



Cover Art By: Tom McKeith

# Delphi Reports

*ReportSmith, QuickReport, and Beyond*

## ON THE COVER



**7 Leveraging ReportSmith: Part I** — Mark Ostroff  
ReportSmith is a robust reporting tool that allows you to build powerful, complex reports featuring graphs, crosstabs, drill-downs, live data pivoting, etc. Mr Ostroff begins a two-part series that discusses when and how to use ReportSmith in your Delphi development.



**15 QuickReport** — Cary Jensen, Ph.D.  
If you don't need ReportSmith's full-fledged capabilities in your application, QuickReport is probably a better tool for your reporting needs. Dr Jensen builds a basic QuickReport report, covers report previewing and printing, data module usage, and examines bands.



**20 ReportSmith Secrets** — Jeff Sims  
Does ReportSmith have you puzzled? Mr Sims shares straight talk regarding some common misperceptions about Borland's misunderstood reporting tool. He also reveals a number of secrets to help you put the power of ReportSmith to work in your Delphi application.

## FEATURES



**24 Informant Spotlight** — Gregory Lee  
What's old is new again. Although roughly 25 years old, today's technologies make the Internet the hot new thing. To get you connected, Mr Lee introduces DelphiFinger, a Finger utility that just happens to demonstrate adding Winsock capability to Delphi applications.



**32 OP Tech** — Keith Wood  
Continuing his *TLabelEffect* discussion from our July '96 *DI*, Mr Wood returns to teach another lesson in extensibility. Step-by-step, you'll learn to build prototype and finished property editors for the *EffectStyle* and *ColourScheme* properties, as well as give *TLabelEffect* its own Help file.



**39 From the Palette** — Karl Thompson  
Just drop a *DirectoryListBox*, *FileListBox*, *DriveComboBox*, or *FilterComboBox* on your Delphi form for navigation capabilities. But if you want more functionality, Mr Thompson shows you how to enhance these controls as he builds on his *Table Documentor* tool from the March '96 *DI*.

## REVIEWS



**44 Borland Pascal 7 Insider** — Book Review by James Callan

**44 Mastering Delphi 2** — Book Review by Larry Clark



**46 Ace Reporter** — Product Review by Bill Todd

## DEPARTMENTS

**2 Delphi Tools**

**5 Newslite**

**50 File | New** by Richard Wagner

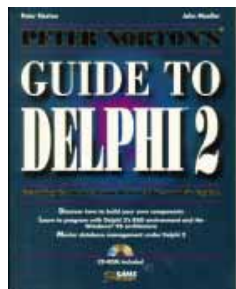


New Products and Solutions



## New Delphi Book

Peter Norton's Guide to Delphi 2  
Peter Norton, John Mueller  
SAMS



ISBN: 0-672-30898-3  
Price: US\$49.99  
(788 pages, CD-ROM)  
Phone: (800) 428-5331

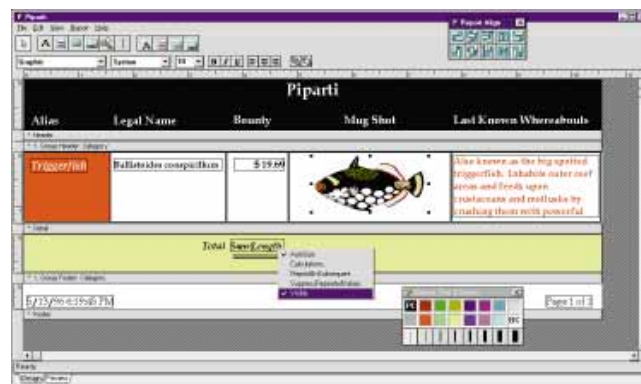
## Digital Metaphors Releases Piparti 1.0 Report VCL for Delphi

Digital Metaphors of Dallas, TX announced the release of *Piparti 1.0*, a report VCL for Delphi. Written in Object Pascal, Piparti is a VCL component that includes an interactive Report Designer.

When installed, Piparti adds two components to Delphi's Component Palette, *TppReport* and *TppViewer*. Reports are created by dropping a *TppReport* component onto a form and double-clicking. This invokes the Report Designer, which can be used to visually design and preview reports.

Working like Delphi's Form Designer, the Report Designer supports multi-selection, cut-and-paste operations, undelete, control sizing, and positioning. Additionally, several Report Designer tools augment the standard Delphi Object Inspector, and provide access to report control properties. These include the Format Toolbar, Alignment Palette, Color Palette, and various Speed Menus.

Piparti also allows devel-



opers to preview reports at design time through a built-in print preview form. After a report has been rendered, the previewer can see the design layout changes without running the report again. The previewer provides standard Zoom and Print functions, and has a caching scheme that allows users to view or print any page in the report. The Print Preview feature can be customized for the runtime environment. *TppViewer* is a print preview component that assists developers in customizing reports.

Additionally, Piparti supports data access through Delphi's standard *TDataSource* component.

Report controls include: ppLabel, ppMemo, ppCalc, ppImage, ppShape, ppLine, ppDBText, ppDBMemo, ppDBCALC, and ppDBImage. Control Groups and Title/Summary bands are also supported. Piparti 1.0 is available in both 16- and 32-bit versions. A demonstration copy is available at Digital Metaphor's Web site.

**Price:** US\$199 (US\$149 introductory price), plus shipping and handling

**Contact:** Digital Metaphors, Infomart, 1950 Stemmons, Suite 5001, Dallas, TX 75207

**Phone:** (214) 746-4703

**Fax:** (214) 746-4704

**E-Mail:** Internet: info@digital-metaphors.com

**Web Site:** http://www.digital-metaphors.com

## Eagle Research Links Delphi 2 to Access with JETset

Eagle Research, Inc. of San Francisco, CA has begun shipping *JETset*, a tool that connects Delphi 2 to Access through Microsoft's JET database engine. JETset supports Access database features such as transactions, stored queries, reporting, macros, database repair, verification, and replication. JETset also supports JET's use of external data sources such as Excel and Lotus spreadsheets, Btrieve and FoxPro

databases, and text files.

With JETset, programmers can deploy Delphi applications that use Access without having to ship the Borland Database Engine.

According to Eagle Research, JETset's design-time interface can be used to make any Delphi component. JETset for Delphi requires Microsoft JET version 3.0. JET 3.0 and its documentation are included in current versions of Microsoft Visual C++,

Visual Basic, and Access.

**Price:** US\$150, includes a 30-day money-back guarantee, and no distribution royalties. JETset is also bundled at no charge with the professional edition of Eagle's VB2D Visual Basic to Delphi translator (US\$450).

**Contact:** Eagle Research, Inc., 360 Ritch Street, Ste. 300, San Francisco, CA 94107

**Phone:** (415) 495-3136

**Fax:** (415) 495-3638

**E-Mail:** Internet: sales@xeaglex.com

**Web Site:** http://www.xeaglex.com

New Products  
and Solutions



## New Delphi Book

### Delphi Database Development

Ted Blue, et al.  
M&T Books



ISBN: 1-55851-469-4  
Price: US\$44.95  
(968 pages, CD-ROM)  
Phone: (800) 488-5233

## New Import/Export Tool

Nutshell Software of Westlake Village, CA has released *IM/EX ASCII*, an import/export tool for Delphi. *IM/EX ASCII* exports ASCII data to and from Paradox or dBASE with one line of code. It is designed to support 82 percent of ASCII files, including variable length formats with comma or custom delimiters.

In addition, *IM/EX ASCII* supports use of Memo or large text fields. It can also double the speed of ASCII operations by exporting the results of a SQL query.

To reduce run-time errors, *IM/EX ASCII* has a new Flex-Field feature that allows an application to handle ASCII files with an unknown or changing struc-

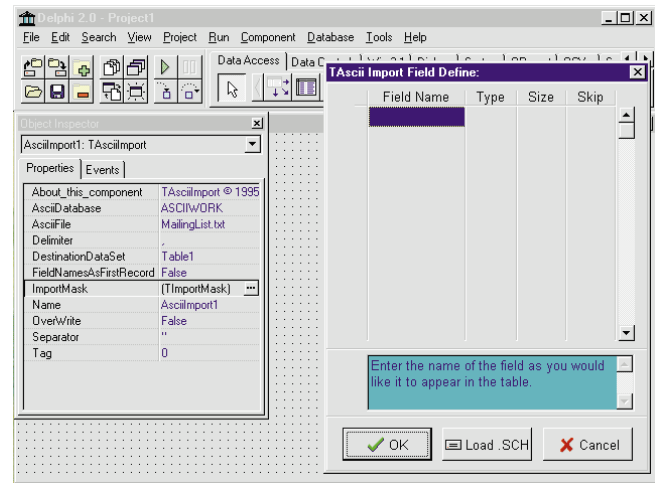
## Tamarack Announces Version 2.0 of TtaDBMRO for Delphi

Tamarack Associates of Palo Alto, CA is shipping version 2.0 of *TtaDBMRO*, a data aware control for Delphi.

Usable in 16-and 32-bit environments, *TtaDBMRO* is a DBCtrlGrid-like control that allows the developer to display data aware controls in a scrollable manner. It supports all Borland field data aware controls, and is compatible with Delphi 1 and Delphi 2.

*TtaDBMRO* also allows developers to use data aware controls with different DataSources, provides several ways to customize the appearance of records, and supports titles.

*TtaDBMRO* is compatible with InfoPower 1.2, Orpheus 2.0, *TDBLookupComboPlus* 4.1, and *TDBComboBoxPlus* 2.1. It operates under Windows 3.11, Windows 95, and Windows NT 3.51. The new version also adds support



ture. The components also allow users to skip fields, customizing the conversion process. Additionally, *IM/EX ASCII* offers the ability to quit a job in progress via the Cancel property.

A demonstration version is available at Nutshell's Web site.

**Price:** US\$99.95 (available in both 16- and 32-bit versions)

**Contact:** Nutshell Software, 31316 Via Colinas, Ste. 106, Westlake Village, CA 91362

**Phone:** (818) 865-7945

**Fax:** (818) 865-0012

**E-Mail:** Internet: support@nutshell-software.com

**Web Site:** http://www.nutshellsoftware.com

for *DBLookupComboBox* and *DBLookupListBox*.

Trial and demonstration versions of *TtaDBMRO* can be found on the Delphi and Bdelphi forums on CompuServe, file names: MRO.ZIP and MRO-DEMO.ZIP.

**Price:** *TtaDBMRO*, US\$25 (including free version 2.0 updates and support via e-mail). *TtaDBMRO* is available

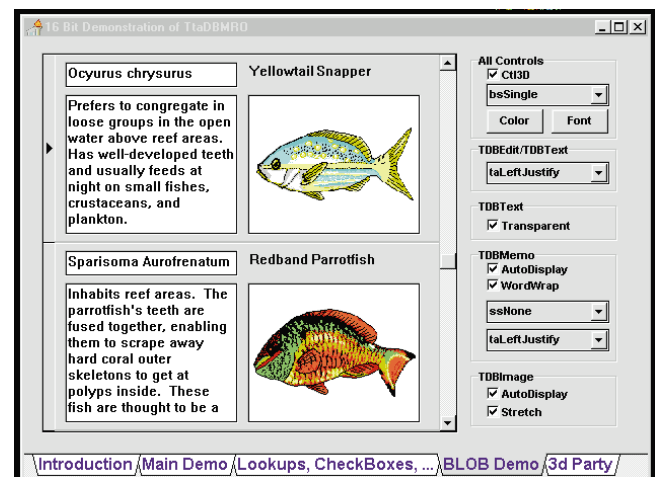
via CompuServe shareware registration, ID 8213 for US\$29.95. The *TtaDBMRO* 1.x upgrade is available from Tamarack Associates for US\$15.00. (This upgrade offer is not available through CompuServe shareware registration.)

**Contact:** Tamarack Associates, 868 Lincoln Ave., Palo Alto, CA 94301

**Phone & Fax:** (415) 322-2827

**E-Mail:** CIS: 72365,46

**Web Site:** http://ourworld.compu-serve.com/homepages/deven







## New Delphi Book

### Delphi 2: A Developer's Guide

Bill Todd, Vince Kellen  
M&T Books



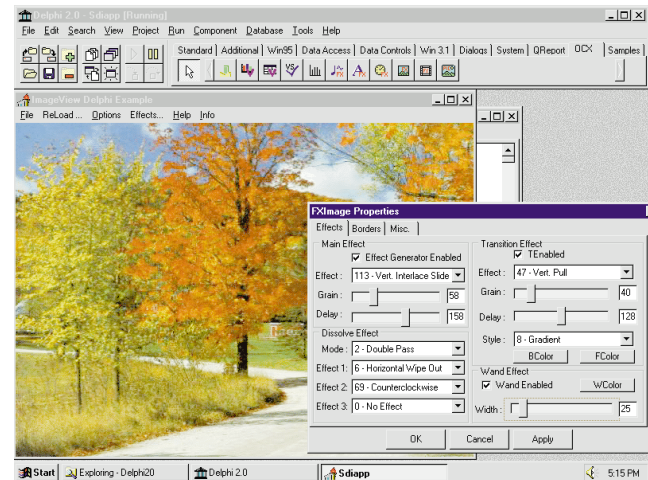
ISBN: 1-55851-476-7  
Price: US\$44.95  
(937 pages, CD-ROM)  
Phone: (800) 488-5233

## ImageFX Introduces FXTools 4.0 Professional Edition

ImageFX of Rochester, NY has released *FXTools 4.0 Professional Edition*, a royalty-free tool kit that adds imaging and multimedia special effects technology to Delphi 2 or any other environment that can host 32-bit ActiveX controls in Windows 95 or Windows NT. It also features 16-bit VBX controls for use with Delphi 1 under Windows 3.x.

FXTools 4.0 contains eight reusable ActiveX and VBX components that give developers control over the display of images, text, and shapes, as well as playback of sound and video. The image control gives developers support for most image file formats including .BMP, .DIB, .GIF, .JIF, .JPG, .PCX, .PNG, .RLE, .TGA, .TIF 5.0, .WMF, and .WPG.

Images can be rotated, cropped, resized, flipped, mirrored, dithered, and printed. The label controls have five background styles, including tiled images and color gradients. Additionally,



text can be rotated, animated, filled with an image or gradient, and displayed with a 3D style. The shape control can be used to create buttons that include text and images. The sound control supports MCI wave and MIDI audio, including 32-channel sound mixing using DirectSound. The video control supports .AVI, .MOV, and MPEG file formats. In addition, individual video frames can be displayed with special effects.

A demonstration version can be downloaded from ImageFX's Web site or CompuServe forum.

**Price:** US\$399

**Contact:** ImageFX, 3021 Brighton-Henrietta TL Road, Rochester, NY 14623-2749

**Phone:** (800) 229-8030 or (716) 272-8030

**Fax:** (716) 272-1873

**E-Mail:** Internet: imagefx@-imagefx.com, or CIS: 74431,3331

**CIS Forum:** GO IMAGEFX

**Web Site:** <http://www.imagefx.com>

## MKS Source Integrity 7.2 Released, Adds Web Functionality

Mortice Kern Systems of Waterloo, ON, Canada has begun shipping an update to its change management tool, *MKS Source Integrity*. In version 7.2, MKS Source Integrity includes a security and administration model, as well as integration with Web

servers, including Netscape Enterprise Server and Microsoft Internet Information Server, enabling intranet change management and distributed development over the World Wide Web.

MKS Source Integrity also includes: visual merge, visual differencing, event triggers, a configuration language, an automated building process, file promotion, NetWare-specific functionality, and integration into Delphi, Borland C++, PowerBuilder, Netscape Navigator Gold Premium Internet client software, Visual C++, Visual Basic, and Web servers.

MKS Source Integrity automates the change process in files and docu-

ments through a project-oriented release management system and parallel development capabilities. Version 7.2 allows development teams to fine-tune access permissions by user and user roles, and create user groups and SCM policies. It also features a new reporter with graphing and custom report capabilities.

**Price:** US\$499; or US\$149 for an upgrade.

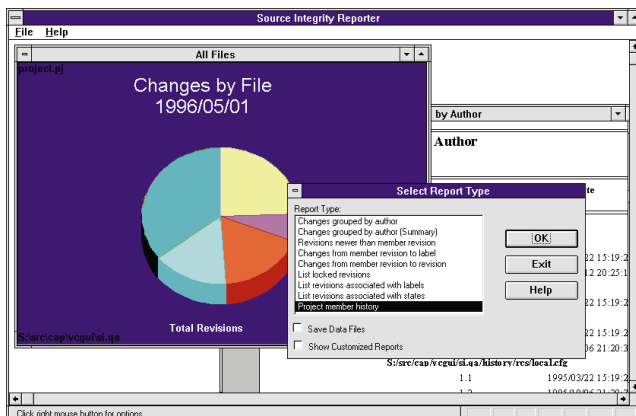
**Contact:** Mortice Kern Systems, 185 Columbia St. W., Waterloo, ON, Canada, N2L 5Z5

**Phone:** (800) 265-2797 or (519) 884-2251

**Fax:** (519) 884-8861

**E-Mail:** Internet: info@mks.com

**Web Site:** <http://www.mks.com>



August 1996



### Visual Components Licenses Technology to Borland

Visual Components, Inc. (VCI), a wholly-owned subsidiary of Sybase, Inc., has announced licensing arrangements and development alliances with Borland International, Netscape Communications Corp., and Powersoft. Borland has licensed OEM editions of the Formula One spreadsheet, the First Impression charting and business graphics component, and VisualSpeller, a 100,000-word, multi-language dictionary, for its Delphi 2, Paradox 7, and C++ 5.0 products.

For more information, visit VCI's Web site at <http://www.visualcomp.com> or (800) 884-8665.

## Borland Offers Rebate on Delphi through September

Scotts Valley, CA — Borland International Inc. has announced a US\$100 rebate to new owners of Delphi Desktop 2 through September 30. In addition, Borland will include a free copy of DeltaPoint Inc.'s QuickSite Web site manager with every purchase of Delphi 2, Borland C++ 5.0,

Paradox 7, and Visual dBASE Professional, a US\$99 value.

QuickSite is a Web site development component that allows users to create, publish, and manage Web sites without programming in HTML.

The estimated street price (before rebate) for a competitive upgrade of Delphi Desktop 2 is US\$199.95.

For more information, call Borland at (800) 932-9994.

## Delphi Informant Wins Computer Press Award

Elk Grove, CA — Informant Communications Group, Inc. announced *Delphi Informant* was named Runner Up in the Rookie of the Year category for the Computer Press Awards.

The awards featured 26 categories and over 100 winning submissions from publications such as *Computerworld*, *Infoworld*, *PC World*, *C|Net*, Ziff-Davis Press, and others.

The select judging panel included representatives from *PC/Computing*, ZD-Net, *MacUser*, *Hotwired*, *PC Games Magazine*, McGraw-Hill, and others.

In their remarks, the judges said "*Delphi Informant* earns reader loyalty by offering must-read tutorials and commentary in a clear, well-written format. A reader-friendly layout helps pinpoint key information; a mix of review and opinion round out a balanced package."

For more details, visit CPA's Web site at <http://www.conterra.com/webmagic/cpa-main.htm>.

## Borland Announces InterClient for InterBase

San Francisco CA — Borland International Inc. has announced InterClient for the InterBase cross-platform SQL database server. Written in the Java programming language, InterClient is platform independent and provides JDBC-compliant connectivity software for InterBase.

The Web server is used for storing and delivering Java applications while InterClient provides a client/server architecture, database access, and control over the end-user application interface. This is aimed at improving end-user response time, as well as reducing Web server and network traffic.

InterBase enabled with the Java-based InterClient

is targeted at IS managers using Web technologies for information distribution and application development. Organizations can also provide updated applications via the Web.

InterClient enables distributed transaction processing by both the Java client and database server; eliminates the need to install and maintain platform-specific client database connection and client libraries; and reduces overall Web server traffic on the network.

Pre-release versions of InterClient for Windows 95, Windows NT, and Solaris will be available for customer evaluations. For more information, visit Borland's Web site at <http://www.borland.com>.

## SQA & Borland Announce SQA Suite for Delphi Client/Server Suite 2

Woburn, MA — SQA, Inc. and Borland International Inc. announced a version of SQA Suite that supports Delphi Client/Server Suite 2 on the Windows NT and Windows 95 platforms.

SQA will make this support available in an update to SQA Suite 5, the 32-bit version of the automated testing solution that began shipping last May. The updated version of SQA Suite 5 will integrate with Delphi via an object-level API

provided by Borland.

Previously, Borland selected SQA, Inc. as a premier automated test tool vendor to participate in the worldwide launch of Delphi. With this new announcement, Borland has extended their Open Tools API to include the interfaces and information that SQA needs to bring its Object-Oriented Recording and Object Testing technologies to Delphi Client/Server Suite. This object-level inte-

gration will allow SQA Suite users to create position-independent test scripts for Delphi applications, as well as test the properties and data of objects in 32-bit Delphi applications, including ActiveX and OLE controls.

In addition, SQA Suite will provide integrated load, stress, and multi-user testing of Delphi applications in LAN and WAN environments.

*"SQA Suite for Delphi Client/Server Suite 2"*  
continued on page 6

August 1996



### Crystal Becomes Seagate Software

Crystal, a wholly-owned subsidiary of Seagate Technology, Inc. and vendor of Crystal Reports and Crystal Info, has changed its name to Seagate Software, Information Management Group (IMG). As part of Seagate Software, IMG will target the business intelligence market, expanding its query and reporting tool offerings. For more information, visit the Seagate Software IMG Web site at <http://www.img.seagate-software.com>, or call (800) 877-2340 or (604) 893-6392.

Washington, DC	Aug 12-15
New York/New Jersey	Aug 19-22
Chicago, IL	Aug 20-23
San Francisco, CA	Aug 26-29
Orlando, FL	Sep 10-13
Boston, MA	Sep 16-19
Houston, TX	Sep 23-26
Raleigh, NC	Sep 23-26
Minneapolis, MN	Sep 30-Oct 3
Atlanta, GA	Oct 8-11
Seattle, WA	Oct 8-11
Philadelphia, PA	Oct 14-17
Denver, CO	Oct 21-24
Los Angeles, CA	Oct 22-25
Dallas, TX	Oct 29-Nov 1
New York/New Jersey	Nov 4-7
Columbus, OH	Nov 11-14
Chicago, IL	Nov 18-21
Washington, DC	Nov 19-22
San Francisco, CA	Dec 3-6
Frankfurt, Germany	TBA
Amsterdam, the Netherlands	Oct 22-25
London, England	Oct 28-31

## Informant Communications Group Announces *Web Informant Magazine*

Elk Grove, CA — Informant Communications Group, Inc. has announced *Web Informant* magazine. Targeted at Web developers, *Web Informant* will contain technical how-to articles on Java, JavaScript, and Perl programming, as well as HTML, VRML, database application development, server administration, and firewalls and security.

The Premiere Issue of *Web Informant* will be available in September 1996 via paid subscription, or newsstand outlets across the United States and Canada. The magazine will be published monthly, and carry a cover

price of US\$5.95. The annual subscription rate for *Web Informant* is US\$39.95. The magazine will be available at Barnes & Noble, B Dalton Bookseller, Waldenbooks, Tower Books, Borders, CompUSA, Egghead, and Software Etc.

Informant Communications Group is also publishing a "Preview" issue of *Web Informant*. The Preview issue will have a smaller page-count than the regular magazine, but will provide a typical cross-section of technical articles a subscriber will see each month.

The Preview issue will be published in early August

1996, and distributed to subscribers of other Informant Communications Group publications.

*Web Informant* is the fourth title in a line of technical magazines from Informant Communications Group. The others are: *Oracle Informant*, *Delphi Informant*, and *Paradox Informant*.

### SQA Suite for Delphi Client/Server Suite 2 (cont.)

SQA expects to have field test versions of SQA Suite for Delphi available this summer. Final pricing and release date were unavailable at press time.

## Softbite Announces 1996 Delphi World Tour

Addison, IL — Borland International Inc., Softbite International, and Informant Communications Group have announced the 1996 Delphi World Tour. The four-day seminar will take place in both the US and Europe.

The first day of the Delphi World Tour is geared towards those new to Delphi 1 and 2. It covers essential techniques, tools, and tricks of Delphi application development. Day two provides a look at the techniques used to create working applications in Delphi 2.

Day three and four each offer an advanced Delphi seminar and review key features, including advanced database techniques, creating components, and using resources.

Delphi World Tour attendees will receive a free copy of Delphi 2 Developer version, a starter subscription (three issues) to *Delphi Informant*, documentation and diskette containing Delphi code, and

a copy of *Delphi Informant* at the event.

There is also a chance to meet with local third-party consultants, trainers, and book vendors during the Delphi Networking Lunch on day two.

Pricing for the US offering of the Delphi World Tour is US\$1,095. Those attending only two days of the event pay US\$595, or US\$295 to attend one day.

## Borland Supports the Java Beans Initiative

San Francisco, CA — Borland International Inc. has announced support for a component model standard, the Java Beans Initiative, proposed by JavaSoft for Sun Microsystems Inc.'s Java programming language.

As previously announced, Borland's Latte development tool will bring rapid application development, reuse, and scalable database access to Java developers.

In recent demonstrations of Latte, Borland has previewed a platform-indepen-

Discounts are available for three or more attending from the same company.

To receive a complete brochure via fax, call (708) 833-9122 (or 630-833-9122 after Aug. 3) and request document number 5. The brochure can also be requested from Softbite at (708) 833-0006 (or 630-833-0006 after Aug. 3). Softbite's Web site is located at <http://www.softbite.com>.

dent component model for Java, code-named BAJA. The demonstrations showed how the BAJA component model addresses platform-independent implementation, compound components, support for RAD, simple UI widgets to non-visual complex remote services, third-party component development, and component persistence.

In addition, Borland will detail its support for integrating COM and Open Doc component model standards into Java.





## ON THE COVER

Delphi 1 / Delphi 2 / ReportSmith 2.5 / ReportSmith 3.0



By *Mark Ostroff*

# Leveraging ReportSmith: Part I

## Advanced Delphi Reporting

**R**eportSmith is the Rodney Dangerfield of Delphi. And just as Mr Dangerfield is actually a very savvy performer, ReportSmith is an extremely powerful part of Delphi. ReportSmith can accomplish many reporting tasks that no other reporting tool can match. The key is in knowing how to leverage ReportSmith to its best advantage in your Delphi applications.

### Why Have Multiple Report Writers?

To complicate matters, Delphi 2 features two reporting tools: ReportSmith, and a set of native VCL components named QuickReport. It would seem to be redundant to have multiple ways of creating reports. There are good reasons, however, for including both ReportSmith and QuickReport in the current version of Delphi. The challenge is to know when to use each.

#### QuickReport: simple, fast reporting.

QuickReport is designed for applications where simple reporting is needed, and the reporting facility needs to be a part of the application's .EXE. QuickReport follows the Delphi development model very closely. It is, however, a design-time, developer-only solution. As such, QuickReport is best suited for applications where all reporting needs are limited to a specified set of reports, and those reports must be controlled from within your application.

This internal reporting model provides for easier distribution. It also supplies a faster report start-up, since the Windows startup code has already executed by the time your user runs a report. QuickReport's smaller overhead results in reports that generally run faster as well. Ultimately, applications that use QuickReport will have a total disk

space requirement that is significantly smaller than what would be needed for using the Delphi/ReportSmith combination. Total system resource requirements are also less when using QuickReport. And, of course, since QuickReport components are Delphi VCL components, you can sub-class them to add your own reporting functionality.

QuickReport does have a few limitations. Live data display during report design in QuickReport is restricted to using a report preview. This requirement to switch between design mode and preview mode makes the design process more complex. In addition, certain reporting needs can't be accomplished with QuickReport. For example, leveraging server resources for large data sets, creating stand-alone reports, and inclusion of crosstabs and data graphs are simply not within the capacity of QuickReport.

#### ReportSmith: robust, full-featured reporting.

ReportSmith, on the other hand, is the power choice. It is well suited for creating complex reports that require the inclusion of data graphs, multiple crosstabs, live data pivoting, drill-down selection criteria, or other ad hoc analysis capabilities. ReportSmith can also be used to build "smart" reports. A single report design can serve double duty. It can be run either within the context of a



Delphi program or as a separate stand-alone report. Thus, ReportSmith is the better choice when you have an application that needs to support both data entry/reporting users and report-only users with the same reports.

ReportSmith also offers a flexible method whereby end-users can design their own reports. These new reports can then be run from within your application. Since ReportSmith is designed for this kind of dual role, it has a more flexible design interface. Its WYSIWYG design mode, along with its live data display, makes report design much simpler than with QuickReport. In fact, although both QuickReport and ReportSmith can build mailing label reports, the live data display design interface of ReportSmith makes it a much better choice for the task.

The design of ReportSmith is tuned for client/server applications. Its power and performance are well suited to reporting on very large data sets. The use of ReportSmith also means that your total disk space requirements can be more efficient when you have multiple applications that reference the single installation copy of ReportSmith. The ability to deploy reports using either the ReportSmith Run-Time or a complete copy of ReportSmith means that you can split your user community with even more finely tuned control. You can support users that just need to run pre-defined reports, as well as people who need to run those reports and create their own. This kind of user-community tailoring is simply not possible with QuickReport.

**Which Do I Use?**

The key to maximizing the use of any tool is to know when to use it, and how to apply it properly. Some guidelines are given here.

**When to use QuickReport.** Are load-time performance and distribution size the overriding decision factors? If so, QuickReport is the right choice. With smaller data sets, QuickReport reports will also run faster than ReportSmith reports. Another important issue is whether you need to completely control the creation and execution of reports. Since QuickReport reports are compiled directly into your .EXE, the end-user has no way to modify the report, and no way to run it outside of your application. If you need to create a totally closed solution, QuickReport is the way to do it.

QuickReport's better coordination with the Delphi environment is also a consideration. However, this is more of a Delphi 1 development issue. Delphi 2 has much better integration between Delphi and ReportSmith.

**When to use ReportSmith.** Are flexibility and power the most important reporting features for your application? If so, ReportSmith is the better choice. It is also the only way to build an open-ended solution. ReportSmith is the only way to create "smart" double-duty reports from a single design. If allowing end-users

Feature	Description
Delphi Connection type	Allows reports to use the data already buffered in the Delphi application's BDE connection.
ReportSmith API	A programmable API for directly interfacing other tools with ReportSmith. (The API documentation is included on the Delphi 2 CD for editions of Delphi which ship with ReportSmith.)
32-bit performance	Record processing is visibly faster due to 32-bit data processing.

**Figure 1:** New ReportSmith 3.0 features.

to create their own reports is a system requirement, you will need to include ReportSmith as part of your solution. Otherwise, developers will be constantly asked to build new reports as users think of them.

**Using ReportSmith 3.0 with Delphi.** The Developer and Client/Server editions of Delphi 2 include a new 32-bit version of ReportSmith. ReportSmith 3.0 adds some new capabilities that will affect how you approach the integration of ReportSmith reports into your Delphi applications. These new features are summarized in the table shown in [Figure 1](#).

**ReportSmith 3.0 differences that affect Delphi development.** Some 16-bit versus 32-bit integration issues also exist. You will need to plan for these differences between ReportSmith 2.5 and 3.0 if you need to support both 16-bit and 32-bit Delphi applications. The table in [Figure 2](#) summarizes them.

**The New Delphi Connection**

Coordination between ReportSmith 2.5 and Delphi 1 has to be implemented programmatically through the use of report parameters, report variables, and/or DDE. The new 32-bit ReportSmith 3.0 adds the ability to directly use the same data connection as your Delphi 2 applications. This feature is referred to as the Delphi Connection.

For most reports, this ability to use the same data buffer results in much better report performance. It also means that report coordination between Delphi and ReportSmith is greatly simplified.

The specific advantages of using the Delphi Connection are:

- Faster data selection. A large part of the start-up time in a traditional ReportSmith report involves running the report's underlying SQL code. ReportSmith uses this code to select which records to include in the report. With the Delphi Connection, both Delphi and ReportSmith use the same data connection. The records to be reported are already selected by the time the report runs. ReportSmith just needs to format the data with a Delphi Connection report.
- Single development language. Traditional reporting with ReportSmith involves the use of either ReportSmith Basic or standard SQL code to perform any necessary



Feature	Description
ReportSmith Data Dictionary	ReportSmith 2.5 has a Data Dictionary Utility that allows a developer to pre-define table and column aliases, field visibility rights, etc. There is no 32-bit version of this utility. However, the editions of ReportSmith that are included in the Delphi 2 Update release do allow you to use Data Dictionaries created in the 16-bit tool when you develop 32-bit reports in ReportSmith 3.0.  Delphi 2 includes a Delphi 1 installation directory on the CD. You will have to install the ReportSmith Data Dictionary tool from the Delphi 1 directory to enable this feature in ReportSmith 3.0.
Menu Structures	The text of some menu items has changed from ReportSmith 2.5 to 3.0 to reflect the Win32 standards. For example, 2.5 uses "Character" where 3.0 uses "Font" in the SpeedMenus. (ReportSmith 3.0 currently is the only Windows 95 logo-compliant reporting tool.)  These differences may affect program code that relies on specific text in a menu selection when you have to support both 16-bit and 32-bit applications.
Picture Support	Both versions of ReportSmith support the use of .BMP, .PCX, and .DIB graphic formats. Due to a change in the copyright license for certain graphic formats, the 32-bit version no longer supports the .GIF and .TIF formats. If you need to support both 16-bit and 32-bit reporting, you will need to avoid using .GIF or .TIF graphics.

**Figure 2:** 16-bit features supported differently in ReportSmith 3.0.

code-based operations. This is not a big issue with some people. Others, however, prefer to learn just one language. The Delphi Connection is the way to place total control over data calculations into your Delphi application. Adding the power of DDE and the new ReportSmith API means you can then control virtually everything in ReportSmith with Object Pascal code.

- More powerful calculated fields. The entire data set of a Delphi Connection report is controlled by your Delphi application, including any calculated fields you define in your data set component. While ReportSmith itself has the ability to create calculated fields, the power of these "derived fields" is limited to what you can accomplish with either ReportSmith Basic or SQL code. Delphi can, of course, take advantage of the full power of Object Pascal. You can use your own custom code, commercial statistical packages, and any function call contained in DLL libraries or the Windows API in creating derived fields. There are *no* limits to the calculated fields you can create in Delphi.

**Using Delphi-supplied parameters.** The Delphi Connection is only available for building 32-bit reports in ReportSmith 3.0. Using report parameters is, therefore, the only option available for 16-bit applications. There are reasons why you might still want to use this capability — even in 32-bit applications — rather than using the direct Delphi Connection:

- Powerful report control. ReportSmith report variables can be used for a wide variety of functions that would not be possible by simply using the same data connection as your Delphi application. Report variables can be used for internal report processing that reaches outside the current Delphi data set. For example, you may want to highlight in red any purchase orders greater than a certain target amount. Report variables give you the ability to build a report that gets this threshold level at report time by passing a value from a Delphi variable, a value stored in an .INI file or the Windows 95 Registry, or by prompting the user. You can even take advantage of this kind of report variable processing with a report that *does* use the Delphi Connection to select the records to include in the report.
- Creating context-sensitive "smart" reports. Creating reports that can be run by themselves is one of the primary reasons to use ReportSmith. If you also need these reports to respond to the current Delphi environment when run from within your application, report variables provide the ability to create context-sensitive reports. These "smart" reports will feed off the parameters passed by Delphi from within your application. Then, when the same report is run outside of your application, it will automatically prompt the user for the necessary information to run the report in stand-alone mode. (Delphi Connection reports cannot be run by themselves. Your Delphi application must be running to be able to run a Delphi Connection report. Otherwise, the report will auto-exit with an error message.)
- Creating cross-platform reports. Many organizations recognize that the migration from 16-bit to 32-bit Windows platforms will take a while. The applications that support this transition best are the ones that provide for a "develop once, deploy on both" strategy. Since the Delphi Connection is a 32-bit only feature, cross-platform reports must rely on report variables and Delphi *TReport* parameters.
- Reporting on huge data sets. ReportSmith was originally designed as a client/server reporting tool. It provides tuning parameters in the Options dialog box for reporting on very large sets of data (see Figure 3). These Dynamic Data Access features are turned off when the Delphi Connection is used. (All data selection is controlled by the BDE buffer of your Delphi application.) As such, you may want to carefully weigh the performance ramifications of creating Delphi Connection reports when selecting huge data sets.

### The New ReportSmith API

ReportSmith 3.0 also ships with the new ReportSmith API. This standard programming interface can be used by Delphi to control ReportSmith directly. You are no longer restricted to using the less powerful DDE route to direct the operation of ReportSmith from within your Delphi applications. The documentation for this API is included on the Delphi 2 CD for editions that include ReportSmith 3.0.

## Using ReportSmith with Delphi

As noted earlier, two main methods exist for integrating ReportSmith

reports into your Delphi applications. The first method, using Delphi-supplied report parameters, applies to both 16-bit and 32-bit versions. The second method is to use the new 32-bit Delphi Connection.

**Using Delphi-supplied report parameters.** The key to using ReportSmith is in coordinating two separate executables, your Delphi applications and ReportSmith. All versions of ReportSmith support the use of report variables. Delphi's *TReport* component supports a mechanism whereby you can pass values from your Delphi application into report variables contained within your report.

The basic development sequence for using Delphi-supplied report parameters is as follows:

- Build a ReportSmith report that uses report variables. Define report variables for any of the items you want your Delphi application to use for controlling the report's execution.
- Add one or more *TReport* components to your Delphi application to control the report execution.
- Add Delphi code to your application to pass the appropriate values to the report's report variables when the reports are run or refreshed.

**Building a report with report variables.** ReportSmith allows you to create variables within the report design that are not tied to the data. Instead, these report variables have values that are typically supplied by the user and then become available within the report for logic tasks such as record selection. For example, you can create a report variable to store a user-supplied state. This value can then be used in a selection criteria to limit the report to those records from the state the user entered into the report variable. In the case of executing these reports from a Delphi application, your application becomes the report's user, and can supply the necessary values for any report variables defined in the report.

**"Smart" reports.** Report variables thus become the mechanism whereby you can create "smart" reports. The values for report variables can be supplied directly by the end-user, or indirectly by passing them from a Delphi application. When the report is run from the Delphi application, ReportSmith verifies if all report variables have already been supplied values. If they have, the report proceeds and automatically uses the values supplied by Delphi. If any of the report variables have not been assigned a value, or if the report is run "stand-alone" (i.e. outside of any Delphi application), ReportSmith is intelligent enough to prompt the end-user to supply values directly.

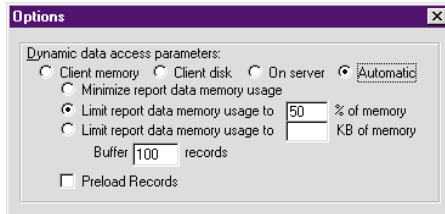


Figure 3: Dynamic Data Access settings.

Start ReportSmith and create a new columnar report based on the CUSTOMER.DB table in the DBDEMOS alias. (This alias is created when you install Delphi; it points to Delphi's sample data directory.) You will use this report as the basis for building your first "smart" report.

**Defining a report variable.** Use the Tools | Report Variables menu selection to bring up the Report Variables definition dialog box. This is where you will create custom prompts and definition specifications for your report variable. ReportSmith will use the information to create a dialog box for prompting the user for each report variable.

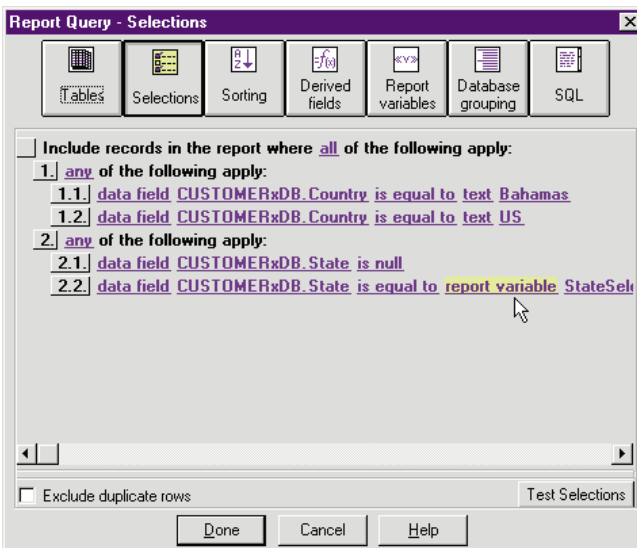
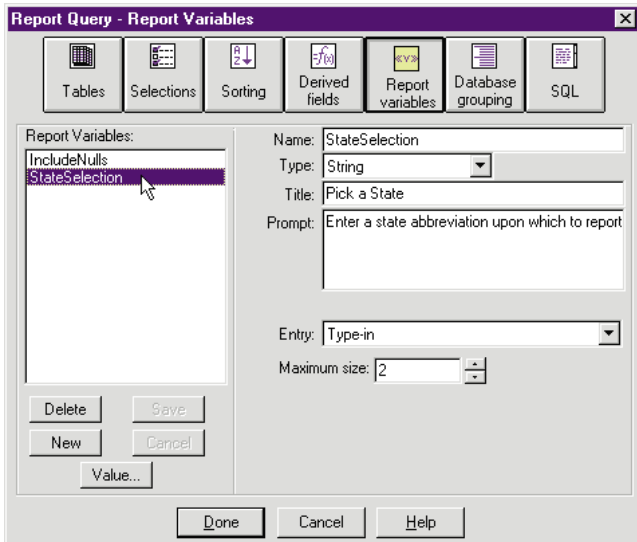
- The **Name** field is where you give your report variable its designated name. This name will appear throughout drop-down lists in ReportSmith wherever a report variable can be used.
- The **Type** of a report variable can be defined as string, number, date, time, or date/time. The variable type will affect the various entry options that will appear below.
- The **Title** refers to the title string that will be displayed in the dialog box ReportSmith uses to prompt the user for a value to assign this report variable.
- The **Prompt** value refers to the string that will appear as the message within the prompting dialog box.

For each report variable, you have a choice of **Entry** options. The report can have the user supply a value by typing it in, choosing from a defined list, choosing the value from a table, or choosing between two values (yes/no, true/false, etc.). When you select **Choose value from a table**, you can use a different data source type than the rest of your report. Various option settings will appear based on the type of the report variable and the entry option chosen.

To save the definition of your new report variable, click the **Add** button. Just editing the current data on the right side of the dialog box will change the currently highlighted report variable, not create a new one. To create a new report variable, you must click on the **New** button first. Create the *StateSelection* report variable that is shown in Figure 4.

When you are finished creating report variables, ReportSmith will prompt you to supply initial values for each variable. However, you won't see any effect on your report; you haven't told ReportSmith how to use your new report variable yet.

**Using a report variable.** Report variables are most often used to provide run-time record selection. In the previous example, say you wanted to make the selection of which state to include — something the user determines at run time. Once you have the *StateSelection* report variable defined, you can add it to the record selection criteria. Just use the drop-down lists in the Selections dialog box to switch from a "hard coded" text value to use your report variable instead. Say your report originally limited the records to those with a country of either the Bahamas or the US. Figure 5 shows how your selection criteria might look after adding the *StateSelection* report variable to control which US state gets included.



**Figure 4 (Top):** The Report Variables definition dialog box.  
**Figure 5 (Bottom):** Using a report variable for record selection.

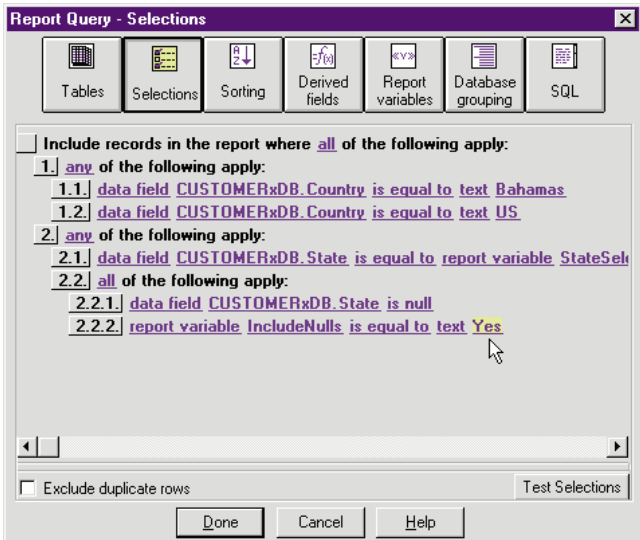
You can also make use of selection lists in conjunction with report variables to create some pretty complex criteria. Say, for example, that you only want null state records included if the user wants to include them. To accomplish this task, first create a *NullSelection* report variable. The parameters should appear as shown in [Figure 6](#).

Once this report variable is defined, open the Selections dialog box and click in the box at the left side of the criteria that says the State field can be null. A pop-up list will appear. Pick the **create a new list and move this item into it** option. Now add a new criteria in that list that states that the *IncludeNull* report variable must be equal to the text “Yes”. Your criteria should now look like the dialog box shown in [Figure 7](#).

Now you should be prompted twice, once for the state to select and once to decide whether to include records with a null value in the State field. Selecting **Yes** to the second prompt will cause the records from the Bahamas to appear. Selecting **No** will limit the report to records from the US.

Parameter	Value to Enter
Name	Include Nulls
Type	String
Title	Include Nulls?
Prompt	Should records with no state be included?
Entry	Choose between two values
Message or Dialog	Message
Yes value	Yes
No value	No

**Figure 6:** Defining a Yes/No report variable.



**Figure 7:** Expanding the selection criteria with a list.

**Changing the value of a report variable.** You may want to test multiple parts of a report controlled by a report variable. You might want to print a different set of records without having to reload the report. In either case, you need to alter the value initially assigned to a report variable. Simply select **Tools | Report Variables** from the main menu, click on the name of the report variable whose value you want to change, and click on the **Value** button that appears near the bottom of the dialog box. ReportSmith will prompt you for a different value in the same manner it prompted for the initial value.

### Creating the Delphi Application

The next step in creating ReportSmith reports that can run from your Delphi application is to build the necessary capabilities into your Delphi application. This involves the use of Delphi’s *TReport* component.

**The TReport component.** The *TReport* component resides in the Data Access page of Delphi’s Component Palette. It is what you will use to control all aspects of running ReportSmith reports. While other properties exist, the table in [Figure 8](#) shows the key properties upon which most of your development attention will be focused. Please note the *LaunchType* property is new to Delphi 2.

The *TReport* component also supplies several methods for dealing with ReportSmith from within your Delphi application. The key methods are listed in [Figure 9](#). All *TReport* methods operate via DDE, but you do not need to pro-

Property	Function
<i>AutoUnload</i>	Determines whether ReportSmith exits or stays in memory once the report is finished. Keeping ReportSmith resident can significantly increase the performance of applications that print multiple reports. Use the <i>CloseApplication</i> method to shut down ReportSmith if <i>AutoUnload</i> is set to <i>False</i> .
<i>InitialValues</i>	Use this array of <i>TStrings</i> to specify initial values for any report variables included in the report. See the discussion of passing report variable values for complete details.
<i>LaunchType</i>	Delphi 2 only. Determines whether ReportSmith or the ReportSmith RunTime will be launched. The default value of <i>ItDefault</i> will launch the ReportSmith 3.0 design environment when you double-click on the <i>TReport</i> in Delphi's IDE, and will execute the ReportSmith 3.0 RunTime when you launch a report from within a running Delphi application.
<i>Preview</i>	Determines whether the <i>Run</i> method launches a print preview or hard copy output. If <i>Preview</i> is set to <i>True</i> , use the <i>Print</i> method instead of <i>Run</i> to force hard copy output.
<i>ReportDir</i>	The directory where the report resides. At run time, use the function <code>ExtractFilePath(ParamStr(0))</code> if you want to set this property to the current application directory. Do not leave this property blank.
<i>ReportName</i>	The name of the .RPT file to run. Place only the name of the file here. All directory information should be placed in the <i>ReportDir</i> property. Any directory information placed in the <i>ReportName</i> property will be ignored.

Figure 8: Key properties of the *TReport* component.

Method	Function
<i>CloseApplication</i>	Tells ReportSmith to perform a <b>File   Exit</b> . Use this method when your <i>TReport's AutoUnload</i> property is <i>False</i> . Make sure you call <i>CloseApplication</i> before exiting your Delphi 1 applications. In Delphi 1, ReportSmith will not exit even when your application shuts down. This behavior has changed with 32-bit applications. ReportSmith will automatically shut down when you exit a Delphi 2 application, even if <i>AutoUnload</i> is <i>False</i> .
<i>Connect</i>	Allows you to connect to a database and bypass the ReportSmith database login. The <i>Connect</i> method is not needed for reports that use the Delphi Connection.
<i>Print</i>	Forces a hard copy output of the report, regardless of the value of the <i>Preview</i> property.
<i>RecalcReport</i>	Tells ReportSmith to refresh the report. Use this method whenever you change the value of a report variable or navigate to a new record set in a Delphi Connection report.
<i>Run</i>	Runs the designated report, looking at the value of the <i>Preview</i> property to determine if a preview or a hard copy should be made.
<i>RunMacro</i>	Lets you run a ReportBasic macro.
<i>SetVariable</i>	Lets you set or change the value of a report variable. If you specify a report variable name that is not used in the current report, the variable will simply be ignored.
<i>SetVariableLines</i>	Similar to <i>SetVariable</i> , this method lets you set or change the value of a report variable by using an array of type <i>TStrings</i> .

Figure 9: Key *TReport* component methods.

gram any of the DDE operations yourself. They are automatically encapsulated within the *TReport* component. You also do not need to use any of the Delphi DDE components when you work with *TReport*.

All *TReport* methods provide a True/False return value. This return value gives you feedback on the success or failure of *TReport's* action. If the method's action gets a "command received" confirmation from ReportSmith, the return value will be *True*. A return value of *False* indicates some kind of problem with the DDE conversation.

**Integrating ReportSmith report variables into your Delphi application.** Delphi applications can control ReportSmith by pre-defining the values of any report variables used. Set up these report variable values in any way you want within Delphi. For example, you can even create a "Query-by-Form" interface to control all the specifics of a report, assign any values the end-user has set to the appropriate report variables, then launch the report. Any report variable you supply a value for will be skipped in ReportSmith's prompting process. ReportSmith will still prompt the user for any report variables that do not yet have a value.

Keep in mind that Delphi deals with report variables in two ways, depending on whether the report is already open. If the report hasn't been opened, you need to assign values by setting up the *InitialValues* property of the *TReport*. Once the report is running, set or change the values of report variables using the *SetVariable* or *SetVariableLines* method.

Do not use *SetVariable* or *SetVariableLines* without an open report. These methods try to establish a DDE conversation with ReportSmith. If ReportSmith isn't running, these methods will launch it. Your *TReport* may not have all its properties set properly for your report, so ReportSmith will not be launched with the proper setup. You might not even have a report open, in which case your report variable changes will be ignored with no way for Delphi to know they have been lost. Make sure you use the *InitialValues* property to establish report variable values before launching a report.

**Assigning initial values to report variables.** You can assign initial values to report variables either at design time using the Delphi Object Inspector, or in application code. To set initial values at design time, double-click on the edit box for the *InitialValues* property and enter the report variable name and its value in the string editor.

Notice that the proper syntax for each entry in the *InitialValues* property is:

```
@ReportVarName=<ValueToAssign>
```

No spaces are allowed. You also need to remember that report variable names are case-sensitive. You must duplicate the report variable's name *exactly* as it was defined in ReportSmith. In the example shown in Figure 10, enter-



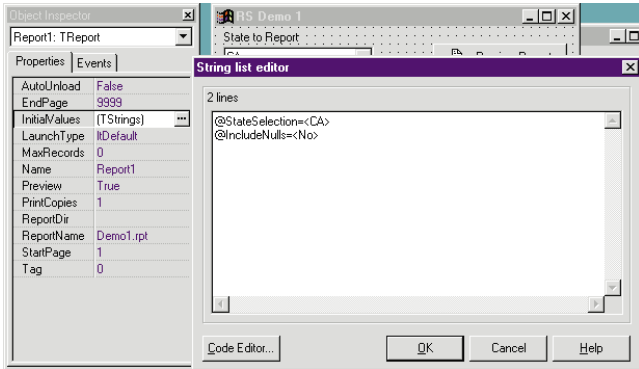


Figure 10: Assigning initial report variable values at design time.

ing a variable name of *StateSelection* would result in the value being ignored. ReportSmith's own prompting logic would then take over and ask the user to select a state. So, if you think your Delphi application is supplying a value to a report variable but the report still prompts for it, check that you are using the correct report variable name.

You can also attach code to a Delphi event that sets up the initial values. Use the *Add* method of the *InitialValues* property to set initial values into a report variable. For example, you might place the following code on a *Print* button to initialize a report and then launch it:

```

procedure TForm1.BtnPreviewClick(Sender: TObject);
begin
    { Set to current application directory }
    Report1.ReportDir := ExtractFilePath(ParamStr(0));
    Report1.ReportName := 'Demo1.rpt';
    Report1.InitialValues.Add('@StateSelection=<CA>');
    Report1.InitialValues.Add('@IncludeNulls=<No>');
    Report1.run;
end;
    
```

**Changing the value of a report variable.** Once a report is opened, you must change report variables in a way that can be recognized by a running report. Delphi uses *TReport* methods to change report variables via a two-way DDE conversation. Changing a report variable does not automatically refresh the report based on the new value.

This separation allows you to set multiple new values before using the *RecalcReport* method to force a refresh of the report. You should usually test the success of any changes before trying to tell ReportSmith to recalculate the report. Here's an example:

```

if Report1.SetVariable('IncludeNulls', 'Yes') then
    Report1.RecalcReport;
    
```

It is extremely important to note the syntax used by the *SetVariable* and *SetVariableLines* methods is very different from what you use to set the *InitialValues* property.

**An Example Delphi Application**

The RS\_DEMO1 project gives you an example of how to control a ReportSmith report from within a Delphi application. This project makes use of the techniques described

earlier to control the running of the DEMO1.RPT report file via report variables.

DEMO1.RPT is an example of a "smart" report. First, start up ReportSmith and open the DEMO1.RPT report file by itself. You will see that the report auto-

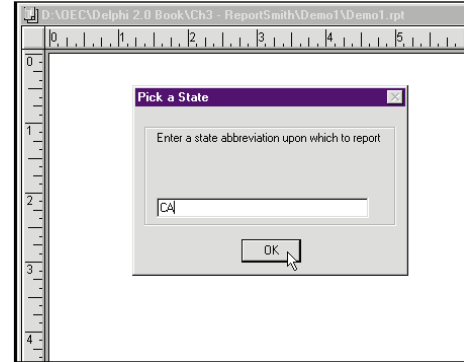


Figure 11: DEMO1.RPT running on its own.

matically prompts you to supply a state abbreviation (see Figure 11). It then asks if you want to include records with a blank in the state field. Once you answer these questions, the report proceeds.

Now try running the Delphi application RS\_DEMO1.EXE and see the difference in running the same report controlled by this Delphi application (see Figure 12). Since Delphi is supplying the values for the two report variables, the report already has all the information it needs before it starts. The end-user is never prompted by ReportSmith.

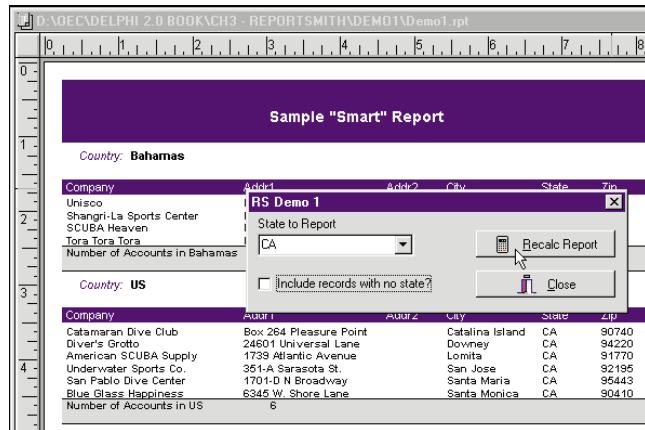


Figure 12: DEMO1.RPT run by a Delphi application.

The main code that runs the RS\_DEMO1 application is listed in Figure 13. Take particular note of the *SetVars* procedure. It tests whether the application has run ReportSmith before deciding on how to implement changes made to the items supplying values to the report variables. If the report hasn't been started, the code uses the *InitialValues* property. If the report is already running, the *SetVariables* method is used and the *Recalc Report* button is activated. Try changing the value of one or both items in RS\_DEMO1 and watch the effect of hitting the *Recalc Report* button. The report will update behind the scenes.

**Until Next Month ...**

Despite its Rodney Dangerfield reputation, ReportSmith is

```

procedure TForm1.SetVars(Sender: TObject);
begin
  if BtnPreview.Visible then
    begin
      Report1.InitialValues.Add(
        '@StateSelection=<' + cbStateList.Text + '>');
      if CbxNulls.checked then
        Report1.InitialValues.Add('@IncludeNulls=<Yes>')
      else
        Report1.InitialValues.Add('@IncludeNulls=<No>');
    end
  else
    begin
      Report1.SetVariable('StateSelection',
        cbStateList.Text);
      if CbxNulls.checked then
        Report1.SetVariable('IncludeNulls', 'Yes')
      else
        Report1.SetVariable('IncludeNulls', 'No');
      BtnRecalc.Enabled := True;
    end;
end;

procedure TForm1.BtnRecalcClick(Sender: TObject);
begin
  Report1.RecalcReport;
  BtnRecalc.Enabled := False;
end;

procedure TForm1.BtnPreviewClick(Sender: TObject);
begin
  Report1.ReportDir := ExtractFilePath(ParamStr(0));
  Report1.Run;
  BtnRecalc.Top := 16;
  BtnPreview.Visible := False;
  BtnRecalc.Visible := True;
end;

```

**Figure 13:** The main code that runs the RS\_DEMO1 application.

a powerful reporting tool that can be easily integrated into your Delphi applications. Hopefully this article has shown you a few techniques you can use in your own applications. This month, we tackled the issues of when to use ReportSmith, and how to call ReportSmith reports using report variables. Next month we'll address the use of the new Delphi Connection.  $\Delta$

*Some of the material in this article is excerpted by permission from the author's chapter on ReportSmith in the book "Delphi In Depth", copyright 1996 by Osborne/McGraw-Hill. All code examples given here are found on the CD that accompanies the book.*

*The demonstration project and report referenced in this article are available on the Delphi Works CD located in INFORM\96\AUG\DI9608MO.*

Mark Ostroff has over 16 years experience in the computer industry. He began by programming mini-computer medical research data acquisition systems, device interfaces, and process control database systems in a variety of 3GL computer languages. He then moved to PC's using dBASE and Clipper to create systems for the US Navy and IBM's COS Division. He volunteered to help create the original Paradox-based "InTouch System" for the Friends of the Vietnam Veteran's Memorial. Mark has worked for Borland for the past five years as a Systems Engineer, specializing in database applications.





## ON THE COVER

QuickReport / Delphi 1 / Delphi 2



By *Cary Jensen, Ph.D.*

# QuickReport

## An Introduction to Delphi 2's Alternative Report Generator

**N**early every edition of Delphi features Borland's ReportSmith, a tool that allows developers to create extremely flexible and powerful reports. However, not every Delphi application requires all the features ReportSmith provides. And, certainly, not all applications need the overhead that including ReportSmith reports entails.

For example, when shipping a Delphi application with ReportSmith reports, you must also ship ReportSmith Runtime. Not only does this mean your application's installation disk set must include the three ReportSmith Runtime installation disks, but your program will occupy an additional 5 to 6MB of hard disk space. (The 16-bit version of ReportSmith Runtime requires a little over 5MB, while the 32-bit version requires more than 6MB.) Granted, this additional storage is a one-time concern, since only one copy of ReportSmith Runtime is required per machine.

To give Delphi developers the ability to include reports in applications without ReportSmith's overhead, a number of third-party developers have made available (often through shareware) component-based reporting tools. Some of these products include:

- ReportPrinter by Nevrona Designs
- Ace Reporter by SCT [Bill Todd's review of Ace Reporter begins on page 46]
- QuickReport by QSD (Quick Soft Development AS, located in Oslo, Norway)

Each of these reporting tools allow Delphi developers to define reports they can print and preview from an application without the need for additional files. Specifically, these tools permit all the code necessary for producing the reports to compile into your application's .EXE.

One of these component-based report tools, QuickReport, is included with Delphi 2. (A 16-bit shareware version is

available for download. For more information, check out QSD's Web site at <http://www.qsd.no/products/quickrep>.)

This article is an introduction to building QuickReport reports and how to use them in your applications. Among the issues we'll cover include the essential components needed to make a form into a QuickReport report; the names and uses of the various components you can include in a report; and, a more detailed discussion of report bands. For additional information, refer to Delphi's online Help, the QSD Web site, or Chapter 10, "Reporting with QuickReport in Delphi 2" in the book *Delphi In Depth* [Osborne/McGraw-Hill, 1996].

### A QuickReport Overview

Using the components on the QReport page of the Component Palette, you can create reports ranging from very simple lists that include data from a single DataSet, to highly sophisticated reports, e.g. those displaying memo fields and bitmap data, as well as reports showing data drawn from multiple DataSets. These reports are compiled into your applications, and can be easily printed and previewed at run time.

Six essential components are needed to create a report with QuickReport:

- a form
- a QuickReport component
- a DataSource component
- a DataSet descendant (i.e. a Table, Query, or StoredProc component)
- a QRBand component

## ON THE COVER

- one or more of the QuickReport-printable components. The most basic of the QuickReport-printable components are QRLabels for printing a static text label, and QRDBLabels for printing data fields from a DataSet.

The form component will be the container on which the QuickReport is constructed. In most cases, you'll use one form for each QuickReport report in your application. You can, although it's less common, make one form the basis for multiple reports.

Residing on the QReport page of the Component Palette, the QuickReport component is the central component for any QuickReport report. Conceptually, you can consider a QuickReport component the ingredient for changing a form into a report. Via its *DataSource* property, the QuickReport component links to the required DataSource component. This DataSource, in turn, must point to the required DataSet. The data in this DataSet is the report's main source of data.

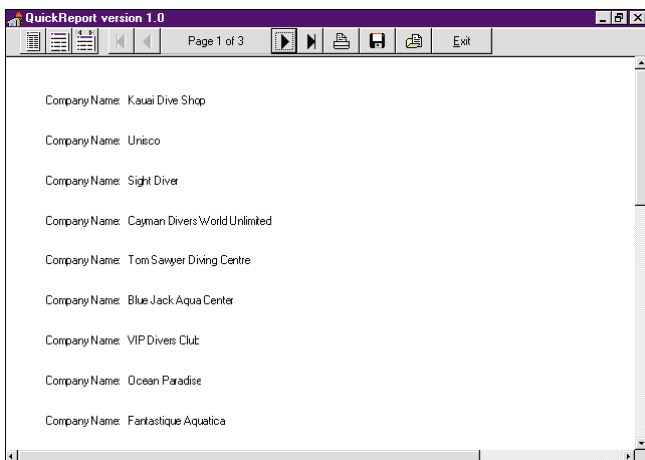
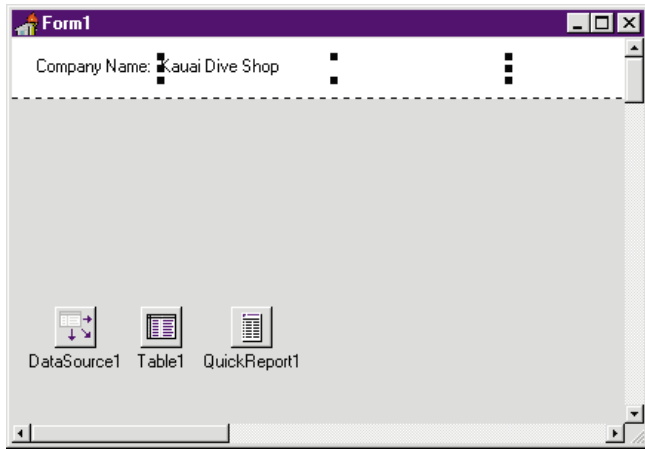
At a minimum, a QuickReport report must include at least one QRBand component, but most include more than one. The QRBand is a container where you'll place the report's printable elements. As mentioned, these include the QRLabel and QRDBLabel components.

The display of the printable elements you place in a QRBand primarily depends on the *BandType* property of the QRBand. If individual records from the QuickReport's DataSource are being printed, at least one QRBand component in the report will have a *BandType* of *rbDetail*. The printable elements within this QRBand are printed for every record in a DataSet. Since most reports also include a page header and footer, you'll probably include at least two additional bands in your report: one with its *BandType* set to *rbPageHeader* and another with its *BandType* set to *rbPageFooter*.

### Creating a Simple Report Example

To create our simple report example, follow these instructions:

- Create a new project.
- Place a DataSource, Table, QuickReport, and QRBand on the project's main form. The default *Align* property for the QRBand is *alTop*, so the QRBand will automatically move to the top of the form.
- Set the *DataSet* property of the DataSource to *Table1*. Set the Table component's *DatabaseName* property to *DBDEMOS*, its *TableName* property to *CUSTOMER.DB*, and its *Active* property to *True*.
- Set the QuickReport's *DataSource* property to *DataSource1*. Set the QRBand's *BandType* property to *rbDetail*.
- Place one QRLabel and one QRDBLabel in the QRBand. Set the QRLabel's *Caption* property to *Company Name*. Set the QRDBLabel's *DataSource* property to *DataSource1*, and its *DataField* property to *Company*. Since *Table1* is active, when you have set the *DataField* property of the QRDBLabel component to *Company*, the company name associated with the first record in the *CUSTOMER.DB* table will appear in this component. The form should now resemble [Figure 1](#).



**Figure 1 (Top):** A form has been made into a basic QuickReport report by placing several QuickReport components on it.

**Figure 2 (Bottom):** A QuickReport report in the default previewer.

### Previewing a Design Time QuickReport

You don't have to run a project to preview a QuickReport report. Provided the DataSets used for the data in the report are active, you can view the report at design time. To do this, double-click the QuickReport component, or right-click the QuickReport component and select **Preview report** from the displayed SpeedMenu. The QuickReport component will format your report and display it in the default QuickReport previewer (see [Figure 2](#)).

While previewing your QuickReport reports at design time is convenient, you should always rely on run-time tests to verify your QuickReport reports. Most QuickReport components include event methods to which you can assign event handlers to control many aspects of the report's display and printing.

For example, you can use these event handlers to perform conditional highlighting of fields, add conditional page breaks, or load and print custom data based on the contents of the report. However, these event handlers are not executed when you preview a report at design time. Consequently, if the event handlers have any effect on the report's content, format, or page control, a design-time previewed report will be different from the same report when previewed or printed at run time.

### Using a QuickReport at Run Time

To allow the user to preview or print a QuickReport



report, you must explicitly call the QuickReport *Preview* or *Print* methods. However, this is the minimum you must do. Using a QuickReport report from a Delphi application should include several additional steps to properly configure it. For example, it's not likely you'll display the form containing the QuickReport components to the user. Instead, you'll call the QuickReport's *Preview* or *Print* methods from another form.

Furthermore, while Delphi automatically calls the constructor of the form that contains the QuickReport, you may decide to explicitly call the form's constructor at run time, and then release the form when it's no longer needed. This reduces the resources your application requires.

Both of these techniques can easily be performed with the QuickReport report created in the preceding section.

### Previewing and Printing a QuickReport Report

Use the following steps to demonstrate printing or previewing at run time with the report created in the preceding section:

- Select **File | New Form** from Delphi's main menu.
- Place two Button components on the new form. Change the *Caption* property of *Button1* to **Preview Report**, and change the *Caption* property of *Button2* to **Print Report**.
- Double-click the **Preview Report** button to create an *OnClick* event handler for it. Enter the following code:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  Form1 := TForm1.Create(Self);
  Form1.QuickReport1.Preview;
  Form1.Release;
end;
```

- Next, double-click the **Print Report** button and enter the following code into its *OnClick* event handler:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  Form1 := TForm1.Create(Self);
  Form1.QuickReport1.Print;
  Form1.Release;
end;
```

- While in *Unit2*, select **File | Use Unit**. Select *Unit1* from the displayed dialog box. This makes the declarations that appear in the **interface** section of *Unit1* available to *Unit2*. (This is required to refer to the type *TForm1* and the variable *Form1* from *Unit2*.)
- Designate *Form2* as the Main form. Do this by selecting **Project | Options** (Delphi 2) or **Options | Project Options** (Delphi 1) to display the Project Options dialog box. From the Forms page of this dialog box, select *Form2* from the **Main Form** drop-down menu.
- Remove *Form1* from the **Auto-create forms** list. Do this by selecting *Form1* from the **Auto-create forms** list on the Forms page of the Project Options dialog box, and then click >> to move *Form1* into the **Available forms** list. Now close the Project Options dialog box.
- Run the project. When you click the **Preview Report** but-

ton, the default QuickReport previewer dialog box will appear, and when you click **Print Report**, the QuickReport will be sent to the printer.

### Using Data Modules with QuickReport

While the preceding report example includes both the *DataSource* and *DataSet* components on the same form as the QuickReport components, Delphi 2 provides you with a more attractive option — *data modules*. Using a data module to hold all *DataSets* used by your QuickReport reports simplifies the creation and maintenance of the reporting side of your applications.

After you define the data module, select **File | Use Unit** from the QuickReport form (or its associated unit), and select the unit associated with the data module. From that point, all *DataSources* and *DataSets* defined for the data module are available to the QuickReport components.

In most cases, however, the data module you use for reporting should be separate from those you use for the interactive elements of your user interface (e.g. data entry forms). Consequently, a Delphi 2 application will often include at least two data modules — one for the user interface, and another for reporting.

The reason for using separate data modules for data entry and printing is fairly simple. When a QuickReport is either previewing or printing the data in a *DataSet*, it must navigate that *DataSet*. While this behavior is harmless if the QuickReport is not using a *DataSet* accessible to the user, it can pose problems if the QuickReport and the user are sharing the same *DataSet*.

For example, if a user is editing a record, and then, before posting changes to the record, attempts to print or preview the report, QuickReport causes the changes to be explicitly posted. Furthermore, after previewing the report, the user will be returned to the last record in the table (actually, the last record printed or previewed). In other words, the user will probably not be on the same record he or she was editing. By comparison, if the user interface and QuickReport use different *DataSets* — even if they point to the same table — previewing a QuickReport report will have no effect on the current record's state.

### Components Used in QuickReport

The example we built represents the simplest report you can create with QuickReport. Consequently, it employed only a few of the available QuickReport components. The QReport page of the Component Palette lists a total of 11 components that you can use in your QuickReport reports. In addition to these, the Component Palette features two other components that you can include in your reports. **Figure 3** lists and describes all 13 of these components.

In addition to those shown in **Figure 3**, there are two more QuickReport components that do not appear on the Component Palette. The non-visual QRPrinter component is created automatically for every QuickReport. QRPrinter

Component	Description
QuickReport	Required component that makes a form a QuickReport.
QRBand	Panel-type object on which you place printable components. The <i>BandType</i> property defines what feature a particular QRBand provides.
QRLabel	Static, single line of text.
QRDBText	Use to display a field from a DataSource.
QRDetailLink	Defines the link between a master and detail QRBand, as well as between a Detail and a SubDetail QRBand.
QRMemo	Use to define multiple lines of static text.
QRShape	Shape component for a QuickReport.
QRDBCalc	Component for displaying simple summary statistics. Can also be used for single fields inside the Detail and SubDetail bands for the purpose of applying picture (display) formats.
QRPreview	Panel-like object used to display the image of a report preview. Use to create a custom report previewer.
QRGroup	Use to identify which QRbands are the Detail, Header, and Footer bands for groups of records.
QRSysData	Field used to display system and report data, such as the date, the time, page number, record number, and so on.
DBImage	Field used to display bitmaps from a DataSet.
Image	Use to display bitmaps.

**Figure 3:** The components for use with QuickReport reports.

provides you with low-level access to your printer's features, such as issuing a form feed. The second component, `QRCustomControl`, is the base class for the QuickReport components. Normally, you won't use this component in your applications.

## Using QRbands

All QuickReport reports must use at least one QRband, and most use at least three. As mentioned, the most important property of a QRband is *BandType* because it defines the role that the band will play. [Figure 4](#) displays a list of the various *BandType* values, and how they affect the QRband.

Most reports use at least three QRbands:

- 1) one acts as the page header
- 2) one acts as the detail
- 3) one acts as the page footer

The page header and page footer provide the top and bottom margin for the page, while the detail band displays the data from each record.

After the *BandType* property, the second most important property of a QRband is *Height*. The *Height* of a QRband determines how much space it will occupy on the report. For example, the *Height* property of a band designated as the page header (by setting its *BandType* to *rbPageHeader*) determines the size of the top margin of each page of your report. Likewise, the *Height* of a Detail band determines how much

Band Type	Description
<i>rbColumnHeader</i>	Optional ColumnHeader band that contains column headings for a columnar report. These column headings can alternatively be placed in a Header band, such as a PageHeader or GroupHeader.
<i>rbDetail</i>	The Detail band is printed once for each record in a DataSet. Most reports have one Detail band. In a master/detail report, you can use one Detail band for each DataSet (an alternative is to use a SubDetail band for detail records). Only summary reports do not include at least one Detail band.
<i>rbGroupFooter</i>	When groups are defined using either QRGroup or QRDetailLink components, an optional GroupFooter band can be used to print elements at the end of a group.
<i>rbGroupHeader</i>	When groups are defined using either QRGroup or QRDetailLink components, an optional GroupHeader can be used to print elements before each group.
<i>rbOverlay</i>	The optional Overlay band contains printable elements that will be printed over all other bands in a report. For example, an Overlay can contain a logo that elements from other bands will print on top of.
<i>rbPageFooter</i>	The optional PageFooter band contains elements that will be printed at the bottom of each page of the report.
<i>rbPageHeader</i>	The optional PageHeader band contains elements that will be printed at the top of each page of the report.
<i>rbSubDetail</i>	The SubDetail band is essentially the same as a Detail band, but is designed to be used for a detail table in a master/detail report. While you always use a Detail band for the master table in a report, you can use either a Detail or a SubDetail band for the detail records. You use a QRDetailLink component to identify a SubDetail band.
<i>rbSummary</i>	The optional Summary band is printed once at the end of a report, after the last master record in a single table report, or after the last detail record in a master/detail report.
<i>rbTitle</i>	The optional Title band is printed once at the beginning of the report. The QuickReport property <i>TitleBeforeHeader</i> defines whether the Title band is printed before the page header on the first page of the report.

**Figure 4:** QRband *BandType* property values.

vertical space is used for each record in the master table, and consequently, influences the maximum number of records that can appear on any given page.

Here's a tip: You can set the *Ruler* property of a QRband to a value other than *qrrNone* to display a vertical or horizontal ruler (or both vertical and horizontal rulers) in either inches or centimeters. These rulers can provide invaluable assistance for accurately placing objects in your QRbands.

## Creating a Multi-Band Report

When your QuickReport reports contain more than one band, they can become much more difficult to create and maintain. This is because there is nothing that visually distinguishes one type of band from another, other than the band's published properties in the Object Inspector. Moreover, these properties are displayed for only one band at a time — the selected band.

By following the preceding rules, you'll find your multi-band reports much easier to create, as well as easier to modify later: **First design your report on paper.** This allows you, in advance, to identify how many bands, and of what types, you'll place onto your form.

**Place your bands onto the form in the logical order they'll be printed.** For example, if you're going to have a title band on your report, place it on the form first, then place your page header, detail band, page footer, and summary band. The QRBand components are top-aligned. The first band placed will always appear at the top of the form, and the second will appear in the second position.

While the actual order of the bands has no influence on their behavior, a multi-band report is much easier to use if the order of the bands on the form duplicates the role they play in the report. (Note that if you misplace a band, you can drag it to a new position. For example, if you add a Title band and it appears as the last band on the form, you can drag it to the top position.)

**Always give your bands meaningful names that indicate their role in the report.** For example, set the *Name* property of your Title band to `Title`, or another name identifying how the band is being used. This is especially important when creating master/detail reports, or reports containing groups that include headers and/or footers.

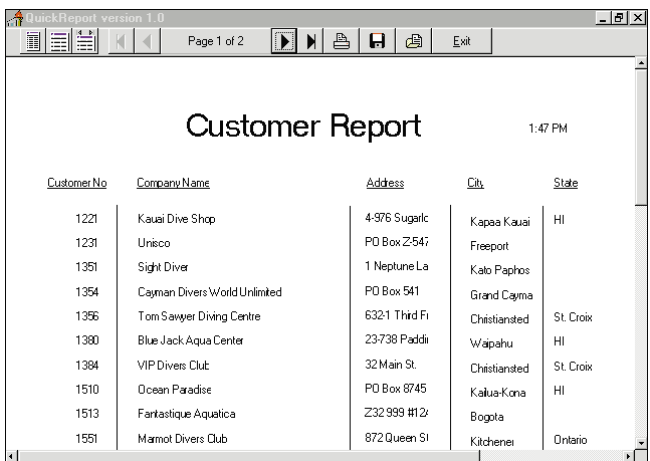
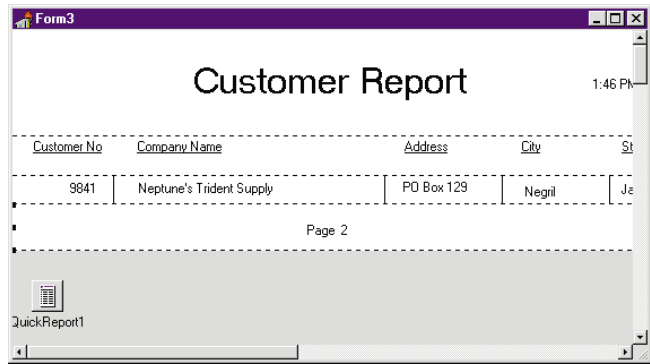
Figure 5 shows the basic design of a multi-band report that contains four bands: an *rbPageHeader*, *rbColumnHeader*, *rbDetail*, and *rbPageFooter*. The Header and Footer bands provide the top and bottom margins of the report, respectively. The *rbColumnHeader* band is used to display the column headings for the Detail band. Please note, however, the column headings in this case could just as easily have been placed in the page header. The Detail band contains the fields that will be printed for each report in the DataSet.

The page header includes the report title — a QRLabel component. It also includes the time of report printing, which is displayed using a QRSysData component with its *Data* property set to *qrsTime*. The page footer also includes two components. One is a QRLabel with its *Caption* property set to *Page*, and the other is a QRSysData with its *Data* property set to *qrsPageNumber*.

The QRLabel components that appear in the ColumnHeader band are underlined. This was achieved by choosing an underlined font for these components. The separator lines appearing between the fields in the Detail band are actually QRShape components — its width was set to be very narrow. Figure 6 is the resulting report displayed in the Report Previewer.

## Conclusion

This article briefly examines some of the reporting capabilities available from QuickReport reports. It only scratches the surface. QuickReport allows you to design completely cus-



**Figure 5 (Top):** The basic design of a report with four bands. **Figure 6 (Bottom):** Our completed report in the previewer.

tom report previewers, save report output to file for display at a later time, create event handlers to produce enormously flexible reports, etc. If your Delphi applications require reports, and you're not already committed to using ReportSmith, you may find that QuickReport gives you everything you need in a report generator, and more. ▲

Parts of this article are based on, or reprinted by permission, from *Delphi In Depth* by Cary Jensen, Loy Anderson, Joseph Fung, Ann Lynnworth, Mark Ostroff, Martin Rudy, and Robert Vivrette, published by Osborne/McGraw-Hill, Inc. Copyright 1996 by The McGraw-Hill Companies Inc. For more information about this book, please visit the Web site <http://gramercy.ios.com/~jdsi/did.html>.

*The demonstration forms and report referenced in this article are available on the Delphi Informant Works CD located in INFORM96\AUG\DI9608CJ.*

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including *Delphi In Depth* [Osborne/McGraw-Hill, 1996]. He is also Contributing Editor of *Paradox Informant* and *Delphi Informant*, and is 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 via CompuServe at 76307,1533.





## ON THE COVER

ReportSmith

By *Jeff Sims*

# ReportSmith Secrets

## Tips for Increasing Your Productivity with Borland's Misunderstood Reporting Tool

**R**eportSmith is misunderstood. Granted, it's different from other reporting tools, so some complaints are understandable: "Why should ReportSmith be so different?" "It has no design mode." "It doesn't compile." And perhaps most damning: "It's too slow. I wait too long for the report to format."

But these are misperceptions. It's true that ReportSmith has no design mode (i.e. those cryptic "9s" and "As" across a banded report specification). And yes, opening a report to modify it requires the associated query to execute, which in turn triggers ReportSmith to reformat the report. But that's just its default behavior. To open a ReportSmith report and make a change without triggering a query and reformat, simply deselect the **Run Report** check box in the Open Report dialog box. Presto! The query doesn't run until you say OK. (Already, a secret revealed!)

So ReportSmith isn't slow — once you learn a few secrets. Which is what this article is all about.

More than anything else, ReportSmith suffers from an image problem. And it's my contention that if you would take the time to learn a few tricks of the trade, you'd find ReportSmith as useful and powerful as I do. And frankly, I greatly prefer the WYSIWYG "live" data approach.

I've culled the following "secrets" from what I keep promising as the soon-to-be published book, *Report Writers Explained* [Prentice Hall]. These techniques will help you better understand ReportSmith and increase your Delphi reporting productivity.

### Secret 1: Make Sure You've Got the Horsepower

Are you trying to run ReportSmith on a 386-based system with 4MB of RAM? For the record, ReportSmith is *not* a low-end product. Period. Sure, ReportSmith is bundled with almost every version of Delphi (except for Delphi 2 Desktop), and one of Delphi's strengths is its ability to create a lean, mean executable that can run on yesterday's hardware. ReportSmith, however, needs a modern processor and lots of memory to perform well.

If your clients want reports that involve complexity, ReportSmith is the tool that will allow you to deliver the product. You will need at least 12MB of RAM and a 486 processor. (Alternatively, Delphi 2 now ships with QuickReport if you want to create a simple, lightning-fast, executable report.) If you're short of RAM, try to free some memory (e.g. unload unnecessary programs, etc.). If, however, you are calling ReportSmith (even RunTime) from a Delphi application, you probably want 16MB of RAM.

Now let's consider the choice of the operating system (OS). With a 32-bit OS, ReportSmith 3.0 (it will *not* run on Windows 3.x) can truly multi-task. Periodically, the DBMS will take a while to respond to your query. So if you have enough memory (i.e. 32MB), then run Windows/NT. Otherwise, Windows 95 is



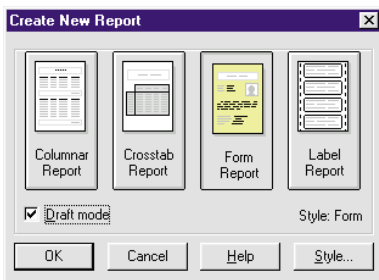
## ON THE COVER

a better choice than Windows 3.x. Remember, both Windows NT and Windows 95 will use every last byte of memory (Note that Windows 3.x can only use 16MB). So, get off to a good start with adequate resources and a robust multi-tasking operating system.

In brief, don't believe what it says on the box. To run ReportSmith, I recommend a Pentium-based system running 32-bit Windows with plenty of RAM (16MB for Windows 95 and 32MB for Windows NT). In addition, you should avoid using middleware; run native drivers instead. And, for those of you accessing dBASE files, don't forget to create indices on columns used in joins.

### Secret 2: Draft and Presentation

I'm amazed at how often I encounter users and developers who don't take advantage of ReportSmith's Draft mode.



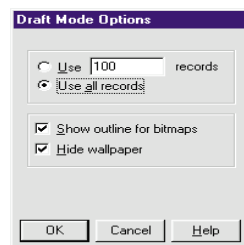
**Figure 1:** Enable Draft mode at the Create New Report dialog box.

Let's say you are creating a new report in ReportSmith 3.0. After selecting **File | New**, the Create New Report dialog box appears. It allows you to select the type of report you want to create. To enable Draft mode, check the **Draft mode** box in the lower

left corner of the screen (see [Figure 1](#)).

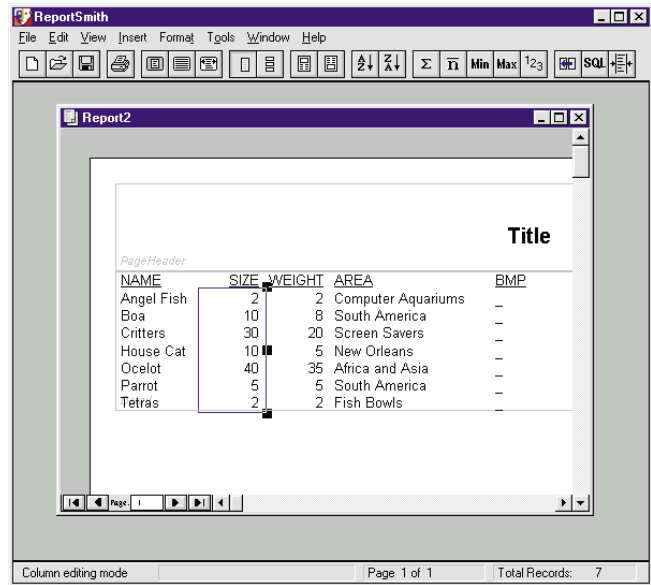
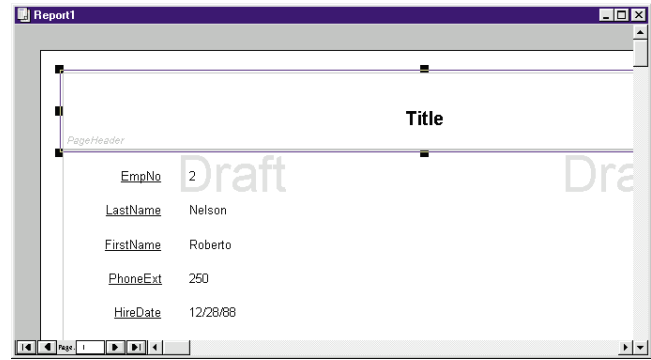
While you're designing your report, you can also enable Draft mode by selecting **View | Draft** from ReportSmith's main menu. You can then switch to Presentation mode — by selecting **View | Presentation** — to check the final appearance of the report as it will be printed.

The Draft Mode Options dialog box (see [Figure 2](#)) lets you control the number of records to display, and whether to show "placeholder" outlines for graphic images. Reducing the volume of data (number of records) and showing light gray outlines instead of the graphic image(s) improves your system's performance as you create and modify your report. As you're working in Draft mode, ReportSmith will include the "Draft" watermark (see [Figure 3](#)).



**Figure 2:** The Draft Mode Options dialog box.

A correctable problem (specific to ReportSmith 3.0) occurs when ReportSmith RunTime is used with a report created in Draft mode — the Draft watermark continues to appear. The solution is to open the report, select **View | Presentation** to place the report in Presentation mode, and then save it. Now, subsequent RunTime operation will not include the watermark.



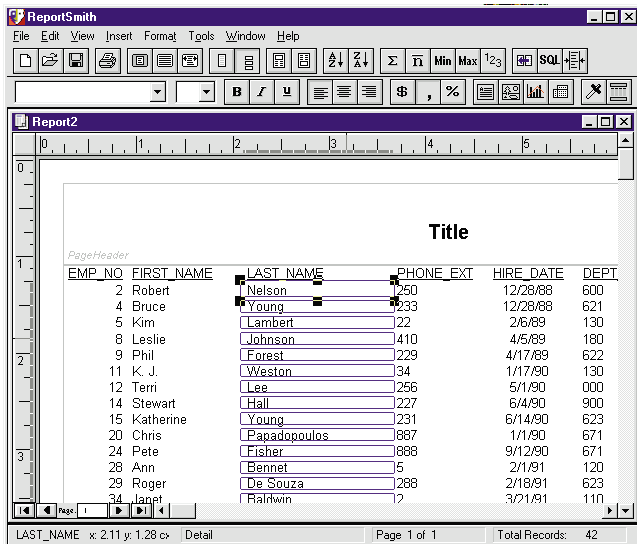
**Figure 3 (Top):** This design-time report shows the Draft watermark. **Figure 4 (Bottom):** This columnar report is shown in Column editing mode. The selected data from the SIZE column can now be moved horizontally.

### Secret 3: Column Editing and Form (Field) Editing

The next secret has to do with the arrangement of data on the report. Let's say you want to move various columns or fields to different positions on your report. To move columns, you can enable Column editing mode and relocate columns and their labels horizontally across the report page. As you might expect, this is the default mode for columnar reports. In addition, it's the default mode for crosstab and label report types. [Figure 4](#) shows a columnar report in Column editing mode.

Form editing or Field editing mode (see [Figure 5](#)) allows you to move individual fields (not entire columns) freely around the report page (horizontally and vertically). Use Field editing mode when inserting fields into headers and footers. Please note that when moving values in Field editing mode, the associated field labels do not automatically move along with their respective columns. (However, if you hold down **Shift** and click on the field and its label, you can drag both to a new location). Field editing mode is the default for form reports.

Here's another secret I learned the hard way: don't switch to Field editing mode, and then drag and drop from a column to the header or footer. For some reason, this technique may



**Figure 5:** The report from Figure 4 in Field editing mode. The selected field can be moved freely around the report.

damage the report, and I know of no way to repair it. Instead of moving the column, insert it using the Insert Field dialog box. You can delete the column if it's no longer necessary.

#### Secret 4: Don't Push the Done Button Unless You Are

A critical (and often overlooked) part of ReportSmith is its Report Query interface.

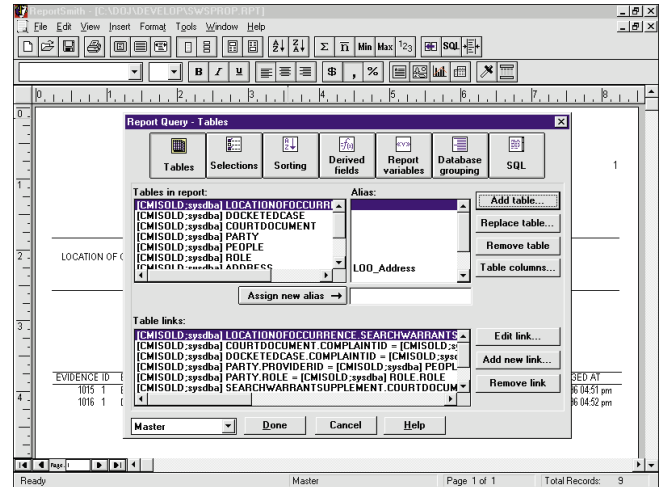
When developing a report, I prefer to see it with actual data as I fine tune the formatting. The good news is that ReportSmith is second-to-none when it comes to displaying "live data" in a report. The bad news is that you can spend time waiting for ReportSmith to reformat the report's display, *after* executing the query. Fortunately, you don't have to wait because you can enable Draft mode (as already described).

Retrieving the data and formatting the report are really different activities. ReportSmith is designed such that creating the report query (i.e. identifying the tables, columns, links, selection and sorting criteria; creating derived fields, report variables, and database groups; and accessing the actual SQL code) can be done from one dialog box.

The Report Query - Tables dialog box has seven main buttons across its top. By selecting the appropriate button (labeled accordingly), you can instantly change the display (see Figure 6). This is an excellent interface — and while you don't need to use every aspect — it features everything you need for query definition. And, there's no hesitation between these dialog boxes.

Also, the Report Query - Tables dialog box features five other important buttons:

- Add table
- Replace table
- Table columns
- Edit link
- Add new link



**Figure 6:** The Report Query - Tables dialog box.

These provide access to additional dialog boxes. Report performance, by the way, can be improved by intelligently excluding unnecessary table columns.

Specify the query as completely as you can. Only then should you begin work on the report's format. Pressing the Done button will require the query to run and ReportSmith to format the report. The more times you access the Report Query - Tables dialog box, the more times the query will run and ReportSmith will reformat the report.

The bottom line? Don't thrash between the query and the report.

#### Secret 5: It's Not Where You Go, It's the Named Connections

This quasi-secret is explained in the documentation (Chapter 1 of the ReportSmith 2.5 *User's Guide*, or ReportSmith 3.0 *Creating Reports*). Since almost nobody reads the manuals, here is my take on it.

This issue seems especially troublesome for those familiar with Borland Database Engine (BDE) aliases, and is probably related to ReportSmith connecting to many other sources (e.g. DB2, Oracle, SQL Server, etc.) that do not depend on the BDE. Its "connection model" is based on Named Connections.

The scenario typically involves a developer who creates a report accessing Paradox tables on a development system where the files are located in a system other than the production system. The developer spends some time creating a report (without a Named Connection) and finally gets it looking the way the user wants. So the developer gives a copy of the report (the .RPT file) to the user and — it doesn't work ("SQL Execution Error - Table does not exist"). The developer then learns that without a Named Connection the report is dependent on the file's physical location.

Fortunately, all is not lost. The report can be fixed by replacing each of the tables using the Report Query - Tables dialog box. This report, however, is hard coded and will require

## ON THE COVER

maintenance (i.e. replacing the tables as just described) whenever the location changes. The lesson is: create reports with Named Connections.

### **Conclusion**

ReportSmith is a powerful tool, but its fundamentally different design often keeps developers from taking advantage of its power. Hopefully this article sheds some light on the all-too-often hidden power of ReportSmith, and explains some of the fundamentals that will increase ReportSmith developer productivity. ▲

Jeff Sims is a Northern California-based management and information technology consultant. He can be reached at (415) 359-7851, or 70253.422@compuserve.com.





## INFORMANT SPOTLIGHT

Delphi 1 / Delphi 2 / Object Pascal / Internet



By Gregory Lee

# Delphi Finger

## Adding Winsock Capability to Your Delphi Applications

**T**he Internet. The next wave in the computer revolution — or just a shooting star? The answer may well depend on programs currently being developed. It's an exciting time to be a software developer — potentially a very profitable time, if you can capitalize on the promise of the Internet.

Unfortunately, the books focusing on programming for the Internet have been written for the C/C++ and Java communities. The good news, however, is that you can program for the Internet using Delphi — you just have to be persistent. If you've never used Delphi to call a DLL, nor created a special message handler, you may think Winsock programming is a complex — perhaps even insurmountable — task.

Take your time. Read the information thoroughly to get a broad understanding of the topic. Then reread any portions that remain unclear to you. If you need a more thorough explanation of implementation details, you can always refer to your Delphi manuals. Above all, stick with it. In the end you'll be glad you made the effort.

### DelphiFinger

Perhaps the best way to learn any programming technique is by example. So let's dive right in with one of the simpler Internet

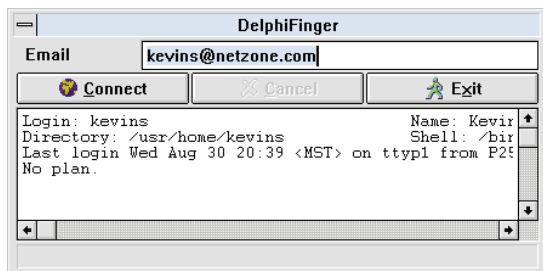
programs — a *Finger* client. Simply put, the Finger protocol allows you to obtain basic information about another Internet user. Supply the Finger client with a valid Internet e-mail address, and the pro-

gram locates the host system, connects to the Finger server, sends it the user's name, and displays the reply. The process is similar to making a phone call, and an effective way to visualize the process. **Figure 1** shows the results of a typical DelphiFinger session.

Before we delve into this implementation of Finger, a caveat is in order: Finger is an optional Internet service that many of the large online services, and some private Internet providers, have decided not to offer. For example, if you enter a CompuServe or an America Online e-mail address you may receive a message such as "Connection refused." In fact, you may not receive any reply. The most likely reason for this is that the system's administrator sees the Finger service as a potential security problem. Rather than limiting the amount of information available, the administrator simply decided not to offer it.

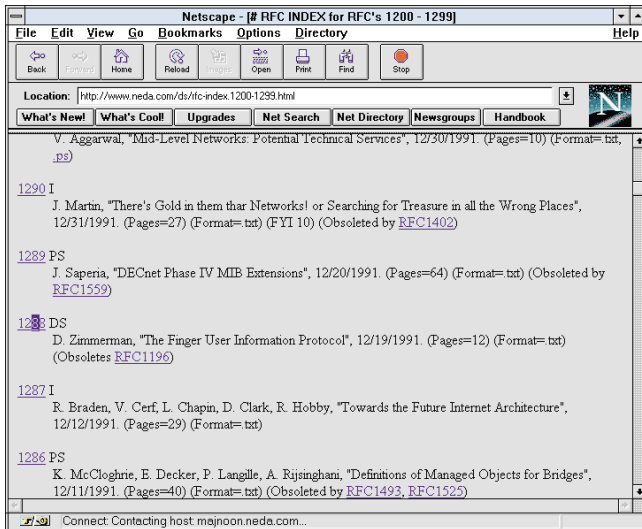
For a complete description of the Finger protocol, I recommend you read RFC 1288, "The Finger User Information Protocol" by D. Zimmerman (see **Figure 2**). RFC is the acronym for *Request For Comment*, and virtually every Internet standard is documented somewhere in an RFC file. Here are some Web sites containing indexes you can browse to find RFC documents:

- <http://www.rasip.etf.hr/rfc/rfc.1.html>
- <http://www.neda.com>
- <http://www.rfc-index.html>



**Figure 1:** A typical DelphiFinger session.





**Figure 2:** The Web site, [www.neda.com](http://www.neda.com), contains an index to all RFC documents.

## The Windows Sockets Library

Winsock is the Windows version of the original Berkeley sockets interface. The sockets interface was developed to provide a simple application programming interface (API) for network applications based on the TCP/IP network protocol.

You don't need to understand what the TCP/IP protocol is or how it works to use Winsock. However, you do need WINSOCK.DLL and a basic knowledge of the functions available. In this article, we'll touch on some of the commonly used functions with enough explanation to get you started. For a more detailed description and a complete list of the functions available, you should consult the *Windows Sockets Specification*. This document is available on the Internet in Windows Help file format, HTML, and a variety of other common formats.

Because Winsock is a normal DLL, you can access its functions as you would with any other DLL — either declare the functions as “externals” and let Delphi handle the details, or use the *LoadLibrary* and *GetProcAddress* functions to do it yourself. The trick, of course, is knowing *which* of Winsock's 44 functions to use and *when* to use them.

In our DelphiFinger project, WINSOCK.PAS contains all the Winsock function prototypes, and the WINSOCK.DLL index numbers for these functions (see [Listing One](#) on page 29). The included file for Delphi 2 users is WSOCK32.PAS. Although most of the functions aren't used, they've all been included in WINSOCK.PAS so you can use that file in future projects without having to worry about adding new prototypes. You will also notice many Winsock constants, record types, and error messages have also been defined in WINSOCK.PAS and WSOCK.PAS.

If you're using Delphi 2, please see [“A Note to Delphi 2 Users”](#) on page 26. This sidebar outlines considerations for those using the 32-bit version of Delphi.

## Getting Started

In a typical DelphiFinger session the user enters an Internet

user name and host name and then clicks on the **Connect** button. This is our cue to initiate the Finger conversation. Before we can do anything useful with the Winsock library, however, we have to call the *WSAStartup* function:

```
function WSAStartup(Version: Word;
                   WSDDataAddr: PWSADATA): Integer;
```

*WSAStartup* takes two arguments: the first indicates the version of Windows Sockets you're looking for, and the second is the address of a *TWSADATA* record buffer. The *TWSADATA* record is a special structure defined in the *Windows Sockets Specification*. It's used by the *WSAStartup* function to store information about the version of WINSOCK.DLL you're using.

Whether you're interested in the information returned in the *TWSADATA* record or not (and we're not), you still have to call *WSAStartup* before calling any other Winsock function. If an error occurs, *WSAStartup* will return an error code. A return value of zero indicates success.

## Searching for the Host Address

We now know Winsock is alive and well, so we can begin the process to locate the remote system and Finger server. To accomplish this we use the Winsock function, *WSAAsyncGetHostByName*. Don't be intimidated by its long name; although it's a mouthful, *WSAAsyncGetHostByName* is a fairly simple function. Given the name of a host system, *WSAAsyncGetHostByName* searches for that system's IP address; you can compare it to calling directory assistance to find a telephone number.

The *Async* part of this function's name refers to the fact that the task is performed in the background. When we call *WSAAsyncGetHostByName*, it initiates the task but returns before the job is done. Winsock will inform the user when the task has completed and the information is ready. This is very convenient because locating the host system's IP address could be time consuming. If the search takes too long, the person using the Finger client may think the program has locked up.

A major benefit of asynchronous processing is that the system can do other things while it's waiting. Remember, Windows is a cooperative multi-tasking system. Calling functions that “take over” the system for an extended period of time is not just bad manners — it violates the prime directive of this operating system.

*WSAAsyncGetHostByName* takes five parameters:

```
function WSAAsyncGetHostByName(Window: HWND;
                               Msg: Word;
                               Name: PChar;
                               Host: PHostInfo;
                               Size: Integer): THandle;
```

- 1) The handle of the window you want Winsock to notify when the process is finished.
- 2) The message you want Winsock to use when it calls you back. Unless you're writing Delphi components, you gen-

erally don't need to worry about window handles and message numbers. Rest assured, however, Delphi is keeping track of all this stuff and you can access it if you really want to. In our example, we want Winsock to call us back through the main form and we'll create a new message number for it to use.

- 3) The name of the host we're trying to find.
- 4) A pointer to the record where Winsock can store the host information.
- 5) The record's size.

Like the *TWSAData* record used with *WSAStartup*, the *THostInfo* record is based on a special structure defined in the *Windows Sockets Specification*. This record is used by *WSAAsyncGetHostByName* to store information about a host system.

If you check the *Windows Sockets Specification*, you'll notice that our *THostInfo* record doesn't quite match the original **hostent** structure. Specifically, a 1K buffer has been added at the end because Winsock needs some extra room to store the data to which the other members of the structure will point.

As long as *WSAAsyncGetHostByName* doesn't immediately run into a brick wall, the function will return a handle for the new task. If the handle is **nil**, something went wrong. We're dead in the water and the only thing left to do is report the error. Luckily, Winsock provides a function to determine exactly what went wrong — *WSAGetLastError*. It takes no arguments and simply returns an error code corresponding to the last Winsock function called. In the DelphiFinger function *WinsockError*, this error code is used to find an appropriate message string and display the error to the user.

For now, let's assume everything has performed without a hitch: *WSAStartup* returned cleanly and our *WSAAsyncGetHostByName* call returned something other than **nil**. Now what?

We wait.

Somewhere in the background the wheels are turning. Eventually, the host name lookup will finish and Winsock will tell us the result. We'll intercept the message with a special message handler, check out the new IP address, and proceed to the next step.

There's more than one way to create a message handler in Delphi. The simplest way, however, is to create the procedure somewhere in the program's body. Then, you declare the procedure, along with the message it will intercept, in the **protected** section of the form's definition. In this case the message will be returned to *Form1*, so that's where we've placed the declaration.

Message handlers always receive a record structure containing the *wParam* and *lParam* values associated with a typical Windows message callback. For Winsock lookup

## A Note to Delphi 2 Users

Some minor changes are required to make DelphiFinger work with Delphi 2:

- Since the size of the *Integer* data type in Delphi has increased from 16- to 32-bits, all the integers used in the WINSOCK records and function prototypes must be changed to the new type *Smallint*. Items already declared as *Smallint* or *Longint* should be left as they are.
- Because the 32-bit version of WINSOCK is called WSOCK32, this name must be changed in each DLL function prototype.
- The default Delphi calling convention has been changed. This means you must add the **stdcall** declaration to the end of each DLL function prototype.
- Delphi 2 now includes a WINSOCK.PAS unit. Therefore, you should rename the WINSOCK.PAS file included to WSOCK32.PAS and make the corresponding change to the **uses** section of FINGER.PAS.

— Gregory Lee

functions, *wParam* contains the task handle returned by the original function call and the high word of *lParam* indicates the task completion status. If the *WSAAsyncGetHostByName* task succeeds, the status code is zero and our record will contain at least one IP address for the host system.

### Locating the Finger Server

Great! We have the host system's IP address. Now we need to find the location of the Finger server. In our phone call analogy, this step is similar to using a company directory to find an extension number. To accomplish this, we use the Winsock function *WSAAsyncGetServByName*:

```
function WSAAsyncGetServByName(Window: HWND;
                               Msg: Word;
                               ServiceName: PChar;
                               ProtocolName: PChar;
                               Server: PServerInfo;
                               Size: Integer): THandle;
```

As you've probably already guessed, *WSAAsyncGetServByName* is another callback function and takes six parameters. The first two are a window handle and a message value. The third and fourth parameters are strings indicating the name of the service we're interested in, and the name of the underlying protocol (in this case the values should be **finger** and **tcp**). The fifth parameter is the address of a record where Winsock can store the server information, and the sixth parameter indicates the record's size.

As was the case with *THostInfo*, the *TServerInfo* record is based on a special structure defined in the *Windows Sockets Specification*. *WSAAsyncGetServByName* uses this record to store information about a server. Again we've added a 1K buffer to the end of the record for Winsock to store the data it collected.

When the function call is made, *WSAAsyncGetServByName* sets the new task in motion and quickly returns. Just as Winsock did with the previous function,

*WSAAsyncGetServByName* will send a message to our form when the task is complete and we'll have another special message handler waiting to intercept it.

### Making the Call

At this point we know the location of the host's address and where to find the Finger server. Of course, before we can actually communicate with the Finger server, we must open a line for communication.

In Winsock, this process requires three steps. Creating the socket itself is accomplished by calling the Winsock function *socket*:

```
function socket(AddressFormat: Integer;
               SocketType: Integer;
               Protocol: Integer): Integer;
```

The *socket* function takes three parameters: the address family, communication type, and the protocol to use with the new socket. For this application (and just about any other you'll probably write), we use the Internet address family, creating a stream type socket, and using the TCP protocol. For each of these arguments, constant values are defined in `WINSOCK.PAS`, and, as you can see, we're using them in the call to *socket*. If the *socket* function is successful, it returns a valid socket handle. Otherwise, it returns `INVALID_SOCKET`.

Before we can place the call, we must tell Winsock to handle this socket 100 percent asynchronously. The Winsock function designed for this purpose is *WSAAsyncSelect*:

```
function WSAAsyncSelect(Socket: THandle;
                       Window: HWnd;
                       Msg: Word;
                       Event: LongInt): Integer;
```

*WSAAsyncSelect* takes four parameters: the socket descriptor; the handle of the window that should receive notification messages; the message we want Winsock to use; and an event bitmask specifying the events we want handled in this way.

### There Can Be Only One

Notice that although *WSAAsyncSelect* can apply to various operations — connecting, reading, writing, closing the socket, etc. — you can only specify a single callback message. This means that from this point, only one message handling procedure can handle all the messages.

You might guess that by calling *WSAAsyncSelect* a number of times, passing a different message and event bitmask pair each time, you could configure a whole series of message handling procedures tied to different events — this sounds logical. Unfortunately, that's not the way it works.

Each call to *WSAAsyncSelect* completely resets the status of the socket's event handler. For example, if you call *WSAAsyncSelect* once with the event bitmask set for **read** events and then immediately call it a second time with the event bitmask set for **write** events, the second call will reset the event handler and only **write** events will be handled asynchronously.

An Internet Glossary	
<b>Address Format</b>	Describes the format of an Internet address. The only address format currently supported by Winsock is the ARPA Internet address format.
<b>Client</b>	An application that initiates a connection and communicates with a server.
<b>HTML</b>	Hypertext Markup Language. The language used to create pages on the World Wide Web. Many public domain documents and specifications are currently available in this format.
<b>IP Address</b>	A number used to locate a user or service on the Internet. In the current specification, this is a 32-bit number and is commonly displayed in dotted decimal notation. Dotted decimal notation disseminates the address into individual bytes and lists the decimal equivalent of each byte. For example, the IP address \$FFFF0001 is represented in dotted decimal form as 255.255.0.1.
<b>Server</b>	An application that provides a service; it does not generally initiate connections. Instead, the server waits at a well-known location and waits for client applications to connect with it.
<b>Socket</b>	The line of communication used on the Internet to communicate with another user or service.
<b>TCP/IP</b>	Transmission Control Protocol/Internet Protocol. This is the basic, underlying protocol suite used to control the flow of information over the Internet. For a detailed description see the documents RFC793 (TCP) and RFC791 (IP).
<b>Winsock</b>	The Windows implementation of the original Berkeley sockets interface.
<b>WINSOCK.DLL</b>	The library containing the functions described in the <i>Windows Socket Specification</i> .
<b>WSOCK32.DLL</b>	The 32-bit version of WINSOCK.DLL.

### Which Is Which?

If we can't assign separate event handlers for the different operations, how do we distinguish callbacks? When a Winsock makes a notification call, a portion of the record it passes to our callback routine contains the bitmask corresponding to the reason for the callback. For example, when incoming data is waiting and ready to be received, the bitmask passed into our callback routine will contain the constant value *FD\_READ*.

### Making the Connection

We've set the stage with *WSAAsyncSelect*. Now we're ready to place the call. The Winsock function *connect* is designed to do just that:

```
function connect(Socket: THandle;
                Address: PSocketAddress;
                Size: Integer): Integer;
```

The *connect* function takes three parameters:

- 1) our socket handle
- 2) the address of a *TSocketAddress* record
- 3) the size of the *TSocketAddress* record

The *TSocketAddress* record is based on a special structure defined in the *Windows Sockets Specification*. The address member is the IP address we want to connect to and the port number is the Finger server's port number. When *connect* is called, the process is initiated and it returns immediately with

an error code or a zero. A zero indicates the task has been started. Eventually, we'll receive a message indicating whether *connect* has succeeded.

When the call goes through, Winsock sends the message *MsgAsyncEvent* to *Form1* and the low word of *lParam* contains the constant *FD\_CONNECT*. It's now time to communicate with the Finger server.

### Hangin' on the Telephone

The Finger server's job is pretty boring — it just sits there, listening and waiting for someone to call. Periodically, the phone rings and the Finger server picks up the line. To save time and keep things simple, the Finger server doesn't bother saying "Hello." The client is responsible for keeping things going by sending a query to the server. The *query* in this case is just the name of the user we're interested in, followed by a carriage return and linefeed characters.

The Winsock function for sending our query is called *send*:

```
function send(Socket: THandle;
             Buffer: PChar;
             Size: Integer;
             Flags: Integer): Integer;
```

The *send* function requires the socket descriptor, a pointer to the message we want to send, the message's length, and an options flag. If everything looks good, *send* starts the task and returns immediately. If something goes drastically wrong, *send* returns an error message. Eventually, when the task is complete, our message handler receives yet another callback.

This time Winsock sends *Form1* another *MsgAsyncEvent* message. Now, however, the low word of *lParam* will contain *FD\_WRITE*. Since there is no additional data to send, we'll ignore the *FD\_WRITE* callback.

It's up to the Finger server on the host system to process the query and send us a reply. When the reply arrives, we receive another *MsgAsyncEvent* message. This time the low word of *lParam* contains *FD\_READ*. The Winsock function *recv* is used to obtain the reply:

```
function recv(Socket: THandle;
             Buffer: PChar;
             Size: Integer;
             Flags: Integer): Integer;
```

### Closing the Connection

We have the reply and are now done talking with the Finger server. It's time to hang up the phone. Typically, the Finger server will hang up its end of the line before we've even had a chance to process all the data it's sent. When that happens, we receive an *MsgAsyncEvent* message for the *FD\_CLOSE* event and, after we've handled any remaining data in the buffer, we can close our end of the connection.

Winsock provides the function, *closesocket*, to close the line and free all the resources associated with the socket. If it's successful, *closesocket* returns zero. If not, it returns *SOCKET\_ERROR*.

Even if *closesocket* works, we're still not quite off the hook. There's one last function we're obligated to call: *WSACleanup*. This function de-registers the Finger client and allows Winsock to release any resources it has allocated on our behalf.

### Conclusion

At this point you might be thinking, "If DelphiFinger is one of the simplest Internet applications, I don't even want to know what's involved in creating something like an e-mail client." Our little Finger client *is* one of the more basic Internet applications, but it uses many of the core Winsock functions. On the surface, an e-mail or FTP client may seem exponentially more complex, but from a programming standpoint, it's really not. Once you understand the fundamentals, writing Internet applications in Delphi is not very difficult. ▲

*The demonstration files referenced in this article are available on the Delphi Informant Works CD located in INFORM96\AUG\DI9608GL.*

Gregory Lee is a programmer with over 15 years of experience writing applications and development tools. He is currently the president of Software Avenue Inc., which has just released a package for Delphi developers called the "Internet Developer's Kit" (see the "Tools" section of the June 1996 *Delphi Informant*). Greg can be reached by e-mail at 76455.3236@compuserve.com.



**Begin Listing One — WINSOCK.PAS**

```

unit Winsock;

interface

uses
  WinTypes, WinProcs, Messages;

const
  { New messages for asynchronous callbacks }
  MsgHostInfoReady = WM_USER+0;
  MsgServerInfoReady = WM_USER+1;
  MsgAsyncEvent = WM_USER+2;

  { Event bitflags defined in
    Windows Sockets Specification }
  FD_READ = 1;
  FD_WRITE = 2;
  FD_OOB = 4;
  FD_ACCEPT = 8;
  FD_CONNECT = 16;
  FD_CLOSE = 32;
  FD_ALL = 63;

  { Socket types currently supported }
  SOCK_STREAM = 1;
  SOCK_DGRAM = 2;

  { Socket option values }
  SO_DEBUG = 1;
  SO_ACCEPTCONN = 2;
  SO_REUSEADDR = 4;
  SO_KEEPALIVE = 8;
  SO_DONTROUTE = 16;
  SO_BROADCAST = 32;

  SO_USELOOPBACK = 64;
  SO_LINGER = 128;
  SO_OOBINLINE = 256;
  SO_DONTLINGER = 65407;
  SO_SNDBUF = 4097;
  SO_RCVBUF = 4098;
  SO_SNDLOWAT = 4099;
  SO_RCVLOWAT = 4100;
  SO_SNDTIMEO = 4101;
  SO_RCVTIMEO = 4102;
  SO_ERROR = 4103;
  SO_TYPE = 4104;

  { Protocol ID numbers }
  IPPROTO_IP = 0;
  IPPROTO_ICMP = 1;
  IPPROTO_GGP = 2;
  IPPROTO_TCP = 6;
  IPPROTO_PUP = 12;
  IPPROTO_UDP = 17;
  IPPROTO_IDP = 22;
  IPPROTO_ND = 77;
  IPPROTO_RAW = 255;
  IPPROTO_MAX = 256;

  { Other WINSOCK constants }
  AF_INET = 2;
  INVALID_SOCKET = -1;
  SOCKET_ERROR = -1;
  INADDR_ANY = 0;
  MAXGETHOSTSTRUCT = 1024;
  SHUTDOWN_RECV = 0;
  SHUTDOWN_SEND = 1;
  SHUTDOWN_BOTH = 2;

type
  { New basic pointer types }
  PChar = ^PChar;
  PInteger = ^Integer;
  PLongInt = ^LongInt;
  PPLongInt = ^PLongInt;

  { Record types used by WINSOCK functions }
  TWSAData = record
    Version: Word;
    HighVersion: Word;
    Description: array[0..255] of char;
    Status: array[0..127] of char;
    MaxSockets: ShortInt;
    MaxUdpDatagramSize: ShortInt;
    VendorInfo: PChar;
  end;

  THostInfo = record
    Name: PChar;
    AliasList: PPChar;
    AddressType: Integer;
    AddressSize: Integer;
    AddressList: PPLongInt;
    Reserved: array[1..MAXGETHOSTSTRUCT] of char;
  end;

  TServerInfo = record
    Name: PChar;
    Aliases: PPChar;
    Port: Integer;
    Protocol: PChar;
    Reserved: array[1..MAXGETHOSTSTRUCT] of char;
  end;

  TProtocolInfo = record
    Name: PChar;

    Aliases: PPChar;
    ProtocolID: Integer;
    Reserved: array[1..MAXGETHOSTSTRUCT] of char;
  end;

  TSocketAddress = record
    Family: Integer;
    Port: Word;
    Address: LongInt;
    Unused: array[1..8] of char;
  end;

  TSocketList = record
    Count: Integer;
    DescriptorList: array[1..64] of Integer;
  end;

  TTimeValue = record
    Sec: LongInt;
    uSec: LongInt;
  end;

  { New WINSOCK pointer types }
  PWSAData = ^TWSAData;
  PHostInfo = ^THostInfo;
  PServerInfo = ^TServerInfo;
  PProtocolInfo = ^TProtocolInfo;
  PSocketAddress = ^TSocketAddress;
  PSocketList = ^TSocketList;
  PTimeValue = ^TTimeValue;

```

```
{ Special type for WINSOCK error table }
TErrorMessage = record
    ErrorCode: Integer;
    Text: String[50];
end;

const
{ Message table - maps WINSOCK error codes to
messages strings }
WinsockMessage: array[0..50] of TErrorMessage = (
    (ErrorCode:10004; Text:'Interrupted system call'),
    (ErrorCode:10009; Text:'Bad file number'),
    (ErrorCode:10013; Text:'Permission denied'),
    (ErrorCode:10014; Text:'Bad address'),
    (ErrorCode:10022; Text:'Invalid argument'),
    (ErrorCode:10024; Text:'Too many open files'),
    (ErrorCode:10035; Text:'Operation would block'),
    (ErrorCode:10036; Text:'Operation now in progress'),
    (ErrorCode:10037;
        Text:'Operation already in progress'),
    (ErrorCode:10038;
        Text:'Socket operation on non-socket'),
    (ErrorCode:10039; Text:'Destination address required'),
    (ErrorCode:10040; Text:'Message too long'),
    (ErrorCode:10041;
        Text:'Wrong protocol type for socket'),
    (ErrorCode:10042; Text:'Bad protocol option'),
    (ErrorCode:10043; Text:'Protocol not supported'),
    (ErrorCode:10044; Text:'Socket type not supported'),
    (ErrorCode:10045;
        Text:'Operation not supported on socket'),
    (ErrorCode:10046; Text:'Protocol family not supported'),
    (ErrorCode:10047; Text:
        'Address family not supported by protocol family'),
    (ErrorCode:10048; Text:'Address already in use'),
    (ErrorCode:10049;
        Text:'Can't assign requested address'),
    (ErrorCode:10050; Text:'Network is down'),
    (ErrorCode:10051; Text:'Network is unreachable'),
    (ErrorCode:10052;
        Text:'Network dropped connection or reset'),
    (ErrorCode:10053;
        Text:'Software caused connection abort'),
    (ErrorCode:10054; Text:'Connection reset by peer'),
    (ErrorCode:10055; Text:'No buffer space available'),
    (ErrorCode:10056; Text:'Socket is already connected'),
    (ErrorCode:10057; Text:'Socket is not connected'),
    (ErrorCode:10058;
        Text:'Can't send after socket shutdown'),
    (ErrorCode:10059;
        Text:'Too many references, can't splice'),
    (ErrorCode:10060; Text:'Connection timed out'),
    (ErrorCode:10061; Text:'Connection refused'),
    (ErrorCode:10062;
        Text:'Too many levels of symbolic links'),
    (ErrorCode:10063; Text:'File name too long'),
    (ErrorCode:10064; Text:'Host is down'),
    (ErrorCode:10065; Text:'No route to Host'),
    (ErrorCode:10066; Text:'Directory not empty'),
    (ErrorCode:10067; Text:'Too many processes'),
    (ErrorCode:10068; Text:'Too many users'),
    (ErrorCode:10069; Text:'Disc quota exceeded'),
    (ErrorCode:10070; Text:'Stale NFS file handle'),
    (ErrorCode:10071;
        Text:'Too many levels of remote in path'),
    (ErrorCode:10091;
        Text:'Network subsystem is unavailable'),
    (ErrorCode:10092;
        Text:'Incompatible version of WINSOCK.DLL'),
    (ErrorCode:10093;
        Text:'Successful WSASStartup not yet performed'),
```

```
(ErrorCode:11001; Text:'Host not found'),
(ErrorCode:11002;
    Text:'Non-Authoritative Host not found'),
(ErrorCode:11003; Text:'Non-Recoverable error: FORMERR,
    REFUSED, NOTIMP'),
(ErrorCode:11004; Text:'Valid name,
    no data record of requested type'),
(ErrorCode:0; Text:'Unrecognized error code')
);

{ Prototypes for WINSOCK.DLL functions listed
alphabetically }
function accept(Socket: THandle; Address: PSocketAddress;
    Size: Integer): Integer;
function bind(Socket: THandle; Address: PSocketAddress;
    Size: Integer): Integer;
function closesocket(Socket: THandle): Integer;
function connect(Socket: THandle;
    Address: PSocketAddress; Size: Integer): Integer;
function gethostbyaddr(Address: PLongInt; Size: Integer;
    AddressFamily: Integer): PHostInfo;
function gethostbyname(Name: PChar): PHostInfo;
function gethostname(Name: PChar;
    Size: Integer): Integer;
function getpeername(Socket: THandle;
    Peer: PSocketAddress; Size: PInteger): Integer;
function getprotobynumber(
    ProtocolID: Integer): PProtocolInfo;
function getprotobyname(Name: PChar): PProtocolInfo;
function getservbyport(Port: Integer;
    Protocol: PChar): PServerInfo;
function getservbyname(Name: PChar;
    Protocol: PChar): PServerInfo;
function getsockname(Socket: THandle;
    Address: PSocketAddress; Size: PInteger): Integer;
function getsockopt(Socket: THandle; Level: Integer;
    OptionFlag: Integer; OptionValue: PChar;
    Size: PInteger): Integer;
function htonl(Address: LongInt): LongInt;
function htons(Address: Integer): Integer;
function inet_addr(IP: PChar): LongInt;
function inet_ntoa(Address: LongInt): PChar;
function ioctlsocket(Socket: THandle; Command: LongInt;
    var Argument): Integer;
function listen(Socket: THandle;
    BackLog: Integer): Integer;
function ntohl(Address: LongInt): LongInt;
function ntohs(Address: Integer): Integer;
function recv(Socket: THandle; Buffer: PChar;
    Size: Integer; Flags: Integer): Integer;
function recvfrom(Socket: THandle; Buffer: PChar;
    Size: Integer; Flags: Integer;
    Address: PSocketAddress; Size: PInteger): Integer;
function select(Unused: Integer; ReadList: PSocketList;
    WriteList: PSocketList; CheckList: PSocketList;
    Timeout: PTimeValue): LongInt;
function send(Socket: THandle; Buffer: PChar;
    Size: Integer; Flags: Integer): Integer;
function sendto(Socket: THandle; Buffer: PChar;
    Size: Integer; Flags: Integer;
    Address: PSocketAddress;
    AddressSize: Integer): Integer;
function setsockopt(Socket: THandle; Level: Integer;
    OptionFlag: Integer; NewValue: PChar;
    Size: Integer): Integer;
function shutdown(Socket: THandle;
    Options: Integer): Integer;
function socket(AddressFormat: Integer;
    SocketType: Integer; Protocol: Integer): Integer;
function WSAAsyncGetHostByAddr(Window: HWnd; Msg: Word;
    Address: PLongInt; Size: Integer;
    ProtocolFamily: Integer; Host: PHostInfo;
```

```

    Size: Integer): THandle;
function WSAAsyncGetHostByName(Window: HWND; Msg: Word;
    Name: PChar; Host: PHostInfo; Size: Integer): THandle;
function WSAAsyncGetProtoByName(Window: HWND; Msg: Word;
    ProtocolName: PChar; Protocol: PProtocolInfo;
    Size: Integer): THandle;
function WSAAsyncGetProtoByNumber(Window: HWND;
    Msg: Word; ProtocolID: Integer;
    Protocol: PProtocolInfo; Size: Integer): THandle;
function WSAAsyncGetServByName(Window: HWND; Msg: Word;
    ServiceName: PChar; ProtocolName: PChar;
    Server: PServerInfo; Size: Integer): THandle;
function WSAAsyncGetServByPort(Window: HWND; Msg: Word;
    Port: Integer; ProtocolName: PChar;
    Server: PServerInfo; Size: Integer): THandle;
function WSAAsyncSelect(Socket: THandle; Window: HWND;
    Msg: Word; Event: LongInt): Integer;
function WSACancelAsyncRequest(
    TaskHandle: THandle): Integer;
function WSACancelBlockingCall: Integer;
function WSACleanup: Integer;
function WSAGetLastError: Integer;
function WSAIsoBlocking: Boolean;
function WSASetBlockingHook(
    BlockingFunction: TFarProc): TFarProc;
procedure WSASetLastError(ErrorCode: Integer);
function WSAStartup(Version: Word;
    WSDataAddr: PWSADATA): Integer;
function WSAUnhookBlockingHook: Integer;

```

**implementation**

```

{ External directives WINSOCK.DLL routines listed by
  index number }
function accept;           external 'WINSOCK' index 1;
function bind;           external 'WINSOCK' index 2;
function closesocket;   external 'WINSOCK' index 3;
function connect;       external 'WINSOCK' index 4;
function getpeername;   external 'WINSOCK' index 5;
function getsockname;   external 'WINSOCK' index 6;
function getsockopt;    external 'WINSOCK' index 7;
function htonl;         external 'WINSOCK' index 8;
function htons;         external 'WINSOCK' index 9;
function inet_addr;     external 'WINSOCK' index 10;
function inet_ntoa;     external 'WINSOCK' index 11;
function ioctlsocket;   external 'WINSOCK' index 12;
function listen;        external 'WINSOCK' index 13;
function ntohl;         external 'WINSOCK' index 14;
function ntohs;        external 'WINSOCK' index 15;
function recv;          external 'WINSOCK' index 16;
function recvfrom;      external 'WINSOCK' index 17;
function select;        external 'WINSOCK' index 18;

```

```

function send;           external 'WINSOCK' index 19;
function sendto;        external 'WINSOCK' index 20;
function setsockopt;    external 'WINSOCK' index 21;
function shutdown;      external 'WINSOCK' index 22;
function socket;        external 'WINSOCK' index 23;
function gethostbyaddr; external 'WINSOCK' index 51;
function gethostbyname; external 'WINSOCK' index 52;
function getprotobyname; external 'WINSOCK' index 53;
function getprotobynumber;
    external 'WINSOCK' index 54;
function getservbyname;
    external 'WINSOCK' index 55;
function getservbyport;
    external 'WINSOCK' index 56;
function gethostname;
    external 'WINSOCK' index 57;
function WSAAsyncSelect;
    external 'WINSOCK' index 101;
function WSAAsyncGetHostByAddr;
    external 'WINSOCK' index 102;
function WSAAsyncGetHostByName;
    external 'WINSOCK' index 103;
function WSAAsyncGetProtoByNumber;
    external 'WINSOCK' index 104;
function WSAAsyncGetprotoByName;
    external 'WINSOCK' index 105;
function WSAAsyncGetServByPort;
    external 'WINSOCK' index 106;
function WSAAsyncGetServByName;
    external 'WINSOCK' index 107;
function WSACancelAsyncRequest;
    external 'WINSOCK' index 108;
function WSASetBlockingHook;
    external 'WINSOCK' index 109;
function WSAUnhookBlockingHook;
    external 'WINSOCK' index 110;
function WSAGetLastError;
    external 'WINSOCK' index 111;
procedure WSASetLastError;
    external 'WINSOCK' index 112;
function WSACancelBlockingCall;
    external 'WINSOCK' index 113;
function WSAIsoBlocking;
    external 'WINSOCK' index 114;
function WSAStartup;
    external 'WINSOCK' index 115;
function WSACleanup;
    external 'WINSOCK' index 116;

```

end.  
**End Listing One**





By *Keith Wood*

## Completing the Component

### Adding Property Editors and Online Help to *TLabelEffect*

Last month we extended the *TLabel3D* component to make it more flexible and enable it to be rotated [see Keith Wood's article "3-D Labels with a Twist" in the July 1996 *Delphi Informant*]. (The original *TLabel3D* component was introduced by Jim Allen and Steve Teixeira in the June 1995 issue of *Delphi Informant* in their article "A 3-D Label Component.") The result was the *TLabelEffect* component. As promised, this month we'll finish the component production cycle by providing property editors for two of its properties and by adding Help to the component.

Property editors provide a graphical way of changing a property's value. Delphi has many examples, including the Font and Color dialog boxes. In this article, we'll examine ways of prototyping an editor before configuring it for use.

An integrated Help file enables users to get information when they need it. A Help template is provided with this article for use with other components, and includes the appropriate styles and images to blend with Delphi's Help system.

#### Graphical Editing

Adding a property editor to a component is easy, and provides a simple way to modify the value of the property.

The basic requirement of a property editor is that it takes the current value of a particular property, and displays it visually. It must then allow changes to the property to be transferred back to the component.

The Delphi documentation covering property editors is relatively sparse (pages 38-43 of the *Component Writer's Guide*), with the online

Help simply duplicating the text in the book. With a bit of experimentation, however, this is enough to produce what we want.

Typically, a property editor is supplied for those properties that have a visual content. In our case, we are writing editors for the *EffectStyle* and *ColourScheme* properties in our *TLabelEffect* component. Since these editors are similar, we'll examine only the *EffectStyle* editor in detail.

Before generating the property editor itself, we'll construct a prototype of the final product to ensure it works properly.

In this case, our property editor is simply a dialog box. To communicate with this dialog box, it must have a value that we can initially set and later read. (Sounds like a property!)

Looking at the declaration of the form, we can see that it's a class derived from *TForm*. Components are also classes, and have properties, so we should be able to add our property to the form in the same way. First, we create a new form, setting its *BorderStyle* property to *bsDialog*, and chang-



ing the *Caption* to *Effect Style*. Then we add a number of *TLabelEffect* components (each with different *EffectStyles* we might want to use), an OK button, and a **Cancel** button.

The buttons are *TBitBtn* components, so we merely have to set the *Kind* property appropriately (e.g. *mrOK* and *mrCancel*) for them to function as required. They will automatically close the dialog box when pressed, and return a value indicating which button was selected.

We then add the form's property directly into the code. In the **private** section of the form's definition, add our internal variable *FEffectStyle* and declare an access method for it. Then in the **public** section declare the property itself, *EffectStyle*, and how to access it. A **write** method is required since we want to update the form to reflect the value given.

We also need to override the constructor method since we are setting an initial value. The property is not declared as **public** because it will not be available at design time (as would be likely if we were defining a component).

The method chosen to highlight the currently selected style is to toggle the transparency of the appropriate *TLabelEffect* component. Initially, all the labels are set to have a *Color* of *clBackground* (the standard background color for Windows), and to be *Transparent*. The **write** method for our property must then make the correct label opaque while ensuring that all the others are transparent.

To do this, we use the list of components maintained by the form itself, *Components[]*. We cycle through all the components on the form, and for each one that is a *TLabelEffect* we compare its *EffectStyle* with our new property. If they match, we set that label's *Transparent* property to *False*. Otherwise we set it to *True* (see [Figure 1](#)).

We also need to react to the user selecting an *EffectStyle* to use. The user does this by clicking on the one he or she wants. So we select all the *TLabelEffect* components, go to the Events page in the Object Inspector, and enter a name for a method to call when a label is clicked, e.g. *LabelEffectClick*.

In the code we simply want to set the form's *EffectStyle* property to the value of the one clicked. Since any one of the labels could trigger this method, we must use the *Sender* parameter to provide access to the correct label.

We ensure that it is treated as a *TLabelEffect* object with the **as** operator, and set the property accordingly:

```
{ Set the form's EffectStyle property based on the
  value of the label selected }
procedure TEffectStylePropEd.LabelEffectClick(
  Sender: TObject);
begin
  EffectStyle := (Sender as TLabelEffect).EffectStyle;
end;
```

It would also be nice if the dialog box responded to a double-click by selecting the style and closing (as if the OK but-

```
{ Set the EffectStyle property of the form and highlight
  appropriate label }
procedure TEffectStylePropEd.SetEffectStyle(
  EsStyle: TEffectStyle);
var
  i: Integer;
begin
  if FEffectStyle <> EsStyle then
    begin
      FEffectStyle := EsStyle;
      for i := 0 to ComponentCount - 1 do
        if Components[i] is TLabelEffect then
          with Components[i] as TLabelEffect do
            Transparent := (EffectStyle <> EsStyle);
        end;
    end;
end;
```

**Figure 1:** The **write** method for the *EffectStyle* property.

ton had been pressed). Since a *Click* event is triggered as part of a double-click, the correct style has already been set.

All we need to do is simulate pressing the OK button, which is done by invoking that button's *Click* method.

As before, we select all the *TLabelEffect* components and enter a method name in their *OnDblClick* event:

```
procedure TEffectStylePropEd.LabelEffectDblClick(
  Sender: TObject);
begin
  btnOK.Click;
end;
```

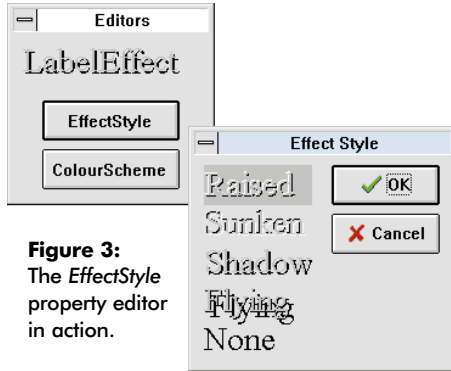
We can now test the editor by attaching it to another form that invokes it on request. This form contains a *TLabelEffect* component (to show the effects of any changes), and a button to request a change to its *EffectStyle* property. Following the example on page 41 of the *Component Writer's Guide* for editing the property as a whole, the button creates a new instance of the editor, sets its initial value to that of the original label, transfers the new value back to the original if OK was selected, and then destroys the editor (see [Figure 2](#)).

Note that all references to the dialog box are contained within a **try..finally** block. This ensures that the dialog box

```
procedure TForm1.BtnEffectStyleClick(Sender: TObject);
var
  EffectStylePropEd: TEffectStylePropEd;
begin
  { Create a dialog box as defined elsewhere }
  EffectStylePropEd :=
    TEffectStylePropEd.Create(Application);
  try
    { Initialize with the current style }
    EffectStylePropEd.EffectStyle := LblEffect.EffectStyle;
    if EffectStylePropEd.ShowModal = mrOK then
      { If OK then update with the new value }
      LblEffect.EffectStyle :=
        EffectStylePropEd.EffectStyle;
  finally
    EffectStylePropEd.Free; { Tidy up }
  end;
end;
```

**Figure 2:** Invoking the prototype *EffectStyle* property editor.

will be destroyed at the end of the method, regardless of any errors that may occur.



**Figure 3:** The *EffectStyle* property editor in action.

We can check that the editor works as expected (see [Figure 3](#)) and that the

*TLabelEffect* component on our main form reflects the style chosen in the prototype editor. (As mentioned, the property editor for the *ColourScheme* property is virtually identical to that for the *EffectStyle* property.)

### Putting It in Place

Now that we have the prototype property editor functioning as required, it's time to build it into a real property editor and register it with Delphi. We start by copying across the prototype property editor developed above. This is the form that we want to display when editing the *EffectStyle* property. We must add the appropriate code so that Delphi can invoke it when the user requests.

In the **interface** section of the unit add a declaration for the property editor itself (as distinct from the form that it displays). This is derived from the *TEnumProperty* class. Many property editor classes are available to use as the base class (see the *Component Writer's Guide* or the online Help).

In this case, the property we are altering is an enumerated type, so this class is appropriate. It is derived from *TOrdinalProperty* and provides all the processing for generating the drop-down list of all possible values of the enumeration.

We are adding to this functionality, and so must override two of the inherited methods: *GetAttributes*, which tells Delphi what capabilities the property editor has; and *Edit*, which will invoke our new form. Also in the **interface** section, we must declare the *Register* procedure that informs Delphi about our changes:

```
{ The property editor for the EffectStyle property }
TEffectStylePropertyEditor = class(TEnumProperty)
public
    { Public declarations }
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
end;
```

The *Register* procedure then appears in the unit's **implementation** section. A call is made to the *RegisterPropertyEditor* procedure, passing four parameters. The first provides information about the property being altered by this editor, and is always a call to the built-in function *TypeInfo*. The second parameter is the type of component to which this editor applies. If this is set to **nil** then the editor will apply to all

properties of the given type, regardless of the component in which it appears.

Next comes the name of the particular property within the component. Again, if this is left blank (a null string), the editor applies to all properties regardless of name (provided they are of the appropriate type). Finally the class of the editor is supplied.

```
{ Define the property editor to Delphi }
procedure Register;
begin
    { Only applies to properties of type TEffectStyle
      in TLabelEffect component }
    RegisterPropertyEditor(TypeInfo(TEffectStyle),
                          TLabelEffect, '',
                          TEffectStylePropertyEditor);
end;
```

The *GetAttributes* function of the property editor supplies Delphi with information about the editor's capabilities. It returns a set of values from the table in [Figure 4](#). Our property editor allows for multiple selection of components, and presents a dialog box and a list of values.

```
{ Tell Delphi that we also have a dialog box for editing }
function TEffectStylePropertyEditor.GetAttributes:
    TPropertyAttributes;
begin
    Result := [paMultiSelect, paDialog, paValueList];
end;
```

Note that the dialog box is all that we've added. The rest is handled by the *TEnumProperty* class from which we inherited, making good use of Delphi's object-oriented design.

The *Edit* method of the property editor is the interface between the user double-clicking on a property and showing our dialog box. As for the button in the prototype example, it creates a new instance of the dialog box, ini-

Attribute	Meaning
<i>paValueList</i>	The editor can give a list of enumerated values. The <i>GetValues</i> method builds the list.
<i>paSubProperties</i>	The property is an object with subproperties that can display. The <i>GetProperties</i> method handles the subproperty lists.
<i>paDialog</i>	The editor can display a dialog box for editing the entire property. The <i>Edit</i> method opens the dialog box.
<i>paMultiSelect</i>	The property should appear when the user selects multiple components. Most property editors allow multiple selection. The notable exception is the editor for the <i>Name</i> property.
<i>paAutoUpdate</i>	Calls the <i>SetValue</i> method after each change in the value, rather than after the entire value is approved.
<i>paSortList</i>	The Object Inspector should sort the list of values alphabetically.
<i>paReadOnly</i>	The property is read-only in the Object Inspector. This is used for sets and fonts that the user cannot enter directly.

**Figure 4:** Property editor attributes (from the Delphi online Help).

```

{ The procedure invoked when the property is
  double-clicked }
procedure TEffectStylePropertyEditor.Edit;
var
  EffectStylePropEd: TEffectStylePropEd;
begin
  { Create a dialog box as defined above }
  EffectStylePropEd :=
    TEffectStylePropEd.Create(Application);
  try
    { Initialize with the current style }
    EffectStylePropEd.EffectStyle :=
      TEffectStyle(GetOrdValue);
    if EffectStylePropEd.ShowModal = mrOK then
      { If OK then update with the new value }
      SetOrdValue(Ord(EffectStylePropEd.EffectStyle));
  finally
    EffectStylePropEd.Free; { Tidy up }
  end;
end;

```

Figure 5: Invoking the *EffectStyle* property editor.

tializes it with the property’s current value, transfers any new value back to the component after executing, and then destroys the dialog box (see Figure 5). Note that the *GetOrdValue* function returns the current value of the property (for any ordinal type), but that it must be converted into the appropriate enumerated type before it can be used. Similarly, the value returned for the property must be converted back into an ordinal value before Delphi can handle it. Both these conversions are handled by using the corresponding type as if it were a function, i.e. *casting*.

As before, similar processing applies to the *ColourScheme* property editor. All that remains is to incorporate the new editors into Delphi. Having generated the .DCU files, copy these into the library directory (DELPHI/LIB). Then select **Options | Install components** from the menu. Add the editor units to the list and click **OK** to start the installation. The new property editors can then be invoked by placing a *TLabelEffect* component on a form and double-clicking on either the *EffectStyle* or *ColourScheme* properties.

### A Little Help

Providing Help for a component adds a certain professionalism to the product, and should answer questions about the component without the need for constant reference to the designer. To produce a Help module it’s necessary to write the source document so that it can be saved as a Rich Text Format (.RTF) file. The document must then be compiled to generate the module that can be interpreted by the Windows Help engine.

Windows Help consists of a series of screens, or *topics*, that are connected by hypertext links. Each screen is a separate page in the source document. Different screens are separated by hard page breaks (entered manually).

Information relating to that screen is encoded in footnotes, with different footnote markers having specific meanings (see Figure 6). The “B” footnote is peculiar to Delphi, and enables it to automatically provide access to the appropriate

Footnote marker	Meaning	Comments
#	Context string	The name of this page or topic, which must be unique and serves as the destination for hypertext links.
\$	Title	The title of this page, which appears in the bottom half of the Search dialog box.
K	Keyword	The keywords for this page which appear in the top half of the Search dialog box.
B	Delphi keyword	Keywords used by Delphi when requesting help on a component, property, or event from the IDE.
+	Browse sequence number	Specifies the sequence of related screens, which can then be accessed through the << and >> buttons in the Help toolbar.

Figure 6: Help footnotes and their meanings.

screen of Help when requested from the IDE. The naming convention that must be followed is:

- “class\_” + component name for the main component screen
- “prop\_” + property name or “event\_” + event name for generic properties or events
- “prop\_” + component name + property name or “event\_” + component name + event name for component-specific properties or events

Hypertext links are encoded by underlining or double-underlining the text that denotes the jump and immediately following it with the context string of the destination. The context string must be formatted as hidden text. The difference between single and double underlines lies in how the destination screen is shown. A single underline results in a pop-up window overlaying the original, while a double underline displays the destination screen in the original window.

Graphics can be incorporated into the Help file by embedding commands for their placement or by inserting them directly into the document. The commands are enclosed in braces ( {} ), and consist of a keyword indicating the positioning of the image, followed by the name of the image to include.

The keywords are *bm1* for a left aligned picture, *bmr* for right-aligned, or *bmc* for in-line (as a character). The first two keywords cause the text to start at the top of the image and to wrap around it. The *bmc* keyword inserts the image directly into the text at the position specified. Images can be used as hypertext links by formatting them as for normal text, but this is not used in component Help.

Most formatting will be carried across to the Help file, but tables (as in Microsoft Word) should be avoided. Text can be prevented from scrolling in the Help window by formatting the paragraph as “keep with next.”

## Help Screens

Help for a component within Delphi consists of a number of standard screens described below. All use a sans-serif font, such as Arial or MS Sans Serif. Most of them have the title in blue that is set as a banner that does not scroll.

There is a screen for the component as a whole, describing its purpose and key properties. It also provides pop-up lists of related components and this component's properties, methods, and events. For components that appear on the Component Palette, a copy of the bitmap used to identify it should appear alongside the component name. Small images are used in the lists of properties, events, and methods to denote key elements and elements that are accessible only at run time.

Instructions regarding how to accomplish common tasks with this component fills another screen, with links to and from the component screen. It describes the component's purpose and use, and explains which properties or methods to use to make it do what we want.

There is another screen for the unit that defines the component, showing what components and types are declared therein. Links are provided to these items for more detailed information.

Screens for each of the component's properties appear next, describing their effects and valid values. Links are provided to related properties (in a pop-window), relevant types, and example code for using the property at run time. Similarly, methods and events also rate separate screens, explaining their purposes and functions. They also link to related methods or events and example code.

Finally, there are screens for any special types declared for use within the component. These detail valid values for the type and provide links to the properties defined for it.

### TLabelEffect Help

Help for the *TLabelEffect* component follows the guidelines previously described. Since it is similar to a *TLabel* component, those properties and events that are common are not described. Only the added properties have detailed descriptions. No new methods or events have been added.

A naming convention has been adopted to make the coding of links easier. Topics are all in upper case and consist of the item name followed by a screen type, separated by an underscore character ( \_ ). For example, the items include TLABELEFFECT, the component; DEPTHHIGHLIGHT, a property; and EFFECTDEPTH, a type. Screen types include COMPONENT, PROP, and TYPE. Figure 7 shows the first page of the Help source document with its corresponding footnotes.

Note that the link destinations are shown, whereas normally they would be formatted as hidden text. Figure 8 shows the same page of Help at run time.

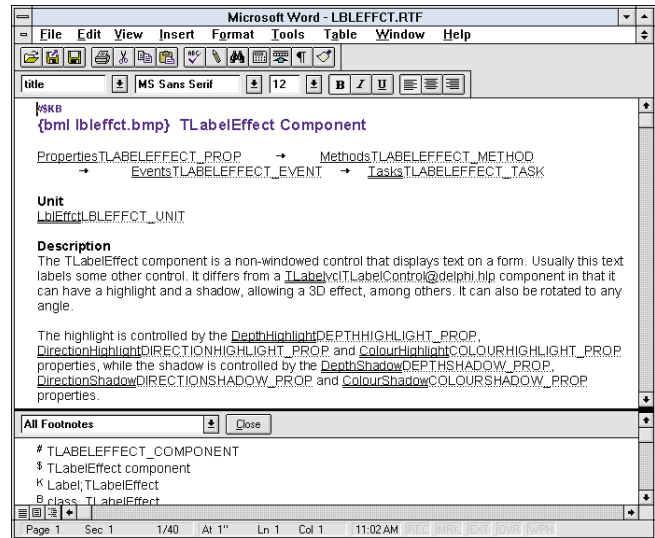


Figure 7: The first page of the *TLabelEffect* Help document and its footnotes (shown in Microsoft Word for Windows 6.01). Note that jump targets are shown — normally they are hidden. The resulting Help screen is shown in Figure 8.

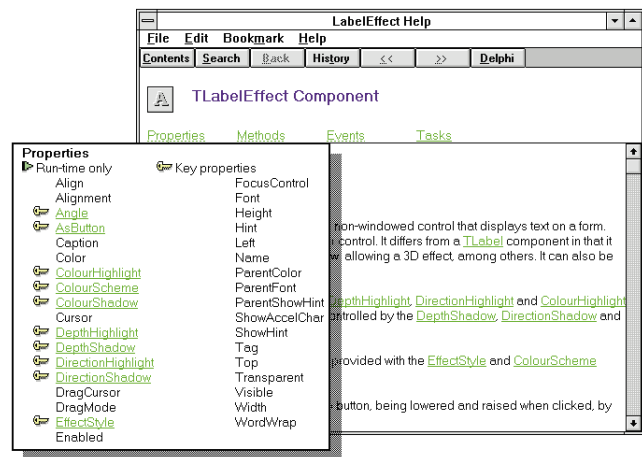


Figure 8: The online Help described in Figure 7 at run time.

The “B” footnotes for integrating with Delphi use the component-specific format of “prop\_” + component name + property name. This ensures they will only be referenced for the *TLabelEffect* component, even if another component contains properties of the same name.

Navigational connections are made between all the properties and between all the types for the component. This is done through the “+” footnote, with the associated value being a group identifier, followed by a sequence number, separated by a colon. The numbers increase by five each time to allow for easy insertion at a later stage, and all consist of three digits since they are all treated as alphabetic values for comparison (i.e. so “5” is not less than “10”). This means that the properties and types can be read as a group by pressing the << or >> buttons in the Help toolbar from any one of them.

Several small standard images appear in Delphi's Help, such as the “key” image displayed beside key properties of a component. These images are included in the source



document and as separate bitmap files in the files accompanying this article.

### Some Detective Work

For the *TLabelEffect* Help, we want the ability to refer back to the *TLabel* entry provided in Delphi's online Help. We can provide a hypertext link to another file by specifying the context string followed by the "at" symbol (@), and the name of the Help file where it's found (including the extension).

This is fine if we know the appropriate context string. Guessing at what it might be proves fruitless, but the context strings must be available somewhere to allow access to them from programs and the Help engine itself. After some exploring of the Help related files, we find that the keyword file (.KWF) contains these strings. Unfortunately, they are not obvious from examining the file. Searching using "TLabel" eventually locates a section of text that refers to *vcITLabelControl*. Testing this reveals that it is the context string we want.

Further examination gives a general overview of the sort of context strings that are encoded in Delphi Help. These are shown in the table in Figure 9, along with examples of specific values. Text between the angle brackets (< >) should be replaced with the name of the item being referenced. The difference between controls and components is not clear, but appears to be related to whether the component is a standard Windows control.

Type of reference	Format	Example
Control	vc1<component>Control	vcITLabelControl
Component	vc1<component>Component	vcITMemoComponent
Object	vc1<object>Object	vcITListObject
Property	vc1<property>Property	vcICaptionProperty
Method	vc1<method>Method	vcIDraggingMethod
Event	vc1<event>Event	vcIOnClickEvent
Procedure	vc1<procedure>Procedure	vcIShortCutToKeyProcedure
Function	vc1<function>Function	vcIInputBoxFunction
Type	vc1<type>Type	vcITAlignType

Figure 9: Table of Delphi Help context strings.

We also want to provide a button on the Help toolbar to allow access to the rest of the Delphi online Help. This is done in the Help project file, which is described further below.

### The Help Project File

Along with the Help source, we need a project file (.HPJ) to tell the Help compiler where to find everything and what to do with it. The one used in this case has the bare minimum of entries since the Help is contained in a single source file without any special processing. The basic layout of the project file resembles an .INI file, with named sec-

```
[OPTIONS]
title = LabelEffect Help
root = C:\Delphi\Comps\LblEffct
compress = false
report = on
warning = 3

[FILES]
lblreffct.rtf

[CONFIG]
BrowseButtons()
CreateButton("btn_delphi", "&Delphi", "JumpContents('Delphi.HLP')")
```

Figure 10: The *TLabelEffect* Help project file.

tions followed by various parameters. The contents of the *TLabelEffect* project file are shown in Figure 10.

The [OPTIONS] section contains parameters that control the Help compiler. In our case, we specify the text to appear in the Help window's title bar, the directory in which to find the source files, that we do not want any compression done, and that error messages are to be displayed during the build process.

The [FILES] section contains a list of all source files to be used in the compilation. All will be combined into the one Help file. An external list of files can be added by using the include directive (#) and enclosing the file name in angled brackets (< >).

Finally, the [CONFIG] section contains macros that we want to run when the Help file is opened. The *BrowseButtons* macro adds the << and >> buttons to the Help toolbar and enables them appropriately to browse through related topics. To allow access to Delphi Help, we can add a *CreateButton* macro to this section of the project file with the action being to jump to the contents page of the DELPHI.HLP file. Note that the initial "single quote" is actually a grave accent mark (`). A single quote (or apostrophe) will not work.

There are many more sections and parameters that can be added to the project file, but these are sufficient for our purposes. Have a look in the API Help for lists of the Help macros, and in the Help compiler documentation for the remainder.

### Compiling the Help File

The Help compiler is then invoked and passed the name of the Help project file. From this, it locates the necessary source documents and generates the compiled Help file. The compiler may be called HC31.EXE or HCP.EXE, depending on which version is available.

Note that it is not a Windows program, and so it must be run from a DOS window or from the command line available through File | Run in the Program or File Managers. In this case the command is:

```
hc31.exe lblreffct.hpj
```

We need to extract a list of keywords defined in the compiled Help file. These are used by Delphi when integrating the Help into its own system. This can be done by selecting the Keyword Generate icon in the Delphi group.

Enter the name of the Help project file (LBLEFFCT.HPJ in our case) and the name of the output file will be automatically generated with an extension of .KWF. Then click on the OK button to produce the keyword file.

Now that we have the compiled Help and keyword files, they can be integrated with Delphi. To do this, copy the .HLP file to \DELPHI\BIN, and the keyword file (.KWF) to \DELPHI\HELP. Then merge the component Help keywords into Delphi's list by selecting the HelpInst icon in the Delphi group, opening file DELPHI.HDX in \DELPHI\BIN, adding the LBLEFFCT.KWF file to the list, and compiling and saving the .HDX file.

The Help can now be accessed by selecting a *TLabelEffect* component on a form, or one of its new properties in the Object Inspector and pressing **[F1]**. The appropriate screen of Help should appear. Help for the properties and events inherited from *TCustomLabel* are still available, and originate from Delphi's own Help system.

### Conclusion

Adding property editors to a component enables particular properties to be altered graphically. Although not applicable to all properties, it does enhance those with a strong visual presence — such as the two presented here — and can

complement selecting values from a list or entering them directly. This lets the user update the property in various ways, allowing the user to select the most appropriate one.

Help can explain the use and limitations of a component and its properties. Having this available online enables it to be accessed when it is most needed, i.e. when the component is being used. Delphi allows for this additional Help to be integrated into its own Help structure so that the user sees no difference between the two. Copying the style of Delphi's own Help means the component's Help blends in that much better.

Together, these additions to your component give it that professional finish and — perhaps — an edge over the competition. **Δ**

*The demonstration forms and files referenced in this article are available on the Delphi Works CD located in INFORM\96\AUG\DI9608KW.*

Keith Wood is an analyst/programmer with CSC Australia, based in Canberra. He started using Borland's products with Turbo Pascal on a CP/M machine. Although not working with Delphi currently, he has enjoyed exploring it since it first appeared. You can reach him via e-mail at [kwood@netinfo.com.au](mailto:kwood@netinfo.com.au) or by phone (Australia) 6 291 8070.





## FROM THE PALETTE

Delphi 2 / Object Pascal



By *Karl Thompson*

# Table Documentor: Act 32

## Enhancing Some Native Delphi Components for Use in a Comprehensive Utility

**D**irectoryListBox, FileListBox, DriveComboBox, and FilterComboBox. These four components ship with Delphi, and give developers an easy way to allow their users to navigate drives and directories, and to select files. Generally these components work just fine, although they do have one major design limitation: What happens when you want to give your users access to database files on a UNIX server, or to data files via an alias? With these components, you're out of luck; since they don't support aliases, it's impossible to use them to access tables on database servers.

We'll enhance these components to allow this access. Besides giving you three improved components (the functionality of DirectoryListBox and DriveComboBox are combined into one component), I've also upgraded the **Table Documentor utility** that first appeared in the **March 1996** issue of *Delphi Informant*. This utility provides a good example of how to use the enhanced components.

In addition to server access via an alias, the new version of Table Documentor has been updated to recognize the new field types that became available with the release of Delphi 2. (The new version of Table Documentor, as well as the three enhanced components, require Delphi 2 to compile.)

Also, a multi-table dynamic viewer was incorporated into the program. Now, not only can you document the structures of database tables, you can also view the contents of up to three tables at once. In addition, you can optionally print structures to text files. I've found this feature to be very useful; when working on team projects, I've been able to easily include in e-mail to team members copies of table structures when necessary.

All of this — the updated utility, the enhanced components, and over 2,000 lines of source code — is available for download (see end of article for details).

### Let's Get Started

This article will be divided into two broad areas. First we'll look at the development details of the new components, then we'll review the new functionality of the Table Documentor. If you are a component developer, you'll be particularly interested in one component: KtDrvDirLstBx. A unique kind of list box, it exhibits different behavior depending on the type of item selected.

Also, while studying the development of the components, we'll take a look at a code error that led to a rather nasty bug (nasty in that the error messages Delphi returns were not very helpful in finding the problem).

However, after the side effects of this error are brought to light, you'll be able to instantly know when you've been bitten by this bug.

In addition to the standard components not providing access to files via an alias, there is another shortcoming. As we all know,

## FROM THE PALETTE

Microsoft sets design standards with respect to the Windows interface. One of these calls for the user of a combobox to make a drive selection and a list box to navigate the directories on the selected drive.

In pre-Delphi versions of Object Pascal, Borland made a considerable enhancement to this standard by incorporating the ability to select a drive and navigate the directories in one list box. When I had the opportunity to beta test Delphi 1, I urged Borland — to no avail — to maintain this ability, even though it deviated from the gospel according to Microsoft. Anyway, if you don't like the tool, Delphi gives you the power to create your own, and that's what we'll do this month.

The functionality of the original Delphi components will be replaced with `ktDrvDirLstBox`, `ktFileListBox`, and `ktFilterComboBox`. Naturally, the original components will remain on the Component Palette. The new components will be installed on the System page, and their bitmaps look exactly like the original Delphi ones, except for the yellow 'E' on them.

The functionality of the `DriveComboBox` and `DirectoryListBox` has been combined in the `ktDrvDirLstBx` component. `ktFileListBox` and `ktFilterComboBox` function just like their counterparts, although if you're going to use `ktDrvDirLstBox` in an application, you must use the corresponding 'kt' components and not the original Delphi ones.

A descendant of `CustomListBox`, the `ktDrvDirLstBx` component is the focal point. A student of Object Pascal might wonder why the component doesn't descend from `DirectoryListBox`. After all, isn't one of the great benefits of OOP that the programmer doesn't have to re-invent the wheel, but rather merely extend existing functionality? At first it would seem to be a perfect opportunity to inherit the behavior of `DirectoryListBox`, and add to it our new features.

Although that's what I had originally set out to do, it soon became apparent it would not work. One problem is that Object Pascal doesn't allow for multiple inheritance. Because we want the custom directory list box to have both the features of Delphi's `DirectoryListBox` and `DriveComboBox`, it became necessary to descend from `TCustomListBox`.

The other hurdle to overcome is more subtle. Normally, list boxes handle only one type of item. They may present the user with a list of similar choices for configuring an application, such as a list of modems. Or in the specific case at hand, one standard Delphi list box allows the user to select a directory, and another list box allows for the selection of a specific file. However, our design objective requires that a list box handle available drives, directories, and aliases.

You might be thinking this really doesn't present a problem, because the items in question really are represented by initialized strings that are handled by the `Items` property which

is of type `TStrings`. And you would be correct — except for one minor detail. The list box must exhibit different behavior depending on which type of item is selected by the user. This certainly is not a characteristic of a normal list box.

### Distinguishing Code

Now that we understand this problem, how will the code know how to distinguish between `Items[10]` which may be a drive and `Items[16]` which may be an alias or a directory? The answer is simple. Each item has associated with it a bitmap object. The bitmaps are different for a drive, an alias, an open directory, and a close directory. By evaluating the bitmap associated with any item, it is easy to determine the nature of the selected item. Pretty cool, huh?

Let's go to the code for an example. The following is found in the `TktDrvDirLstBx.DblClick` method:

```
if not(TBitmap(Items.Objects[ItemIndex]) = FAliasBMP) then
  OpenCurrent
else
  begin
    ...
```

When the user double-clicks on an item in an instance of the `ktDrvDirLstBx`, the item is selected, and if it is a drive or directory, it then becomes the working or default system path.

However, if the item is an alias that is mapped to a UNIX server, for example, then the directory on the UNIX server can't be the working directory of the DOS environment. Therefore, the above `if` statement checks first to see if the user has double-clicked on an alias, and if not, then 'normal' DOS behavior is executed with a call to `OpenCurrent`.

Notice that the `Object` field of the `Items` property can be any one of a number of different type of objects such as `TGraphic`, `TOutlineNode`, `TBlobStream`, or `TCanvas` (to name a few), in addition to the `TBitmap` that interests us. Therefore, because we know we are dealing with a bitmap (after all, that is the kind of object that was inserted into the object field when the `Items` property was initialized), we must type cast the left side of the equality test to that of a `TBitmap`. (See the `TktDrvDirLstBx.BuildList` method if you want to see how the objects and the strings are initialized.)

The `ktDrvDirLstBx` adds several new properties. Among them are:

- `AliasChar` (Char)
- `AliasLeadSep`, `AliasTrailSep` (Boolean)
- `DrivesFirst` (Boolean)
- `ShowAliases` (Boolean)
- `AliasBMP` (`TBitmap`)

If either `AliasLeadSep` (alias leading separator) or `AliasTrailSep` (alias trailing separator) are `True`, then the alias name (assigned to the `Directory` property) is padded accordingly with the value of `AliasChar`. The default for



each is *True*. This behavior is implemented so an application that processes the string values of the *Item* property can parse the value to determine if it is an alias. Again, one may wonder: Why not test the value of the object field? You should note the bitmaps are not public; so in keeping with the original Delphi design, I chose not to make them public. Also, Paradox programmers are accustomed to *:AliasName*: so I wanted to be able to provide this functionality. Again, check out the Table Documentor for an example of how this feature is used (the *All2DocClick* method provides one such example.)

Two other properties that are useful for controlling the runtime behavior of your application are *DrivesFirst* and *ShowAliases*. When *DrivesFirst* is *True*, the *ktDrvDirLstBx* instance will list drives and directories first; otherwise aliases will be listed first. If *ShowAliases* is *False*, the list box will show only drives and directories.

I assume there may be someone who doesn't care for the glyph I used for the alias bitmap. (Maybe you'll want the glyph used for an alias in the Database Explorer that ships with Delphi.) By surfacing the *AliasBMP* property, the programmer can change the glyph. Simply assign a properly initialized bitmap to this property. (Remember, it is the programmer's responsibility to free the bitmap after the assignment.)

Finally, I should discuss the *ktDrvDirLstBx* property. In the original Delphi version, a programmer can assign a value to this property, and if it is a valid directory name, the system default directory will become that directory. This is also how *ktDrvDirLstBx* works. However, the *Directory* property will return the name of the currently active alias (an alias is made active by double-clicking on it). So far, so good. The property is initialized within the scope of the component by assigning the alias name to the *FDirectory* variable. However, *FDirectory* is not available to the application that uses *ktDrvDirLstBx*. If the application attempts to assign an alias name to the *Directory* property, the program will get a "Not a valid file name error". This is because the *SetDirectory* method (the write method that assigns a value to *Directory*) will attempt to change to the alias as if it were a valid directory).

So on one hand, *ktDrvDirLstBx.Directory* can return a valid BDE alias name, but the program cannot assign an alias name to that property. This did not seem to be a major drawback, because I could not think of a case when an application would need to assign such a value to the *Directory* property. However, I'm sure that some enterprising *Delphi Informant* reader will come up with a good reason to be able to do so!

## A Component Error

I mentioned at the outset that I would reveal a coding error I made while developing these components — one that proved to be quite tricky to track down.

The error is nasty and renders your library useless! *Therefore, make sure you've made a backup of your CmpLib32.DCL file before attempting to learn about this error.*

Once you've successfully installed the enhanced components, and *ktDDBug.res* is in the same directory as the unit, there are two steps needed to recreate the bug.

First, activate the define statement at the head of the *ktDrvDir* unit file ( `{.$DEFINE BUG1}` ) by removing the initial period. Then, rebuild the component library by selecting **Component | Rebuild Library**. The component library will be trashed! And the only message you'll get will be "Invalid ImageList Index." The problem arises because the *ReadBitmap* method tries to load a resource named *ALIAS*:

```
FALiasBMP.Handle := LoadBitmap(HInstance, 'ALIAS');
```

The name apparently conflicts with some other resource name in the component library, or is some kind of reserved identifier for some other purpose. Either way, changing the name alleviates the problem.

By the way, if you do test the library with the above error activated, you can simply restore the working library by copying the backup copy of your library to *CmpLib32.DCL*, selecting **Component | Open Library**, and selecting *CmpLib32.DCL*.

How did I find this error? Unfortunately, only through detective work and good luck. It's interesting that the error didn't surface until the unit was compiled into the component library. That is, if you follow the Component Writer's Help file instructions for "Testing Uninstalled Components," this error won't surface.

## The Table Documentor Program

Enhancing the Table Documentor utility provides a perfect example for our enhanced components. In the first article I suggested some changes to the interface. To a large extent, those changes have been incorporated into this version. The most visible change is the use of a *TPageControl*. (Using a *TPageControl* rather than a new form to display the structure results saves a lot of screen real estate.) You'll find the three new components on the first page of the notebook.

Using the utility is straightforward. Simply navigate drives, directories, and aliases in the *ktDrvDirLstBx*. As you move through the directories, you'll see files displayed in the center *ktFileListBox*. If you wish to filter your files to show only Paradox or dBASE tables, you can do so by using the *ktFilterCombo*.

Aliases are distinguished in the *ktDrvDirLstBx* by the alias bitmap. If you have aliases defined in your BDE that connect you to a database server, when you double-click on one of them, you'll be prompted for your user name and

## FROM THE PALETTE

password, as well as any other information the database requires for logging in.

You may select files for processing using one of several methods. If you scroll through the file list and double-click on a file name, the name will appear in the right-most list box. You can also use the standard Windows selection technique of pressing **⇧** while clicking on one file name, and then clicking on another file name further down the list to select a contiguous list of files. Or you can hold **Ctrl** and single-click random files to select. After files are selected, you can click the **>** button to move them to the **Table to document** list box. Finally, you can click on the **>>** button to move all of the files to the list box. This also works in reverse to remove files from the **Table to document** list box. By the way, you may select files from more than one directory or alias at a time. Table Documentor keeps track of the file's location internally. This way, you can document or view two tables in different locations at the same time.

After one or more table selections are made, the user can view the table structure of each selected table by clicking the **Show Structures** button (see [Figures 1 and 2](#)). [The table structure retrieval process is described in Karl Thompson's [March 1996, Delphi Informant](#) article.]

If the **Browser** tab is clicked, the first three selected tables are dynamically linked to a database grid (see [Figure 3](#)). I've found it useful to have a listing of a table's structure and a view of the table's data on one form, with access controlled by just a few mouse clicks.

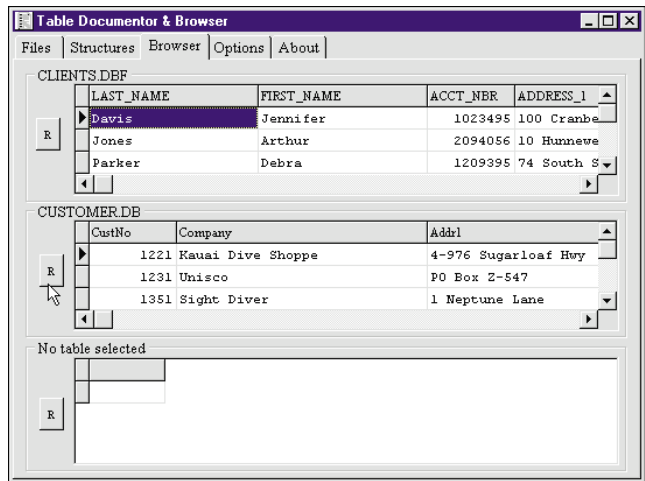
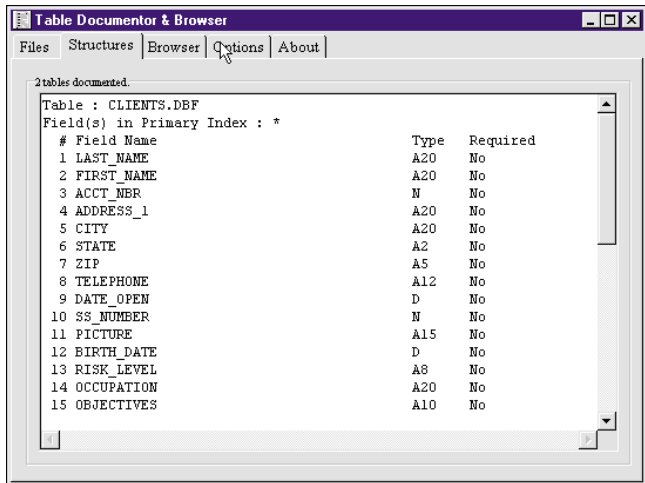
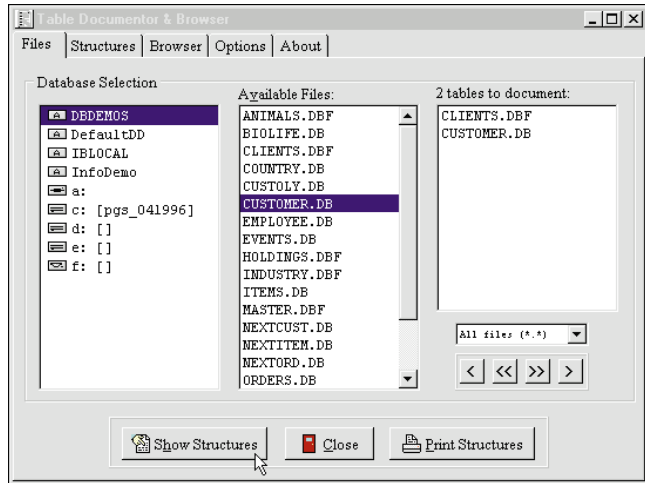
The **Options** page (see [Figure 4](#)) deserves a few brief comments as well. The **Print** options are self explanatory. If the **Browser Status** is set for **Auto refresh**, the views of the data tables will be refreshed every time the **Browser** tab is selected. Normally this is OK.

If the view is to a large table on a server, however, it may take some time to refresh the tables, so the user is given the option of preventing a refresh. Of course, a manual refresh can be accomplished by clicking the appropriate **R** button (again, see [Figure 3](#)).

The **Path/Alias Options** allow fine tuning of the environment. The **Save path/alias** option only affects the display in the list boxes and the **GroupBox** captions. If the option is selected, then the path or alias will be displayed along with the file name. Although this is handy for reference, often the **FileListBox** will not be wide enough to display a full path and file name. In such instances, you'll want to turn this option off. You can also control whether aliases will be listed first or last in the **DrvDirLstBx** with the **List alias first** option; or you can choose to not display aliases at all.

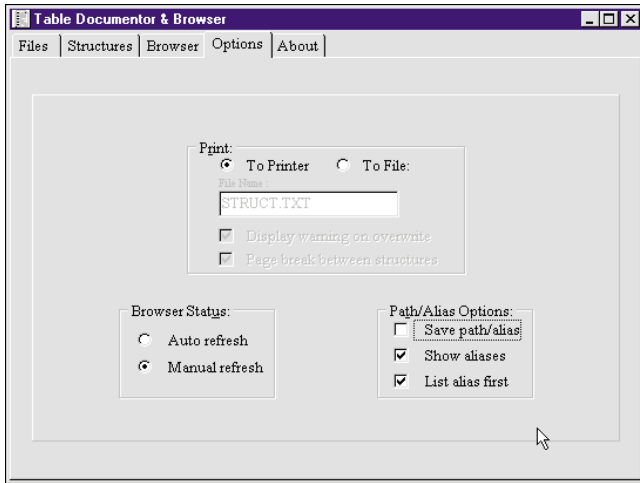
## Enhancements

We've reached the point where I normally make suggestions for further enhancements to the application. The obvious



**Figure 1 (Top):** The enhanced Table Documentor utility for 32-bit environments. It allows you to select a table by referencing an alias or a specific drive. Select the number of tables to document in the **Available Files** list box, then select the appropriate arrow buttons to move or remove files from the list box at the far right. **Figure 2 (Middle):** In [Figure 1](#), CLIENTS.DBF was one of the tables the user selected to document. By clicking on the **Structures** tab, the user can view the structure of this table. **Figure 3 (Bottom):** The **Browser** page allows the user to view the data stored in the tables.

suggestion is to document more about the table's structure. For example, it would be helpful to know what fields have



**Figure 4:** Want to print the selected tables' structures? The **Options** page allows you to do this, as well as set **Browser Status** and **Path/Alias Options**.

validation checks. This requires BDE calls, which are beyond the scope of this article.

Additionally, the code that dynamically changes the links between the table components and the databases is a good candidate for execution in a thread. Such a change would probably make it possible to leave the **Auto refresh** option on.

There is one other significant enhancement I can think of: a horizontal scrollbar. Unfortunately, list boxes don't have the ability to display them. In the case of Table Documentor, where listing of paths and file names can easily be wider than the allotted space, it would be nice to have the ability to scroll left and right to see the entire file name.

Finally, there is that perpetually deferred task, online Help. Both the components and Table Documentor would benefit from concisely written Help.

I hope you find these components as useful as I have. ▲

*The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\AUG\96\DI9608KT.*

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



## TEXTFILE



### An Insider Look at Delphi's Roots

I recently read Paul Cilwa's *Borland Pascal 7 Insider* [John Wiley & Sons, 1993], and I'm glad I did. Cilwa is a Windows programming consultant with solid development experience — and it shows; he offers excellent advice on how to write “good” Pascal. *Insider* was written for Pascal programmers by a Pascal “insider.” As such, it's a clearly written guide to the undocumented and hard-to-find topics that are barely addressed, and are more often ignored, in Delphi literature.

Programmers new to Pascal beware: *Borland Pascal 7 Insider*, and other Pascal 7 titles, should only be considered *after* you have worked with Delphi for a while and are comfortable examining advanced language topics. Delphi's online Help and printed documentation detail the differences between Borland Pascal 7 and Delphi syntax. Delphi programmers who do not have experience with Borland Pascal should familiarize themselves with these differences before reading *Insider*.

Delphi, with the inclusion of the form designer and .DFM files, has rendered many difficult and error prone Windows programming techniques obsolete. Pascal 7 pre-dates these developments, and therefore requires Resource Workshop and the need to map resource constants in source files. For this reason, studying *Insider* with an eye on the Delphi

source code included in the Developer release of version 2 can reveal many of the internal workings of Delphi. Unless you have Resource Workshop, don't expect to work through the code examples contained in this book.

*Insider* is divided into three parts. Part 1 begins with a discussion of “good” Windows programming, and moves quickly into defining an enhanced string class to illustrate some object-oriented techniques. It then moves to resources and the construction of a set of OWL derivative classes. These sections include a “from-the-ground-up” look at

### One of the Best Gets Better

I enthusiastically reviewed Marco Cantù's *Mastering Delphi* in the pages of *Delphi Informant* last November. It quickly became the book I turned to when trying to solve Delphi problems. Therefore, I expected a great deal from Cantù's sequel, *Mastering Delphi 2 for Windows 95/NT* [SYBEX, 1996]. And I was not disappointed.

However, I was surprised to find the page count had gone *down* by more than 30 percent. My concern was whether any important material had been omitted. A quick check revealed the overall outline was largely unaltered; only a few chapter titles were changed, generally reflecting differences between Delphi 1 and 2. Still, most chapters had significantly fewer pages than before.

*TApplication* and *TForm*.

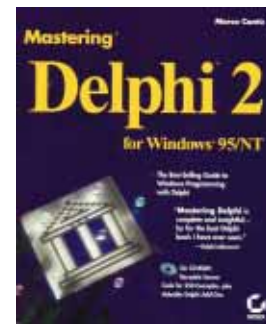
Part 2 continues with a behind-the-scenes view of *TMenuItems*, creation of a standard file management dialog, and building a wrapper class for managing .INI files. These sections will be of particular interest to component writers, as well as those seeking special features of the Windows 95 look and feel. Part 2 concludes with an .INI editor and a detailed discussion on how to integrate online Help into your applications.

Part 3 addresses working with the Clipboard, DDE, and OLE. The Clipboard discussion alone is probably



worth the price of the book. Cilwa clearly explains how a robust Windows application should — and can — support the most feature-rich data formats. If you've ever used cut-and-paste unsuccessfully, you'll now learn why. Cilwa's work on DDE and OLE is equally good. Part 3 concludes with an

*“An Insider Look at Delphi's Roots”  
continued on page 45*



Some careful searching revealed why: tighter editing, which improves the presentation; slightly reduced inter-line leading, which does not affect readability; and the omission of some longer code fragments, although they still appear on the accompanying CD. All in all, this seems like a prudent approach.

Unlike some Delphi 2 books, this one is far more than the result of a few global phrase substitutions, or the addition of a Delphi 2 appendix. Throughout the book, the material is updated to reflect Delphi 2 practices.

For example, one major change is in chapter 18, “Client/Server and Advanced Database Applications” (formerly “Building Client/Server Applications”). Here, the old material is compacted signifi-

cantly by omitting a lengthy code example (which is available on the CD). Additionally, new sub-sections discuss three-tier architecture, data modules, filtering, and Delphi's data dictionary.

Chapter 22, “Adding Printing Capabilities to Delphi Applications,” covers not only ReportSmith, but also QuickReport, a subject often missing from quickie updates of other Delphi 1 books. The chapter concludes

*“One of the Best Gets Better”  
continued on page 45*



## An Insider Look at Delphi's Roots (cont.)

excursion into serial communications. In a cleverly constructed exercise, the reader works through an example of how to set a system clock by dialing into the National Institute of Standard's atomic clock. Cilwa clearly has a sense of humor.

Although you won't find fancy component design or slick graphics in *Insider*, you will find a frank, well-written discussion of serious Pascal and Windows application development. As an "insider," you'll learn how to construct frameworks, such as the VCL, using API calls and messaging, and you'll gain a better understanding of how to manage memory, as well as explore the differences between virtual and dynamic methods. You'll also learn coding efficiencies, such as passing C-style API

parameters in Pascal programs without the tedious type conversions.

*Insider* is a coding resource for serious developers looking to deeply examine Delphi and general Windows development. If you've read other Delphi books, and have writ-

## One of the Best Gets Better (cont.)

ten programs, but you want to examine your Pascal roots, consider Paul Cilwa's *Borland Pascal 7 Insider*.

— James Callan

*Borland Pascal 7 Insider* by Paul Cilwa, edited by Jeff

by demonstrating how QuickReport sometimes provides a better solution than ReportSmith.

The CD provides code for examples from both editions. It also offers a generous collection of third-party controls and other information. Unfortunately, many of these extra goodies haven't kept current. As the book's introduction states, "the third-party components on the CD are the 16-bit versions, so you cannot install them in

Delphi 2 (unless they include 32-bit compatible source code)." For a book that is specific to Delphi 2, this seems curious.

One area where the Delphi 2 edition outweighs its predecessor is the index — it is now 73 pages. The completeness of this enlarged index is greatly appreciated.

I previously stated *Mastering Delphi* deserved a place on your short list of candidates if you intend to buy only one book on Delphi. If you're

Duntemann. John Wiley & Sons, Inc., 1 Wiley Drive, Somerset, NJ 08875-1272, (800) 225-5945.

**ISBN:** 0-471-59894-1

**Price:** US\$26.95 (537 pages, source code disk available)

now venturing into 32-bit territory, the same holds true for *Mastering Delphi 2*.

— Larry Clark

*Mastering Delphi 2 for Windows 95/NT* by Marco Cantù, SYBEX Inc., 2021 Challenger Dr., Alameda, CA 94501, (800) 227-2346 or (510) 523-2373.

**ISBN:** 0-7821-1860-7

**Price:** US\$49.99 (1,043 pages, CD-ROM)





## NEW & USED

By *Bill Todd*

# Ace Reporter

## A Band-Oriented Report Writer Written in Delphi for Delphi

**T**ired of creating reports with report generators that are huge, do not integrate well with Delphi, and make distribution — especially in stages — a nightmare? Ace Reporter may be exactly what you are looking for.

Produced by SCT Associates, Inc., Ace Reporter is a report writer, written in Delphi exclusively for use with Delphi. And its familiar band orientation will make most developers feel right at home.

### Getting Started

To create a report, just drop a DataSet (e.g. *TQuery* and *TTable*), DataSource, and

SctReport component onto a form. Your form should resemble **Figure 1**. After connecting the DataSet and DataSource components, open the Object selector and choose the ReportPage component, or click on the white square at the end of the horizontal ruler to select the page.

This component is not visible on the form, but was created automatically when you placed the SctReport component on the form. Set the ReportPage component's *DataSource* property to the DataSource component.

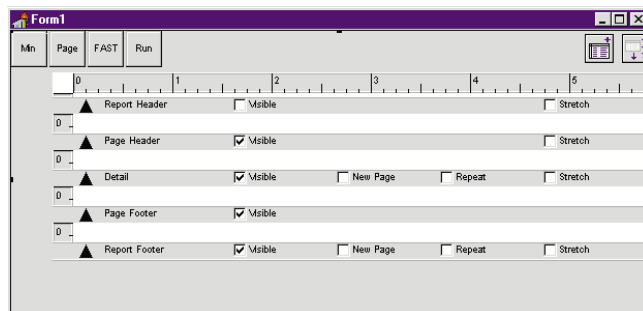
Although you can drop individual fields and labels on a report, it's quicker to press the **FAST** button on the report to display the dialog box shown in **Figure 2**. Clicking the **Next** button displays a page of column headings that are generated automatically, and allows you to edit them.

Another click of the **Next** button displays the page shown in **Figure 3**. This enables you to select within which bands the text labels (column headings) and variable labels (fields) will be placed.

The report is ready to run. All that's needed is a menu choice or button to call the SctReport component's *Run* method.

### Running the Report

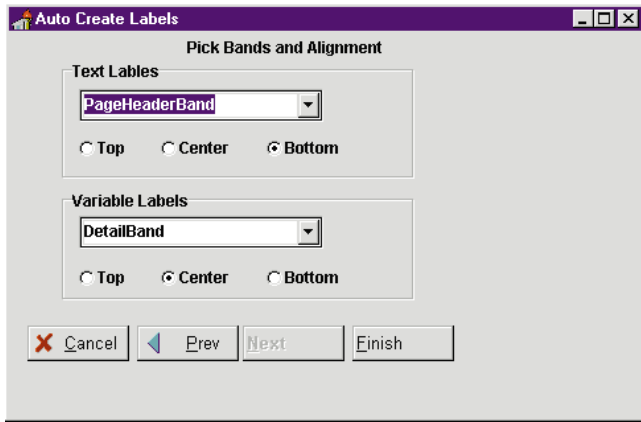
When the report is run, the Report Destination dialog box is displayed (see



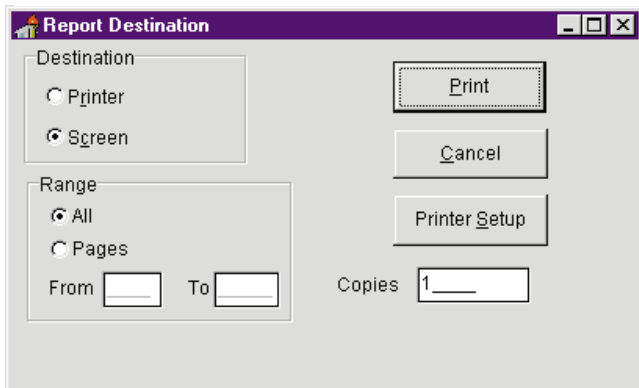
**Figure 1:** Getting started with Ace Reporter.



**Figure 2:** Click the **FAST** button on the report to display the Auto Create Labels dialog box.



**Figure 3:** After selecting **Next**, the Auto Create Labels dialog box allows the user to select bands for placement of text labels and variable labels.



**Figure 4:** The Report Destination dialog box.

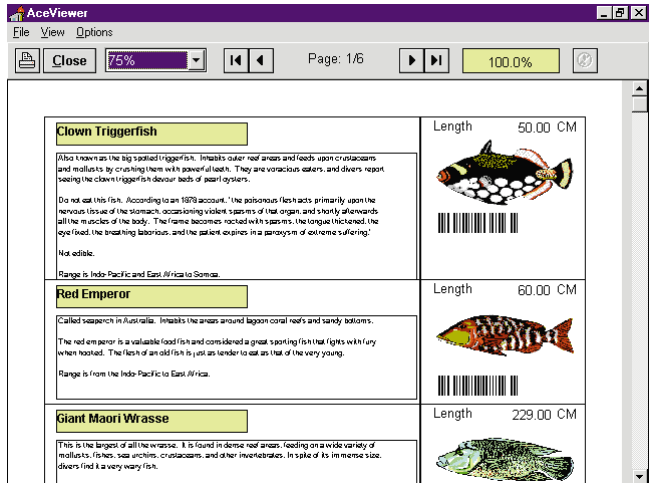
Figure 4). It gives you a chance to set the number of copies, page range, printer, and send the report to the screen or printer. Using code, you can set the initial defaults for this dialog box, or you can suppress the dialog box altogether.

When you send the report to the screen, the AceViewer will be displayed (see Figure 5). This preview window is exactly what you would expect, with controls to move from page to page, change the zoom factor, and print the report. As you can see, Ace Reporter can handle memo fields and graphics without a problem.

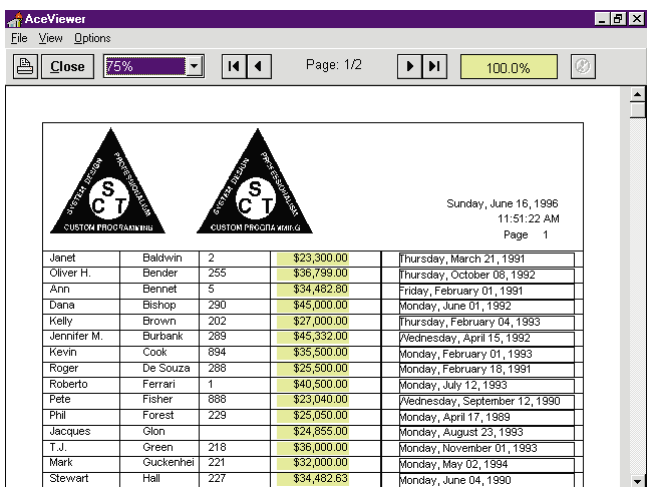
Ace Reporter's advanced features include its ability to send a report to the screen and immediately display the first page, even before the entire report has been run. Page one appears in the previewer as soon as it has been generated, and an indicator gauge in the previewer's toolbar displays a percentage of the report generated. In addition, you can use the previewer's split screen mode to view two sections of the same report, or two reports at once. If you don't like the appearance or features of the preview window, you can also design a custom previewer.

### The Feature Set

Ace Reporter has an excellent suite of tools to enhance the appearance of your report. These tools assist developers in



**Figure 5:** AceViewer, Ace Reporter's print preview window.



**Figure 6:** The finished example report.

adding horizontal and vertical lines, boxes, and other shapes. Each band features a *BorderType* property that lets you turn on a single line border around the band. Returning to the report design in Figure 1, it's easy to select the page header band by clicking on it and dragging the bottom edge down to add space below the column headings. Then drop a SctLine component on the report to place a horizontal line below the headings.

SctLine is one of the small jewels in this product. The line is actually contained in a rectangular component. You can set a property to determine if the line lies on the rectangle's top or bottom edge. This offers two benefits. First, since the line is part of a rectangle, it's always perfectly straight as you drag it horizontally across the report. Second, by adjusting the height of the rectangle you can add space above or below the line. This is much easier to use than the line drawing tools in any other report writer I have used.

To create a tabular look for the report, choose the option on the FAST button that automatically places a vertical divider line between each field. If you want to add vertical lines after the fields are placed, [Shift]-click the SctVerticalDivider on the

Component Palette, then click between each field in the detail band of the report. This places a vertical line between each field, and the line automatically sizes itself to the height of the detail band. This guarantees there will be no gaps in the vertical lines, and is another example of the thought and care that has gone into designing Ace Reporter.

If you want a grid style report with vertical lines between the columns and horizontal lines between the rows, set the detail band's *BorderType* property to provide the horizontal lines. In addition to the line and vertical divider, there is also a *SctShape* component that lets you quickly add boxes and other shapes to a report.

Figure 6 shows the finished report with the added horizontal and vertical lines. Another great feature is Ace Reporter's **Run** button. Click the **Run** button on the report and it will run to the screen while you are still in design mode in Delphi.

### Building on Delphi's Capabilities

When you first place the *SctReport* component on a form (again, see Figure 1) it provides report header and footer, page header and footer, and detail bands. Each band has a check box that lets you quickly toggle its visibility.

By clicking the **Page** button, you can add one or more group bands to the report. Ace Reporter's groups are unique because when adding a group on a field, it doesn't sort the data on that field. The Ace design philosophy — and it's a good one — is not to provide functionality already available in Delphi. To use a group in a report when the data in the table isn't in order by the field to group on, use a *TQuery* component as your *DataSet* and let the *ORDER BY* clause in the query do the sorting.

### Aggregates without Aggravation

A report just isn't a report without subtotals, running totals, final totals, minimum and maximum values, averages, sums, and counts. And Ace Reporter provides them all. However, the way Ace Reporter displays information on a report is different from any report writer I've used before, and it's a very flexible and powerful approach.

Ace Reporter's basic architecture accumulates and stores information in variables and then displays the information in labels. You can define three types of variables for a report: *field*, *expression*, and *total*. You will rarely need to define field variables since Ace Reporter creates a field variable for every field in the *DataSet* used.

You create a total variable when computing the sum, count, minimum, maximum, or average of a field. To create the variable, click the **Page** button on the report to open the Page Manager. From there, select the Variables tab and press the **Add** button. When you define a total variable you don't need to specify the type of summary information needed (sum, minimum, maximum, etc.), only define the field variable to summarize.

Next, place a *TotalVarLabel* component from the Component Palette on the band where you want the summary to appear. Set the *Variable* property to the variable previously defined in the Page Manager, and the *TotalType* property to the type of summary needed.

If you want a different summary value on the same field or in a different band, you do not need to define another total variable. Just place the necessary *TotalVarLabel* components and connect them to the same variable.

### Express Yourself

Expression variables allow you to display any piece of information from anywhere. After defining an expression variable in the Page Manager, select it in the Object Inspector's drop-down list and set its *UpdateLevel* property. *UpdateLevel* is set to one of the bands on the report. Whenever Ace Reporter is about to print that band, it will trigger the variable's *OnGetData* event.

Then, in the event handler, write code to determine the value displayed, and assign that value to a parameter passed to the event handler. This means you can perform any type of computation, locate values in other tables, or do anything else that Object Pascal can do.

You display the value of an expression variable on your report the same way you did with a total variable. Place a *VarLabel* component on the report and set its *Variable* property to the name of the expression variable. You can also include any calculated fields that you defined using the Delphi Fields editor and *OnCalcFields* event handler in your report.

### The Devil's in the Details

Another unique and powerful feature of Ace Reporter is sub-bands. A *sub-band* is a band that is linked to another band in the report. The sub-band is printed each time the band it is linked to is printed. Sub-bands can be used to display data in one-to-many relationships.

For example, if you want to print all the orders for a customer, you would display the customer information in the detail band, then create a sub-detail band connected to the *Orders* table's *DataSource* to display the order records.

Another example of the power and flexibility of Ace Reporter is its ability to allow you to set the *ReserveSpace* property of the detail band to the minimum number of detail records to appear on a page.

In this example, *ReserveSpace* is set to 3, instructing Ace Reporter to print this customer on the next page, unless there is room on the current page for the customer data and at least three order records. You can also add sub-header and sub-footer bands to print column headings, other header information, or totals for the detail records.

What's really impressive is that Ace Reporter can be set to do this for more than one level of detail. It also includes an over-

## NEW & USED

lay band that allows you to print anything anywhere on the page. This is handy for doing preprinted, form style reports, or watermark background graphics.

Ace Reporter features a full suite of events that you can attach code to for custom computations, as well as run reports from arrays or any other source that's not a database table.

For example, each band has an *OnPrintWhen* event that is triggered each time the band is about to print. The event handler returns *True* or *False* to control whether the band prints. The *OnDataFilter* event lets you write code that examines each record and determine if it should be included in the report.

In addition to sending reports to the screen or printer, you can also send a report to a file. This is particularly useful because you can send a report to a file in Rich Text Format (.RTF). This enables you to send a report to a file with its fonts and formatting preserved, and users can import the report into a word processor or e-mail message without altering its appearance.

### Drawbacks

About the only feature that's missing from Ace Reporter is the option to automatically suppress printing blank lines in a band or within a container. This is not a major problem. It just means you must format addresses or other text in code to drop blank lines. I do this often, and some report writers I've used include this feature, so I miss it.

The only complaint I have about Ace Reporter is the documentation. The manual is a scant 20 pages, and less than helpful. For example, on page 7 in the section on creating your first report you will find:

On the newly created form, you will drop a Dataset component, a TDataSet component, and a TSCTReport component. You should then set the DataSet property of the DataSource to point to the Dataset you just dropped.

*What DataSource?*

Fortunately there is a good tutorial on disk in Microsoft Word for Windows format to help get you going, as well as an adequate online Help system. Take the time to work through the tutorial. It will answer a lot of questions and show you how to do many things not covered in the manual.

Ace Reporter also ships with a demonstration program that shows a wide variety of reports.

Running the demonstration file provides a good introduction to the power and flexibility of Ace Reporter, as well as the code to learn exactly how a specific report is done.

Ace Reporter includes both 16- and 32-bit versions, so maintaining a common code base for both environments or converting an application from 16- to 32-bit is easy.


### Conclusion

Learning to use Ace Reporter is one of the best time investments I've made. This is a great product. It's not only powerful, but the **Fast** button's features allow you to create reports in record time.

With Ace Reporter I can create almost any report my clients want with none of the problems I have had with ReportSmith or Crystal Reports.

There are no DLLs or other external files to distribute. Everything is in your application's .EXE. The only connection to the database is the one opened by your Delphi program — and printing reports is fast. I ran a complex 115-page report that included a grid between each row and column on my Pentium 100MHz notebook in 20 seconds.

If you want fast, flexible, and completely integrated reporting for your Delphi applications, try Ace Reporter.

There is a demonstration version in the file ACETRIAL.EXE on SCT Associates' Web page at <http://ourworld.com-puserve.com/homepages/sct>. 

## INFORMANT FACT FILE

Ace Reporter from SCT Associates, Inc. is a band-oriented report writer, written in and for Delphi. To begin designing a report, just drop a DataSet, DataSource and SctReport component on a form. In addition, Ace Reporter has an excellent suite of tools to enhance a report's appearance that allows developers to add horizontal and vertical lines, boxes, and other shapes. Ace Reporter also allows you to include subtotals, running totals, final totals, minimum and maximum values, averages, sums, and counts on your reports. Highly recommended.

**SCT Associates, Inc.**  
9221 South Kilpatrick Ave.  
Oak Lawn, IL 60453-1813  
**Phone:** (708) 425-0205  
**Fax:** (708) 422-3877  
**E-Mail:** CIS: 73766,1224  
**Web Site:** <http://ourworld.com-puserve.com/homepages/sct>  
**Price:** US\$245

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* [QUE, 1995]; 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.





## Java Changes Everything

In the past year, Delphi picked up two formidable competitors: Optima++ and Java. They represent two modes of change occurring in the development tools arena — one evolutionary, one revolutionary. Let's examine these changes and their implications for you as a Delphi developer.

**Evolutionary Changes.** Before 1996, most change within the visual tools market was the result of an evolutionary process. The first phase began in the early 1990s when Visual Basic and PowerBuilder emerged as clear leaders in this area. Their respective languages weren't necessarily robust, but they became corporate standards because they were "good enough" to get the job done *quickly*.

Delphi's entry into this market last year introduced a second evolutionary step. Delphi's technological edge was so striking that the two dominant 4GLs were no longer considered untouchable. The recent release of Optima++ only re-inforces the trend to add true object orientation and optimized code compiler technology to this maturing marketplace.

**Revolutionary Changes.** As neat and tidy as this evolutionary process has been, one technology trend has emerged that throws the whole market up for grabs: the Web and its programming *wunderkind*, Java. The frenzy for Web tools — and paranoia over the Java hype — has undoubtedly kept product teams from all vendors working late nights "Web-enabling" their products. Borland's Delphi 2 Internet/Intranet Update is such an example.

But if "Web-enablement" was all we were talking about, we could call the Web craze just another evolutionary step. What is truly radical about the

Web is that it re-adjusts our thinking about application development. The premise in the 1990s was always a client-side application working with server-side data. But Web proponents are discussing a far different "recentralized" model. Java is a principle means of doing just that. If such a model were to succeed *in toto*, then the client-side programs we are currently creating would be effectively replaced by Java applets housed on a server.

**I Don't Think So.** Undoubtedly, the Web offers a compelling solution for many problems not solvable by standard client/server approaches (see "File | New" in the *April '96 DI*). But a complete "recentralization" revolution will never occur. The whole notion is based on a false premise, best typified in Sun's well-known phrase: "The network is the computer." As I have previously discussed on this page (in the *July '96 DI*), this vision runs counter to all our experience in the software world.

Java will not render Delphi, Optima++, or other tools obsolete, but its importance in the marketplace can hardly be underestimated. Java will find many practical uses in the server-based paradigm, but the language will surely go beyond the Web and be used for developing native Windows .EXEs as well. In spite of the endless hype, I am convinced that Java will survive as a language, regardless of the winner of the Microsoft vs. Sun/Netscape wars.

**More Tools.** The Web revolution has caused the visual tools market to broaden in scope more than ever before, with the distinctions between Web development and client/server tools fading daily. During the evolutionary phases, a developer could get away with concentrating on a single tool, be it Visual Basic, C++, Delphi, or whatever. This practice is not so simple today; the world of client/server computing continues to encompass an ever increasing number of technologies: Internet/intranet architecture, Web database access, distributed objects, and "active" Web content. Within this context, we as developers will need to use multiple tools to provide sound solutions in the marketplace. Delphi will be among those tools. So will Java.

Next month, we'll look at how you can use Delphi to create Web applications. **Δ**

— Richard Wagner

Surfing the Web? If so, visit the "File | New" home page at <http://www.acadians.com/filenew/filenew.htm>. In addition to downloading past articles, you can get the latest tips and information from the world of software development.

*Richard Wagner is Contributing Editor to Delphi Informant and Chief Technology Officer of Acadia Software in the Boston, MA area. He welcomes your comments at [rwagner@acadians.com](mailto:rwagner@acadians.com) or on the "File | New" home page (see above).*

