on CompuServe at 75020,3617 and on the Internet at optimax@optidevl.com.

Begin Listing Five — The XString Unit

```
unit XString;
{ A collection of string-handling routines to comple-
ment
  those from the SysUtils unit }

interface
function FillStr(C : Char; N : Byte): string; export;
{ Returns a string with N characters of value C }
function UpCaseFirstStr(const s: string): string; export;
{ Capitalizes the first letter of every word }
function LTrimStr(const S: string) : string; export;
{ Trims the leading blanks }
function RTrimStr(const S: string) : string; export;
{ Trims the trailing blanks }
function StripStr(const s: string): string; export;
{ Strips all blanks }

implementation

function FillStr(C : Char; N : Byte): string;
{ Returns a string with N characters of value C }
begin
  FillChar(Result[1],N,C);
  Result[0] := Chr(N);
end;

function UpCaseFirstStr(const s: string): string;
{ Capitalizes the first letter of every word }
var
  Index: Byte;
  First: Boolean;
begin
  Result        := S;
  First         := True;
  for Index := 1 to Length(s) do
    begin
      if First then
        Result[Index] := UpCase(Result[Index]);
      if Result[Index] = ' ' then
        First := True
      else
        First := False;
    end;
end;

function LTrimStr(const S: string): string;
{ Trims the leading blanks }
var
  Index, MaxIndex: Integer;
begin
  Index    := 1;
  MaxIndex := Length(S);
  while (Index <= MaxIndex) and (S[Index] = ' ') do
    Inc(Index);
  Result := Copy(S,Index,MaxIndex-Index+1);
end;

function RTrimStr(const S: string) : string;
{ Trims the trailing blanks }
var
  Index: Integer;
begin
  Index := Length(S);
  while (Index > 0) and (S[Index] = ' ') do
    Dec(Index);
  Result := Copy(S,1,Index)
end;

function StripStr(const s: string): string;
{ Strips all blanks }
var
  Index : Integer;
begin
  Result := '';
  for Index := 1 to Length(S) do
    if S[Index] <> ' ' then
      Result := Result + S[Index];
end;

end.
```

End Listing Five

By Walker Lipscomb

Who Owes Whom?

Creating an Attractive and Intuitive User Interface

Have you ever spent a relaxing vacation at the beach with several friends, only to have the idyllic experience spoiled at the end by a heated argument over *who owes whom*?

This article provides a software solution to this problem. Along the way, the sample program illustrates several examples of Delphi components and features — what they are, and how they can be used to construct a user interface that is both attractive and intuitive to use. As Marshal McLuhan wrote, "The Medium is the Message," and in this case, the medium is a program, which by its nature, will be used only occasionally. However, this program will be indispensable on those occasions.

The Scenario

Four families of varying sizes rent a large beach house for a week. One couple pays the US$800 deposit, and another pays the remaining US$2600 due on arrival. One family picks up US$125 worth of drinks at the discount liquor store. During the week, each family buys groceries several times (and keeps the receipts).

As the week progresses, other friends and relatives show up to stay a few days. On Thursday, some of the group charters a fishing trip and others rent a ski boat. At the end of the week, there is a minor argument about whether to count the kids at "full fare." The family with the most kids points out that the kids slept four to a room, and did not have any of the drinks (or so they think). The family without children notes that *they* weren't the ones who left the refrigerator door open overnight with US$150 worth of shrimp in it (or made that mess on the carpet). A compromise is reached: children are to be counted at 50 percent for rent and 75 percent for food. Now — *who owes whom*?

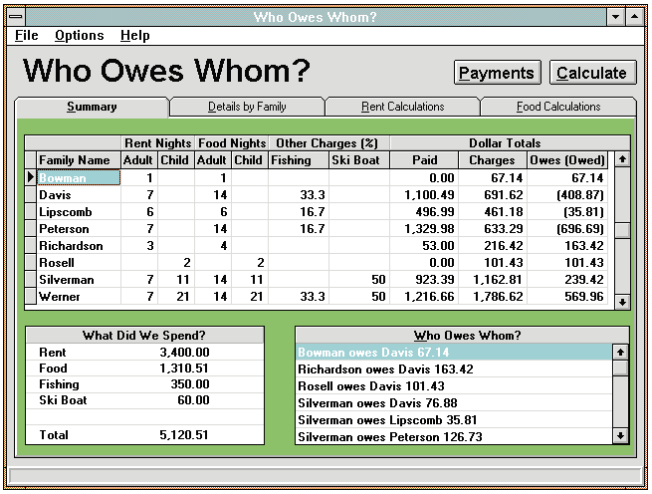


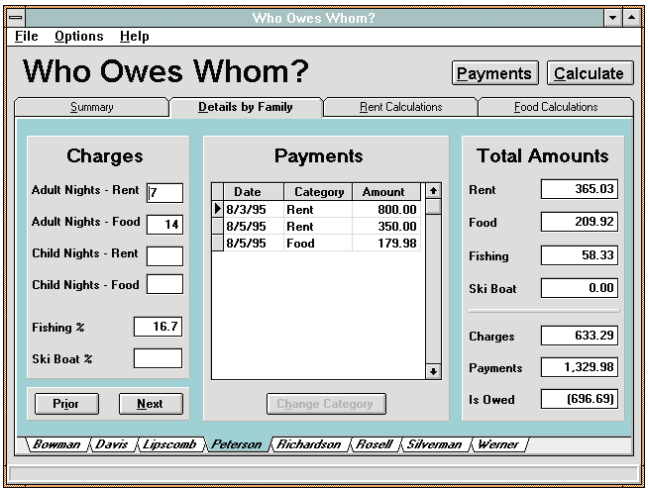
Figure 1: The Summary page.

Who Owes Whom (WOW) is a Delphi program designed to simplify the job of allocating shared expenses and determining who should pay whom, and how much. Information is arranged on four notebook pages. The Summary page (see Figure 1) is the primary mechanism for entering families and charges. Use of the Details by Family page (see Figure 2), the Rent Calculations page (see Figure 3), and the Food Calculations page is not required, but they will help in the inevitable and unenviable task of explaining to people why they owe so much. You move to each page by pressing the appropriate notebook tab with the mouse, or by holding down Alt and pressing the underlined letter.

Operation

Follow these steps to enter a simple scenario (Rent and Food only). Operational details are provided in the usual Windows Help file:

- If the Summary Page is not displayed, press Alt S to display it.



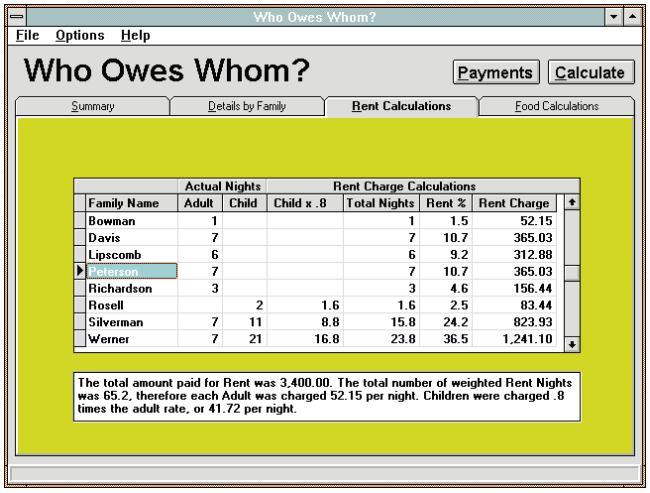


Figure 2 (Top): The Details by Family page.
Figure 3 (Bottom): The third notebook page is for calculating rent.

- Enter the first family's name in the **Family Name** column at the left of the grid.
- Under **Rent Nights/Adult**, enter the number of adults multiplied by the number of nights they should be charged rent (e.g. 2 adults for 7 nights equals 14 nights).
- Similarly, enter the appropriate information into **Rent Nights/Child**, **Food Nights/Adult**, and **Food Nights/Child**.
- Using the ↓ to move to each subsequent row, enter the remaining families.
- Press the **Payments** button (or Alt P) to display the Payments form (see Figure 4).
- For each payment, use the drop-down edit boxes to select the appropriate family who made the payment, and the correct category (Rent or Food). Enter the amount and press **OK** or Enter.
- Press the **Cancel** button or Esc to return to the Summary page.
- Note that a status box at the bottom of the screen suggests re-calculation. Press the **Calculate** button (or Alt C or F9) to calculate who owes whom.

Figure 4: Calculating food expenses per family.

Design Considerations

By its nature, WOW is not likely to be used on a huge database or by multiple simultaneous users. Therefore, little consideration was given to efficiency or multi-user access. Due to Delphi's inherent speed, WOW does run quite nicely with up to a couple of hundred families entered — surely larger than any conceivable beach party — even if you invited the Kennedys.

Since this program will probably be running on a laptop at the beach or in the mountains, printing capability was not deemed a requirement. (It's bad enough that you're carrying your laptop on vacation, much less a printer!) On the other hand, WOW will probably be run only once or twice a year, and without a manual handy. Thus, the screens should provide sufficient visual clues for easy operation. This also makes having adequate context-sensitive help a requirement. Since laptop pointing devices are often less than optimal, it must be easy to perform all operations from the keyboard.

Now, we'll discuss the implementation of WOW in Delphi, and how that implementation satisfies the design goals.

Implementation

TTabbedNotebook. Delphi offers myriad visual components "out of the box" and a virtually unlimited number of add-in VBX controls are available. Many additional Delphi components are already on the market, and the number is growing. The trick is to choose the right mix of components that will maximize usability.

The TabbedNotebook component (*TTabbedNotebook*) is the primary "motif" for WOW. TabbedNotebooks offer the neophyte user a familiar real-world visual metaphor, and can immediately show the user what major application areas are available — in this case the Summary, Details by Family, Rent, and Food calculations. A TabbedNotebook effectively provides sub-forms that can be used to change the type of information shown in an area of the main screen. In the design phase, the TabbedNotebook will be typically sized to occupy a large portion of the form — a quarter to a half or more. Then, to create the multiple sub-forms, you add two or more Page names to the *Pages* property of the component.

As many other visual components (e.g. labels, grids, edit boxes, etc.) as desired can be placed on each page. At run time, when the user or the program changes the notebook page, the components on the selected page are displayed and the components on the other pages are hidden.

Note that while "page" is the correct programming terminology, the Help file uses the term "sheets." If the user thinks in terms of pages, he or she will naturally expect to move from one to the other by pressing PgUp and PgDn. (Wouldn't you?) The Windows standard use for these keys is to "scroll grid" to a new set of records, and we happen to need that functionality. Therefore, WOW provides the F6 / Shift F6 keys to rotate among Notebook "sheets," and accelerator keys

```
procedure TMainForm.DepartNotebookPage;
begin
  case TabbedNotebook1.PageIndex of
    0: begin
         CurrentFamilyName:=
           FamilyTableFamilyName.AsString;
         if FamilyTable.State<>dsBrowse then
           FamilyTable.Post;
       end;
    1: begin
         CurrentFamilyName:=
           FamilyTable2FamilyName.AsString;
         if FamilyTable2.State<>dsBrowse then
           FamilyTable2.Post;
         if PaidTable.State<>dsBrowse then
           PaidTable.Post;
       end;
    2: begin
         CurrentFamilyName:=
           RentTableFamilyName.AsString;
         if RentTable.State<>dsBrowse then
           RentTable.Post;
       end;
    3: begin
         CurrentFamilyName:=
           FoodTableFamilyName.AsString;
         if FoodTable.State<>dsBrowse then
           FoodTable.Post;
       end;
  end;
end;
```

Figure 5: The *DepartNotebookPage* procedure is called by the *OnChange* event.

to select a page directly. Clicking the appropriate tab with the mouse works as well, of course.

Two *TTabbedNotebook* events are particularly important. The *OnChange* event occurs before leaving a page, and the *OnClick* event occurs after arriving on the new page. In WOW, each page displays data from the Family table, but each references a different *TTable* component. The *OnChange* event calls the procedure *DepartNotebookPage* (see Figure 5) which stores the Family Name from the "old" page, and posts any outstanding edits. The *OnClick* event calls the *ArriveNotebookPage* procedure that synchronizes the table on the "new" page to the same family (see Figure 6).

TTabSet. On the Details By Family page, you will see a different kind of tab, the TabSet. Unlike the *TTabbedNoteBook*, the *TTabSet* has no "built-in" functionality other than changing the highlighted tab. You must write code to affect any other controls. The TabSet could be used to change any value, for example the color of another component, or to modify the field's value in a record. WOW uses TabSet for perhaps its most obvious use — changing position within a table.

When the table referenced contains a small number of records, the TabSet is great because the user can immediately see the available choices. For larger data, however, TabSet would be a poor choice. Delphi provides scroll keys if all choices cannot be seen, but scrolling through more than two page-widths of tabs would be inconvenient to say the least.

TLabel. Labels? Why write about labels in *Delphi Informant*? Labels are the Rodney Dangerfield of components — they

```
procedure TMainForm.ArriveNotebookPage;
begin
  case TabbedNotebook1.PageIndex of
    0: begin
         SyncToFamily(FamilyTable,CurrentFamilyName);
         FamilyGrid.SetFocus;
       end;
    1: begin
         LoadFamilyNames(FamilyTabSet.Tabs);
         FamilyTabSet.TabIndex:=
           FamilyTabSet.Tabs.IndexOf(CurrentFamilyName);
         AdultNightsRentBox.setfocus;
       end;
    2: begin
         SyncToFamily(RentTable,CurrentFamilyName);
         RentGrid.SetFocus;
       end;
    3: begin
         SyncToFamily(FoodTable,CurrentFamilyName);
         FoodGrid.SetFocus;
       end
  end;
end;
```

Figure 6: The *OnClick* event works with the *ArriveNotebookPage* procedure to synchronize each new page to the same specified family.

get no respect. And, actually they are smarter than they seem. They provide accelerator keys to set the focus to another control. These accelerator keys are indicated by putting an ampersand (&) in the Label's *Caption*. Herein lies the problem: Sometimes you may not want an ampersand in the *Caption* to create an accelerator — you want it to be a plain ol' ampersand. To display ampersands normally (and not as accelerators) in a Label component, simply set its *ShowAccelChar* property to *False*. Or if that is unworkable, simply enter && to let Delphi know you simply want the ampersand character.

TPopup. The Details by Family page provides a method of changing payments once they have been entered. Direct entry works fine for the date and amount, but to change a payment category by typing it in, the user would have to remember the exact spelling and punctuation of the desired category.

A PopupMenu component provides an easy method for entering the correct category name. Since PopupMenus are normally accessed by right-clicking, a keyboard alternative is required to meet our design goals. By providing the **Change Category** button, we create the Alt T accelerator key and document it at the same time. Similarly the **Prior** and **Next** buttons are included primarily to provide a keyboard method of accessing the details of other families.

When the PopupMenu is initiated with a right-click, it appears next to the mouse pointer. When you display it using the *Popup* method, you must include the screen coordinates where the menu should appear. On the other hand, the menu should appear near the **Payments** grid on the form, and the user may move the form around on the screen. The *ClientToScreen* method (see Figure 7) provides the needed translation.

```
procedure TMainForm.ChangeCategoryButtonClick(
  Sender: TObject);
var
  PopupPoint: TPoint;
begin
  PopupPoint :=
    { Get SCREEN coordinates }
    FamilyDetailPanel.ClientToScreen(
      Point (PaymentsPanel.Left+trunc(
      { 2/3 accross panel }
      PaymentsPanel.Width*0.66),
      { Mid-way down }
      PaymentsPanel.Top+trunc(PaymentsPanel.Height/2)));
  CategoryPopup.Popup(PopupPoint.X,PopupPoint.Y);
end;
```

Figure 7: The *ClientToScreen* method.

Conclusion

Designing a pleasing and functional user interface is a matter of picking the right mix of components for the job and customizing those components as needed. Delphi's extensive set of user-interface components and infinite capacity for expansion can provide any component required. Delphi's strong object orientation and well-organized programming environment makes customization of these components easy, and its inherent speed makes the resulting program nimble and efficient. Δ

The Who Owes Whom project is available on the Delphi Informant Works CD located in INFORM\96\APR\DI9604WL.

Walker Lipscomb is an independent developer in the Washington, DC area, working with both commercial and governmental organizations. Mr Lipscomb can be reached at (703) 524-9232, or on CompuServe at 74132,2225.

Delphi 1.0 / Object Pascal / Windows API / Perseus VCL

By Karl Thompson

A Walk on the Wild Side

Getting at Windows Secrets Using Windows API Functions and Perseus

Delphi takes Windows programming to a higher plateau, for the most part because it hides the complexity of Windows programming. With Delphi's component architecture, the Object Pascal developer can easily create an application by dropping components on a form, setting the initial values of some properties, and adding code to selected event handlers. Delphi also makes pointers obsolete for the great majority of applications — no small feat for a sophisticated optimizing compiler. Undoubtedly however, there will be times when some coders need to dig in and get into the guts of Windows.

One of Delphi's strengths is that it allows programmers easy access to Windows API functions and data structures. The API function calls are encapsulated in various Delphi units. (For more details, click on Delphi's **Help** menu and select **Windows API**.)

This article will explore some of the API functions and data structures of the ToolHelp unit. These API functions enable you to query Windows about available system resources and how those resources are being used. The sample Walker project (see Figure 1) demonstrates how to use the ToolHelp unit to get information about the currently loaded modules, running tasks, registered Windows classes, and how memory is being allocated on the Windows global heap. Walker will also do a memory dump of any global memory block. Both a hex and ASCII dump can be done for any block up to 64K. Walker will also provide additional resource statistics including:

- available GDI and user heap,
- total available memory and largest available contiguous memory block,
- the number of free items and least-recently used (LRU) items,
- and the number of virtual pages and free virtual pages.

Figure 1: Walker displays information about the currently loaded modules, running tasks, the registered Windows classes, and how memory is being allocated on the Windows global heap.

Note that the Walker program is definitely a work in progress. Time permitting, we'll add additional functionality and robustness in future articles. And naturally, you are free to take on the task yourself.

Some Terms Defined

Before covering the program in detail, let's review some Windows terminology that often causes confusion:

- A *module handle* (or more specifically, an entry in the module table) contains information that can be shared by multiple instances of an application.
- A *task*, or *entry* in the task database (TDB), contains information specific to each execution (instance) of an application.

The module table will contain information about program resources (bitmaps, etc.) while the TDB will contain such information as the current working directory for a specific task — thus two executions of a program can have different, current directories.

To understand the next term, it's important to first understand a basic Windows concept. Anyone who has programmed in Windows for long realizes that nearly every visible interface item is a window. From the perspective of a Delphi program, a window is (obviously) a window. What might not be so obvious, however, is that a scrollbar, button, listbox, checkbox — and about every other display item — is a window too, as far as Windows itself is concerned.

Why is this important? Windows must be able to send messages to any of these and other window types (classes). Each window is created based upon a *window class*, and the class tells Windows what procedure is going to process the message.

These are enough terms to get started. Other terms are covered in the online help included with Walker. For now, let's move on and take a look at Walker's organization.

Walker's Structure

Walker's files are organized so that if you need to use the functionality of the ToolHelp unit, you'll be able to easily locate the appropriate initialization code. There is one global unit, uGlobal.PAS, that initializes all the pointers and ToolHelp data structures used by Walker. Normally you would not want to write a function just to initialize a pointer, particularly if it's initialized only once. In this case, however, it was done to clarify how the pointers are initialized.

Even if you don't understand pointers, but *do* understand that the functions in *uGlobal* return properly initialized pointers, then you can easily use the ToolHelp unit. Other files include: the main program module, *Walker*; the Task Walker, *uTasks*; the Class Walker, *uClasses*; the Module Walker, *uModule*; and the Global Heap Walker, *uGlobalW*. The unit that handles the display of the global data dump is *wGlobalD*. [The source code is too extensive to list in its entirety. However, it is available on diskette and for download. See end of article for details.]

Module, Task, and Class Walkers

I use the term *Walkers* because each one iterates, or "walks," over the appropriate linked list of data structures maintained by Windows. Let's examine the Class Walker first (see Figure 2). Note that we call the routines that list the classes, *Tasks*, *Modules,* etc. The list of the class structures is maintained in the user segment's default local heap.

When you run the Class Walker, you may see classes with names such as `#34XXX`. These are integer atoms. Borland's WinSight program will provide a translation of these atoms to string names. Atoms are discussed in the book, *Undocumented Windows*, by Andrew Schulman, et al. (see end of article for details). For now, suffice it to say that an atom is a 16-bit word and that the Windows atom manager provides a means for translating an ASCII string into an atom.

The real work for this unit, and the others, occurs in the *OnShow* event handler.

```
procedure TClassesFrm.FormShow(Sender: TObject);
var
  OK: Bool;
begin
  pOurClassEntry := InitClassEntry(pOurClassEntry);
  OK := ClassFirst(pOurClassEntry);
  while OK do begin
    ListBox1.Items.Add(Format(' %-11s  %5.4x  %-s',
      [GetModuleNameFromHandle(pOurClassEntry^.hInst),
      pOurClassEntry^.hInst,
      pOurClassEntry^.szClassName]));
    OK := ClassNext(pOurClassEntry);
  end;
  GroupBox1.Caption   := ' Module Name  hInst Class Name ';
  ClassesFrm.Caption := ' ' +
    IntToStr(ListBox1.Items.Count) + ' Classes Found ';
end;
```

First, a pointer called *pOurClassEntry* is initialized with a call to *InitClassEntry* (located in *uGlobal*). The *pOurClassEntry* is of type *pClassEntry* and is a pointer to a *TClassEntry* structure. *pOurClassEntry* must be initialized before the Class walk can begin.

The call to *InitClassEntry* not only initializes a pointer but also allocates memory on the heap to hold one *TClassEntry* structure. One of the requirements for using many of the ToolHelp data structures is that the structure's *dwSize* field must be initialized to the size of the structure. The Walker *InitXXXX* functions also handle this requirement. Notice that all we have to do is call the Object Pascal function *SizeOf* while passing it the data type of the Windows API structure.

If Walker were to be a truly robust application (i.e. suitable for commercial distribution), the code should handle the run-time error that's generated if there isn't enough memory available to allocate the structure. For more information, see Delphi's help topic "RTL Exceptions." For now however, we'll overlook this shortcoming so that we can proceed with learning about these Windows data structures.

Start Walking

Basically, all the Walkers work in the same manner — once a pointer is initialized, the walk begins. To begin the walk, we must get the first structure in the list. In this case, the call to *ClassFirst* accomplishes this. If the call is successful, we enter a **while** loop that first adds an entry to a *TListBox* and then tries to again initialize the *Class* data structure (pointed to by *pOurClassEntry*) with a call to *ClassNext*.

When the call to *ClassNext* returns *False*, the walk is over. The title row above the list box is the value of the *Caption* property of the *TGroupBox* that is the parent of the *TListBox*. Finally, the memory allocated for the *TClassEntry* structure is

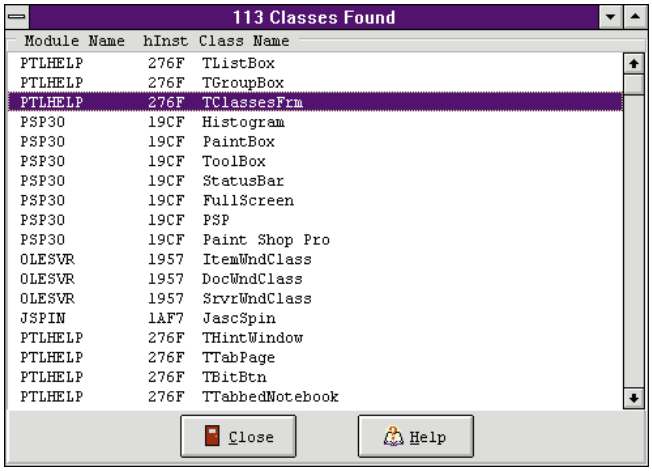


Figure 2: The Class Walker.

freed in the *OnDestroy* event handler (the *FormDestroy* method). This is accomplished with the call to *Destroy*.

When you review the code in the other units, you'll see that the Class, Task (see Figure 3), and Module (see Figure 4) Walkers work the same way. The Global Walker, however, is different.

The Global Walker

The Global Walker (see Figure 5) presents us with some interesting challenges. As Charles Petzold points out in his book, *Programming Windows 3.1*, "global memory ranges from the spot where MS-DOS first loads Windows to the top of available memory." This means that Windows is itself running in its global memory (as the Global Walker demonstrates).

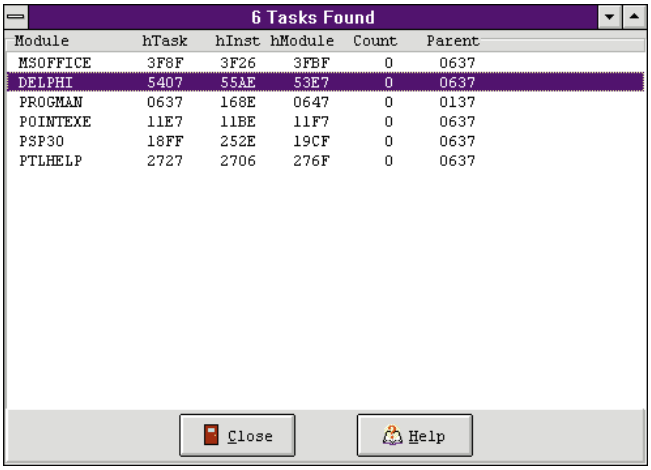
Windows manages blocks of memory on the global heap known as *segments* or *items*. As the name implies, the Global Walker will walk the global heap to document these segments. The difficulty comes from trying to walk the heap without modifying it.

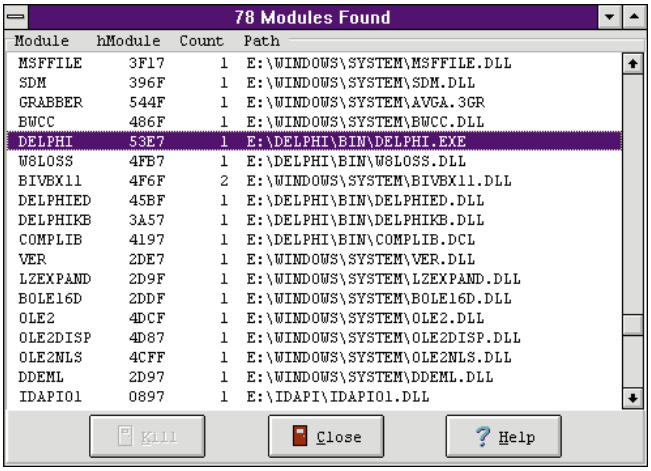
As you may know, Delphi's *TListBox* allocates memory to store strings on the global heap. Thus, each call to *TListBox.Add* allocates some additional memory from the global heap. This means as we try to walk the global heap and add a formatted string to a listbox (as we did with the other Walkers), the heap itself is changing. Particularly since segments can be discardable, moveable, etc., allocating memory during the walk renders the results of a global heap walk useless.

Perseus to the Rescue

This problem was overcome by using Perseus, a virtual listbox shareware component written by Sebastian Modersohn. [A trial version of Perseus is included in the .ZIP file that contains the Walker program. See end of article for details.] Perseus' installation instructions are in its help file. Essentially, however, it (VListBox.DCU) is installed just as any other component.

Perseus is not a descendent of Delphi's *TListBox*, and therefore does not use a *TString* variable to manage the strings it displays. In fact, it's the programmer's responsibility to supply the strings to Perseus, whose job in turn, is to efficiently display those strings.





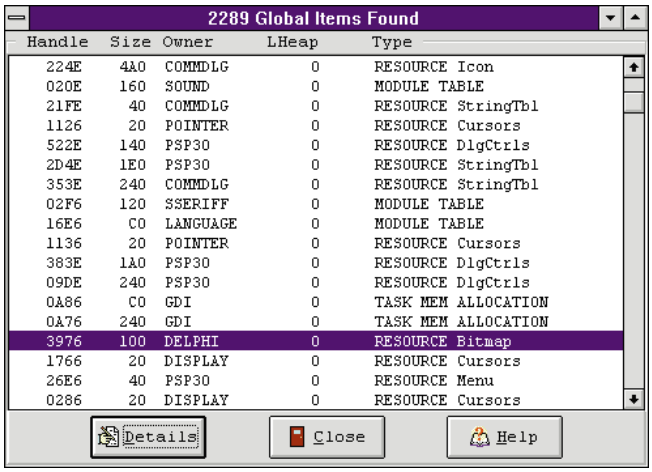


Figure 3 (Top): The Task Walker. **Figure 4 (Middle):** The Module Walker. **Figure 5 (Bottom):** The Global Walker.

And it does so superbly. While developing Walker on a computer with 20MB of memory, I constantly exhausted available memory trying to use a *TListBox* component while working on the Global Walker module. This was never a problem when using Perseus, as it can handle up to 2,147,483,647 strings.

Picking the Right Event

All memory allocation is completed in the *FormShow* method (see Listing Six beginning on page 45). Since we want to walk the global heap, we want to grab the items on the heap at the last possible moment after all allocation has been completed. Since Delphi allocates memory for forms and child controls in

the *FormCreate* method, we should perform the Global walk in the *FormShow* method after the form that is used to show the contents of the heap has itself been allocated.

The sequence of the allocation is important. First, a *TStringList* is created and initialized. *GlobalItemString* is used to manage the strings that Perseus displays. Memory is allocated by calling *TStringList*'s *Add* method in a **for** loop that iterates *MaxGlobalItems* times.

Next, a *TGlobalEntry* structure and a pointer are initialized, and finally, a *TGlobalInfo* structure and pointer are initialized. (Remember, this memory must also be deallocated. This is done in the *FormDestroy* method.) The *TGlobalInfo* structure is initialized last because we'll need a count of all items on the global heap. Since the strings being managed by *TStringList* are also allocated in global memory, the items they occupy must also be included in the count.

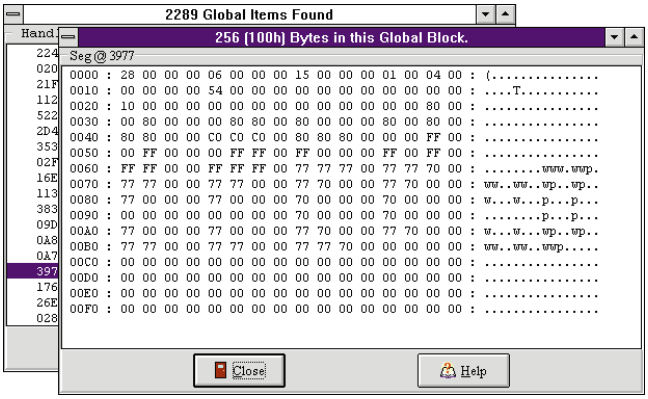
Now that all the memory needed for this module has been allocated, the global walk can begin. We use the same **while** loop construct to walk the global heap as we have used in the other Walkers. As the loop is executing, one string is being formatted with information about each item on the heap. Notice that the *Add* method is not being called. Since memory was previously allocated, we can use a simple assignment statement. As before, when a call to *GlobalNext* fails, the walk is complete.

Finally, Perseus must display the strings formatted during the walk and initialize a couple of title strings. For Perseus to handle the display of the strings managed by the *GlobalItemString* object, Perseus' *RowCount* property must be set to the number of strings it will need to display. Since that number is equal to the number of items on the global heap, we use the value in *TGlobalInfo.wcItems*. With Walker, the *TStringList.Count* property could not have been used to provide this number. Remember we added *MaxGlobalItems* to the string list, which of course, is not the same as the number of actual items on the heap.

When Perseus needs a new string to display, the *OnGetItem* event is fired and the string is assigned in the *VListBox1GetItem* event handler method.

```
procedure TGlobalW.VListBox1GetItem(Sender: TVListBox;
  Index: Longint; Col: Integer; var Strg: OpenString;
  State: TOwnerDrawState; var Font: TFont;
  var BGColor: TColor);
begin
  Strg := GlobalItemString.Strings[Index];
end;
```

When browsing the list of items, the user can double-click on an item to see a memory dump of the segment (see Figure 6). Again, because of the efficiency with which Perseus displays strings, it's used to display the memory dump. The size of any memory dump is limited to 64K. Note that an item on the global heap can be larger than 64K, but a segment of such a size would be quite rare.



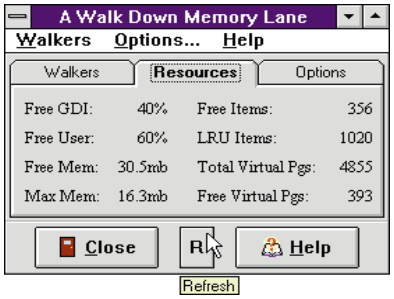


Figure 6 (Top): Double-click an item to view the segment's memory dump.
Figure 7 (Left): The Resources page of the *TTabbedNotebook*.

The Resources Page

The Resources page (see Figure 7) of the *TTabbedNotebook* component shows the currently available or used amount of some of the more interesting Windows resources. Each time the user clicks on the Resources tab, the values are refreshed.

```
procedure TMainForm.TabbedNotebook1Click(Sender: TObject);
begin
  if TabbedNotebook1.PageIndex = 1 then
    GetResources;
end;
```

If a user is running Walker with the Resources page displayed and wants to see updated statistics, a simple click on the **R** button does the job. Since it only makes sense to refresh the values when the Resources page is displayed, the **R** button is only enabled while this page has focus. The **R** button's enabled state is handled by the *OnChange* event handler.

The code that actually gets the values is located in the *GetResources* method in the *uTlHelp* unit (see Listing Seven beginning on page 45). Two new ToolHelp data structures are initialized here: *TMemManInfo* and *TSysHeapInfo*. *TSysHeapInfo* provides the free percentage of GDI and User resources, while *TMemManInfo* provides the number of virtual pages and free virtual pages. *TGlobalInfo*, which was introduced in the Global Walker module, provides information about the total items and free items on the global heap. The "Free Mem" and "Max Mem" values are returned by run-time Object Pascal functions. (See Delphi online help for more details.)

Odds and Ends

A common practice among component vendors is to ship completely working copies of their components for trial purposes. Such trial components give the developer the opportunity to fully evaluate the component. There is just one "catch"

— trial components will only work while Delphi is running. This is true for Perseus, the virtual listbox used by Walker.

So how does the component know that Delphi is running? There is no *IsDelphiRunning* function. At least one way to find out if Delphi is loaded is to use the ToolHelp function, *ModuleFindName*. Walker uses this technique in the *MainForm.FormShow* method (see Figure 8). This is because if a form that is using the trial version of Perseus is loaded when Delphi is not running, Perseus will display a polite message saying it cannot be used outside Delphi. It will then close the application.

```
procedure TMainForm.FormShow(Sender: TObject);
begin
  GDI1.Enabled         := False;
  User1.Enabled        := False;
  SpeedButton4.Enabled := False;
  BitBtn10.HelpContext := 0;
  { If Delphi isn't running, disable Global Button
    and menu item.
    Note: Global Walk depends upon Perseus, a virtual
    list box component that must be registered
    separately to run when the Delphi IDE is not running. }
  pOurModuleEntry := InitModuleEntry(pOurModuleEntry);
  if (ModuleFindName(pOurModuleEntry,'DELPHI') = 0) then
    begin
      SpeedButton7.Enabled := False;
      Global1.Enabled      := False;
    end;
  Dispose(pOurModuleEntry);
end;
```

Figure 8: The *MainForm.FormShow* method detects if Delphi is running.

In the case of Walker, we did not want to prevent anyone from using the program when Delphi was not loaded. In the *FormShow* method, the code checks if Delphi is running. If not, then the **Global** walk button and **Global** menu item are grayed. This prevents the user from selecting **Global Walk** and then receiving a message saying that Perseus is not registered, and having Walker close down. Naturally, you can use the same technique for any trial components that you want to ship.

If you happen to select **Options | Project** from the Delphi menu, you'll notice that the only form that is automatically created is *MainForm*. All other forms are created as Walker needs them. This saves from having to allocate memory for the forms before they are used. It also allows Walker to load faster. To see how a form is created/allocated manually as needed, look at the code in each of the *XXXXWalker* methods in *uTlHelp*.

The technique is very simple. Call *Application.Create*, passing it the type of form and the variable instance of the form. Call a procedure that will eventually call *ShowModal* (or *Show* if the form will be disposed differently than what the examples illustrate). When *ShowModal* returns, free the memory allocated.

Why Walker, and Where to Now?

Walker came about because of an interest in learning more about how a developing application was working in Windows. Hopefully, if you spend some time with Walker, you'll gain some insight into the way that Windows works as well.

There is still a lot that can be done with this program. For one thing, the User and GDI Walkers need to be written. A hex dump routine could be written for all the local heaps, and a tool could be developed for monitoring stack space. Beyond that, Walker could be made application specific. That is, Walker would monitor just the resources used by an application (that is part of the purpose of the **Options** page). With further development, Walker could show the developer resources that haven't been properly disposed after the application terminates. If anyone makes any of these enhancements, please send them my way!

References and Acknowledgements

A great deal of the material this article is based upon came from the book, *Undocumented Windows* by Andrew Schulman, David Maxey, and Matt Pietrek [Addison-Wesley, 1992]. I would like to thank Yon Barrenechea who recommended this text to me, and Peter Below for seconding the recommendation. They were right on the money with their advice. Additional information came from Charles Petzold's *Programming Windows 3.1* [Microsoft Press, 1992].

I've known about these books for a long time, but I never bothered to read them because I'm a Pascal/Delphi programmer at heart (I've not written a line of C in 11 years) and I was put off by their C orientation. I figured that I would just buy the Delphi or Pascal books that interested me. This was a mistake.

Unfortunately, the topic of Windows architecture is not covered in-depth in any Pascal-specific book that I've encountered. So if you have even a passing interest in how Windows works "under the hood," pick up a copy of one or both of these books. And don't worry about the C code if you don't know C. It's not that tough to follow. Δ

The demonstration Walker program and trial version of Perseus are available on the Delphi Informant Works CD located in INFORM\96\APR\DI9604KT.

Karl Thompson is an independent Paradox and Delphi developer serving clients from New York City to Philadelphia. He has been writing applications using some vintage of Borland's Pascal since 1984. He can be reached at (800) 242-9192, or on the Internet at 72366.306@compuserve.com.

Begin Listing Six — TGlobalW.FormShow

```
procedure TGlobalW.FormShow(Sender: TObject);
var
  OK: Bool;
  Counter: Longint;
begin
  { Show hourglass mouse pointer }
  Screen.Cursor := crHourglass;
  try
    GlobalItemString := TStringList.Create;
    { Allocate the strings before! walking Global Heap }
    for Counter := 0 to MaxGlobalItems do
      GlobalItemString.Add('');
    pOurGlobalEntry := InitGlobalEntry(pOurGlobalEntry);
    pOurGlobalInfo  := IniTGlobalInfo(pOurGlobalInfo);
    GlobalInfo(pOurGlobalInfo);
    OK := GlobalFirst(pOurGlobalEntry, Global_All);
    if not (OK) then
      MessageDlg('Error getting Global information...',
                 mtWarning, [mbOK], 0);
    Counter := 0;
    while OK do begin
      GlobalItemString.Strings[Counter] :=
        Format('    %4.4x  %5x  %-14s  %-5d  %-14s',
        [pOurGlobalEntry^.hBlock,
        pOurGlobalEntry^.dwBlockSize,
        GetModuleNameFromHandle
         (pOurGlobalEntry^.hOwner),
        ord(pOurGlobalEntry^.wHeapPresent),
        GetGlobalBlockType(pOurGlobalEntry^.wType,
        pOurGlobalEntry^.wData)]);
      inc(Counter);
      OK := GlobalNext(pOurGlobalEntry, Global_All);
    end;
  finally
    Screen.Cursor := crDefault; { Show default pointer }
  end;
  VListBox1.RowCount  := pOurGlobalInfo^.wcItems-1;
  GroupBox1.Caption   :=
    ' Handle  Size Owner     LHeap     Type ';
  GlobalW.Caption     := ' ' +
    IntToStr(VListBox1.RowCount)+' Global Items Found ';
end;
```

End Listing Six

Begin Listing Seven — TMainForm.GetResources

```
procedure TMainForm.GetResources;
var
  OK: Bool;
begin
  Label7.Caption  := format('%4.1nmb',[MemAvail/OneMb]);
  Label8.Caption  := format('%4.1nmb',[MaxAvail/OneMb]);
  pOurSysHeapInfo := InitSysHeapInfo(pOurSysHeapInfo);
  pOurMemManInfo  := InitMemManInfo(pOurMemManInfo);
  pOurGlobalInfo  := InitGlobalInfo(pOurGlobalInfo);
  if SystemHeapInfo(pOurSysHeapInfo) then begin
    Label5.Caption :=
      Format('%d%%',[pOurSysHeapInfo^.wGDIFreePercent]);
    Label6.Caption :=
     Format('%d%%',[pOurSysHeapInfo^.wUserFreePercent]);
  end
  else begin
    Label5.Caption := 'N/A';
    Label6.Caption := 'N/A';
  end;
  if MemManInfo(pOurMemManInfo) then begin
    Label14.Caption :=
      Format('%d',[pOurMemManInfo^.dwTotalPages]);
    Label16.Caption :=
      Format('%d',[pOurMemManInfo^.dwFreePages]);
  end
  else begin
    Label14.Caption := 'N/A';
    Label16.Caption := 'N/A';
  end;
  if GlobalInfo(pOurGlobalInfo) then begin
    Label12.Caption :=
      Format('%d',[pOurGlobalInfo^.wcItemsFree]);
    Label13.Caption :=
      Format('%d',[pOurGlobalInfo^.wcItemsLRU]);
  end
  else begin
    Label12.Caption := 'N/A';
    Label13.Caption := 'N/A';
  end;
  Dispose(pOurGlobalInfo);
  Dispose(pOurMemManInfo);
  Dispose(pOurSysHeapInfo);
end;
```

End Listing Seven

New & Used

By Douglas Horn

DFL's Light Lib Series

Two Delphi VCLs That Know What to Do with Data

As Delphi's popularity among developers skyrockets, many third-party software developers are taking notice. These developers of programming tools are now releasing VCL controls designed for, and written in, Delphi. Such native VCLs are generally more compatible and easier to implement in Delphi applications than VBX and DLL controls that are designed to be used by any number of development platforms.

Two new Delphi-specific controls are now available from DFL Software, Inc. These controls, Light Lib Images and Light Lib Business, promise powerful, nearly effortless implementation of data-aware image handling and business graphing, respectively. Although each has a few minor drawbacks, the components provide interesting and needed solutions to problems that many Delphi developers may be experiencing.

DFL's tools also support other programming environments, such as C/C++, Visual Basic, and CA-Visual Objects. However, unlike many other products that try to cover all the bases, DFL has actually provided language-specific components and class libraries for each environment, and the standard and professional editions include the source code to the VCL components. The professional editions are reviewed here.

Installation

While the people at DFL tried to make the Light Lib components as easy as possible to install and use, the installation is in fact rather puzzling, especially for users who want to use both of the separately sold components. No printed documentation comes with the components. This does not hurt the product because the online help files are very strong, and they're tied directly into Delphi's help system. Although this review is very much in favor of this trend in documentation, its Achilles' heel presents itself when the installation routine becomes unclear and there is no documentation to reference.

Whether installing the Business or Images package, when the installation routine begins, it unexpectedly informs the user that, "The following Light Lib components will be installed: Light Lib Objects, Light Lib Images, Light Lib Business, Light Lib Multimedia."

The program then asks for the serial number and hard disk location for installation, then asks what versions to install: Delphi, Microsoft Foundation Classes, Microsoft Visual Basic, and/or CA-Visual Objects.

The peculiarity begins after the installation, when not only the Business (or Images) tool is installed, as the user intended, but also an evaluation version of the other VCL. If the user tries to install the other VCL into the same directory in order to delete the evaluation version and save some disk space, he or she is informed that the existing directory will be deleted. Actually, the program will only overwrite the evaluation versions, but without some sort of documentation, the user is left wondering.

Apart from this quirk, the installation routine gets most of the job done (albeit slowly). After the necessary files are installed, the user is instructed how to merge the Light Lib and Delphi help files, and how to add the new controls to Delphi's Component Palette. The Light Lib tools are added to their own page of the Component Palette, but using standard Delphi commands, they can be moved anywhere the user chooses.

Light Lib Images

The Images VCL consists of a single component, *TImageWindow*. It contains an area for displaying images, and a toolbar that allows users to access and manipulate those images. As Delphi developers would expect, this toolbar can be configured or concealed altogether. The component's integrated scrollbars and progress gauge can also be hidden.

TImageWindow allows a wide range of image operations, including: scanning or opening existing images; zooming; scaling; rotating and flipping; dithering; and printing. The component also handles conversion between any of its supported image formats. The Standard Edition supports BMP, PCX, TGA, TIF, and PNG. (The public domain PNG format replaces the widespread but proprietary GIF format.) The Professional Edition adds GIF, JPG, and Light Lib's own BLOb format.

Using *TImageWindow* as a standard image viewing component is useful, but not particularly ground-breaking. Much more interesting is the fact that *TImageWindow* is a data-aware component. Using its *DataSource* and *DataField* properties, developers can attach *TImageWindow* to a database just as they would Delphi's native DBImage component. However, by using the *TImageWindow*, developers can access images from any of the supported formats, or from TWAIN (Technology Without An Interesting Name — no kidding!) compliant scanners. This is an excellent feature for databases that require a large number of images (see Figure 1).

The traditional method of entering images into a *TDBImage* component uses the Windows Clipboard to cut and paste them into a database. The *TImageWindow* component eliminates this requirement and makes image-intensive databases easier to use and implement in Delphi.

It does have a shortcoming, however, in that images must be saved to a Binary type field rather than a Graphic type field (in Paradox for Windows tables). Also, to save an image to the database, the *TImageWindow's* **Save** button must be pressed. The more Delphi-esque way of handling this would be to automatically update the image when the record is posted. In fact, this can be added programmatically by trapping the *Post* procedure and calling *TImageWindow's Save* method. However, the documentation does not mention this possibility, so developers are on their own.

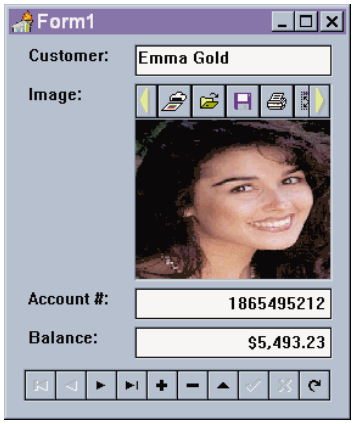


Figure 1: Light Lib Images' *TImageWindow* component allows Delphi developers to scan and import images into a database.

This flaw aside, the VCL's native BLOb (Binary Large Object) handling is very useful for storing large numbers of images. *TImageWindow* offers three choices for BLOb compression: none, speed-optimized, and size-optimized.

Unfortunately, because of the proprietary format and the need to use Binary fields rather than Graphic fields, the Light Lib BLObs are not compatible with existing image BLObs in Paradox tables. *TImageWindow* can write and retrieve its own BLObs from Paradox tables, but cannot read those that have been input using *TDBImage* components. The reverse is also true of *TDBImage* components and Light Lib BLOb images. Although DFL says this will be fixed in future releases, users of existing databases must re-enter all their database images if they use the Light Lib component rather than Delphi's.

Light Lib Images includes a few other features that add to its usability. The first is a stripping algorithm that divides images into 32K memory blocks before processing. The result is faster loading and display than is available in many other image viewers. Even large JPEG images load quickly. The component also takes advantage of what it calls "intelligent dithering" to improve image quality. This allows users to save hi-color images as 16-color images with surprisingly little image degradation.

Although Light Lib Images is a Delphi-specific VCL, in some areas it strays from some of Delphi's established norms, to the chagrin of Delphi users. For example, the *ShowToolbar* property would best be implemented as a nested property, much as *TDBNavigator's VisibleButtons* property is. Instead, double-clicking the *ShowToolbar* property calls a dialog box with numerous check boxes (see Figure 2).

While functional, this variance from the norm does stand out. More troublesome is the fact that the *ImageName* and *ImagePath* properties used to set the image (when not in data-aware mode) do not call Open File dialog boxes. Rather than browsing for the desired file, the developer must manually add the path and file name.

Light Lib Business

As with Light Lib Images, Light Lib Business consists of a single data-aware component. The component, *TGraphWindow*, has one claim to fame — it turns live data into live graphs (see Figure 3). For example, developers can connect a table or query to a *DataSource* with a DBNavigator component. Adding a *TGraphWindow* component will then allow them to scroll through their database, viewing graphs of their data in real time.

TGraphWindow has many strengths, but a few weaknesses as well. The graphing is surprisingly quick — fast enough to render complex graphs in the time it takes to do a basic screen redraw. In addition, the graphs are fully end-user configurable by default. When a user clicks on a section of the graph, the component is intelligent enough to determine what section has been selected (vertical or horizontal axes background, bar, etc.), and open the dialog box controlling that section.

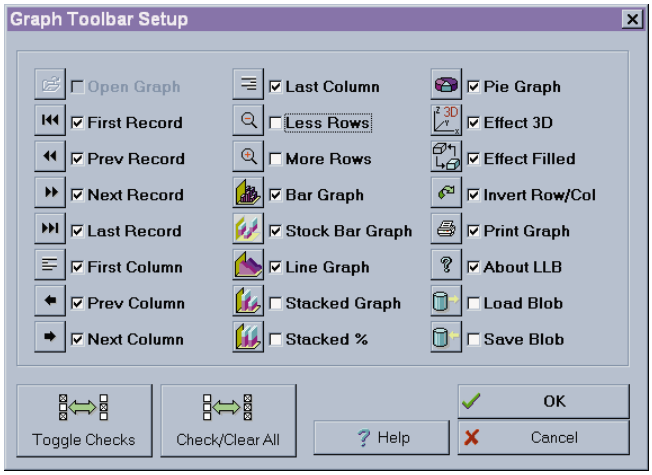


Figure 2: The Light Lib VCLs behave in a non-Delphi-esque way. This dialog box, for example, would be more appropriate as a nested property on the Object Inspector.

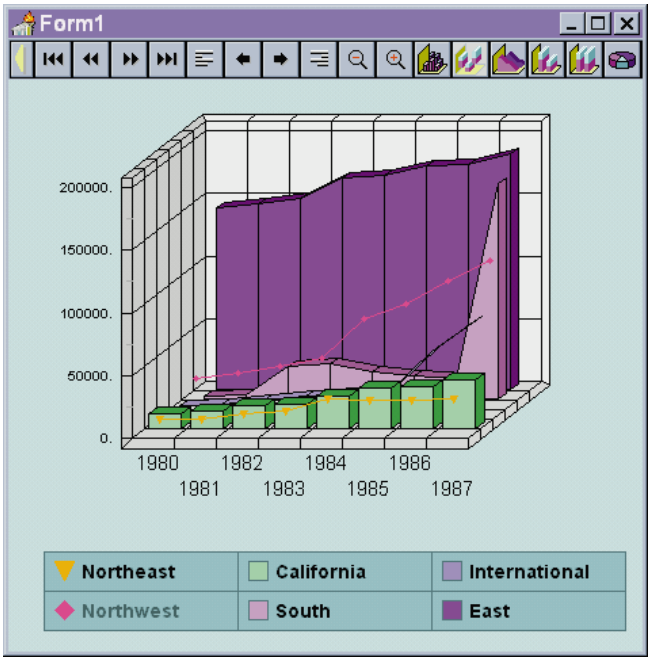


Figure 3: Light Lib Business (Professional Edition) allows 2D and 3D line and bar graphs to be combined, but would be much improved with more graph types.

The *TGraphWindow* component includes a toolbar, which the developer can configure or hide. Using the *DialogAccess* property, the six graph-configuration dialog boxes can be set as either accessible or inaccessible. Other nice features include auto-scaling and auto-sizing of graphs, the ability to add and change graph legends, and the ability to modify graph scale.

One excellent feature of *TGraphWindow* is its ability to save graphs as BLObs that can then be saved to disk or to a database. These BLObs encapsulate the attributes of the graph so that future users can see the data exactly as it was first presented, without having to reformat the graph.

Light Lib Business shares some minor flaws with Light Lib Images, but its major drawback is the number of graph types it offers. The standard version includes bar, line, pie, and stacked graphs, which may be either two- or three-dimensional, filled or non filled. The Professional Edition allows mixed line and bar graphs, and stock tracking graphs. Still, this variety is rather sparse compared to many packages on the market.

The graphs supplied do cover a wide range of situations, but simply are not numerous enough to satisfy the sophisticated graphing needs of many end-users.

The Rest

One real advantage the Light Lib tools offer is that they include complete source code to the VCL components. In point of fact, these components are actually very complex Pascal wrappers that interface with Light Lib's object-oriented DLL, the source code for which is not provided.

However, the source code that developers really need is available. The primary PAS file for Light Lib Business, for example, contains over 6,500 lines of code. As the source code is fairly well documented, many of these lines are program comments.

DFL's technical support line is a toll call, but is not likely to leave customers stuck in voice mail, and the technical support personnel do a good job of solving problems. Technical support is also available via fax, CompuServe, DFL's BBS, and the World Wide Web. DFL issues online upgrades free of charge to users. (According to one DFL representative, many of the concerns addressed in this review will be resolved in upcoming online releases.)

Light Lib Images could be a real godsend to anyone developing image-intensive databases in Delphi. The product's data-aware and BLOb compression capabilities make it a natural choice for such applications. As a graphing tool, Light Lib Business is short on graph types. However, its ability to tie in seamlessly with Delphi's *DataSource* property make it a strong contender in the Delphi graphics world. Δ

INFORMANT FACT FILE

Light Lib Images is a data-aware image scanning, viewing, and conversion VCL component. This sterling tool is only slightly tarnished by its rather non-Delphi way of doing things. It's a vast improvement over Delphi's own *TDBImage* component, making image-intensive database development a real possibility in Delphi.

Price: VCL only (no source code) US$149; Standard Edition US$295; Professional Edition US$495.

Light Lib Business offers data-aware graphing in Delphi database applications. This Delphi VCL offers fast graphing, and can be connected to tables and queries via the *DataSource* property. The current version could be improved with a wider variety of graph types.

Price: VCL only (no source code) US$149; Standard Edition US$249; Professional Edition US$449.

DFL Software, Inc.
55 Eglinton Avenue East, Suite 208
Toronto, Ontario, Canada M4P 1G8

Voice: (416) 487-2660
Fax: (416) 487-3656
BBS: (416) 487-4041
E-Mail: Internet: tech@dfl.com
CompuServe: GO DFLSW
Web Site: http://www.dfl.com

Douglas Horn is a free lance writer and computer consultant in Seattle, WA. He specializes in multilingual applications, particularly those using Japanese and other Asian languages. He can be reached via e-mail at horn@halcyon.com.

table of CONTENTS

It Doesn't Cure the Common Cold

Don't believe everything you read. Not only will the Web not cure the common cold, it is not about to replace most client/server applications either. Listening to some industry prognosticators, however, you would think that we might as well trade in our Pentiums for Internet terminals. The argument is that you will soon forget about running applications locally, opting instead for programs that arrive courtesy of the Web. I beg to differ. A maturing PC marketplace is moving towards "enabling" technologies, not "replacing" technologies. As with the local area network in the 1980s, the Web should be considered an extension of your local operating system, not as its successor. After all, the whole PC revolution brought decentralization to the computer world; I find it a rather cheeky notion that we would ever tolerate a return to a centralized framework.

What then does the Web mean to you as a Delphi developer? If it will not replace your desktop, is it important to pay attention to? Definitely. The Web holds an opportunity to reach domains that mainstream desktop and client/server applications would never work in. Therefore, rather than making your Delphi programming skills obsolete, the Web offers an occasion for you to use those talents in new areas. Savvy developers will embrace the Web, carefully deciding when and how to use it as a development environment.

Intoxicating Connectivity. While its global connectivity is intoxicating, the Web is suited for a relatively narrow genre of applications. But within these camps, no other technology gives you more "bang for the buck." Perhaps the most striking case is an application that needs to reach an external and diverse user base. Some examples include:

Disseminating information. FedEx illustrates the Web at its finest with its Package Tracking system (http://www.-fedex.com/track_it.html). No matter who you are, if you have access to the Web, you can find out the status of a package you sent.

Gathering information. Head hunters and placement firms provide an example of how the Web can be used to allow potential employees or contractors to submit a resume or employment application via the Web.

Providing a direct sales line to consumers. Web applications provide companies a direct sales line to its customers. For intangible goods like software, the Web serves as a store front as well as the delivery channel. For tangible goods like Godiva chocolates (http://www.godiva.com), a Web application offers a convenient alternative to visiting a retail store or ordering by phone.

Providing an alternative advertising medium. Over the past six months, you have probably begun to see ads pop up everywhere on the Web. Just how effective the Web is as an advertising medium remains to be determined, but I suspect it will do well at reaching targeted audiences. While Web advertising is often static HTML pages, there will be an ever increasing number of advertising-oriented applications on the way.

A second area in which the Web excels as an application environment is in *Intranets*. Intranets use essentially the same plumbing as the Internet, but they are private — restricted to an organization or groups within a company. These systems are ideal for Fortune 500 or multinational companies that want to improve corporate communications within the face of geographical and multi-platform challenges. Not only can Intranets serve as a corporate mouth organ, they also provide a fertile environment for groupware applications (à la Lotus Notes).

Without the Web as a backdrop, you really have no practical way of using Delphi to develop solutions in these realms.

Strings Attached. The use of the Web as an application environment is convincing when you need to appeal to a wide or diverse audience, but it is not without a price. Quite literally, Web applications have strings attached to them: they rely on a synchronous means of communication in order to work. Without a connection, you cannot run the application. However, not only are most business PCs still not connected to the Web, but there are many contexts in which that scenario just does not work. For example, an application for a remote salesperson's notebook is likely to involve distributed data. But without mature and affordable wireless technology, the best solution remains asynchronous replication with the home office, not the Web.

Web applications also cannot match the sophisticated user interfaces that Delphi and other visual development tools can provide. Any HTML-based application has a meager supply of UI controls available to them. Even future enhancements to the HTML language will not anytime soon catch up to a Windows 95 Treeview or Listview control. Perhaps as Java visual tools like Borland's Latte or Symantec's Espresso begin to perk, you will begin to see better UIs available across the Web. But even if they can provide the presentation depth and richness that Windows users are now demanding, such a solution would come at a cost: the greater the complexity of a Java application, the longer it will take to download onto the client.

Further, Web applications are ultimately dependent on the speed of the transmission line. A direct or ISDN connection to the Internet may be becoming ubiquitous for Fortune 100 companies, but most small- to medium-size businesses continue to use 14.4 or 28.8 baud access to the Web. And even if you have a blazingly fast connection, heavy network traffic can plague application performance.

Therefore, when designing Web applications, you are often forced to think in terms of simplicity, compromising *usability* and *high performance* to maximize your use of limited bandwidth.

Shed the hype. Look at the Web realistically. The Web offers newfound power, but it is not our panacea. Just like other revolutionary technology advances that came before it, the Web augments, not replaces, our development environment.

Richard Wagner

Richard Wagner is the Chief Technical Officer of Acadia Software in Boston, MA. He welcomes your comments at rwagner@acadians.com.