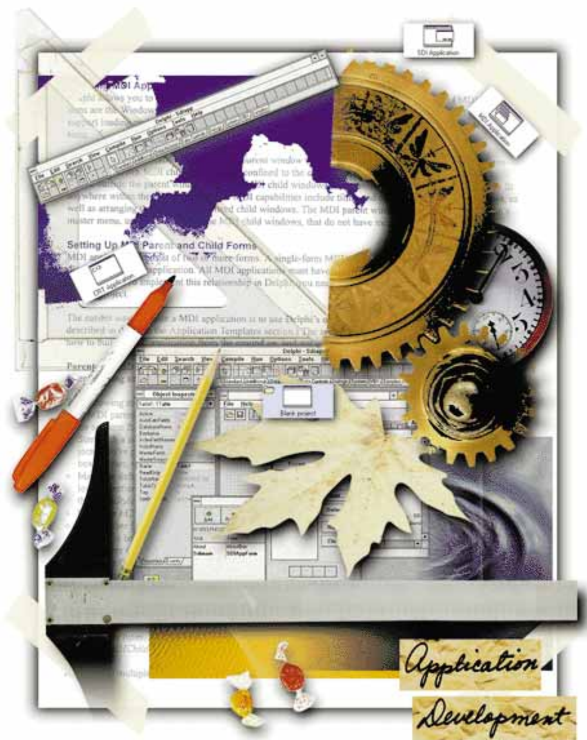
August 1995 - Volume 1, Number 4

# Application Development

## Implementing MDI, On-Line Help, and .INI Files



Cover Art By: *Victor Kongkadee*

## ON THE COVER

9 **Creating MDI Apps** — Jonathan Matcho
In the Microsoft Windows development environment, there are two ways you can go with the basic interface of any application you create: MDI or SDI. This month, Mr Matcho gets us started with the basics of the Multiple Document Interface.

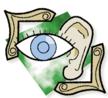
14 **A Topical Search** — Robert Palomo
Now, more than ever, any major Windows applications features an on-line, hypertext, context-sensitive help system based on the Windows help engine. Mr Palomo introduces us to the basics and then gets to the nitty gritty of implementing such a system in Delphi.

20 **Initialization Rites** — Douglas Horn
There's an accepted Windows standard for storing information from one session of an application to another: the humble .INI file. Delphi makes manipulating these initialization files particularly straightforward by encapsulating them with its *TINIFile* object, as Mr Horn explains.

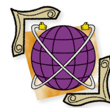

## FEATURES

25 **OP Basics** — Charles Calvert
If you come from a non-Pascal background, you may have already noticed that the Object Pascal string type is unlike its C/C++ cousin. However, Pascal does offer null-terminated strings as well. Mr Calvert makes it all plain in this first of a three-part series on strings.

29 **DBNavigator** — Cary Jensen, Ph.D.
In this month's DBNavigator, Mr Jensen kicks off a two-part series on data validation. The focus this month is on checking non-table data, featuring a **try...except** block and a discussion of field-level versus record-level validation.



34 **Sights and Sounds** — David Faulkner
Mr Faulkner adds a little color to the issue with his "color triangle". Make that, lots of color and a good measure of algebra to boot! In any case, it's a fine example of Delphi's screen-painting capabilities, compiler options, **uses** tricks, and more.



40 **On-Line** — Rand McKinney
Remarkably, a wide assortment of Delphi-centric World-Wide Web sites have already stepped onto the Internet stage. From the official Borland sites to "Delphi Hacker's Corner", Mr McKinney picks his favorites and offers an HTML to make it all easy.

43 **At Your Fingertips** — David Rippy
Our popular bi-monthly tips column is back. This month, Mr Rippy shows us how to implement a "hot regions" user interface, a poor-man's glossary system, and a quick-and-dirty way to copy a record from one table to another.


## REVIEWS


45 **InfoPower**
Product review by Joseph Fung

50 ***Delphi Nuts & Bolts***
Book review by Larry Clark

50 ***Delphi Unleashed***
Book review by Cary Jensen, Ph.D.

51 ***Delphi Programming EXplorer***
Book review by Tim Feldman


## DEPARTMENTS


2 **Editorial/Letters**

4 **Delphi Tools**

7 **Newsline**

# Editorial

***You gotta take it down to the drums. It's the only way you're gonna get it right.***
— Iggy Pop

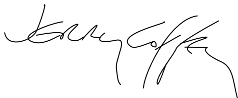
**The obscure Metallic KO is a live recording of the apocalyptic final concert of Iggy Pop and the Stooges. It captures an odd moment in music history and is remarkable in several respects, not the least of which is that it was ever released. It records the Stooges playing abominably in front of an overtly hostile audience. You can even hear bottles breaking as they hit the stage. At one point the band is so out of sync that Iggy Pop shouts over the cacophony and orders them to "take it down to the drums." Slowly, agonizingly, in front of a hooting crowd, the Stooges comply. Apocalyptic — yes. Good music — hardly.**

In this business, the readers set the tempo and I have no intention of letting us get out of sync, so I decided to take it "down to the drums" from the get-go. In June I asked you to let us know how we're doing and what types of articles you'd like to see. And you did.

I was greatly relieved to see from your letters that we're on track so far. In fact the response has been overwhelmingly positive — and for that I thank you. You also did a fine job of burying us in article requests, and they are also very much appreciated. All have been duly noted and a list has been made available to all current and prospective *DI* writers.

Taking your messages as a whole, several points came through loud and clear. For one, you *do* want an editorial section with letters to the editor, so here you are. I've attempted to capture most of the other major points with representative letters, so I'm going to let you do the rest of the talking.

I'd like to avoid the sort of humiliating, public course correction that the Stooges suffered in the early 70's (oh yeah — long before punk), so you can bet I'll be listening. Let's keep in sync.



— Jerry Coffey, Editor-in-Chief
CompuServe ID: 70304,3633

## Love the mag

**Hi Jerry: The last three issues of Delphi Informant have been fabulous. I read a ton of magazines and this one is tops. It is just right in length and wonderful in layout.**

**How about an article that carefully dissects a simple (not too simple) object? An article that hits on the constructor, destructor, and all the parts of a custom component would be very nice. ... I prefer reading about the basics of objects.**

**This is virgin territory for a lot of us out here. And keep hitting the database stuff a lot too.**

**Congrats on a fine publication. Well worth the money. — Chip Mayer**

*Thank you Chip! Although we want to hear all comments — positive and negative — this is the kind of message that keeps us going. Feel free to send them any time. <g> And don't worry — I won't run a couple of pages of "you guys are great" messages. But keep in mind that I can if I need to. <g>*

## Suggestions, suggestions, suggestions

**Jerry. A few comments on your new mag as requested. ... (1) Yes, I do like the editorial page. A useful communication that makes the magazine seem more complete. Nice to hear from the busy Editor-in-Chief. (2) I'm a beginner, but can already follow (and nearly understand) most of your articles. This does not mean that the level is not right — it is. A good mix of introductory and more advanced in fact. ... (3) I like the little tips in "At Your Fingertips". Please do keep little snippets like the "incremental search field". It reminds me to not waste money on some third-party product that offers such simple stuff! <smile> I know that lots of powerful stuff is just a matter of reading the manuals and practicing and the tips help with this. ... (4) Third-party product and book reviews are very useful. Yes — I'll buy good-value third-party components. I've also got the *Dummies* book and it is very good as you said (lots of tips again). Still waiting to buy the Sams *Delphi Developer's Guide* and Waite Groups' *Borland Delphi How-To*. ... Maybe you have some reviews coming on these before I buy? (5) Don't waste too much space on VB! Give those fellas a short while to convert, but don't waste valuable space when it's Delphi stuff we want. (6) Object Pascal basic-to-intermediate level articles would be helpful for many. ...**

**Thanks for an interesting and useful magazine. Looking forward to my next one. — Richard Entwistle, Hong Kong**

*This pithy message from Mr Entwistle is fairly typical. Most of you have responded with such thoughtful critiques. Thank you! Thank you! Thank you! Regarding the book reviews — we plan to review every Delphi book as soon as it is available. The two you mention aren't out yet, but three that are available are reviewed in this issue.*

## Not too much database please

First, I like the editorial ... How you are doing? IMHO, wonderfully! It is an information-packed magazine with something for everyone, and don't change that! ... Topic spread: Pretty good, considering you have only published two issues as of yet. Keep it up! Object Pascal: Like many coming from a strong Visual Basic background ... I don't know everything in the Object Pascal language itself yet. Thus, the more of it, the better!

Databases: Eeeiiiii! Everybody seems to assume that VB programmers program databases! As a game/edutainment author, I take umbrage at that. ... There are a lot of database programmers, so I'm not suggesting that you remove those articles from the magazine, but keep it under control if you don't want to lose the general programming contingent. ... More graphics-oriented articles would be nice.

... I noticed something missing from your magazine; a letters department ... it would be nice to see what other people think of *Delphi Informant*.

In general, I really like the magazine, and will most likely subscribe to it soon if this quality continues! Keep up the great work! — Michael

*Thanks Michael. And don't worry — we won't let the magazine be dominated by any one programming area. There's a cool graphics article beginning on page 34. BTW, have you subscribed yet? <g>*

## Give us more!

Mr. Coffey, you asked for feedback. Well, here is mine. I just received my first issue (V1N2) of *Delphi Informant*. In short, I loved it. In fact, I couldn't get enough of it. Could you please get a little more between the covers? That's the only negative thing I can say about *DI*. *DI* seems to have a pretty good balance of material. Anyway, thanks for a great magazine. I've already cleared a spot on the bookshelf for future volumes.

By the way, have you noticed how many WWW sites are forming around Delphi? Amazing! Do you think *DI* will talk about the offerings of these sites? — bt

*One thing virtually everyone agreed on was that DI could be a bit bigger. All of you should be happy to note that this month's issue is 16 pages longer (that's one "signature" in the biz). It certainly makes me happier. From an editorial stand-point it allows us to offer broader coverage. Our WWW coverage begins on page 40.*

## Not technical enough

I just read the first issue of the *DI*. I was very excited to read some articles with deep insight into Delphi and the VCL ... I think, *DI* has a good chance to become a major Delphi developers information magazine, but I would recommend that you offer articles that are more technical. ... One could write about the architecture and implementation of the VCL database components ... OLE programming with Delphi ... a dockable toolbar ... Mail-access components ... There are lots of complicated things to write about.

This criticism should not be taken as a very strong one. I can imagine it is a lot of work to start a new magazine. It should be taken as a friendly hint ... what a Delphi developer with some practice would like to read.

Thanks for your work on *Delphi Informant* — Christian Abeln

*Are there ever lots of technical things to write about! And thank you Christian for the "friendly hint" to supply more advanced articles. (There were a couple of similar letters that weren't as kind.) However, there were also many letters exhorting us not to become too technical and leave those learning Delphi behind (one of them follows). We fully intend to keep a good balance by offering articles for programmers at all levels of expertise.*

## Too technical

Dear Sir,

... I am frustrated with some of the articles provided in the first two issues of *DI*. I am brand new to Delphi, Pascal, and OOP. My background is DOS and BASIC, so Delphi is quite a big undertaking for me. ... My subscription to *DI* is part of this process.

In particular, I find that there are some assumptions made on the part of some writers that their readers know certain terms or lingo. For example, in issue one, you had an article by Dan Ehrmann called "Data-Aware Delphi". It dealt with modifying a query component via a tabbed interface. ... even though this article is identified for "Beginner/Intermediate", nowhere could I find information on what a *table* is! ... Questions arose in my mind: 1) What is a table? 2) How are tables created? Do they float down out of the sky? 3) How does one add data to a table? 4) Is a .DBF file a table? ... then the author throws the term *query* around as if everyone knew what a query was. ...

Thank you for your time.

— Philip Kapusta

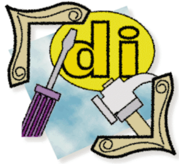
*Too technical, not technical enough — these are the kinds of things that keep editors awake nights. I'm afraid there's little I can do except try to steer down the middle. As long as these letters are in balance, we're on course — and your feedback is critical.*

*Let me close with a final question. Philip refers to a practice we used only in the premiere issue of DI (which is sold out BTW — thank you very much), that of labeling articles as Beginner, Intermediate, etc. In practice, I found this distinction a highly subjective one and simply quit doing it. A number of you have asked for its return however, so please let me know how you feel. I can always bring it back if you find it useful.*

*Thanks again for all your messages. — J.C.*

# Delphi TOOLS

New Products and Solutions



## MKS' Source Integrity Released

**Mortice Kern Systems** of Waterloo, ON has released a new version of ***MKS Source Integrity*** (formerly MKS RCS). It brings users a new and complete configuration management system that encourages multi-user development. Version 7.1 fits into development environments, and will include Visual Merge, enhanced reporting capabilities, event triggers, a new configuration language, integration into Visual C++ and Borland C++, and other project management facilities.

MKS Source Integrity is the only PC version control system offering an interactive, visual merge facility. Developers will save time by merging files with a mouse click.

Source Integrity automates your configuration management system even more with event triggers. Time is saved by configuring particular events to automatically trigger a second event. For example, MKS Source Integrity can be configured to automatically mail a confirmation message to the project leader each time a team member checks in a new file.

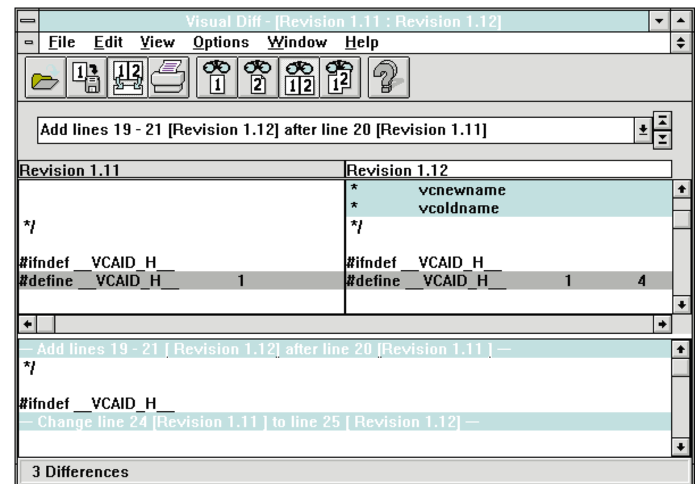
In addition, Source Integrity features a fully-customizable toolbar and easy-to-use configuration.

**Price:** US$449, upgrade for current users US$149.


**Contact:** MKS 185 Colombia Street West, Waterloo, ON, Canada, N2L 5Z5

**Phone:** (519) 884-2251

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


## Access Btrieve Data with Titan for Delphi

**AmiSys Inc.**, of Concord, CA has released ***Titan for Delphi***, a Btrieve interface for Delphi developers. Titan is a drop-in replacement for the IDAPI used in Delphi. It allows Delphi users to access Btrieve data files without using the ODBC layer provided through IDAPI. Also, Titan allows the developer to use all the standard Delphi data-aware components, such as grid, edit, checkbox, and list box controls without modifications.

Titan doesn't require any IDAPI dynamic link library (DLL) files to be installed or configured on the target system. It supports multiple databases through the use of Btrieve standard Data Definition files (DDFs).

Titan supports all the Delphi visual development tools such as the Database Form Expert, Field Editor, and Master/Detail relationships between tables.

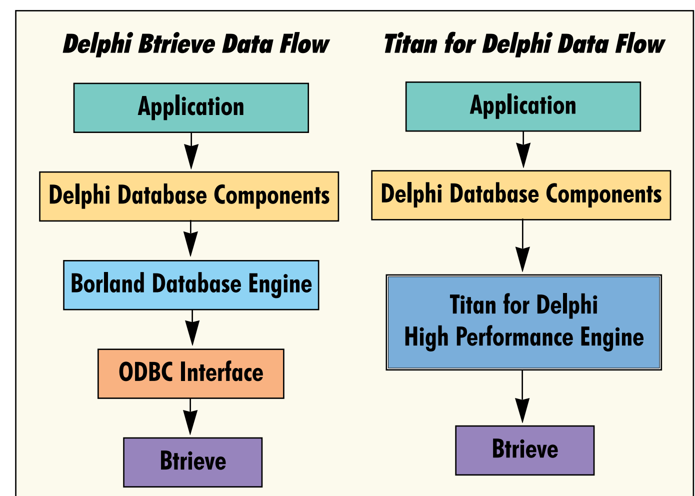
In the future, AmiSys plans to release a TQuery component, integrate calls to the IDAPI, and enable Titan to access non-Borland databases.

**Price:** Titan for Delphi Runtime Version, US$295.


**Contact:** AmiSys Incorporated, 1390 Willow Pass Road, Suite 930, Concord, CA 94520-5253

**Phone:** (510) 671-2103

**Fax:** (510) 671-2104


### New Delphi Books

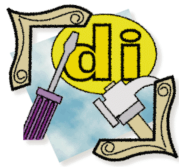
**Teach Yourself Delphi in 21 Days**
*Andrew J. Wozniewicx & Namir Shammas*
Sams Publishing
**ISBN:** 0-672-30470-8
*Teach Yourself* uses example and full code listings to explain how to combine visual programming with Object Pascal. It covers DLLs, DDE, and OLE components, exception handling, program architecture, and more.
**Price:** US$29.99 (912 pages)
**Phone:** (800) 488-5233

**Borland Delphi How-To**
*Gary Frerking, Wayne Niddery, & Nathan Wallace*
Waite Group Press
**ISBN:** 1-57169-019-0
*Borland Delphi How-To* presents over 100 programming problems and their solutions. It covers graphics, document, multimedia, database, OLE, and DDE issues. This book ships with a CD-ROM containing resources, bitmaps, and custom components.
**Price:** US$39.95 (700 pages, CD-ROM)
**Phone:** (800) 368-9369

**Delphi**

**T O O L S**

New Products and Solutions



### Delphi Training

**UDP Solutions Center**, a Borland training member, is offering an advanced course in Delphi Client/Server application development. Borland's official course material is taught at an accelerated pace, and concludes with supplemental materials relating to client/server applications, such as an exercise building an application using an MS SQL server in a true client/server environment.

UDP will be offering client/server application development using Delphi (5 days) US$1875.00 on Aug. 28 - Sept. 1 and Sept. 11-15. For more information and registration materials, contact: Christopher Simmons at (503) 690-6877 or e-mail chriss@udp.com

## Delphi Component Creates Paradox for Windows Reports

**Kirsch & Partners Limited** of Markham, ON has announced the release of ***PdoxRept***, a reporting component for Delphi.

With PdoxRept, Delphi developers can create reports using Paradox for Windows, and deliver executables to users along with the Paradox report files (RDLs or RSLs). PdoxRept will print without displaying splash screens or start-time requests. And with RDLs, developers can be sure the reports won't be accidentally modified.

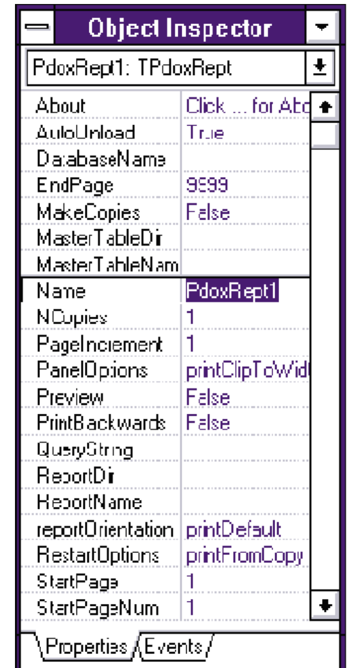
PdoxRept offers approximately 24 properties, giving developers full control over printing, including the number of copies, page orientation, overflow control, the base table for reporting, aliases, page numbering and offsets, and previewing.

PdoxRept ships with the latest run-time version of Paradox for Windows, so users do not need to own or learn Paradox. The license agreement allows developers to distribute executable code without additional license or fee.

**Price:** Introductory price, US$99.95 (regularly US$149.95).

**Contact:** Kirsch & Partners Limited, 3105 Markham Industrial Park, Markham, ON, Canada L3R 6G4

**Phone:** (416) 967-1660




## Syware Announces Dr. DeeBee Tools

**Syware Inc.**, of Cambridge, MA has announced the release of ***Dr. DeeBee Tools***, a suite of utilities for ODBC developers and product support organizations to monitor the operation of ODBC-enabled applications and drivers.

Dr. DeeBee Tools provide insight into the performance and inner workings of ODBC. It includes a suite of seven utilities: Dr. DeeBee Check (semantically checks ODBC applications at run-time), Dr. DeeBee Peek (records ODBC calls made by an application), Dr. DeeBee Timer (counts the ODBC calls an application makes and their duration), Dr. DeeBee Test (runs regression tests on ODBC drivers), Dr. DeeBee Spy (a version of Dr. DeeBee Peek for product support organizations that can be re-distributed), Dr. DeeBee Replay (replays logs generated by Dr. DeeBee Spy), and Dr. DeeBee Info (displays description and characteristics of a driver).

ODBC application and driver writers, along with their customer support organizations, use Dr. DeeBee to monitor, optimize, and test ODBC products and projects.

Dr. DeeBee Tools reports on the ODBC performance of an application. It helps developers determine how the application is using ODBC, and if the application is using ODBC properly. ODBC driver writers use Dr. DeeBee Tools to determine the use and performance of their driver, and to run tests against the driver. Support organizations for ODBC products use Dr. DeeBee Tools to determine their customers' ODBC problems and to pinpoint whether the problem is in the application or driver.

Dr. DeeBee Tools work with all ODBC applications, including those created by Microsoft's Access, Visual Basic, and Borland's Paradox, Delphi, and C/C++. Dr. DeeBee Tools are compatible with both ODBC 1.0 and ODBC 2.0.

**Price:** US$399 for the Super Bundle (all seven tools); or US$199 for the Starter Bundle (Info, Peek, and Timer). Corporate site license and reseller prices are available.

**Contact:** Syware Inc., P.O. Box 91 Kendall, Cambridge, MA 02142

**Phone:** (617) 497-1376

**Fax:** (617) 497-8729

Dr. DeeBee Timer

| Function | # calls | Time(ms) | Ave.(ms) |
|---|---|---|---|
| SQLError | 2 | 2 | 1 |
| SQLExecDirect | 62 | 4438 | 71 |
| SQLExecute | 12 | 343 | 28 |
| SQLExtendedFetch | 0 | 0 | 0 |
| SQLFetch | 46 | 72 | 1 |
| SQLForeignKeys | 0 | 0 | 0 |
| SQLFreeConnect | 8 | 155 | 19 |
| SQLFreeEnv | 5 | 106 | 21 |
| SQLFreeStmt | 59 | 47 | 0 |
| SQLGetConnectOption | 8 | 13 | 1 |

OK

**Delphi**
**TOOLS**

New Products and Solutions

# MicroHelp Releases SpellPro 2 & Thesaurus

**MicroHelp, Inc.** of Marietta, GA has recently released ***SpellPro 2 & Thesaurus***, an upgrade to its SpellPro tool that enables programmers to build spell check functions into visual applications. SpellPro 2 & Thesaurus includes a thesaurus that uses a dictionary of more than 50,000 synonyms. The package contains DLL, VBX, and 16/32-bit OLE controls.

SpellPro 2 now supports multiple custom dictionaries while its new dictionary utility gives more than 50 percent compression and is three times faster when exporting a dictionary to a text file. The common-word cache has been increased from 25 to 1024 words and includes user-definable language settings. Additionally, SpellPro's preload function decreases start-up time.

SpellPro 2 also features built-in dialog boxes to reduce coding time, AutoLinkHwnd functionality that allows applications to be automatically linked to the DLL, and multiple discardable code segments that reduce the amount of memory needed.

SpellPro 2 & Thesaurus can check 60,000+ words per minute using a Pentium/66 processor. It also features utilities for building individual dictionary and thesaurus files, and medical and legal dictionaries. There are no run-time royalties for distributed applications.

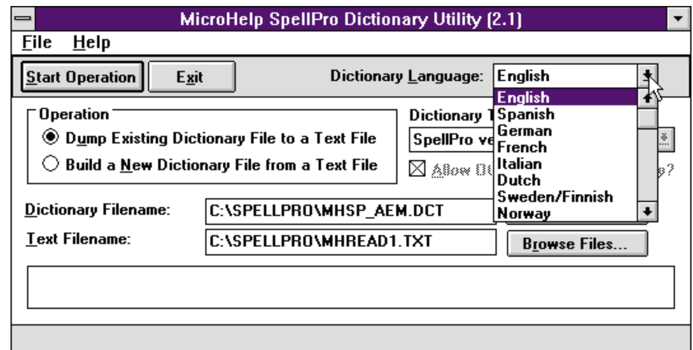
**Price:** US$129

**Contact:** MicroHelp Inc., 4211 J.V.L. Industrial Park Drive, NE, Marietta, GA 30066

**Phone:** (800) 922-3383, or (404) 516-0899

**Fax:** (404) 516-1099

**BBS:** (404) 516-1497




**EMS Professional Shareware** has released ***Professional Shareware Libraries***, adding three new libraries for Delphi, Clarion, and NT, featuring over 100 tools for developers and consultants. Each library includes a search program allowing users to search for a utility by product name, producer, type, release date, or with a free text search across all information in the database. The first library purchased costs US$59.50, and any additional libraries are US$25. For more information contact EMS at (301) 924-3594, or e-mail ems@wdn.com.

# ProtoView Releases DataTable

**ProtoView Development Corp.** of Cranbury, NJ has released ***DataTable version 2.7*** for Delphi. DataTable is a Windows grid control that gives applications the look and feel of a professional spreadsheet. It provides visual access to database tables, and allows Delphi developers to create powerful database manipulation applications without coding.

DataTable supports over 32 different data types, including Microsoft SQL Server, Sybase SQL server and more. The Delphi developer has complete control of the DataTable through design-time Object Inspector settings, API, and numerous notification codes.

DataTable features design time visual setting via the Delphi Object Inspector, Delphi Data Sources support, cell formatting, editing within individual cells, row and column selection, and horizontal and vertical grid lines.

It also includes three-dimensional row and column styles, column re-sizing, hidden columns, support for database null values, multi-line column and row labels, automatic insertion of empty rows, multi-line data entry in a cell, and advanced methods and notifications. With DataTable, developers can set fonts and colors for row and column labels, sort with up to three sort keys, and display cells as check, list, or combo boxes.

**Price:** US$149; source code, US$495. Applications created using the DataTable may be distributed royalty free.

**Contact:** ProtoView Development Corporation, 2540 Route 130, Cranbury, NJ 08512

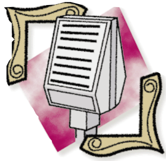
**Phone:** (800) 231-8588 or (609) 655-5000

**Fax:** (609) 655-5353

# News

L I N E

August 1995



**Borland International Inc.'s Sixth Annual Developers Conference**, to be held August 6-9 in San Diego, CA, will focus on helping developers make a smooth transition to Windows 95. The conference is expected to attract over a thousand developers from around the world and will feature over 200 sessions. Registration for a single attendee is US$1,195, and US$1,145 each for three or more people from the same company. For more information or to register, call (800) 350-4244 or (805) 495-7800 (international). Additional information about Borland Developers Conference is also available on Borland Online, Borland's World Wide Web site at http://www.borland.com.

**PC Database Summit**, a national conference for managers, system analysts, and developers of database applications on PCs, is scheduled for August 20-22, 1995 in Seattle, WA. The conference will include seminars and workshops, two database competitions, an exhibition hall, and a computer lab. Summit '95 will be held at the Sea-Tac Red Lion Hotel near the Seattle Airport and costs US$395. Groups who purchase three tickets will receive an additional free pass. For more information, or to register, call (800) 497-7060.

## Visual Components Made Delphi Ready

*Lenexa, KS* — Visual Components, Inc. has announced the release of Delphi-specific declaration fields for Formula One, First Impression, and VisualSpeller. These products are now shipping with current builds of each product.

The declaration files make it easier for developers using Delphi to add high-quality spreadsheet, charting, and spell checking functionality to Delphi applications.

According to the company, the declaration files make it easier for Delphi developers to access the core DLL of each product. This is important because all the functionality of Formula One, First Impression, and VisualSpeller is held in the core DLL. By using the declaration files, Delphi developers can easily access and use this functionality.

For more information visit Visual Components on CompuServe (GO VIS-TOOLS) or contact them at (913) 599-6500.

## Borland Conference Europe 96: A Call for Papers

*Portsmouth, UK* — Desktop Associates Ltd. and Dunstan Thomas Ltd. are accepting papers for the Borland Conference Europe 96, scheduled for April 28 - 30, 1996 in London.

There are three types of presentations: Pre-Conference Tutorials, Technical Presentations, and Case Studies.

The technical sessions are organized into three major tracks: Delphi; Paradox and Visual dBASE; and C++, Decision Support Tools, InterBase, and General Interest. Each track has a number of threads, however, they do not form the definitive list of topics that may be accepted.

The Delphi track will include: Solutions, Programming, Tools and Techniques, Methodologies, Client Server, and Pre-Conference Tutorials.

All overseas speakers will have their airfare and travel paid, and speakers will have their accommodations arranged and paid.

To apply as a speaker, send a personal profile or "bio" for publishing in conference materials, a session title (under 100 characters), a short description (under 30 words) of your presentation, and an abstract of you presentation (under 1000 words) in Microsoft Word for Windows or ASCII format.

*All materials must be submitted no later than September 1, 1995.*

Speakers will be selected and required to submit a written version of each presentation, code samples, and presentation slides.

For submissions please contact Chris Read at Dunstan Thomas Ltd. by phone: +44 (0)1705 822254, fax: +44 (0)1705 8223999, or e-mail: cread@dt.mhs.compuserve.com or 100014,2273.

Or write to: Chris Read, Dunstan Thomas Ltd., The Old Treasury, 1 St. Paul's Road, Portsmouth, PO5 4JU, UK. Please mark correspondence BCE 96.

## Borland Previews Delphi32

*Scotts Valley, CA* — Borland International will be showing the upcoming 32-bit version of Delphi at the Sixth Annual Borland Developers Conference this month. The version has a number of new features such as forward compatibility, and OCX and Windows NT support.

With Delphi32, 16-bit Delphi applications can be converted to 32-bit applications with a simple recompile. The only exception would be if the 16-bit program uses a Windows API call that has been modified between Windows 3.11 and Windows 95 — an unlikely occurrence.

Delphi32 will have complete OCX support, including a new page on the Component Palette with sample OCXes. It will support the entire Windows 95 API (although some Windows 95 API functions will not be "wrapped"), and encapsulate all Windows 95 user-interface elements. Delphi32 will also meet all of Microsoft's Windows 95 logo requirements. For example, Delphi32 will be a fully-functional Windows NT product.

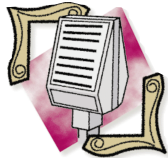
The back-end of the Delphi32 compiler engine will be shared with the Windows 95 version of Borland C++. This enables users to optionally create C++ .OBJ files instead of Delphi's native .DCU files. The compiler will be faster and take advantage of the Windows 95 32-bit flat memory model. All products bundled with Delphi (Borland Database Engine [BDE], Database Desktop, and ReportSmith) will also be 32-bit versions.

Delphi32 and Delphi32 Client/Server will begin shipping within 90 days of Windows 95. Due to the uncertainty of Windows 95 acceptance, Borland will continue to ship the 16-bit version of Delphi.

# News

L I N E

August 1995



## Delphi Seminars Now Available Worldwide

*Scotts Valley, CA* — Borland International Inc.'s Delphi for Windows and Delphi Client/Server application development tools are rapidly gaining worldwide attention following numerous awards from industry publications. The mounting interest in Delphi has led Softbite International, DSW Group, and Zachary Software to offer worldwide seminar series for Delphi developers.

According to Borland, the Delphi series will put developers on the fast track for building the next generation of Windows and client/server applications. With the forthcoming Windows 95 version of Delphi, attendees will learn how they can easily upgrade their applications to full 32-bit performance.

Beginning this summer, Softbite International, DSW Group, and Zachary Software will collectively reach more than 50 cities worldwide to offer developers training on using Delphi to build stand-alone, LAN, and client/server applications. Each company will provide one- and two-day seminars beginning in July and running throughout the year.

For information on seminar locations and dates, log on to Borland's Internet World Wide Web site at htpp://www.borland.com. Or contact Softbite International (708) 833-0006; DSW Group (800) OK-DELPHI; or Zachary Software (800) GO-DELPHI.

## Informant Forum Opens on CompuServe

*Elk Grove, CA* — Informant Communications Group, Inc. (ICG) launched its Informant Forum on CompuServe in late June. The forum was created to foster the exchange of technical information among developers using Borland's Paradox and Delphi, and to give Informant readers an easy way to reach ICG.

"Our forum has already proven to be a great way to reach our readers," said Mark Wiseman, Operations Manager for ICG. "We're able to respond to readers' questions within minutes." If you have a question about your subscription, would like to subscribe, or need other general company information, simply post a message in the customer service area.

For technical information, areas have been designated for various topics. Informant authors, as well as other industry experts, will be surfing Informant Forum to lend their expertise.

Library 19 is dedicated to Peter F. MacGruder fans. Upload your most Mac-worthy news, and if he uses your material in a column, you'll receive an Informant T-shirt or baseball cap.

Informant Forum also has several libraries for Paradox and Delphi downloadables. View the Delphi custom components, utilities, and other files available. For prospective writers, the *Paradox Informant* and *Delphi Informant* Writer's Style Guides and Editorial Calendars are available for download in Library 21 (Editorial Information).

To visit the Informant Forum type "GO ICGFORUM" at any CompuServe GO prompt. If you want to join CompuServe, you can obtain a CompuServe starter kit including a US$15 usage credit by calling toll-free in the United States (800) 524-3388 and asking for REP number 547.

### ICG CompuServe Forum Structure

**Message Sections:**

| # | Title | Description |
|---|---|---|
| 1 | Customer Service | Customer service questions |
| 2 | Pdox DOS (ALL) | Paradox for DOS discussions |
| 3 | Pdox Win Form/Rpts | PW form or report discussions |
| 4 | Pdox Win Queries | PW query discussions |
| 5 | Pdox Win ObjectPAL | PW ObjectPAL discussions |
| 6 | Pdox Win Network | PW networks discussions |
| 7 | Pdox Win Wish List | PW wish list discussions |
| 8 | Delphi Database | Delphi database discussions |
| 9 | Delphi Win API | Delphi Windows API discussions |
| 10 | Delphi Components | Delphi components discussions |
| 11 | Delphi Obj Pascal | Delphi Object Pascal discussions |
| 12 | Delphi Wish List | Delphi wish list discussions |
| 22 | Jobs/Help Wanted | Help wanted/job opportunities |
| 23 | Database Industry | Database industry discussions |

**Library Sections:**

| # | Title | Description |
|---|---|---|
| 1 | General - Cust Srv | Subscription information |
| 2 | New Paradox Uploads | New Paradox Informant code samples |
| 3 | New Delphi Uploads | New Delphi Informant code samples, etc. |
| 5 | Pdox DOS 4.x | Paradox for DOS magazine code samples |
| 6 | Pdox DOS Demo/Share | Paradox for DOS demo/shareware programs |
| 8 | Pdox Win 4.x | PW 4.x magazine code samples |
| 9 | Pdox Win 5.x | PW 5.x magazine code samples |
| 11 | Pdox Win Demo/Share | PW tools/demo/shareware programs |
| 12 | Delphi 1.x | Delphi Informant code samples |
| 14 | Delphi Demo/Share | Delphi tools/demo/shareware programs |
| 17 | General Windows | General Windows utility files |
| 19 | Ltr to Ed/MacGruder | Upload private files to Editorial Dept./MacGruder |
| 20 | Back Issues Info | Back Issue catalog file |
| 21 | Editorial Info | Editorial information downloads, including writer style guides and editorial calendars |
| 22 | Jobs/Help Wanted | Employment opportunities |
| 23 | ICG News | Informant Communications Group news |

## ICG Publishes New Delphi Power Tools Catalog; Readies Paradox and C++

*Elk Grove, CA* — Informant Communications Group, Inc. (ICG) has released a new Delphi Power Tools catalog. In addition, ICG is accepting advertising for the upcoming Paradox and C++ Power Tools catalogs. Each catalog features third-party add-in products and services that complement Delphi, Paradox for Windows (PW), or C++, respectively. These catalogs are independently produced by ICG and inserted into each copy of Delphi, PW, or C++.

The next edition of Delphi Power Tools is already underway. For advertising information, phone Lynn Beaudoin at (916) 686-6610, ext. 17 (e-mail: 74764,1205) or Sheri Birkmaier at (916) 686-6610, ext. 21 (e-mail: 76072,1720).

By *Jonathan Matcho*

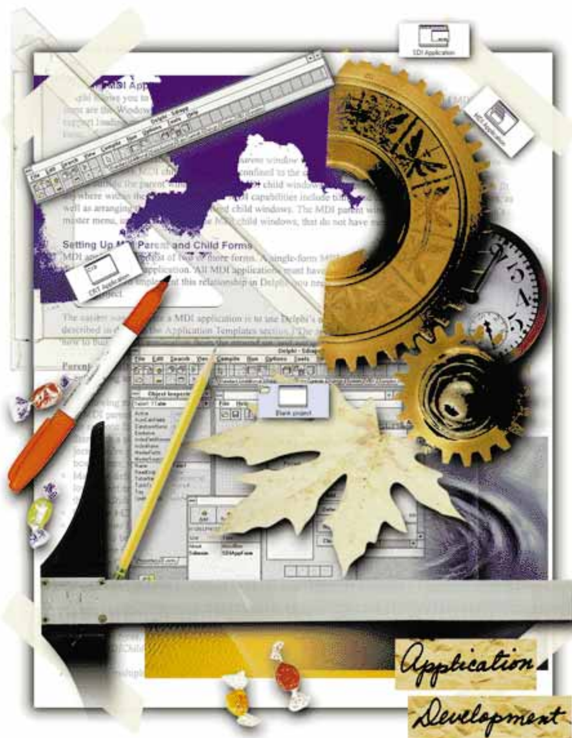
# Creating MDI Apps

## Multi-Form Delphi Applications: Part I

**While a form alone can certainly be considered an application, applications typically consist of several forms. When developing a multi-form Windows application, you can choose from two major ways of presenting the overall application interface: the Multiple Document Interface (MDI), or Single Document Interface (SDI).**

Each style presents an application to the user in a different way. MDI applications have been thought of as "proper" Windows style. However, SDI applications are becoming more popular — Delphi is an excellent example.

In this first part of a two-part series, we'll focus on MDI application design. Specifically, we'll discuss: learning the basics of setting up an MDI application, designing code to control MDI applications, learning about MDI-related properties and methods, and developing an example MDI application.




### Creating MDI Applications

Delphi allows you to create Windows applications that support the MDI standard. Examples of MDI applications are Windows Program Manager, File Manager, Microsoft Word, and most Windows text editors that support loading multiple files simultaneously.

MDI-compliant applications have an MDI *parent window* with a client area that displays one or more MDI *child windows* (or *children*). MDI child windows are confined to the client area, meaning that child windows can't extend outside the parent window's border.

MDI child windows can be minimized, maximized, or sized to fit anywhere within the client area. Typical MDI capabilities include tiling and cascading MDI child windows, as well as arranging the icons of minimized child windows. The MDI parent window contains the application's master menu, used to manipulate the MDI child windows. The child windows do not have menus.

### Setting Up MDI Parent and Child Forms

MDI applications consist of two or more forms. A single-form MDI application is contradictory and should be developed as an SDI application. All MDI applications must have a parent window with at least a single MDI child window. To implement this relationship in Delphi, you need to manage at least two form files in your project.

The easiest way to create an MDI application is to use Delphi's own MDI Application project template. To better understand the process, however, we'll go through the steps of building an MDI project from the ground up. A *project* is a collection of related forms and units that work together to make up an application.

Start with a new project by selecting **File | New Project** from the menu. If you have the Gallery option on for new projects, you're prompted with the Browse Gallery dialog box shown in Figure 1. If the Browse Gallery dialog box appears, select the Blank project template and press the **OK** button.

Make the default form an MDI parent form by setting its *FormStyle* property to *fsMDIForm*. (We'll discuss this more below.) Name the MDI parent form *frmMDIParent*.

Add another form to the project (it will be used for MDI children) by selecting **File | New Form**. This causes a new form and unit to be added to your project. If you have the **Gallery** option on for new forms, you are again prompted with the Browse Gallery dialog box — this time for form templates. Select the Blank form template and click the **OK** button.

Using the Object Inspector, set its *FormStyle* property to *fsMDIChild*, name it *frmMDIChild* and give it a caption, such as "Child". The name is used as the basis for the form's class name, and the caption is the text that's placed in the title bar. (Tip: Giving your form objects, and unit and project files meaningful names makes them easier to work with and document.)

Once this is done, a new unit appears in the Project Manager. Bring the Project Manager on-screen by selecting **View | Project Manager**. Figure 2 shows the Project Manager window with two form types defined. Note the term *form types*. In MDI terms, a form's type indicates whether it is an MDI parent — *fsMDIParent*, or child — *fsMDIChild*.

At run-time, multiple instances of these new form classes can be created and are limited only by system resources. The term *classes* is used here, since when adding a new form to a project, we are essentially sub-classing the basic form class, *TForm*. This is shown by this code from a unit's **type** declaration:

```
type
  TfrmChild = class(TForm)
    ...
```

## Referencing MDI Child Windows

There are many properties and methods that enable you to control MDI child windows within your application. The following example of an imaginary company illustrates several techniques for controlling MDI child windows.

Suppose you have a user request from Bob, the head of the human resource department at a company called XYZ. Bob has just requested that a function be added to XYZ's Bonus Calculation system to enable creation of additional bonus calculation windows within the application. Bob explains that a **New Window** option on the **Window** menu is necessary so he can calculate his bonus multiple times, receiving a bonus check for each additional window he creates. After failing to persuade Bob that this is not the most ethical way to increase one's compensation, you reluctantly agree to implement Bob's new feature.

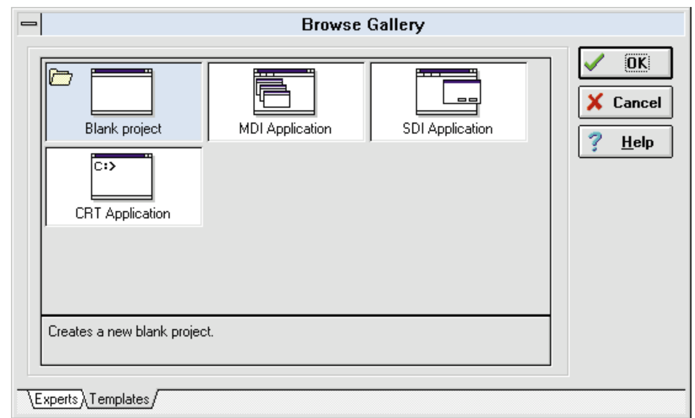



**Figure 1:** The Browse Gallery dialog box for projects is displayed when the **Gallery: Use on New Project** option has been selected at the Environment Options dialog box. The MDI Application template supplied with Delphi will get you off to a great start.

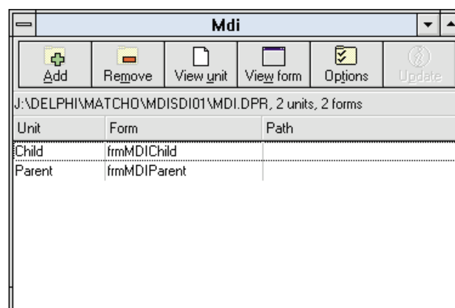


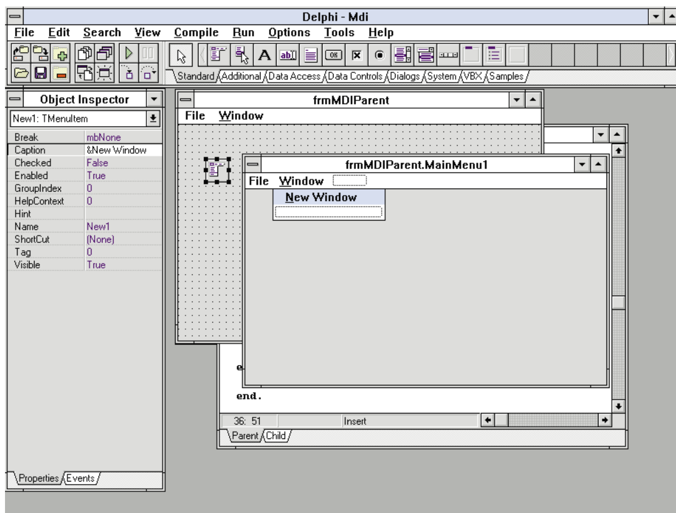

**Figure 2:** The Delphi Project Manager. A project can contain many forms, but only two types are needed for an MDI application.

You determine that you'll simply add a new menu choice to the system, enabling the creation of a new MDI child window by creating another instance (instantiation) of the child window form. You can create this menu by dropping a menu component on the Parent form. Then right-click on the component, select Menu Designer, and define the menu (see Figure 3).

To complete the task, modify the *OnClick* procedure for the **Window | New Window** menu choice (by double-clicking on *OnClick* on the Events page of the Object Inspector, as shown in Figure 4). Here's the modified procedure:

```
procedure TfrmMDIParent.menuNewClick(Sender: TObject);
var
  frmChildTemp: TfrmMDIChild;
begin
  { Create a new instance of the MDI child window form }
  frmChildTemp := TfrmMDIChild.Create(Self);
end;
```

In this procedure, the **var** statement declares a form object named *frmChildTemp* that is of the *TfrmMDIChild* class (the class created when the MDI child form was added to our project). This is also the same syntax by which new variables are defined. The object type *TfrmMDIChild* is available from the



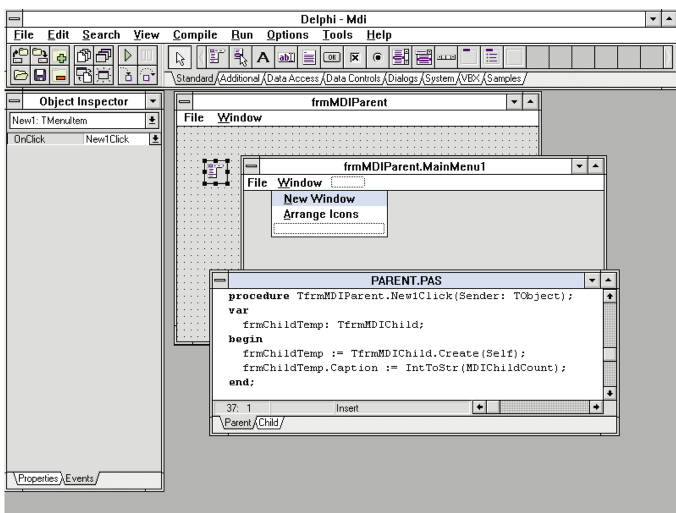




**Figure 3 (Top):** Creating the menu for our example application.
**Figure 4 (Bottom):** Modifying the OnClick procedure for the **Window | New Window** menu item.

MDI child unit that you added to the project earlier. Attached to the child unit is a form named *frmMDIChild*, containing the definition for the *TfrmMDIChild* class.

Now Bob can create new child windows at will (see Figure 5). This example highlights how child windows (which are essentially objects) are created by using a handle of the same object type as that of the actual window's class. Additional properties and methods can be used to control MDI applications. We'll discuss these properties as well as creating new MDI child windows.

## Useful Properties for MDI Window Management

The highest window class, *TForm*, includes a number of properties that support MDI parent and child windows. You can use these properties to affect the behavior of all child forms in your MDI application. Since forms are used as the basis of the user interface, changing overall form-handling properties often has a major effect on interface functionality.

A number of MDI parent window properties that are read- and run-time only are not available in the Object Inspector at design

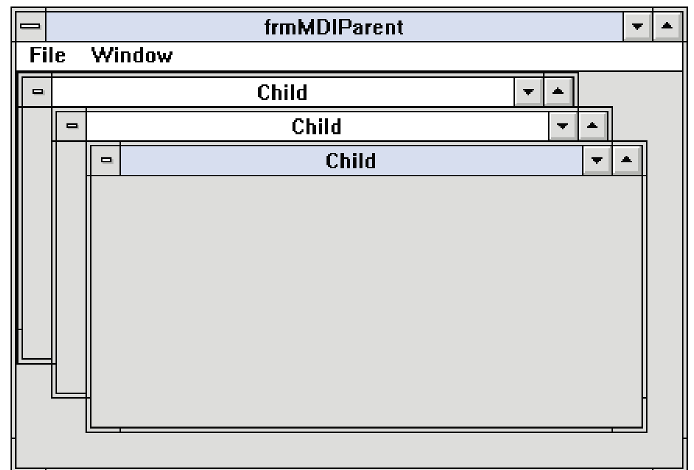



**Figure 5:** The results of the first try at creating child windows.

time. This is because these properties aren't applicable during form design. The read- and run-time properties are:

- *MDIChildCount*: An integer value containing the number of currently open MDI child windows.
- *ActiveMDIChild*: A value of *TForm* type that can be used as a handle to manipulate the active MDI child window.
- *MDIChildren*: An integer-indexed array of *TForm* containing MDI child window handles in the order the child windows were created.

As discussed earlier, two types of forms are required in MDI applications — one for the parent form and another for the child form. We have already discussed how to set a form to become an MDI parent window. Likewise, MDI child windows must have their *FormStyle* property set to *fsMDIChild*.

All MDI applications consist of a parent form having a *FormStyle* property value of *fsMDIForm* that identifies the parent MDI form. The parent MDI form must also be referenced in the application's *CreateForm* method, which Delphi handles automatically. The *FormStyle* property is set at design-time in most cases, but it can also be set at run-time.

Changing a child window's *FormStyle* property from *fsMDIChild* to *fsNormal* enables that window to be brought outside the parent MDI window, as it's no longer a child window. This can also be done at design- or run-time.

The *Visible* property determines whether a visual object is shown or hidden at run-time. Typically, MDI parent forms are not hidden. It's also somewhat odd to consider hiding child forms, and probably the reason this is simply not allowed in a Delphi MDI application.

## Incorporating MDI Window Methods

Now we'll turn our attention to the additional methods of the *TForm* class that work with MDI windows. Along with the *Create* method used in our previous example, there are a number of other methods that are useful when assembling MDI applications. MDI methods are recognized only by

MDI parent windows (those with a *FormStyle* property value of *fsMDIForm*).

To implement Bob's outlandish request, we had to dynamically create MDI child windows at run-time whenever the **Window | New Window** menu choice is selected. The following procedure is identical except that the *MDIChildCount* property is being used to increment the child windows' captions:

```
procedure TfrmMDIParent.menuNewClick(Sender: TObject);
var
  frmChildTemp: TfrmMDIChild;
begin
  frmChildTemp := TfrmMDIChild.Create(Self);
  frmChildTemp.Caption := IntToStr(MDIChildCount);
end;
```

Figure 6 shows an MDI parent form after **Window | New Window** has been selected four times. Note that the child windows now have different captions.

## Arranging Icons

The *ArrangeIcons* method organizes the icons of minimized MDI child windows so they're evenly spaced along the bottom of the parent MDI window. The *ArrangeIcons* method must be sent only to MDI parent windows (again, those having a *FormStyle* property value of *fsMDIForm*).

The *ArrangeIcons* method is typically linked to your MDI application's **Arrange Icons** command on the **Window** menu (if you have one, otherwise you're probably not implementing a true MDI-compliant application). Here is an example of the *ArrangeIcons* method:

```
procedure TfrmMDIParent.menuArrangeIconsClick(
             Sender: TObject);
begin
  TfrmMDIParent.ArrangeIcons;
end;
```

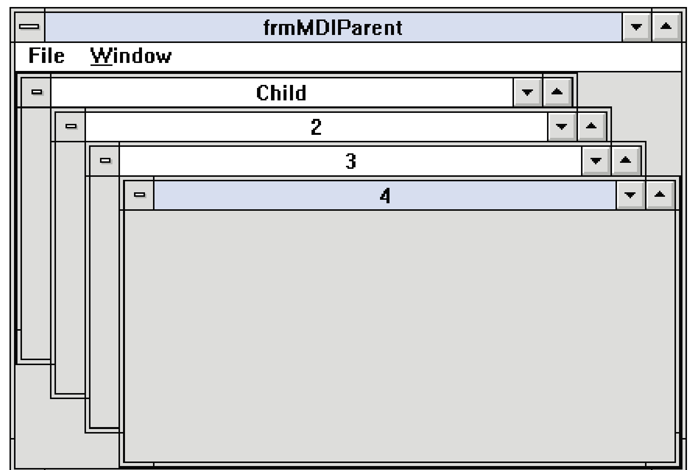
Alternatively, you can abbreviate the *ArrangeIcons* statement by using a more object-oriented syntax:

```
procedure TfrmMDIParent.menuArrangeIconsClick(
             Sender: TObject);
begin
  ArrangeIcons;
end;
```

Abbreviations of this type apply to all MDI methods as well as for all properties belonging to the **procedure** class.

Add this code to your form and then create and minimize some MDI child windows. Then, shuffle their arrangement on the MDI parent window. Selecting **Window | Arrange Icons** causes the MDI child window icons to be arranged as shown in Figure 7.

You may wonder why you have to bother issuing an *ArrangeIcons* method, since virtually all MDI applications provide support for arranging icons, tiling, and cascading their child windows. There is indeed an easier way that involves using an *application template*. Delphi's application templates are simply prebuilt prototypes that can contain code that you can reuse. After reading this article, you will be able to modify and tailor an MDI application template to your specific needs.



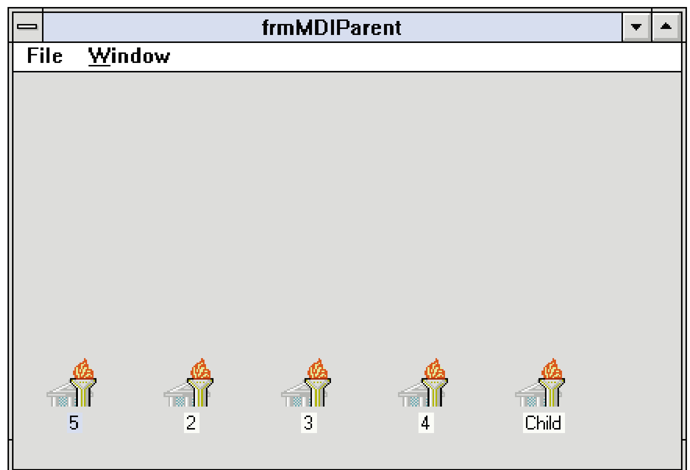




**Figure 6 (Top):** Now the child windows have unique titles based on the *MDIChildCount* property. **Figure 7 (Bottom):** The result of the *ArrangeIcons* method.

## Cascading MDI Windows

The *Cascade* method arranges the child windows so they overlap each other. A cascaded set of windows shows title bars of as many windows that fit on-screen, allowing the user to easily choose an available MDI child window.

Here's an example of how the *Cascade* method is used:

```
procedure TfrmMDIParent.menuCascadeClick(Sender: TObject);
begin
  Cascade;
end;
```

When MDI child windows are first created (e.g. using **Window | New Window**), they are placed in a cascaded arrangement within the parent (refer to Figures 5 and 6).

## Closing the Current Child Window

Most of the work required to close child windows has been done for you by Delphi. There's only one modification you need to make to have the MDI children close as you would

expect them to when Ctrl F4 is pressed. You need to modify the child form's *OnClose* method so that it responds appropriately to the *Close* event:

```
procedure TMDIChild.FormClose(Sender: TObject;
                              var Action: TCloseAction);
begin
  Action := caFree;
end;
```

As you can see, the *OnClose* procedure takes a parameter named *Action*. The value of *Action* determines how the procedure will respond to the *Close* event. We want the window to close, so we assign the *caFree* constant to *Action*. (For more information, see "caFree" or "TCloseEvent Type" in Delphi on-line help.)

You can also add an explicit menu choice to close child windows (that is, a **File | Close** menu option) using the *Close* method. To do so, you need to add a menu item and then modify its *OnClick* procedure:

```
procedure TfrmMDIParent.menuFileCloseClick(Sender:
TObject);
begin
  if ActiveMDIChild <> nil then
    ActiveMDIChild.Close;
end;
```

This code checks the *ActiveMDIChild* property to make sure that there is a child window to close. If there isn't, issuing the *Close* method will cause a General Protection Fault.

## Next and Previous

The *Next* method makes the next MDI child window in the MDI parent window sequence the active MDI child form. The *Next* method treats the MDI child windows as a circular list. For example, if you send the *Next* method to the MDI parent window and the currently selected MDI child window is the last MDI child window out of at least two, the *Next* method causes the first MDI child window in the list to be made active.

The following illustrates use of the *Next* method:

```
procedure TfrmMDIParent.menuNextClick(Sender: TObject);
begin
  Next;
end;
```

The *Previous* method behaves similarly to the *Next* method. However, *Previous* selects child windows in the opposite direction. Like the *Next* method, the *Previous* method also treats the list of MDI child windows as a circular list. If the *Previous* method is issued when the first MDI child is active, the last MDI child becomes active.

## Tiling MDI Children

The *Tile* method sizes the MDI child windows so they don't overlap each other. The client area of the MDI parent window is divided into different regions, each with an MDI child window contained (see Figure 8). This code example uses the *Tile* method:

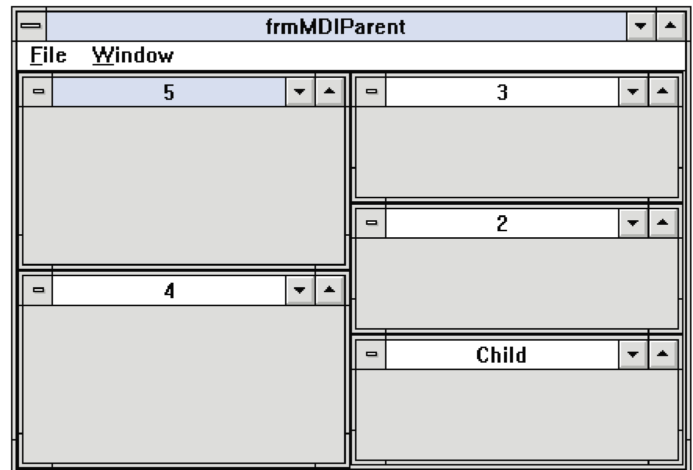



**Figure 8:** The result of the *Tile* method.

```
procedure TfrmMDIParent.TileMenuClick(Sender: TObject);
begin
  Tile;
end;
```

## Conclusion

Of course, now that you have all this information, you have all the license needed to create interesting and unique Delphi applications — right? Not entirely. Keep in mind that your Windows users will expect consistency between applications. And with the impending release of Windows 95, there will be a greater call to provide the user with uniform application interfaces. This does not mean that an MDI-compliant application will always be best. Perhaps an SDI implementation would be better suited to your user's needs.

Fortunately, Delphi makes it simple to create and maintain either interface. The freedom you have to use built-in templates or create custom templates is provided by Delphi. All you have to do is concentrate on the user and the application-specific logic. (In fact, it's important to note that an excellent model for a robust MDI application is available in the form of the MDI Application project template provided with Delphi. You should definitely study it before creating your own MDI applications.)

Next month we'll discuss an interface style that is growing in popularity, the Single Document Interface (SDI). We'll compare SDI and MDI and build a simple SDI application.

This article was adapted from material for *Using Delphi: Special Edition* (Que, 1995) by Jon Matcho, David Faulkner, et al. ∆

*The demonstration project referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\AUG\JM9508.*

Jon Matcho has been building business systems since 1987. Since then, Jon founded Brickhouse Data Systems, Inc., an East Coast consulting firm specializing in software development. In 1993 he joined Professional Computer Solutions, Inc. to assist in the delivery of mission-critical database solutions. You can reach Jon at (908) 704-7300, or on CompuServe at 71760,2720.

By Robert Palomo

# A Topical Search

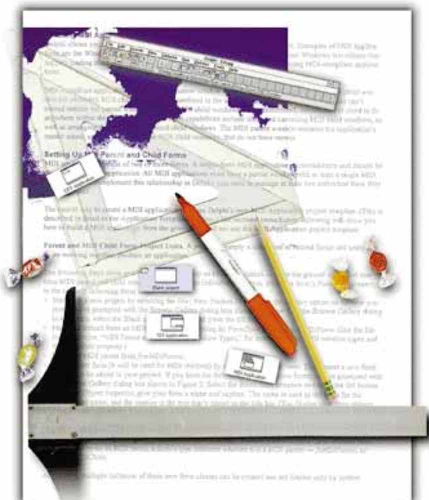
## Integrating On-Line Help in Delphi Applications

**For many people, their first encounter with the term *hypertext* conjures up images of something out of Star Trek. As often happens however, reality is rather mundane as we'll see when we take a peek inside the technology behind hypertext. Ordinary or not, a well-designed hypertext-based help system can be a highly utilitarian, even "sexy" component of an application that adds a great deal of value to your software.**

Software companies are now supplying more of their product's documentation in an electronic format. They do this because the cost of printing and shipping books takes a bite out of their bottom line. On the other hand, software users are often unwilling to relinquish the comforting security of a hefty volume or two (or six). Indeed, sometimes a printed manual is vital. For example, if your system has just crashed, an on-line system administrator's guide would be pretty useless.

The popularity of CD-ROM based multimedia titles — such as Microsoft's Encarta encyclopedia — is doing much to encourage acceptance of the computer screen as a preferred medium for reading and gleaning information. CD-ROM enables software companies to distribute the entire contents of their printed manual sets on disk, as Borland does with Delphi.

On-line documentation solutions range from entire printed manuals "dumped" into a format for "viewer" software (that displays text in a linear fashion similar to a book), to custom-designed hypertext help systems. This article focuses on the latter, since Delphi provides developers with some nicely encapsulated means of integrating this type of help in their applications.



### Authoring vs. Integration

Developing on-line help for Windows applications involves two fundamental processes: creating, or authoring the help file, and integrating help with components of the application. Authoring on-line help files provides the subject matter for entire articles, books, and seminars. We'll look at the process only as it relates to integrating help into a Delphi application.

Authoring on-line help essentially involves writing text, dividing it into discrete topics, formatting it to create hypertext links, creating graphics, and configuring display elements such as secondary windows and menus. A help file is a self-contained Windows application that users can run from Windows independently of any other application. (For example, you can double-click on a .HLP file in the Windows File Manager to run it.) However, that's not always enough to meet the needs of end-users, as we'll see later.

Let's begin by taking a quick look at some of the basic things you, as an application developer, should understand about Windows on-line help. We'll then examine how Delphi makes it easy to integrate help into your applications.

## What's So Hyper about Hypertext?

At the risk of over-simplifying, a *hypertext system* is essentially just a database application. The data are "chunks" of text and/or graphics called *topics*. Each topic is identified by a unique alphanumeric key called a *context string*. The Windows help compiler creates a B-tree type index of these context strings, enabling the WinHelp engine (WINHELP.EXE) to rapidly locate and display any topic in the compiled help file.

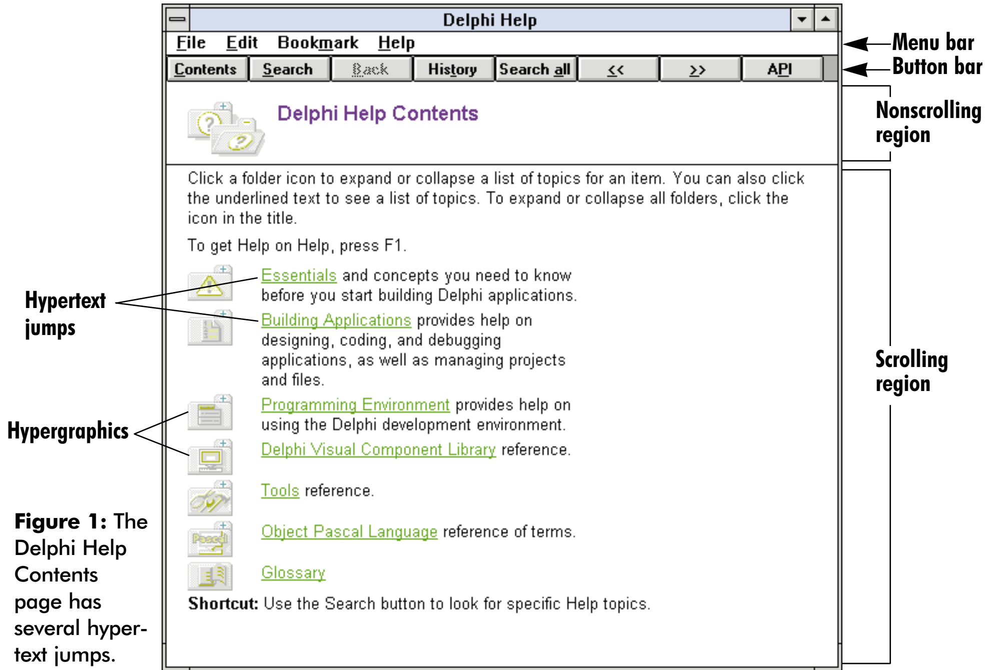



**Figure 1:** The Delphi Help Contents page has several hypertext jumps.

A hypertext system is really no different from a typical customer database where the customer file is indexed on a CustNo field containing a unique value identifying one customer record. For example, searching the index for "ABCCORP" accesses the record for the customer "ABC Corporation". Similarly, you could have a topic titled "Using the Widget Control" in a compiled help file. When the topic was created in the help source file, let's assume it was assigned the context string *UsingWidget*. When the help compiler compiles the source files (text and graphics) into binary format, it creates an index of the context strings defined in the source file(s). Within the help source files, the help author creates and formats text strings that point to a particular context string. When the user clicks on that text in the compiled help file, the help Engine (WINHELP.EXE) looks up the Context ID value in the index and "jumps" to the topic identified by this "key field" value — the context string.

## What Is Context-Sensitivity?

The typical help file has a Contents topic similar to a book's table of contents. This screen summarizes the main subject areas in the help file and provides jumps to the principal topic screen (see Figure 1). These topic screens may in turn contain jumps to other related topics. (Note that topics can be formatted to display in the main help window as a secondary help or a popup window.)

If your users will be content to begin with the Contents screen and muddle around in the help file from there, your life as a developer is easy. You simply develop your application, author a help file, and provide an icon to run the help file from the Windows Program Manager.

Chances are that will not be the case. At some point a user won't understand what is happening, and your job will become more interesting. Experienced Windows users tend to press F1 expecting to see help for the active user interface (UI) element or process that has proved baffling. If nothing happens, or if the user is simply booted into the help file's Contents topic, your client will definitely be disappointed. Even though your on-line help system is fully hypertext-based, it's not context-sensitive.

Briefly, *context-sensitivity* is an application's ability to display a specific topic from a Windows help file at a given point during run-time. We'll focus on the mechanisms that Delphi provides to enable you to "hook" the various components in a Delphi application to specific topics in a Windows help file.

Before discussing the specific techniques, however, there's one technical pitfall inherent to the current state of Windows help development that you must understand so you can avoid it.

## Understanding the Data Type Trap

The Windows application programming interface (API) provides developers with the means to integrate context-sensitive help in Windows applications. To accomplish this, the programmer provides a "hook" in the code that equates to the context string of a specific topic in a help file when that code executes.

It's a simple concept, but not so simple in practice. You go through a lot of gyrations to get values into the right variables of the correct data type: issue a call to the Windows API, start up WinHelp, pass it the ID for the topic, and finally access and display the topic specified by the hook in the source code.

That's the bad news. The good news is that Delphi makes this process easy by providing components, properties, and methods that encapsulate this otherwise aggravating bit of coding.

If you have experience with WinHelp and the Windows API, you're probably familiar with an interesting little glitch in the

communication between these entities. Delphi requires you to specify the help "hook" as an integer value in the *HelpContext* property of visual components. That's because the Windows API requires numeric values for the help Context IDs in a Windows application. Delphi dutifully delivers these values as required.

Now here's a good one: the WinHelp engine requires alphanumeric context strings to locate the topics in a help file's Topic index. In other words, you can't just pass the Context ID from your application through the Windows API to WinHelp. (I know what you're thinking, and yes, the same company did develop both systems. Go figure.)

This means if you have a context string *UsingWidget* in the help file, there's no direct way to call it from a Windows application. For example, if you specify 110 as the help Context ID in the *HelpContext* property for your *Form1.Widget1* component, that value is delivered to the API as a numeric data type.

However, if you specify "110" as the context string for the "Using the Widget Control" topic in the help source file, it will be delivered to WinHelp as a character value. Delphi provides a handy way around this little problem with the *HelpJump* and *HelpContext* methods (which we'll look at a bit later). These methods are fine for use in event handlers, but to provide context-sensitive access to help topics using F1 you need to coordinate your Context IDs in the help project file.

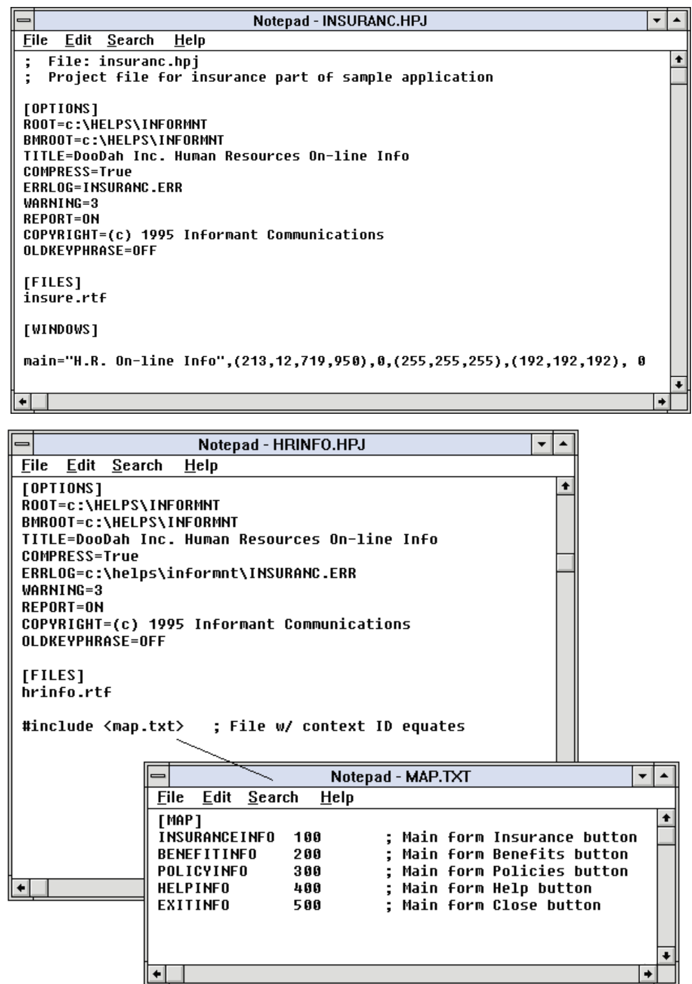
## The Help Project File

The WinHelp compiler begins its work with the help project (.HPJ) file. This is basically just an .INI file. It has different sections, each identified by bracketed keywords (e.g. `[CONFIG]`). Figure 2 shows an example of a simple .HPJ file. The `[MAP]` section of the .HPJ file is where the numeric help Context IDs from your Delphi application are equated to their alphanumeric context strings in the help file.

If your only connection with the help authoring process is to supply the numeric Context ID values from your application components to a help author, you have a cushy job and should try to hang onto it. You only need to coordinate with the help author (who handles the production of the help file) to get the two sets of IDs in sync in the `[MAP]` section of the help project (.HPJ) file. You don't even need to touch the .HPJ file. (In fact, some help authors would do you grievous bodily harm for even thinking about it.) Instead, you can create a separate text file containing the `[MAP]` section that the help author can reference with an **include** statement in the .HPJ file (see Figure 3).

## Hooking Delphi Components to Help Topics

Let's turn our attention to integrating context-sensitivity in a Delphi application. Delphi's architects have thoughtfully provided the *HelpContext* property for every visual component that can receive focus, specifically for integrating context-sensitive help. The default value for this property is zero. At run-time, if a component has a zero value in the *HelpContext* property, has focus,



Notepad - INSURANC.HPJ

```
; File: insuranc.hpj
; Project file for insurance part of sample application

[OPTIONS]
ROOT=c:\HELPS\INFORMNT
BMROOT=c:\HELPS\INFORMNT
TITLE=DooDah Inc. Human Resources On-line Info
COMPRESS=True
ERRLOG=INSURANC.ERR
WARNING=3
REPORT=ON
COPYRIGHT=(c) 1995 Informant Communications
OLDKEYPHRASE=OFF

[FILES]
insure.rtf

[WINDOWS]

main="H.R. On-line Info",(213,12,719,950),0,(255,255,255),(192,192,192), 0
```

Notepad - HRINFO.HPJ

```
[OPTIONS]
ROOT=c:\HELPS\INFORMNT
BMROOT=c:\HELPS\INFORMNT
TITLE=DooDah Inc. Human Resources On-line Info
COMPRESS=True
ERRLOG=c:\helps\informnt\INSURANC.ERR
WARNING=3
REPORT=ON
COPYRIGHT=(c) 1995 Informant Communications
OLDKEYPHRASE=OFF

[FILES]
hrinfo.rtf

#include <map.txt>   ; File w/ context ID equates
```

Notepad - MAP.TXT

```
[MAP]
INSURANCEINFO   100     ; Main form Insurance button
BENEFITINFO     200     ; Main form Benefits button
POLICYINFO      300     ; Main form Policies button
HELPINFO        400     ; Main form Help button
EXITINFO        500     ; Main form Close button
```

**Figure 2 (Top):** A sample .HPJ file. **Figure 3 (Bottom):** Using an included .TXT file to integrate Context IDs with the .HPJ file.

and the user presses F1, nothing happens. This is great, because if there are components that you don't want or need to provide context-sensitive help, you don't have to do anything. Just leave the default zero value in the *HelpContext* property.

By entering a positive, non-zero value in the *HelpContext* property, you set up the means for accessing a topic in the help file (which for now, we'll assume already exists). The value need not be unique within your application. For example, the **OK** buttons in five different dialog boxes might all have the same value in the *HelpContext* property because you want the same help topic to display for all the buttons.

It's important to understand that specifying a Context ID in the *HelpContext* property is only one step in the process of implementing context-sensitivity. You must still coordinate these IDs with the help authoring process as discussed earlier.

## Planning a *HelpContext* ID Scheme

You will probably want to devise some logical scheme for the values of the *HelpContext* property for your application's components. The organization is up to you, but it should be part of your application design.

Depending on the scope of your application, you might define different ranges of values and the parts of the application. For example, one range might be reserved for an entire module, and sub-ranges defined for component windows of those modules. Or, in a small application, you might have one range for each form or dialog box, or one reserved only for menus, and so forth.

As you plan the ranges, make them large enough so that during initial development each component's *HelpContext* ID increments by five. For example, for three button components on a form, you would set the *HelpContext* properties as 100, 105, and 110 respectively (instead of 100, 101, and 102). This leaves room in the range for additional components without having to make major modifications to the `[MAP]` section of the .HPJ file.

## Properties and Methods for Accessing Windows Help

Delphi provides properties and methods that enable you to access specific topics in a help file, or in different help files. In this section, we'll summarize these and take a look at some creative ways you might use them.

Delphi's *TApplication* component encapsulates the functionality needed to access WinHelp via the Windows API. The WinProcs unit contains methods for accessing specific help topics in a given help file.

Before your application can display help, you must specify a help file for it. The *HelpFile* property of *TApplication* specifies the name of the help file that the various help access methods will look at. You normally specify the name of this file by selecting **Options | Project** and choosing the Application page of the Project Options dialog box (see Figure 4).

You can also set or change the designated help file at run-time with an assignment:

```
Application.HelpFile := 'helpfile.hlp';
```

If you don't specify a help file either at design time in Project Options or at run-time in code, the application will not display help at run-time. Without a help file specified, it doesn't matter that you may have assigned Context IDs in the *HelpContext* property of components, or that a .HLP file exists.

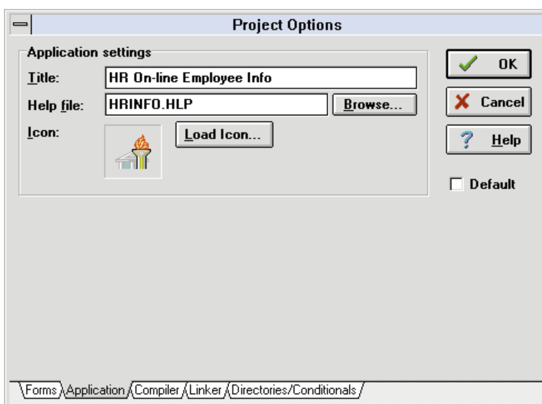



**Figure 4:** The Application page of the Project Options dialog box.

*TApplication* has three methods that pertain to the display of on-line help: *HelpCommand*, *HelpContext*, and *HelpJump*. If you have experience in other environments working with WinHelp, then you may prefer the *HelpCommand* method for invoking and controlling help from a Delphi application. This method provides a quick hook-up from the Delphi environment to any of the WinHelp command macros.

Besides accessing topics in context, you can control properties of the help window — such as size, color, or position — and display topics in secondary windows. *HelpCommand* is well documented in the Delphi on-line help system, and you should be able to learn and use it quickly.

If you want to access a specific help topic with code, then you'll appreciate the simplicity of the *HelpContext* and *HelpJump* methods. These provide the same result: the display of a specific topic in the help file. The difference is in how you specify the Context ID for the help topic you want to access.

An application developer will tend to approach context-sensitivity from the standpoint of the application user-interface. You know what components are in a specific dialog box or window, and you know (or can easily look up) the numeric Context ID in the Object Inspector. Therefore, you would know the Widget component in the Foo dialog box has a Context ID of 110 in its *HelpContext* property. You may not know (nor care) that the help file topic associated with it is "Using the Widget Component" with a context string of *UsingWidget*. You simply want to display whatever topic is in the help file for the Widget component when it has the focus and the user presses F1.

If you want to display this topic in response to another event, simply call the *HelpContext* method in your event handler:

```
Application.HelpContext(110);
```

Help authors tend to approach context-sensitivity from the standpoint of the help source file. They may not even know how to run Delphi, much less deal with the values of the *HelpContext* property of components. They may know only that the topic "Using the Widget Control" has a context string *UsingWidget* in the help source document, and this is the value they can supply to you, the application developer. Not a problem — you can display the same help topic with a call to *HelpJump*:

```
Application.HelpJump('UsingWidget');
```

Remember, you need to use these methods only if you want to invoke a specific help topic in response to some event other than the user pressing F1. F1 context-sensitivity is already built into all components having a *HelpContext* property. You need only specify a non-zero Context ID value in that property, and map the value appropriately in the help project file as we discussed earlier.

## Some Practical Techniques

Let's look at a practical application you might develop using these methods. This is a module for a Human Resources system that

## More Information about Help Authoring

To create a Windows help file, you need: a word processor capable of saving files as Rich Text Format (.RTF) files; a Windows help compiler (HCP.EXE); a hotspot editor for creating hypergraphics (SHED.EXE); and a paint program that can create .BMP files. The help compiler and Hotspot Editor both ship with both versions of Delphi. Also available with Delphi is CWH.HLP, an on-line help-authoring guide. This file resides in the \DELPHI\BIN directory in a default Delphi installation. Another good resource is the Microsoft Help Authors Forum on CompuServe.

I can also recommend *Developing On-Line Help for Windows* by Scott Boggan, David Farkas, and Joe Welinske (SAMS). This book will take you from neophyte to consultant-level expertise in Windows help development. It's well-organized and indexed, very readable, and covers all the issues. It includes a disk containing Word templates and macros optimized for creating Windows help source files, example help projects, help project file templates, and bitmap graphics you can use in your own projects. It also presents a comprehensive review and comparison of several third-party help authoring tools.

### Third-Party Tools

**Windows Help Authoring Utility**
(on the Microsoft Developer Network CD)
Microsoft Corp.
1 Microsoft Way
Redmond, WA 98052
(206) 882-8080

**RoboHelp**
Blue Sky Software Corp.
7486 La Jolla Boulevard, Suite 3
La Jolla, CA 92037
(619) 459-6365

**Doc-To-Help**
WexTech Systems, Inc.
310 Madison Avenue, Suite 905
New York, NY 10017
(212) 949-9595

**Help Magician**
Software Interphase, Inc.
82 Cucumber Hill Road
Foster, RI 02825-1212
(800) 542-2742
(401) 397-2340

provides information and answers to employee questions on-line. The information is contained in several Windows help files. The text of these files are maintained by different people in the HR department. There is also a help file for the module itself.

The help files are as follows:

- INSURE.HLP — Information on medical, dental, and life insurance plans.
- BENEFITS.HLP — Information on all other employee benefits.
- POLICIES.HLP — Information on general employment policies and regulations.
- HRINFO.HLP — Help file for the information application itself.

You could certainly combine the source documents for these files and compile a single Windows help file for the module. However, let's assume there's a reason to keep them separate to illustrate some techniques you can use in Delphi to change the *TApplication* component's *HelpFile* property. You also want to keep them separate to address specific topics in the various .HLP files at different times, depending upon what's happening in the application UI.

At design time, we'll specify HRINFO.HLP as the application help file. We'll create a UI that will switch the help file as needed. Then, we'll create a main form for the application called MAINFRM.PAS, as shown in Figure 5.

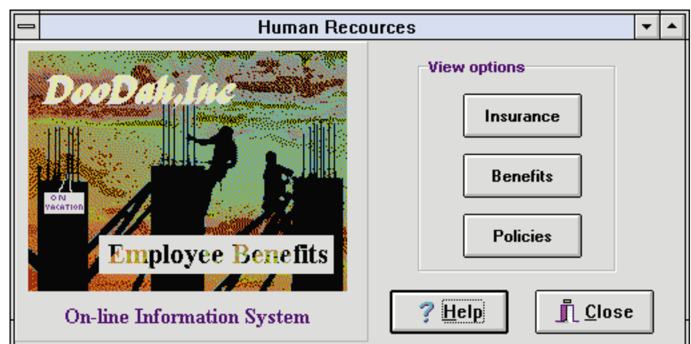



**Figure 5:** The Human Resources sample application.

This form has five BitBtn components labeled **Insurance**, **Benefits**, **Policies**, **Help**, and **Close**. The event code behind the first four buttons hides the main window, displays a button bar, and executes the *OnClick* event of the appropriate Speedbutton component on the button bar:

```
procedure TMainWin.BBtnInsuranceClick(Sender: TObject);
begin
  MainWin.Hide;
  BtnBarWin.Show;
  BtnBarWin.SBInsuranceClick(Sender);
end;
```

Each of the BitBtn components has a similar *OnClick* event handler.

On the button bar, each Speedbutton sets the current help file and displays its Contents screen:

```
procedure TBtnBarWin.SBInsuranceClick(Sender: TObject);
{ User has selected Insurance button in main screen }
begin
  { Display Speedbutton in down state }
  SBInsurance.Down := True;
  { Set current Help file }
  Application.HelpFile := 'INSURE.HLP';
  { Access Help file Contents }
  Application.HelpCommand(HELP_CONTENTS, 0);
  PnlInsurance.visible := True;{ Display button panel }
end;
```

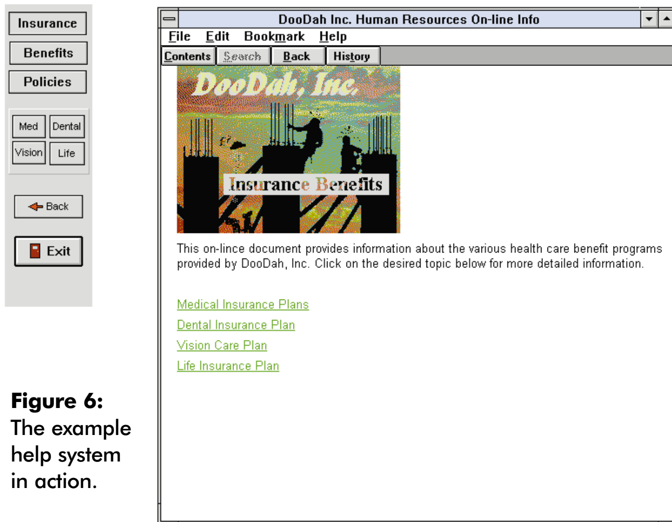



**Figure 6:** The example help system in action.

Each Speedbutton event handler specifies a different help file. The help window is sized in the .HPJ file's `[WINDOWS]` section to allow the button bar to display beside it (see Figure 6).

The **Insurance** button displays a panel on the button bar with several speedbuttons that access a specific topic in the help file using the *HelpJump* method. For example, the **Med** button code looks like this:

```
procedure TBtnBarWin.SBMedInsClick(Sender: TObject);
begin
  Application.HelpJump('MEDICALPLAN');
end;
```

The **Insurance**, **Benefits**, and **Policy** buttons on the button bar have their *GroupIndex* property set to a value of *1*, and the *AllowAllUp* property set to *False*. This eliminates the need to test for the value of the button's *Down* property. You simply set the *Down* property of the selected button to *True*, and the others with the same *GroupIndex* property display in the "up" state. (You could achieve the same effect with a RadioButton group, but this technique enables you to use glyphs on the buttons for a more elegant UI.)

The Windows **Help** button on the main form switches the active help file back to the design time default, HRINFO.HLP:

```
procedure TMainWin.BBtnMainHelpClick(Sender: TObject);
begin
  Application.HelpFile := 'HRINFO.HLP';
  Application.HelpCommand(HELP_CONTENTS,0);
end;
```

The main three BitBtn components are context-sensitive. Pressing F1 while any of them have focus invokes a specific topic in the application's help file. This file (HRINFO.HLP) is set as the active help file whenever the main form's *OnActivate* event occurs.

## Conclusion

Developing a well-planned on-line help system with context-sensitivity can provide an extra measure of value to the software you develop with Delphi. Delphi provides you with some handy short-cuts that reduce the somewhat convoluted integration process for context-sensitivity to a simple property setting. And as we saw in the example application, Delphi enables developers to experiment with Windows help in some interesting and out-of-the-ordinary ways. ∆

*The demonstration application referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\AUG\RP9508.*

Robert Palomo has been a technical writer in the software industry in Seattle, WA and Silicon Valley, CA for the past five years. His most recent job was as a member of the Delphi documentation group at Borland International. The extensive staffing cutbacks at Borland gave him enough free time after Delphi shipped to set up shop part-time as a consultant specializing in integrating context-sensitive help in Delphi and other Windows applications. Robert also does Delphi application development work. You can contact him on CompuServe at 76201,3177 or on the Internet at 76201.3177@compuserve.com.

ON THE COVER

DELPHI / OBJECT PASCAL

By Douglas Horn

# Initialization Rites

## Using Windows .INI Files in Delphi

Nearly every commercial Windows application employs an .INI file. Even a glance at the \Windows directory of most PC's will show how widely they're used — and some programs use several.

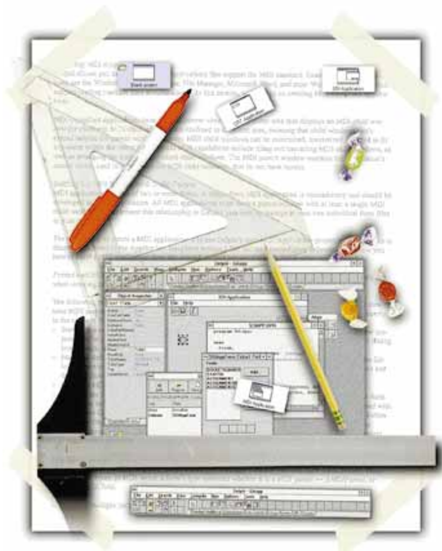
Unfortunately, .INI files are underutilized by most application developers. This is despite the fact that they're extremely useful for storing information that must be saved between program sessions.

Delphi makes it particularly easy to implement an .INI file with an application. This article will introduce .INI files and explain their use with Delphi.

### What Are .INI Files?

INI files are named for their common three-letter file extension, an abbreviation for *initialization*. They're used to store configuration parameters, user preferences, program status, and other information that should persist from one execution of an application to the next.

Two of the best-known .INI files are WIN.INI and SYSTEM.INI. Both of these files are used to store important parameters for the Windows operating system. While most Windows applications use an .INI file named for the executable (e.g. DELPHI.INI), many use the WIN.INI file instead of, or in addition to, their own file.

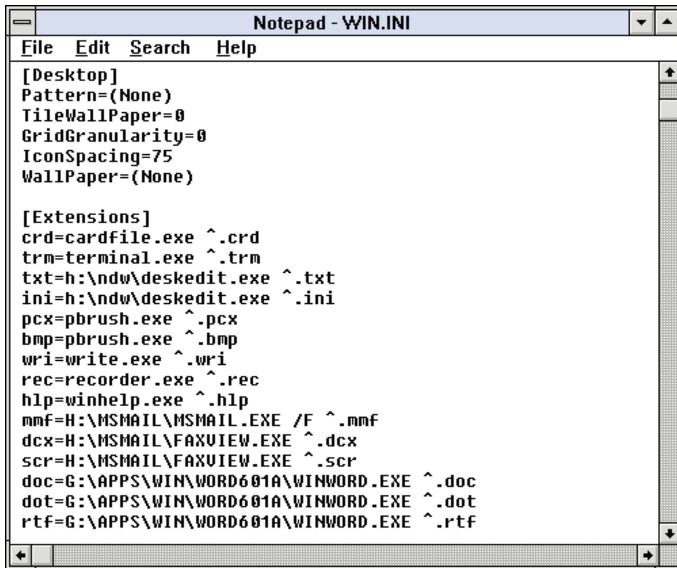


.INI files are simple text files (see Figure 1). Each .INI file is divided into sections, easily recognizable by a section title enclosed in brackets (e.g. `[Boot]`). Within each section are a number of statements. Each statement begins with a variable or *identifier*, followed by its value (e.g. `spooler=True`). .INI files are not case-sensitive.

### How Delphi Handles .INI Files

Delphi interacts with .INI files using its *TIniFile* object. The *TIniFile* object has 16 methods, about half of which programmers will use regularly in implementing .INI files (see Figure 2). The first task when using .INI files in Delphi applications is to add the *IniFiles* statement to the **uses** clause of the program unit that will call the .INI file:

```
uses
  IniFiles, SysUtils, WinTypes, WinProcs, Messages,
  Classes, Graphics, Controls, Forms, Dialogs, ExtCtrls,
  StdCtrls, Menus, Buttons;
```



Notepad - WIN.INI

File Edit Search Help

```
[Desktop]
Pattern=(None)
TileWallPaper=0
GridGranularity=0
IconSpacing=75
WallPaper=(None)

[Extensions]
crd=cardfile.exe ^.crd
trm=terminal.exe ^.trm
txt=h:\ndw\deskedit.exe ^.txt
ini=h:\ndw\deskedit.exe ^.ini
pcx=pbrush.exe ^.pcx
bmp=pbrush.exe ^.bmp
wri=write.exe ^.wri
rec=recorder.exe ^.rec
hlp=winhelp.exe ^.hlp
mmf=H:\MSMAIL\MSMAIL.EXE /F ^.mmf
dcx=H:\MSMAIL\FAXVIEW.EXE ^.dcx
scr=H:\MSMAIL\FAXVIEW.EXE ^.scr
doc=G:\APPS\WIN\WORD601A\WINWORD.EXE ^.doc
dot=G:\APPS\WIN\WORD601A\WINWORD.EXE ^.dot
rtf=G:\APPS\WIN\WORD601A\WINWORD.EXE ^.rtf
```

| Method | Description |
|---|---|
| **EraseSection** | Erases a section of an .INI file. |
| **FileName** | The name of the .INI file the *TIniFile* object encapsulates. |
| **ReadBool** | Retrieves a Boolean value from an .INI file. |
| **ReadInteger** | Retrieves an Integer value from an .INI file. |
| **ReadSection** | Reads all variables of a section into a string. |
| **ReadSectionValues** | Reads all variables and their values of a section into a string. |
| **ReadString** | Retrieves a string value from an .INI file. |
| **WriteBool** | Writes a Boolean value to an .INI file. |
| **WriteInteger** | Writes an integer value to an .INI file. |
| **WriteString** | Writes a string to an .INI file. |

**Figure 1 (Top):** A section of a typical Windows .INI file.
**Figure 2 (Bottom):** Key methods of the *TIniFile* object.

Once *IniFiles* is added to the **uses** clause, the .INI file need only be declared as a variable and then created.

The following illustrates the beginning of a typical procedure that accesses an .INI file:

```
procedure TForm1...
var
  AppIni: TIniFile;
begin
  AppIni := TIniFile.Create('APP.INI');
...
```

After the *TIniFile* handle (`AppIni`) is declared, the .INI file is opened using the *TIniFile.Create* method, where `APP.INI` is the name of the file to be read from or written to. The *Create* method opens an existing .INI file if the one specified exists, or creates a new one if it doesn't. (Unless a specific path is supplied, Delphi will search for `APP.INI` in the \Windows directory.)

When it's no longer necessary to access the .INI file, the *Free* method is used to release the .INI file, destroy the *TIniFiles* object, and free the resources reserved for the object:

```
...
  AppIni.Free;
end;
```

These minor preliminaries are necessary whenever *TIniFiles* are used. Between the *Create* and *Free* methods, however, lie a large number of possibilities.

## Reading, Writing, and 'Rithmetic

The real workhorses of the *TIniFile* methods are the "reads" and the "writes". The three write methods are *WriteBool*, *WriteInteger*, and *WriteString*. These methods are similar; the only factor that differentiates them is the type of value each handles.

Each write method requires three parameters: the section, identifier (or variable), and value. For example, the following statement would write the string value `RIVETS.BMP` to the `WallPaper` identifier in WIN.INI's `[Desktop]` section (assuming a link to WIN.INI was already created). Note that the brackets are not used when specifying the section:

```
WinIni.WriteString('Desktop','WallPaper','RIVETS.BMP');
```

Of course the value may be literal, as above, or can be derived from a property just as any other value in Delphi. The following line would then set an .INI file property to the corresponding program property's current state:

```
AppIni.WriteBool('Settings','ShowButton',Button1.Visible);
```

The read methods are *ReadBool*, *ReadInteger*, and *ReadString* — each corresponding to a write method shown above. The syntax is also quite similar between corresponding read and write methods.

Conversely, while a write method specifies a value, a read method uses a default value that is used in case the target .INI file statement is blank (or non-existent). The following two read methods correspond directly to the write methods previously shown:

```
Canvas.TextOut(10,10,'Wallpaper Image = ' +
  WinIni.ReadString('Desktop','WallPaper','none'));

Button1.Visible :=
  AppIni.ReadBool('Settings','ShowButton',True);
```

There are two more read methods: *ReadSection* and *ReadSectionValues*. These methods function somewhat differently than those already described. Instead of reading a single specified value, *ReadSection* and *ReadSectionValues* read an entire .INI file section into a *TStrings* object.

The difference between the two methods is demonstrated in Figure 3, where the ListBox component on the left uses the *ReadSection* method, while the one on the right uses

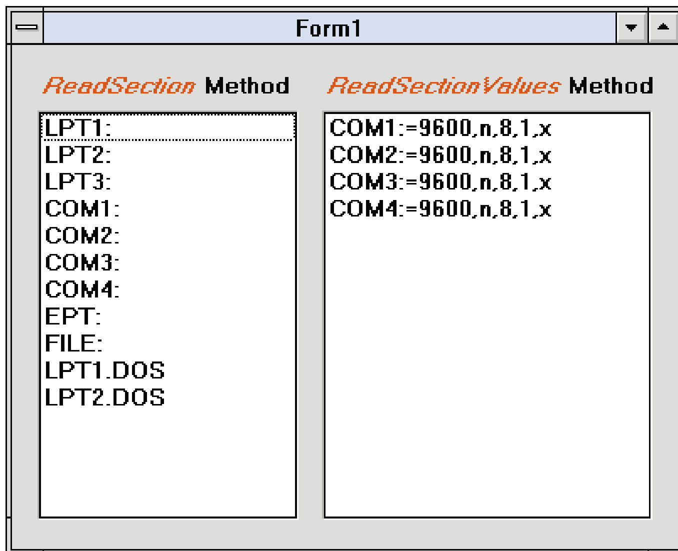



**Figure 3:** A simple pair of ListBox components illustrates the differences between the *ReadSection* and *ReadSectionValues* methods.

*ReadSectionValues*. *ReadSection* displays only the variables in the section, and displays them regardless of whether they have values, or are preceded by the REM keyword. On the other hand, the *ReadSectionValues* method displays the variable and its value (the complete statement), and ignores blank variables or "REM-ed out" statements.

This Object Pascal code produces the list box contents shown in Figure 3:

```
procedure TForm1.FormActivate(Sender: TObject);
var
  AppIni: TIniFile;
begin
  AppIni:= TIniFile.Create('WIN.INI');
  AppIni.ReadSection('Ports',Listbox1.Items);
  AppIni.ReadSectionValues('Ports',Listbox2.Items);
  AppIni.Free;
end;
```

The final *TIniFile* method of note is *EraseSection*. It's not useful under most circumstances, and in fact, could do a lot of damage to a system if used improperly.

## Sample Application

The sample application that accompanies this article (see Figure 4) shows how to implement an .INI file that stores a number of program parameters when it closes. The complete unit file (INI_APP.PAS) is shown in Listing One beginning on page 24. Let's take a look at the code.

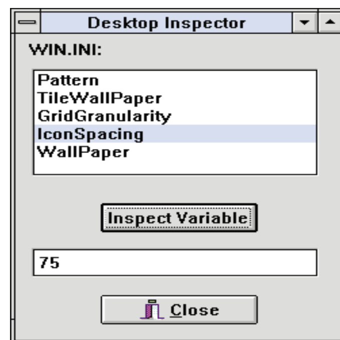



**Figure 4:** The sample application illustrates a variety of practical ways to use .INI files.

When the application opens, the *Create* method (`TFormSA.FormCreate`) allocates memory to create a *TIniFile* object, and passes it the file name of an .INI file. Also upon opening, the application accesses the SAMPLEAP.INI file and uses parameters that .INI file contains to reset itself.

The application's main function is to read all the entries of the `[Desktop]` section of the WIN.INI file into a ListBox component. Then, the user can select a variable to inspect. If the user resizes the window at any time, the user-interface objects reposition themselves relative to the application's new dimensions.

This sample application relies heavily on the most useful form of .INI file manipulation — simple, single-statement read and write method calls. While the *ReadSection* and *ReadSectionValues* methods have their merits, they are less useful than the more straightforward methods. This application uses *ReadSection* to fill the ListBox component with the variables in WIN.INI's `[Desktop]` section. However, these methods are rarely useful for manipulating .INI files. (Again, the *EraseSection* method is used even less and should be implemented with caution.)

To view the code section by section, the initial lines of code consist of the necessary preliminaries. Note the *IniFiles* reference in the **uses** clause. This statement must be added whenever *TIniFile* methods will be used.

The first procedure, *FormClose*, records all the desired information to the SAMPLEAP.INI file. First it declares *TheIni* as a *TIniFile* object, and then creates the link to the application's own .INI file, SAMPLEAP.INI. Next, the procedure writes information consisting of the form's size and coordinates, stored as integers, and the value of the current variables list (*ListBox1*) setting, stored as a string.

Note that the information could more easily be stored as an integer, using the *ItemIndex* property. However, using this property, this would only store a position on the list, not an actual variable. In case WIN.INI is changed, the variable setting might not remain the same.

The next procedure, *FormCreate*, not only accesses SAMPLEAP.INI but WIN.INI as well, using the same *TheIni* object. The first operation is to create a link to WIN.INI, and then read the contents of the `[Desktop]` section into the *TStrings* object, *ListBox1.Items*. After this is accomplished, the link to WIN.INI is destroyed and *TheIni* object is linked to the sample application's own .INI file.

Next, the sample application reads the values stored during *FormClose* (using the *ReadInteger* method), and uses them to set the form to the size and position it held when it was last closed. Using the similar *ReadString* method, the procedure resets the *ListBox* selection to that recorded in the .INI file.

The *FormResize* procedure is then called to establish the proper size and position of the user-interface objects, based on the form dimensions stored in the .INI file. Because the Object Pascal code exists in

the *FormCreate* procedure (rather than *FormActivate*), these changes occur before the form is drawn, and remain invisible to the user.

*Button1Click* is the last procedure to employ .INI file operations. When called (when the user clicks the **Inspect** button) the procedure creates a link to WIN.INI as above, finds the value of the entry selected in the list of variables (`ListBox1`), and displays the value in the edit box (`Edit1`).

The two final procedures provide simple housekeeping. *FormResize* is called by *FormCreate*. It's also called whenever the user changes the form's dimensions. This procedure simply resizes and repositions user-interface objects relative to the form's size. This is done to give users a logical reason to resize the form (i.e. to resize the ListBox depending on the length and number of the `[Desktop]` variables) so that new coordinates and dimensions can be stored in the .INI file.

The last procedure, *BitBtn1Click*, simply closes the sample application.

## Conclusion

I hope this article has helped to demystify Windows .INI files. These files can be easily implemented, and add an extra level of user-friendliness and sophistication to any Delphi application. Although for clarity's sake, the sample application stores only simple coordinates and settings, .INI files can be used to store almost any type of program information.

Next month, we'll extend the .INI file concept further by combining it with MDI (multiple-document interface) applications and menus to create a list of most-recently used files (or MRU). This is an advanced feature that enables users to quickly and easily pick up where they left off. ∆

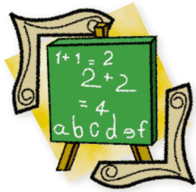
*The demonstration project referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\AUG\DH9508.*

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

**Begin Listing One — INI_APP.PAS**

```
unit Ini_app;

interface

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

type
  TFormSA = class(TForm)
    ListBox1: TListBox;
    Label1: TLabel;
    Button1: TButton;
    Edit1: TEdit;
    BitBtn1: TBitBtn;
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
    procedure FormResize(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  FormSA: TFormSA;

implementation

{ $R *.DFM }

procedure TFormSA.FormClose(Sender: TObject;
                            var Action: TCloseAction);
var
  TheIni: TIniFile;
begin
  TheIni := TIniFile.Create('SAMPLEAP.INI');
  TheIni.WriteInteger('Settings','Top',FormSA.Top);
  TheIni.WriteInteger('Settings','Left',FormSA.Left);
  TheIni.WriteInteger('Settings','Height',FormSA.Height);
  TheIni.WriteInteger('Settings','Width',FormSA.Width);
  WriteString('Settings','Variable',
              ListBox1.Items[ListBox1.ItemIndex]);
  TheIni.Free;
end;

procedure TFormSA.FormCreate(Sender: TObject);
var
  TheIni: TIniFile;
begin
  TheIni := TIniFile.Create('WIN.INI');
  TheIni.ReadSection('Desktop',ListBox1.Items);
  TheIni.Free;
  TheIni        := TIniFile.Create('SAMPLEAP.INI');
  FormSA.Top    := TheIni.ReadInteger('Settings',
                                      'Top',0);
  FormSA.Left   := TheIni.ReadInteger('Settings',
                                      'Left',0);
  FormSA.Height := TheIni.ReadInteger('Settings',
                                      'Height',100);
  FormSA.Width  := TheIni.ReadInteger('Settings',
                                      'Width',100);
  ListBox1.ItemIndex := ListBox1.Items.IndexOf(
    TheIni.ReadString('Settings','Variable',''));
  TheIni.Free;

  FormSA.FormResize(FormSA);
end;

procedure TFormSA.Button1Click(Sender: TObject);
var
  TheIni: TIniFile;
begin
  if ListBox1.ItemIndex > -1 then
    begin
      TheIni     := TIniFile.Create('WIN.INI');
      Edit1.Text := TheIni.ReadString('Desktop',
      ListBox1.Items[ListBox1.ItemIndex],'');
      TheIni.Free;
    end;
end;

procedure TFormSA.FormResize(Sender: TObject);
begin
  if FormSA.Width  < 250 then
    FormSA.Width  := 250;

  if FormSA.Height < 300 then
    FormSA.Height := 300;

  ListBox1.Height  := FormSA.ClientHeight - 179;
  ListBox1.Width   := FormSA.ClientWidth - 32;
  Button1.Top      := FormSA.ClientHeight - 122;
  Button1.Left     := FormSA.ClientWidth div 2 –
                        Button1.Width div 2;
  Edit1.Top        := FormSA.ClientHeight - 81;
  Edit1.Width      := FormSA.ClientWidth - 32;
  BitBtn1.Top      := FormSA.ClientHeight - 40;
  BitBtn1.Left     := (FormSA.ClientWidth div 2) –
                       (BitBtn1.Width div 2);
end;

procedure TFormSA.BitBtn1Click(Sender: TObject);
begin
  Close;
end;

end.
```

**End Listing One**




OP BASICS

DELPHI / OBJECT PASCAL

By Charles Calvert

# Strings: Part I

## An Introduction to Object Pascal Strings

Strings. You work with them in virtually every application you develop, yet there are probably a number of features associated with strings that you haven't had a chance to explore.

This first installment of a three-part series on strings will demonstrate that strings are really a form of array. However, a number of interesting rules specific to strings do not apply to arrays, and we'll investigate most of these in detail. In particular, you'll learn how to search for a sub-string in a string, and how to parse a lengthy string.

### A String Is Just a Form of Array

A string is very similar to, but not identical to, an array of characters. Consider the following Object Pascal declarations:

```
type
  TNearString = array[0..255] of Char;

var
  NearString: TNearString;
  MyString: string;
```

Based on this code fragment, *MyString* has all the traits of *TNearString*, plus a few special qualities. In other words, a Delphi string is a superset of an array of Char that is 256 characters long. Let's examine the differences.

All characters in an array of Char are equal; no one character has any special properties. In a string however, the first character is called a *length byte*. It designates how many characters exist in the string. Because the first character has this special task, the first letter in the string is always at offset one.

It's important to remember that most Chars can be represented in two ways. For instance, the letter "A" can be printed verbatim, or it can be represented by the 65th member in certain character sets. If you want to refer to the letter "A" by its place in a character set, you can enter `#65`. The `#` in this example designates the item in question is a Char and not a simple numerical value. Therefore, the number five is represented as `5`, but the fifth character in a character set is represented by `#5`.

For example, consider a string containing the word "Hello". This string is five characters long, so the first byte in a string containing this word would be set to the fifth character in a table of characters:

```
NearString[0] := #5;
```

The next character in the array would be an “H”:

```
NearString[1] := 'H';   { H = #72 }
```

The rest of the letters would immediately follow:

```
NearString[2] := 'e';
NearString[3] := 'l';
NearString[4] := 'l';
NearString[5] := 'o';
```

The result is an array of six characters that looks like:

```
#5,'H','e','l','l','o'
```

If you took the whole process to an imaginary “memory theater”, you would see six seats aligned one behind the other. The person in the first seat would be told to remember the total number of letters in the word(s) that are part of the string. The person in the second seat would remember the first letter of the string (in this case it’s an “H”). The person in the next seat would remember the letter “E”, and so on.

So far so good. But what about the remaining 250 characters in the string? It doesn’t matter what information is stored in those bytes. They can be zeroed out, or hold nothing but garbage. It doesn’t matter what those bytes hold provided the first byte is correctly set to the total number of valid characters in the string.

Let’s say the length byte in the above example was accidentally set to `#6` instead of `#5`.

Then, the letter the person in seat seven is supposed to remember would become part of the string — usually with disastrous results. For instance, the string Hello may suddenly become any of the following:

- `Hello1`
- `Hellob`
- `Hello#`
- `Hello+`

In short, the behavior is unpredictable in such situations.

The code in Figure 1 will compile and run without error. It prints the word `Hello` inside an Edit component. You can view this code fragment as the anatomy of a string. It explicitly shows the elements that comprise a string. The following code behaves identically to the code in Figure 1:

```
var
  S: string;
begin
  S := ‘Hello’;
  Edit1.Text := S;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S: string;
begin
  S[0] := #5;
  S[1] := 'H';
  S[2] := 'e';
  S[3] := 'l';
  S[4] := 'l';
  S[5] := 'o';
  Edit1.Text := S;
end;
```

**Figure 1:** Assuming an Edit component (named *Edit1*) and Button component (named *Button1*) exist on a form, this Object Pascal code compiles and displays `Hello` in the Edit box.

Clearly, this fragment is easier to write. However, the fact that you can write code like this is a special feature of the compiler. What the compiler actually does is shown in Figure 1. However, it’s laborious to write all that code each time you want to assign a value to a string. Therefore, the compiler allows you to write code like the above example.

A brief sample program, named EASYSTR.DPR, demonstrates these ideas. Specifically, EASYSTR shows what happens if you don’t treat a string’s length byte carefully. The form for the EASYSTR program includes two buttons and an Edit component (see Figure 2). The code for the EASYSTR program is shown in Listing Two on page 28.

The EASYSTR program enables you to display a valid or invalid string inside an Edit component. The invalid string is flawed because it has an incorrect length byte. Specifically, it sets the length byte to 150, although the string you want to print is only five characters long.

This simple mistake would cause trouble in your program. However, so you can clearly see what is going wrong, another *ScrambleString* procedure was added to the program.

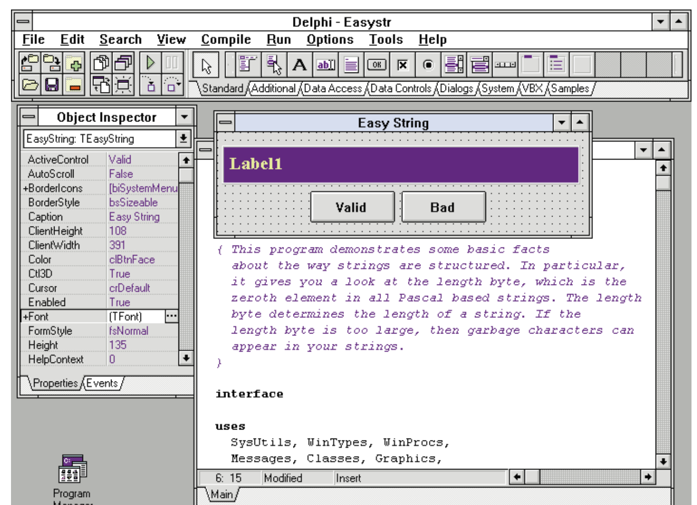



**Figure 2:** To create the form for the EASYSTR program, simply drop two buttons, a panel, and a label onto a form.

This ensures all the characters in a string are set to random values:

```
procedure ScrambleString(var S: string);
var
  i: Integer;
begin
  for i := 0 to 255 do
    S[i] := Chr(Random(255));
end;
```

This function sets all the characters in a string to various values between zero and 255 by using the *Random* function. *Random* returns a number between zero and the value passed in its sole parameter. However, in this case the goal is not to produce a random number, but a random character. To achieve this result, the *Chr* function is used. *Chr* is used to convert a numerical value into a character.

Note that there is no significant difference between using the *Chr* function and simply typecasting the value returned from the *Random* function. For example, this code produces the same result as using the *Chr* function:

```
S[i] := Char(Random(255));
```

Figure 3 depicts what may happen if you try writing a string to an Edit component when the length byte is set to an arbitrarily large value. Clearly the result is less than optimal. Characters are scattered across the component like some kind of hieroglyphic, and the word `Hello` is discernible only after you closely study the output.

You can compare the fiasco in Figure 3 with the orderly results in Figure 4. In this second case, the compiler is passed a valid length byte and the results are precisely defined and readily understandable.

## Null-terminated Strings

Before talking about strings in more depth, we'll discuss PChars, null-terminated strings, and the correct way to use an array of Char as a string.

A *null-terminated string* has no length byte. Instead, the compiler searches for a `#0` character and assumes that it marks the end of a string.

Specifically, if you return to the examples shown earlier, you can easily modify the functions so they will print a properly formatted null-terminated string:

```
procedure TForm1.BCharsClick(Sender: TObject);
var
  S: array[0..25] of Char;
begin
  S[0] := 'H';
  S[1] := 'e';
  S[2] := 'l';
  S[3] := 'l';
  S[4] := 'o';
  S[5] := #0;
  Edit1.Text := S;
end;
```

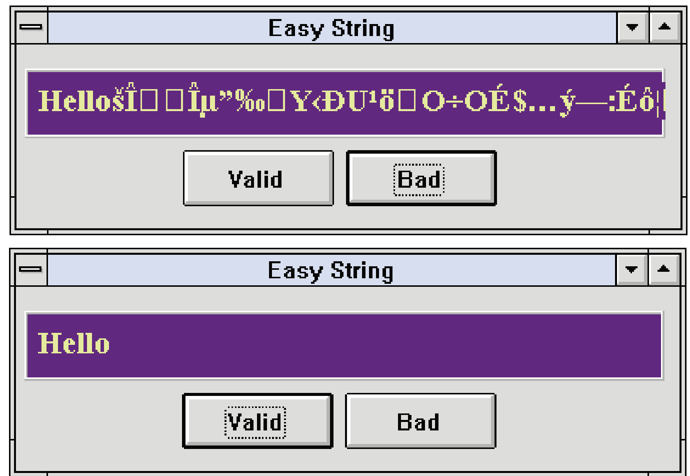



**Figure 3 (Top):** If the length byte of a string is set to the wrong value, the results can be chaotic. **Figure 4 (Bottom):** No matter how you scramble the extra characters in a string, the result shown to the screen is fine provided the length byte is assigned a valid value.

This code prints the word `Hello` in an orderly fashion. To do this, it sets the first character of a null-terminated string to the letter "H" and sets the sixth character to the value of the first member of the currently selected character set.

It's important to remember that you don't end null-terminated strings with the number zero — use `#0`. The number zero is usually the 48th member of a standard character set. It is entirely distinct from the first member of that character set.

Null-terminated strings are frequently referred to as PChars. *PChars* are pointers to arrays of Char. As a result you must allocate memory for them before you try to use them. For instance, the following code explicitly allocates 26 bytes of memory for a PChar before filling it with characters and displaying it on the screen.

The memory is then deallocated:

```
procedure TForm1.BCharsClick(Sender: TObject);
var
  S: PChar;
begin
  GetMem(S,26);
  StrCopy(S,'Hello');
  Edit1.Text := StrPas(S);
  FreeMem(S,26);
end;
```

When you allocate memory for a pointer, it's similar to a person entering the "memory theater" and telling a specific number of people they are part of a string. In this example, for instance, 26 members of the audience are grouped under the aegis of a single PChar.

Devotees of C/C++ will notice Delphi has a function called *StrCopy* that mirrors the job performed by *strcpy* in the land created by AT&T. That is, *StrCopy* copies one string into another. There are also functions called *StrCat*,

**Begin Listing Two — EASYSTR**

```
unit Main;

interface

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

type
  TEasyString = class(TForm)
    Panel1: TPanel;
    Label1: TLabel;
    Valid: TButton;
    Bad: TButton;
    procedure BValidClick(Sender: TObject);
    procedure BBadClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  EasyString: TEasyString;

implementation

{$R *.DFM}

procedure ScrambleString(var S: String);
var
  i: Integer;
begin
  for i  := 0 to 255 do
    S[i] := Chr(Random(255));
end;

procedure TEasyString.BValidClick(Sender: TObject);
var
  S: string;
begin
  ScrambleString(S);
  S[0] := #5;
  S[1] := 'H';
  S[2] := 'e';
  S[3] := 'l';
  S[4] := 'l';
  S[5] := 'o';
  Label1.Caption := S;
end;

procedure TEasyString.BBadClick(Sender: TObject);
var
  S: string;
begin
  ScrambleString(S);
  S[0] := #150;
  S[1] := 'H';
  S[2] := 'e';
  S[3] := 'l';
  S[4] := 'l';
  S[5] := 'o';
  Label1.Caption := S;
end;

end.
```

**End Listing Two**

*StrPos*, *StrCmp*, and so on, if you need them. The *StrPas* function converts a PChar into a string. PChars become particularly important when you start working with Windows API functions.

## Conclusion

Next month's article will discuss how to store strings in text files and retrieve them from text files. The third article will explore parsing the contents of a text file, and converting the data into fundamental Delphi types.

Next month we'll also explore some real-world examples of challenges you might face when working with strings. For example, one classic problem we'll tackle is the need to strip blanks from the end of a string. See you then. ∆

This article was adapted from material for Charles Calvert's book, *Delphi Unleashed*, published in 1995 by SAMS publishing.

*The demonstration program referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\AUG\CC9508.*

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

DBNAVIGATOR

DELPHI / OBJECT PASCAL

By *Cary Jensen, Ph.D.*

# Data Validation: Part I

## Validating Your Data in Delphi

**Every database application needs the ability to validate data. This can be as simple as ensuring that the user enters data in all upper-case letters, or as complex as verifying that an entered value is consistent with data stored in another table.**

This month's DBNavigator is the first of a two-part series on data validation. Part I introduces the basic concepts of data validation, and describes how to apply field-level validation to standard components (i.e. components from the Standard page of the Component Palette not associated with database tables). In Part II, we'll look at applying both record-level and field-level validation to data-aware components (i.e. components from the Data Controls page of the Component Palette).

### Introduction to Data Validation

*Record-level validation* refers to the application of data-validity rules when a record is being posted (written) to a table. When the user has finished editing a record, record-level validation code verifies the data is accurate. If the code determines the record is not acceptable, the record should not be posted to the table.

*Field-level validation* refers to the validation that takes place when a user enters data into a single field. (The term "field" is used here in its database sense. In Delphi a "field" usually takes the form of an Edit or DBEdit component.) In most cases, the validation process occurs when the user has completed editing the field, although it can also be applied after each keystroke. If the data in the field is determined to be unacceptable, the user is informed and won't be permitted to leave the field.

From the user's standpoint, record-level validation is more convenient than field-level. Specifically, when record-level validation is employed, the user can move freely among the records, leaving some fields only partially complete to move onto other fields. As long as the user completes these partial fields before attempting to post the data (and has also completed the record correctly), record-level validation code will accept the record. This process mimics the way a user interacts with a paper form.

On the other hand, field-level validation can be intrusive. A user attempting to leave a field — or perform another task that will result in the field being posted (such as inserting a new record) — when the field is not complete is interrupted by the validation code. This interruption may be as minor as the display of an error message in a status bar, or as significant as the display of a message in a modal dialog box requiring acknowledgment.

## Field Validation with Standard Controls

In Delphi, you will quickly learn that sometimes less programming is better. Specifically, if you can achieve a particular result using either properties or code, use properties. Sometimes this also means selecting the right component for the job. These rules certainly apply when it comes to field-level validation.

In addition, how you perform field-level validation depends in part on the type of component you are validating. The components (or controls) that you place on a form are available in two basic "flavors": data-aware (i.e. associated with a field in a table), or not. A data-aware edit component (i.e. a DBEdit component) has different events than one that is not (i.e. an Edit component), and consequently, requires distinct techniques for field-level validation.

Let's start by considering standard controls — components that are not linked to a table. There are six of them on the Standard page of the Component Palette:

- Edit
- Memo
- CheckBox
- RadioButton
- ListBox
- ComboBox

Of these six components, field-level validation is easiest with the last five. A Memo component typically contains simple text (in its *Lines* property) that is rarely evaluated programmatically.

The remaining four components can be set so the user cannot enter invalid data. For example, a CheckBox can have only two states; its *Checked* property can be *True* or *False*. There is no way for the user to place this component in an invalid state. Likewise, RadioButton and ListBox components display only those values available to a user. Even a ComboBox can be configured to permit only the selection of valid data.

This leaves the Edit component. Field-level validation is most difficult with Edit components. This is because the user is free to enter almost any value. However, your code may be expecting a value that conforms to a particular data type, such as a number, date, or time. Validation of Edit components often requires validation code (an example of which is provided later in this article).

Note: There is a component similar to Edit that is somewhat easier to validate without requiring code. This is the EditMask component, and it appears on the Additional Component Palette page. This component includes an *EditMask* property which can be used to limit which characters the user can enter into the control. Using the *EditMask* property is described in Part II of this series.)

## An Example of Code-Free Validation

The following example demonstrates how to create a standard control that does not require field-level validation code. Since the ComboBox component is the one that requires the most adjustment, this example will make use of that component.

Begin by creating a new project. (It's always a good idea to first create a directory to save the project in.) On your new form place a Label and a ComboBox component from the Standard page of the Component Palette (see Figure 1). Change the Label's *Caption* property to `&Day of Week:`, and its *FocusControl* property to *ComboBox1*. (A Label component cannot have focus, so the *FocusControl* property determines which component focus will shift to when you press the Label's short-cut key, "D" in this case.)

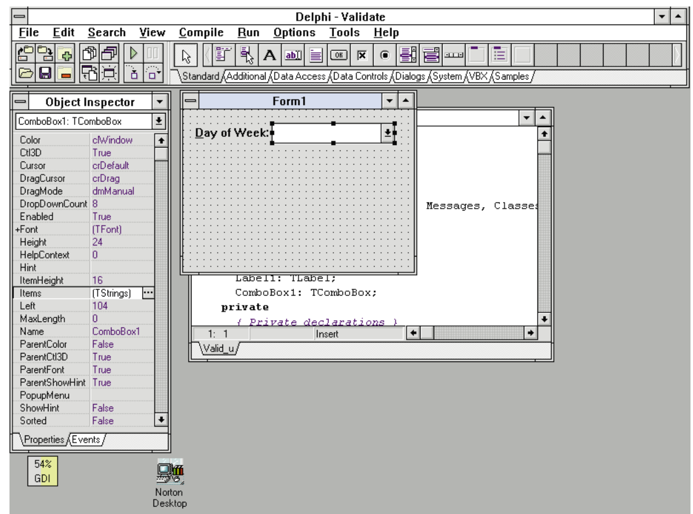



**Figure 1:** Building the example form.

Next, select the ComboBox. Begin by changing its *Style* property to *csDropDownList*. This style requires the user to select from the dropdown list. By comparison, when a ComboBox uses the *csDropDown* style, a user can either select from the dropdown list or enter text. Importantly, this entered text doesn't have to correspond to one of the dropdown list items, so you must validate *csDropDown* style ComboBox components using the same techniques as those used for Edit components.

You are now ready to define the contents of the ComboBox's dropdown list. Select the ComboBox's *Items* property in the Object Inspector and then click on the ellipsis that appears. Delphi displays the String list editor. Now enter the days of the week, one on each line, beginning with `Sunday` (see Figure 2). Click the **OK** button to save the list.

Press F9 to compile and run the program. The running form (see Figure 3) permits you to select any item from the dropdown list. Importantly, it does not permit you to enter a value that is not a day of the week. This is an ideal way of ensuring that your data is valid, without having to write code.

You may notice the running form doesn't have a default value in the ComboBox when the form first opens. When a ComboBox's *Style* is set to *csDropDown*, you can use the *Text* property to define the default value. But when *Style* is set to *csDropDownList*,

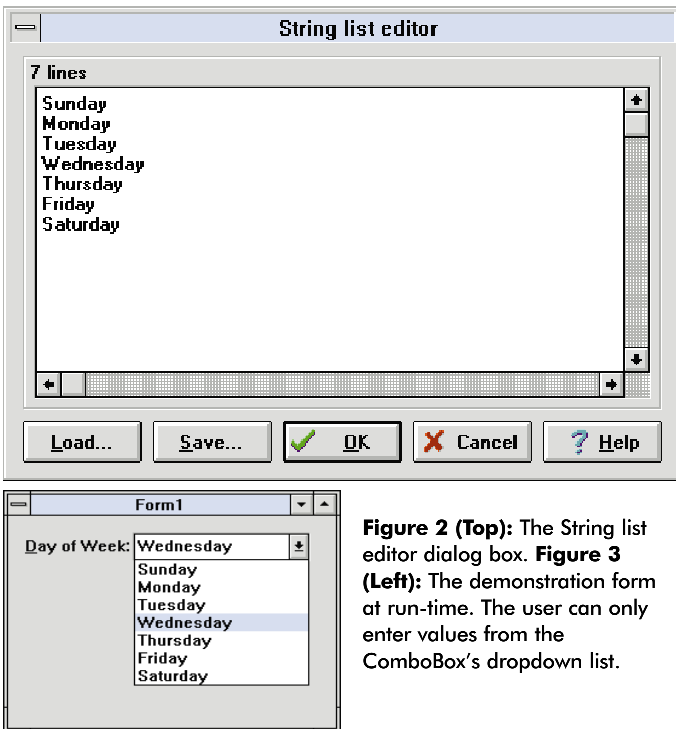



**Figure 2 (Top):** The String list editor dialog box. **Figure 3 (Left):** The demonstration form at run-time. The user can only enter values from the ComboBox's dropdown list.

the *Text* property is blank. If you need to define a default value for your *DropDownList* ComboBox, add the following line to the form's *OnCreate* event:

```
ComboBox1.ItemIndex := 0;
```

This Object Pascal statement sets the ComboBox's default to the first string stored in the *List* property (i.e. Sunday).

## Validating Standard Controls Using Code

Edit components are more complex to validate. On one hand, they possess properties you can use to ensure the user enters appropriate data. On the other hand, there are many situations where you must add code to ensure validity.

Important properties that you should consider using with Edit components include *CharCase*, *MaxLength*, *ReadOnly*, and *Text*. You can use *CharCase* if you want to control the case of data entered into an Edit component. For example, setting *CharCase* to *ecUpperCase* will convert all letters entered into that field to upper-case, while *ecLowerCase* will convert them to lower-case.

The *MaxLength* property enables you to define the maximum number of characters the user can enter into a field. If the user attempts to enter one character more than the defined *MaxLength* value, the form beeps and the character is rejected. When *MaxLength* is set to zero, no limit is enforced.

You use the *ReadOnly* property to prevent the user from changing a value in an Edit component. Obviously, if an Edit component is read-only, the user cannot change the value. Often, you modify the *ReadOnly* property at run-time, changing the Edit component from editable to non-editable, based on events on the form.

The *Text* property permits you to define the default value that will appear in the component. At a minimum, this property should be set to a blank string. It looks strange to the user if the component name (e.g. Edit1) appears by default.

However, these properties provide only limited validation of data. In most cases, the issue reduces to one of data type. Specifically, data entered into an Edit component is necessarily text. However, you will often use an Edit component to permit the user to enter a value of a particular data type, such as a number, an integer, a date, or a time. When the data you want the user to enter into a field is more restrictive than text, it's up to you to verify that the entered data conforms to the desired type.

There are a number of events to which you can attach your Edit component validation code. Some of these require more work than others. For instance, if you add your validation code to either the *OnKeyDown* or *OnKeyUp* event handlers, you must evaluate the entered data after each keystroke. In many cases, this is a lot of work. For example, if the Edit component is used to get a date from a user, your code must account for the fact that the value will not conform to a date value until the user enters the last character.

I have found it easier to validate data when the user attempts to leave the field, or before the value in the Edit component is used by another part of the program. For example, if a button contains code in its *OnClick* event handler that will generate a query based on a value in an Edit component, the *OnClick* code should first validate the Edit component's data. Likewise, if the validation needs to be performed before a form is closed, the validation code can be called from the form's *OnCloseQuery* event handler (and its *CanClose* parameter can be set to *False* if invalid code is found).

The specific technique you employ to validate the Edit component depends on the type of validation required. Since most of the time the validation relates to the data type of the entered text, a **try...except** statement is the most useful. In the **try** block you attempt to cast the *Text* property of the field to the particular data type.

If the value cannot be cast to the specified value, Delphi will generate an exception. You trap this exception in the **except** block, and respond accordingly by displaying a message in a status bar (usually a Panel component), or by raising a custom exception (displaying an error message you've defined).

Even if the Text property of the Edit object can be successfully cast to a date type, it is still possible that the value is not valid based on business rules. For example, you may not want to permit the user to enter a date later than today's date. When an unacceptable date is detected by your code, you can explicitly raise an exception. This can permit the rule violation to be handled by the same exception handler that processes the illegal assignment exception.

The following example demonstrates how to validate a field. Because this type of validation is often called from more than

one event, this code will be placed in a function. This allows it to be called from any event handler that must validate the field. (The specifics of creating a new function are not discussed here. Likewise, exception handling and exception creation is demonstrated without going into detail. If you need additional information about these topics, please refer to the *Delphi User's Guide*.) Begin by adding a Label component and an Edit component to your form. Change the *Caption* property of the Label component to `D&ate:`, set the *FocusControl* property to *Edit1*, and delete `Edit1` from the *Text* property (see Figure 4).

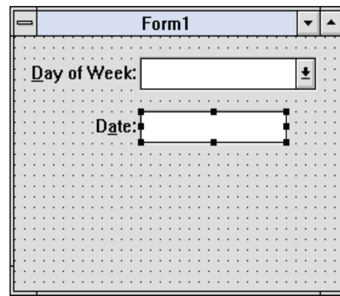



**Figure 4:** Adding an Edit component to the example form.

Next, select the Edit component and set its *MaxLength* property to `10` (10 characters is sufficient to permit the entry of a date). Next, press F12 to display the unit. Move your cursor to the line just above the final statement in the unit (`end.`), and enter the *ValidateDate* function shown in Figure 5.

The *ValidateDate* function takes a single parameter — an Edit component. This permits you to call this validation code from various routines, passing the name of a specific Edit component to validate. If the date in that Edit component is found to be invalid, an error message is displayed and *ValidateDate* returns the value *False*. Otherwise, the function returns the value *True*.

```
function TForm1.ValidateDate(TheField: TEdit): Boolean;
var
  NewValue: TDateTime;
begin
  { Initialize the return value. }
  Result := True ;
  { Do not evaluate if the field is blank. }
  if TheField.Text = '' then
    Exit;
  try
    { Cast the field's text to a date. }
    NewValue := StrToDate(TheField.Text);
    { The value is a date. Perform any additional tests. }
    if NewValue < StrToDate('1/1/95') then
      raise EEarlyDate.Create('Date cannot be before 1995')
    else
      { Valid date. Modify the value of ComboBox1.}
      ComboBox1.ItemIndex := DayOfWeek(newValue)-1;
  except
    on EEarlyDate do
      begin
        Result := False;
        raise;
      end;
    else
      begin
        Result := False;
        raise Exception.Create('Invalid date');
      end;
  end;

end;
```

**Figure 5:** This custom *ValidateDate* function uses a **try...except** block to validate a date value.

As an additional demonstration, this code also sets the value of the Day of Week ComboBox after a valid date has been entered. This is done by modifying the ComboBox's *ItemIndex* property. However, notice it's necessary to subtract one from the *DayOfWeek* function. This is because List components have a zero-based index, while the *DayOfWeek* function returns a value from one to seven. If you want to use the *ValidateDate* function as a generic date validation function, you would remove the following statement:

```
ComboBox1.ItemIndex := DayOfWeek(newValue)-1;
```

We're not finished implementing the *ValidateDate* function. It's still necessary to declare the function in the **interface** part of the unit. Add the header of the function (without the *TForm1* component name) to the **type** declaration.

It is also necessary to declare the exception *EEarlyDate*. This must be done before the form declaration. When you are through the **type** declaration should look like this:

```
type
  EEarlyDate = class(Exception);
  TForm1 = class(TForm)
    Label1: TLabel;
    ComboBox1: TComboBox;
    Edit1: TEdit;
    Label2: TLabel;
    function ValidateDate(TheField: TEdit): Boolean;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

All you need to do now is call this function from the *OnExit* event handler for the Edit component. Return to the form, select the Edit component, and double-click *OnExit* on the Events page of the Object Inspector. Modify the *Edit1Exit* event handler to look like this:

```
procedure TForm1.Edit1Exit(Sender: TObject);
begin
  ValidateDate(Edit1);
end;
```

Compile the project, save it as VALIDATE.DPR, and exit Delphi. Then select **File | Run** from Windows Program Manager and enter the name of the executable project (include the path and the name `VALIDATE`). Once the form is running, move to the **Date** field and enter an invalid date. Then press Tab to exit the field. This will produce an exception resembling that in Figure 6. Move back to the **Date** field and enter a valid date. This time, the value displayed in the **Day of Week** field will be updated.

Note: To see how the exception appears to the user, it is best to run this program from the Program Manager, rather than

from the Delphi IDE. If you run the program from the Delphi IDE, and if **Break on Exception** is enabled on the Preferences page of the Environment Options dialog box (the default setting), each exception puts the program into the Debugger. To see the exception the user sees, you must "step over" the exceptions created by Delphi as well as those you raise.

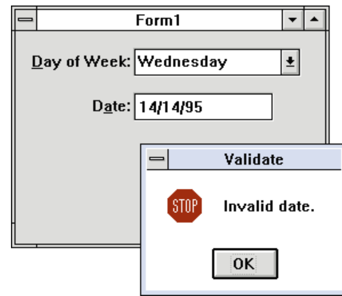



**Figure 6:** The code in Figure 5 produces this exception.

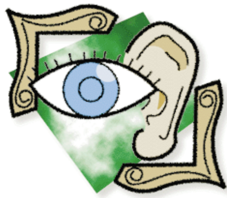
## Conclusion

It is often necessary to ensure that data entered by users conforms to certain parameters. Here we have considered how to perform this task with fields not associated with table data. While some of these components required no code to ensure valid data, Edit components do.

In the next installment, we'll look at techniques for performing field-level and record-level validation with data-aware controls. ∆

*The demonstration project referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\AUG\CJ9508.*

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



Sights and Sounds

Delphi / Object Pascal

By David Faulkner

# Show Your Colors

## Creating Visual Effects with Delphi's Canvas Property

**One of the many nice things about programming in Windows is its device independence. When you write information to the screen or printer, your program does not need to worry about loading, linking, or calling device drivers that may differ from machine to machine.**

The program in this article demonstrates device independence by creating the color triangle shown in Figure 1. Try this program with different video drivers, resolutions, and number of colors. By varying some of the calculations, you can create some spectacular screens.

### Writing to a Form's Canvas

Delphi makes it easy to place text on a form. You just place a Label component on a form and supply the appropriate caption. Because this is so easy, you may not be aware that you can programmatically place text on a form *without* using a Label component. For example, create a new form and place a Button component on it. In the button's *OnClick* method, write the following code:

```
Form1.Canvas.TextOut(10,10,'Hi Mom');
```

Now run the form and click the button. Assuming your button isn't in the wrong place, "Hi Mom" should appear in the screen's upper-left corner.

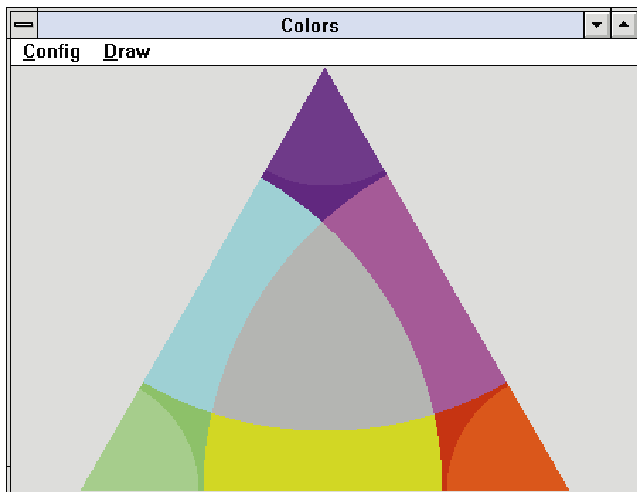



**Figure 1:** The color triangle program in action.

Each form you create has a *Canvas* property of type *TCanvas*. With the form's canvas you can easily draw lines, circles, boxes, and text. The *Canvas* property encapsulates the Windows device context. If you are not already convinced that Delphi is the "cat's meow", the *Canvas* property should convince you. If you had to write the "Hi Mom" code above using just the Windows API, you would need to do quite a bit of work creating and destroying pens and brushes.

The form's canvas also gives you access to the individual pixels on a form. This is done through the *Pixels* property:

```
property Pixels[X, Y: Longint]: TColor;
```

*X* and *Y* represent the horizontal and vertical canvas coordinates of a pixel and *TColor* represents the pixel's color. This property is both read and written at run-time so you can either set or determine the pixel's color.

For a quick example of the *Pixels* property, create a form and enter the following code into the form's *OnClick* method:

```
procedure TForm1.FormClick(Sender: TObject);
var
 x,y : integer;
begin
  for x:=0 to Form1.ClientWidth do
    for y:=0 to Form1.ClientHeight do
      Form1.Canvas.Pixels[x,y] :=
        $02000000 + Trunc($ffffff * Random);
end;
```

Run the form and click on it. The form should fill from left to right with random colors (not exactly exciting, but it gets better).

The *TColor* value of the *Pixels* property determines the color of a pixel. Its type is *Longint* so it is four bytes long. The first byte determines how the color is rendered on a palette. The next three bytes represent blue, green, and red, respectively. A value of zero for any of these colors means that color is not displayed at all, while a value of $FF (that's Delphi's hex notation for 255) means full intensity of that color. For example $02FF0000 is blue, $0200FF00 is green, $020000FF is red, $02000000 is black, and $02FFFFFF (all colors) is white.

## The Color Triangle

To create the color triangle, create a new form and call the *DrawTriangle* method (shown in Listing Three on page 38) from either a button or menu choice. Note that this code is more complicated than it needs to be just to draw a triangle. The complexity is there so the user can configure the application to create those cool screens (which we'll see a bit later).

First, the code in *DrawTriangle* determines the coordinates of the triangle's three points. The triangle's height is set to the form's height minus two pixels to give a one pixel border at the top and bottom of the form. The width is a bit more complicated to calculate, and you might need to brush up on your trigonometry and algebra skills to see where the formula comes from.

The Pythagorean theorem states the square of the hypotenuse of a right triangle is equal to the sum of the squares of the other two sides:

$$h^2 = x^2 + y^2$$

Here $h$ is the length of the hypotenuse, and $y$ is the height of the triangle, and we are looking for $x$, the width of the triangle. Since the color triangle is an equilateral triangle (all its sides are of equal length), we will work with the right triangle created by the line $y$. Since the line $y$ bisects the bottom of the triangle, we know that:

$$x = \frac{1}{2}h$$

By substituting this equation into the above Pythagorean equation, and doing some algebra we have:

$$h^2 = (\tfrac{1}{2}h)^2 + y^2$$

$$h^2 = \frac{1}{4}h^2 + y^2$$

$$\frac{3}{4}h^2 = y^2$$

$$h^2 = \frac{4}{3}y^2$$

$$h = \frac{2}{\sqrt{3}}y$$

In Object Pascal code this becomes:

```
const
  Sqrt3=1.732;    { The square root of 3. }

begin
  with Form1 do
    begin
      intTriangleHeight := ClientHeight-2;
      IntTriangleWidth  := Trunc(2*intTriangleHeight/Sqrt3);
```

Note here that the square root of three is declared as a constant named `Sqrt3`. This is because the square root of three is used in a number of places in the program including once in a loop. There is no need to slow down the program by making the computer do a calculation each time `Sqrt3` is needed.

With the height and width of the triangle determined, it's a simple matter to calculate the coordinates of the triangle's three vertices. A pair of nested **for** loops is used to visit every pixel in the triangle as follows (this code is a simplified version):

```
for y:= intTopY to intTriangleHeight do
begin
  for x:=intTopX-Trunc(y/Sqrt3) to intTopX+Trunc(y/Sqrt3) do
    begin
      { Statements to enter color screen pixels }
    end;
end;
```

The *intTopY* and *intTopX* variables represent the $x$ and $y$ coordinates of the triangle's top vertex. Notice the use of the `Sqrt3` constant again. This time the reader must perform the necessary algebra to calculate the start and end of the $x$ loop.

Within the nested loop, the objective is to vary the intensity of each color in proportion to the distance from that color's vertex. The distance between two points on a plane is calculated with this formula:

$$d = \sqrt{|x_2 - x_1|^2 - |y_2 - y_1|^2}$$

In code this becomes:

```
longBlueDistance := Trunc(Sqrt(Sqr(intTopX-x) +
                         Sqr(intTopY-y)))
```

Since the blue vertex is the top vertex, the *intTopX* and *intTopY* variables are used in this calculation. The *Sqrt* function returns a real number, so the *Trunc* function is used to convert the real number into an integer that can be assigned to the *longBlueDistance* variable. Similar calculations are done for each color.

With the distance known, the amount of blue is proportioned with the following code:

```
realBluePart := ((intTriangleWidth-longBlueDistance) /
                 intTriangleWidth);
longBlue     := Trunc($ff*realBluePart)*$ff*$ff;
```

The *realBluePart* variable is assigned a fraction between zero and one since the minimum *BlueDistance* is zero and the maximum *BlueDistance* is equal to the length of any side of the triangle. This fraction is then multiplied by 255 to obtain the actual blue intensity that will be used to color a pixel. Since the blue part of the *TColor* type is the second byte, the number is multiplied by 255 * 255 to shift it into its proper position.

Actually, the code would be faster and more accurate if it read:

```
longBlue := Trunc($ff*realBluePart) shl 16
```

The shift left operator, **shl**, shifts the value 16-bits (two bytes) to the left to move it into the proper position. The problem with this method is that it removes some of the cross color interference resulting in the various screens you'll create at the end of the article.

After each color is calculated, it can be used to assign a color to a screen pixel as follows:

```
Canvas.Pixels[x,y] := longBlue + longGreen + longRed;
```

That's it for creating the color triangle. Download the code or type it in (it's not that long) and try it.

## Configuring the Color Triangle

While writing the original color triangle code, I made a number of coding mistakes that yielded some wild results. The results were odd enough that I decided to make the program user-configurable so the user could recreate the same effects. I did this by creating the dialog box shown in Figure 2.

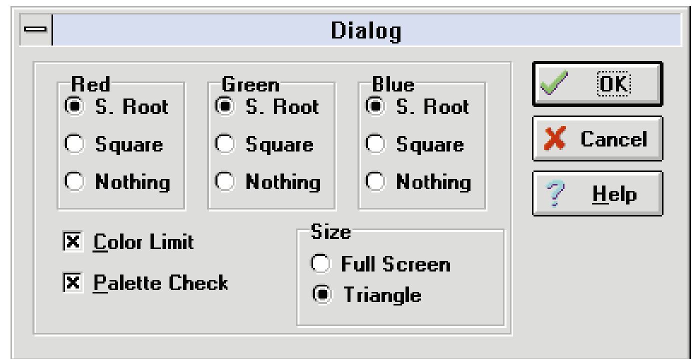



**Figure 2:** The configuration dialog box.

Adding a configuration dialog box to this or any other program is relatively easy, but there are a few things you must know. The configuration dialog box must communicate with the main form.

One way of doing this is with global variables that both the main form and configuration dialog box can access. Global variables that are meant to be accessed by any other program using a unit should be declared in the **interface** section of the unit:

```
interface

uses
type

var
  Form1 : TForm1;
  byteBlueOption : byte;
  byteRedOption : byte;
  byteGreenOption : byte;
  boolFullScreen : Boolean;
  boolColorLimit : Boolean;
  boolPaletteCheck : Boolean;

implementation
```

Before any of these variables can be used, they must be initialized. Since there is no guarantee the user is going to configure the program before they initiate the drawing process, it's a good idea to initialize the variables in the form's *Create* method:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  byteBlueOption   := 0;
  byteGreenOption  := 0;
  byteRedOption    := 0;
  boolFullScreen   := False;
  boolColorLimit   := True;
  boolPaletteCheck := True;
end;
```

To give the user access to the configuration dialog box, either a menu choice or a button can call this code:

```
procedure TForm1.Config1Click(Sender: TObject);
begin
  ConfigDialog.ShowModal;
end;
```

Note the unit that creates the configuration dialog box must be included in the main form's **uses** clause for the above code to compile. The *ShowModal* function shows the passed form and insists the user close that form before continuing.

## Getting Around a Circular Reference

The first problem the configuration dialog box faces is gaining access to the main form's global variables. It seems logical to accom-

plish this by placing the main form's unit in the dialog box's **uses** clause. If you try this, Delphi will respond with the "Error 68: Circular Unit Reference" when you compile. This happens because the main form uses the dialog box's unit, which in turn uses the main form's unit, which in turn uses the dialog box's unit, etc.

To work around this, the main form's unit should be added to the **implementation** part of the dialog box's unit. Declarations made in the **implementation** part of a unit are private and therefore cannot be seen by other units.

With the main form's unit (Coloru.PAS in this example) available to the dialog box, access to the main form's global variables is easy:

```
FullScreenButton.Checked      := Coloru.boolFullScreen;
TriangleButton.Checked        := not Coloru.boolFullScreen;
ColorLimitButton.Checked      := Coloru.boolColorLimit;
PaletteCheckButton.Checked    := Coloru.boolPaletteCheck;
```

This code is used to initialize the values of the various components in the configuration dialog box. When the dialog box is closed, similar code is used to set the values of the main form's global variables.

## Compiler Directives

You may have noticed the following two lines of code at the beginning of the Coloru unit:

```
{$R-}
{$Q-}
```

Although these look like comments, they do a lot more. Any comment beginning with a dollar sign ( $ ) tells Delphi to do something special at compile time. The {$R-} compiler directive tells Delphi to turn off range checking. This quickens the compile and allows the math in the *DrawTriangle* method to go out of range without causing a run-time exception. The {$Q-} compiler directive tells Delphi to turn off overflow checking in integer math. Again, this speeds compilation and allows the integer math in the *DrawTriangle* method to overflow during run-time without causing exceptions.

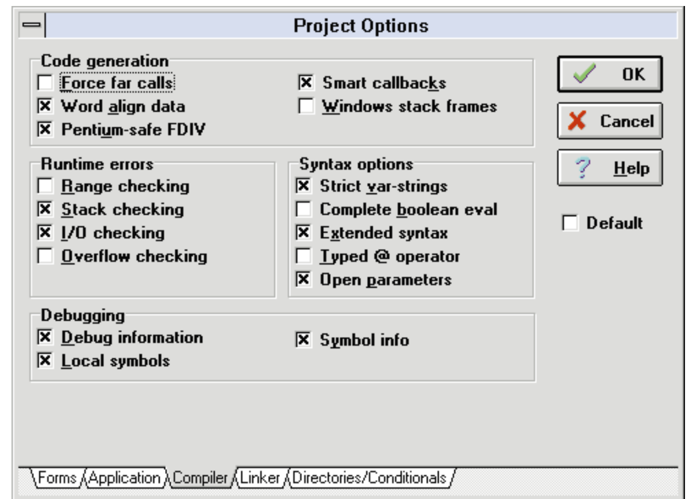
These two options can be set interactively on the Compiler page of the Project Options dialog box (see Figure 3), but it's better to embed the compiler directives in source code. This allows the code to be compiled on other machines without having to check the compiler settings.

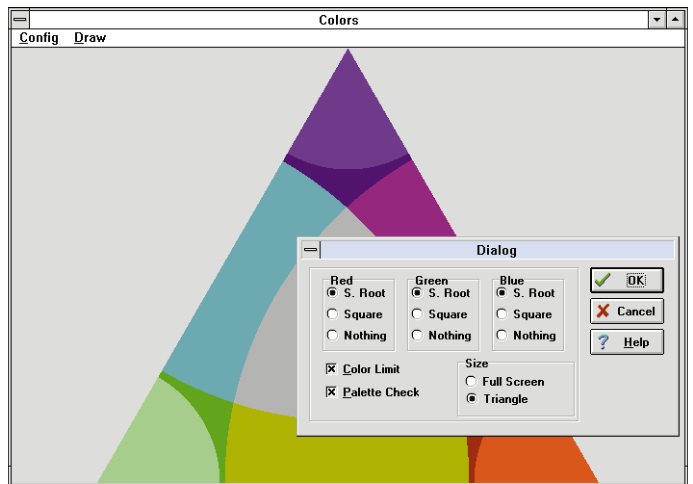
One other compiler directive you might want to play with is {$N-}. This tells Delphi to do real number math in software instead of using the floating point coprocessor on your machine. If you use this directive, you will notice a big slowdown in the program (unless of course, your machine doesn't have a math coprocessor).

## Conclusion

So have some fun. Use the configuration dialog box to produce the special effects shown in Figures 4, 5, 6, and 7.

Delphi gives the programmer quick and easy access to the form's canvas and you can experiment with various lines, boxes, pixels, colors, and fonts. This may seem like a frivolous exercise, but when it comes time to write your components you'll use the skills learned here to write custom *OnPaint* code. ∆



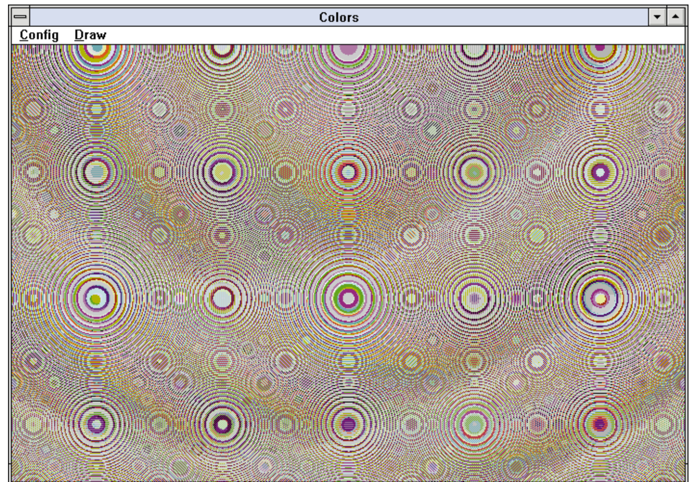
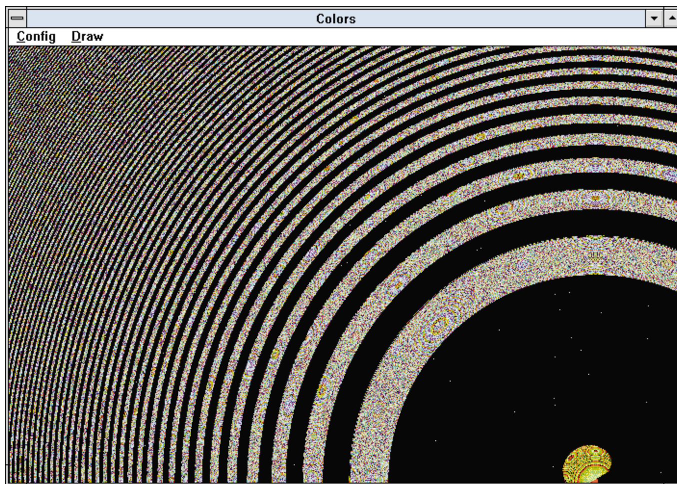




**Figure 3 (Top):** The Compiler page of the Delphi Project Options dialog box. **Figure 4 (Middle):** Fun with colors. Use these settings to create the color triangle as shown. **Figure 5 (Bottom):** To generate this "field of circles", set the **Red**, **Green**, and **Blue** radio buttons to: **S. Root**, **S. Root**, and **Nothing** (respectively). Deselect **Color Limit**, check **Palette Check**, and select **Full Screen**.



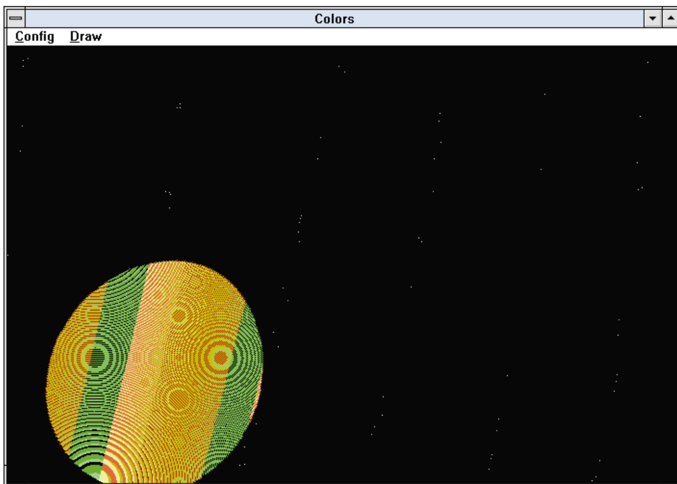




**Figure 6 (Top):** This "super nova" is created by setting **Red** to **Square**, and **Green** and **Blue** to **S. Root**. **Color Limit** and **Palette Check** are not checked. **Figure 7 (Bottom):** To get this "cosmic egg" set **Red** and **Green** to **Nothing** and Blue to **S. Root**. Deselect **Color Limit** and **Palette Check**.

*The demonstration project referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\AUG\DF9508.*

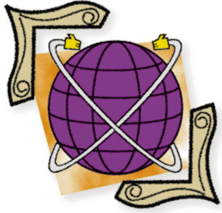
David Faulkner is a developer with Silver Software in Kula, Hawaii. He is also Contributing Editor to *Paradox Informant,* and co-author of *Using Delphi: Special Edition* (Que, 1995). Mr Faulkner can be reached at (808) 878-2714, or on CompuServe at 76116,3513.

**Begin Listing Three — Coloru.PAS**

```
{$R-}
{$Q-}

unit Coloru;

interface

uses
SysUtils, WinTypes, WinProcs, Messages, Classes,
Graphics, Controls, Forms, Menus, Config;

type
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    Config1: TMenuItem;
    Draw1: TMenuItem;
    procedure Config1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure DrawTriangle;
    procedure Draw1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  byteBlueOption: Byte;
  byteRedOption: Byte;
  byteGreenOption: Byte;
  boolFullScreen: Boolean;
  boolColorLimit: Boolean;
  boolPaletteCheck: Boolean;

implementation

{$R *.DFM}

procedure TForm1.DrawTriangle;
var
  { Coordinates of top and bottom left of triangle }
  intTopX,intTopY,intLeftX,intLefty,
  { Coordinates of bottom right of triangle }
  intRightX,intRightY: Integer;
  { Height and width of triangle }
  intTriangleHeight,intTriangleWidth: Integer;
  { Distance current position to triangle point }
  longBlueDistance,
    longGreenDistance,longRedDistance: Longint;
  { Fraction of ColorDistance to MaxDistance }
  realBluePart,realGreenPart,realRedPart: Real;
  { Color of pixel to be placed on screen }
  longNewColor: TColor;
  { Amount of RGB for each color }
  longRed,longGreen,longBlue: Integer;
  { First in last pixel in row of triangle }
  intXBegin,intXLimit,
   { First and last pixel in col of triangle }
    intYBegin,intYLimit: Integer;
  { Loop counters }
  x,y: Integer;

const
  { The Square Root of 3 }
  Sqrt3=1.732;

begin
with Form1 do
  begin
  { One pixel space on top and bottom }
  intTriangleHeight := ClientHeight-2;
  { Get out your trig book }
```

```
  intTriangleWidth  := Trunc(2*intTriangleHeight/Sqrt3);
  { Top is half way across screen }
  intTopX := ClientWidth div 2;
  { and one pixel down }
  intTopY := 1;

intLeftX := (ClientWidth - intTriangleWidth) div 2;
intLefty := intTopY+intTriangleHeight;
intRightX := (ClientWidth + intTriangleWidth) div 2;
intRightY := intTopY+intTriangleHeight;

if boolFullScreen then
  begin
    intYLimit := ClientHeight;
    intYBegin := 0;
  end
else
  begin
    intYLimit := intTriangleHeight;
    intYBegin := intTopY;
  end;

for y:=intYBegin to intYLimit do
begin
  if boolFullScreen then
    begin
    intXLimit := ClientWidth;
    intXBegin := 0;
    end
  else
    begin
    { Get out that trig book again }
    intXBegin := intTopX-Trunc(y/Sqrt3);
    intXLimit := intTopX+Trunc(y/Sqrt3);
    end;

  { For each pixel in this row }
  for x:=intXBegin to intXLimit do
  begin
    case byteBlueOption of
        0: longBlueDistance := Trunc(Sqrt(Sqr(intTopX-x) +
                                   Sqr(intTopY-y)));
        1: longBlueDistance := Sqr(Sqr(intTopX-x) +
                                 Sqr(intTopY-y));
        2: longBlueDistance:=  Abs(Sqr(intTopX-x) +
                                 Sqr(intTopY-y));
    end;

    realBluePart := ((intTriangleWidth-longBlueDistance) /
                     intTriangleWidth);
    longBlue := Trunc($ff*realBluePart)*$ff*$ff;

    if boolColorLimit then
      longBlue := longBlue and $00ff0000;
    case byteGreenOption of
      0: longGreenDistance := Trunc(Sqrt(Sqr(intLeftX-x) +
                                   Sqr(intLefty-y)));
      1: longGreenDistance := Sqr(Sqr(intLeftX-x) +
                                 Sqr(intLefty-y));
      2: longGreenDistance := Abs(Sqr(intLeftX-x) +
                                 Sqr(intLefty-y));
    end;

    realGreenPart := ((intTriangleWidthlongGreenDistance)/
                     intTriangleWidth);
    longGreen := Trunc($ff*realGreenPart)*$ff;

    if boolColorLimit then
      longGreen := longGreen and $0000ff00;

    case byteRedOption of
      0: longRedDistance := Trunc(Sqrt(Sqr(intRightX-x) +
                                 Sqr(intRightY-y)));
      1: longRedDistance := Sqr(Sqr(intRightX-x) +
                               Sqr(intRightY-y));
      2: longRedDistance := Abs(Sqr(intRightX-x) +
                               Sqr(intRightY-y));
     end;

     realRedPart := ((intTriangleWidthlongRedDistance)/
                    intTriangleWidth);
     longRed := Trunc($ff*realRedPart);

     if boolColorLimit then
       longRed := longRed and $000000ff;

     longNewColor := longBlue+longGreen+longRed;

     if boolPaletteCheck then
       longNewColor := longNewColor and $02ffffff;
     Canvas.Pixels[x,y] := longNewColor;
  { End of for x loop }
  end;
{ End of for y loop }
end;

{ End of 'with Form1 do begin' }
end;
{End of procedure}
end;

procedure TForm1.Config1Click(Sender: TObject);
begin
  ConfigDialog.ShowModal;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  byteBlueOption    := 0;
  byteGreenOption   := 0;
  byteRedOption     := 0;
  boolFullScreen    := False;
  boolColorLimit    := True;
  boolPaletteCheck  := True;
end;

procedure TForm1.Draw1Click(Sender: TObject);
begin
  DrawTriangle;
end;

end.
```

**End Listing Three**



By Rand McKinney

# Delphi on the Web

## A Guide to Delphi Resources on the World-Wide Web

**Unless you've been under a rock for the last year, you've at least heard of the world-wide web (WWW), a graphical, hypertext-based interface to the Internet. If you have a "browser" tool such as Netscape or Mosaic (both are available free), and a proper connection to the Internet, you can access a virtually limitless array of information provided on web sites or "pages" by universities, corporations, and individuals.**

In recent months, a number of Delphi-centric web pages have appeared. Created by Delphi programmers, consultants, and writers, these web pages provide a wealth of useful information that's just a few mouse clicks away. Many sites have Delphi demonstration applications and components available for download for free, or as shareware. Some provide forums for questions and answers on Delphi programming, and others are simply marketing vehicles for Delphi application developers. These web sites literally span the globe: there are Delphi web pages on servers in Australia, Sweden, Poland, Germany, and across the United States.

### Web Pages

The following is an incomplete list of some of the best web pages devoted to Delphi. Each page has a URL (Uniform Resource Locator) that is essentially the WWW address of the page. You can type them or use the HTML file provided with this article (see end of article for details). As of this writing, these URLs are accurate. However, the web is dynamic by nature so some may have changed.

### Official Web Pages

The following are the only official Borland web pages.

**The Official Borland Delphi Web Page**

`http://www.borland.com/Product/Lang/Delphi/Delphi.html`

This page includes a Delphi fact sheet, news, and press releases from Borland, downloadable documentation in Adobe Acrobat format, lists of Delphi educational resources, and other Borland information.

**Borland FTP Server on the Web**

`ftp://ftp.borland.com/pub/ftpmenu.htm`

The Borland FTP (File Transfer Protocol) server, presented in living HTML for your browsing pleasure. This is the official place to download a variety of Delphi-related files and other Borland products. If you don't have a web browser, you can access the same files at `ftp.borland.com`.

**Delphi Technical Support Home Page**
`http://loki.borland.com:8080/`
The Delphi technical support web page, including Delphi questions and answers, links to the Borland Database Engine (BDE), ReportSmith, SQL Links web pages, and other relevant information.

## Commercial Sites

The following web pages are maintained by companies or individuals for commercial reasons. Some require payment for their components, books, or applications, but many have freeware and shareware available for download.

**InfoPower Component Library**
`http://www.webcom.com/~wol2wol/infopowr.html`
InfoPower, from Woll2Woll Software, is a library of Delphi data-aware components that are automatically installed into Delphi's Component Palette. [InfoPower is reviewed on page 45.]

**The Coriolis Group's Delphi Explorer**
`http://www.coriolis.com/coriolis/whatsnew/delphi.htm`
Excerpts from the book *Delphi Programming Explorer*, Delphi articles, commercial and shareware Delphi software, etc. are available.

**Delphi RADical Application Development**
`http://super.sonic.net/ann/delphi/`
This page of components enables you to use Delphi for web server CGI programming. It includes descriptions, demonstrations, and downloadable components. There's also a link to 32 downloadable freeware components created by Michael Ax.

**Delphi *TAutoButton* Component**
`http://www.widewest.com.au/delphi/`
A Delphi component built by a Delphi programmer with AeroSoft in Australia.

**The City Zoo**
`http://www.mindspring.com/~cityzoo/cityzoo.html`
Tips and tricks, Delphi announcements, components, etc., from a Delphi consultant in Florida.

**CIUPKC Software**
`http://www.webcom.com/~kilgalen/welcome.html`
CIUPKC Software (pronounced "soo-pack") is a small software development company specializing in software built with Delphi. Their page has a few shareware components, and lots of links to other sites and software.

## Non-commercial Sites

The following web pages are maintained by individuals for their own reasons — primarily out of sheer love for Delphi and its capabilities. These are always informative, sometimes amusing, and occasionally quirky.

**Delphi FAQ**
`http://www.mhn.org/delphi.faq`
Comprehensive, but unofficial, Delphi Frequently Asked Questions list.

**Delphi Bug-List Web Page**
`http://www.cybernetics.net/users/bstowers/delphi-bugs.html`
Unofficial list of known bugs and work-arounds in Delphi, maintained by a Delphi programmer in North Carolina.

**Delphi Hacker's Corner**
`http://tmpwww.electrum.kth.se/~ao/DHC/`
An excellent web page maintained by Anders Ohlsson in Sweden. It includes tips and tricks, demonstrations, a list of major Delphi users' groups, and European mirror sites for Delphi FTP resources.

**The Delphi Station**
`http://www.teleport.com/~cwhite/wilddelphi.html`
With dozens of shareware and freeware Delphi components, there's an incredible amount of useful information, examples, and other Delphi-related stuff on this page.

**Dave's Delphi Destination**
`http://vislab-www.nps.navy.mil/~drmcderm/delphi.html`
Numerous valuable resources, including *The Unofficial Delphi User's Newsletter* (in Windows on-line help format), calendar components, and the famous *TSmiley* component, are available on this page.

**Michael's Delphi Home Page**
`http://linux.rz.fh-hannover.de/~holthoefer/delphi.html`
Lots of components, demonstrations, and other good stuff.

**The Delphi Super Page**
`http://sunsite.icm.edu.pl/archive/delphi/`
Straight from Poland: Warsaw University's Sunsite has a Delphi page loaded with demos, tips and tricks, shareware and freeware components.

**Annotated Bibliography of Delphi Articles and Books**
`http://www.iscinc.com/dugbib.html`
A list similar to this one, but for printed materials on Delphi.

## General Pascal Pages

**Turbo Pascal Programmer's Guide**
`http://www.cs.vu.nl/~jprins/tp.html`
A comprehensive list of web sites and resources related to the Pascal language, maintained by a Pascal programmer in The Netherlands.

**The Perkins Pascal Page**
`http://www.iii.net/users/rexkp.html`
This page provides current information on BugSlay, the Pascal Postmortem Debugger, as well as links to other Pascal related information.

## User Group Sites

There are many Delphi user groups around the world. Anders Ohlsson in Sweden maintains a comprehensive list of them at the following URL:

`http://tmpwww.electrum.kth.se/~ao/DHC/dug.html`
Some of the user groups even have their own web pages.

**New York Delphi Users' Group**
`http://www.iscinc.com/nydug.html`

**North Bay Delphi Special Interest Group**
`http://Super.Sonic.Net/delphisig/index.html`

**Salt Lake City Delphi Users' Group**
`http://www.xmission.com/~uldata/delphi.html`

## Discussion Forums and Newsgroups

**Delphi WWW Forum**
`http://www.pennant.com/delphi/hn/dconn.html`
A newsgroup-like forum in HTML format for Delphi users.

`comp.lang.pascal`
The Usenet newsgroup for discussion of Pascal-related topics, including Delphi.

`alt.comp.lang.borland-delphi`

Alternative-hierarchy Usenet newsgroup for discussion of Delphi. ∆

*The HTML referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\AUG\RM9508.*

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

# At Your Fingertips

By David Rippy

Delphi / Object Pascal

*For the things we have to learn before doing them, we learn by doing them.*

— Aristotle, 384-322 BC

## How can I create "hot regions" on my form?

A common way of presenting options to a multimedia application user is to display an attractive bitmap in place of the usual Windows menus and buttons. For example, the menu screen shown in Figure 1 consists of a single Image object created in CorelDRAW! that displays three pictures representing different areas of the system the user can access.

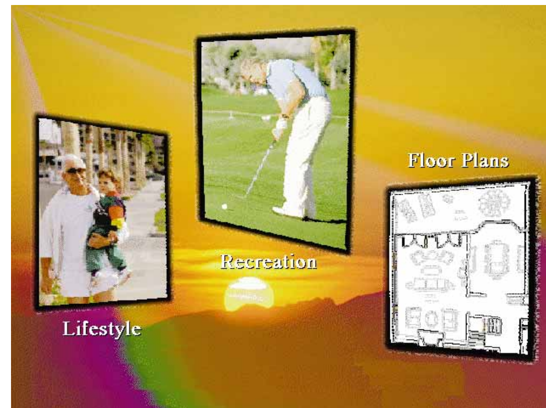



**Figure 1:** This "menu screen" was created in Corel DRAW!

The problem is, we need to be able to detect when the user clicks within the areas of the three pictures. For example, we must load the Recreation form when the user clicks on the man playing golf. To do this, we can use the *TShape* component and define an invisible "hot region" around the golfer. This will enable us to detect when the user clicks within that area. In essence, these hot regions become our "buttons" and can respond to mouse events such as *OnMouseDown* and *OnMouseUp*.

**Figure 2:** Stretch the Shape objects to cover the area that you need to trap for mouse events.

As demonstrated in Figure 2, create a shape object on your form and stretch it until it covers the region that you want to trap for mouse events. You can change the *Shape* property of the shape object to better fit this region. (Note that the icon representing the Shape component is a bit misleading; you can't create a triangle.) Finally, change the *BrushStyle* property to *bsClear* and the *PenStyle* to *psClear*. This will cause our shape to be invisible at run-time. Create as many hot regions as needed for your screen.

As mentioned earlier, the all-important benefit of creating these regions is they provide you with several events you can trap for in your program including the *MouseDown* event. Also, it's likely you'll attach code for loading additional forms, displaying information, playing a sound bite, etc. — *D.R.*

## How can I create a simple glossary system for a memo field?

For applications that rely heavily on terminology that may be unfamiliar to the user, it's often a good idea to include a glossary system to provide definitions. If you're on a shoestring budget, here's a quick and dirty way to give your users the information they need with a minimum of code.

Figure 3 shows a form with a *TMemo* component containing text that explains some highly technical information. To see the definition of a word within the memo field, the user simply highlights the word and clicks on the **Definition** button. A dialog box with the definition of the selected word will be displayed. For example, in Figure 4 the user is unsure of the meaning of "life". However, by clicking the **Definition** button, the meaning of life is made perfectly clear.

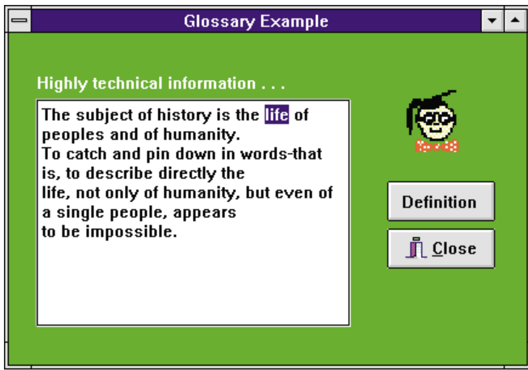
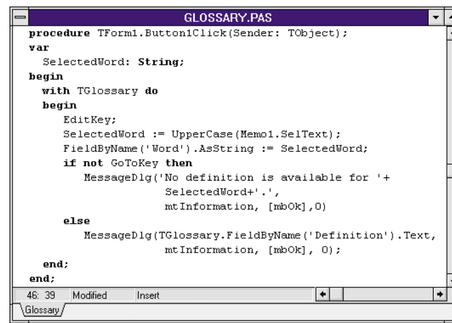



**Figure 3:** You can highlight a word in Windows by double-clicking on it.



```
procedure TForm1.Button1Click(Sender: TObject);
var
  SelectedWord: String;
begin
  with TGlossary do
  begin
    EditKey;
    SelectedWord := UpperCase(Memo1.SelText);
    FieldByName('Word').AsString := SelectedWord;
    if not GoToKey then
      MessageDlg('No definition is available for '+
                 SelectedWord+'.',
                 mtInformation, [mbOk],0)
    else
      MessageDlg(TGlossary.FieldByName('Definition').Text,
                 mtInformation, [mbOk], 0);
  end;
end;
```

**Figure 7:** This code is attached to the *OnClick* event of the **Definition** button.

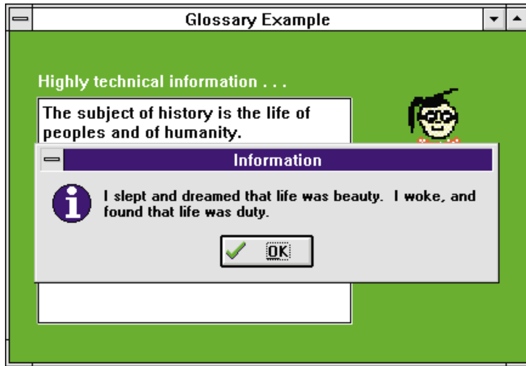



**Figure 4:** Pressing the **Definition** button will display the word's definition.

The first step in implementing the glossary system is to create a table called GLOSSARY.DB with the structure shown in Figure 5. In this table, you'll store all the words and their meanings for the application. As shown in Figure 6, the sample table contains the definitions for several words including "LIFE". It's important to capitalize each letter of the word in the WORD field for the **IndexFields** command to work properly. If you're using a Paradox table, you can automate this task by creating a picture of `*&` for the WORD field (see Figure 5 again).

| Field Name | Type | Size | Key |
|---|---|---|---|
| Word | A | 20 | * |
| Definition | A | 60 | |

**Figure 5:** Table structure for the GLOSSARY table.

Table : GLOSSARY.DB

| GLOSSARY | Word | Definition |
|---|---|---|
| 1 | HUMANITY | Human beings collectively. |
| 2 | LIFE | I slept and dreamed that life wa |
| 3 | SINGLE | One and only. |
| 4 | SUBJECT | A topic of conversation. |

**Figure 6:** Define as many words as needed for your application.

All the code necessary for the glossary system is located in the *OnClick* event of Button1 as displayed in Figure 7. When the user clicks on the **Definition** button, the table object *TGlossary* is immediately placed in a kind of "search mode" by calling the *EditKey* method. The word selected by the user is then converted to upper-case and used to specify the value we want to locate in the WORD column via the **FieldByName** command. Finally, the *GoToKey* method will attempt to locate the selected word. If it is found, the definition of the word is displayed in a dialog box. If the selected word is not contained in the GLOSSARY table, a dialog box informs the user that no definition is available for the highlighted word. — *D.R.*

## How can I easily copy a record from one table to another?

Here's a simple way to copy a single record from one table to another, with only a few lines of code. Figure 8 shows a form containing two table objects, named Table1 and Table2, and their corresponding DataSource and DBGrid components. This example will copy the current record from Table1 into Table2. Further, we will instantly see the new record appear in the DBGrid of Table2 when the **Copy Record** button is pressed.

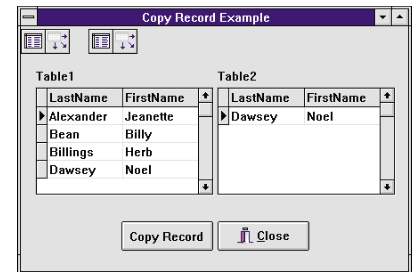
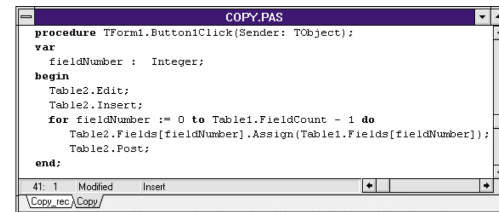



**Figure 8:** This form copies a record from Table1 into Table2.

All the code for copying a record into a destination table with an identical record structure is contained in the *OnClick* event of the **Copy Record** button as shown in Figure 9. Most of the important code is contained within the **for** loop. Basically, it steps through each field in the current record of Table1 and assigns their values to the same set of fields in Table2. Just make sure your tables are open, either interactively or programmatically, before attempting to copy.



```
procedure TForm1.Button1Click(Sender: TObject);
var
  fieldNumber : Integer;
begin
  Table2.Edit;
  Table2.Insert;
  for fieldNumber := 0 to Table1.FieldCount - 1 do
    Table2.Fields[fieldNumber].Assign(Table1.Fields[fieldNumber]);
    Table2.Post;
end;
```

**Figure 9:** This code is attached to the *OnClick* event of the **Copy Record** button.

Note: For simplicity, there is no exception handling for duplicate records — you must only copy one record at a time. In a real application, you would want to handle key violations more gracefully. — *D.R.* ∆

*The demonstration forms referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\AUG\DR9508.*

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

BY JOSEPH C. FUNG

# InfoPower

## Woll2Woll's VCL Components for Database Developers

**If you've been waiting for a third-party vendor to supply an alternative to Delphi's built-in data access components, your wait is over. Woll2Woll Software has released InfoPower, a comprehensive collection of native VCL components for database developers. InfoPower consists of 15 data-aware components that add powerful new capabilities to Delphi, including a sophisticated data grid, enhanced DataSet and DataSource components, filtering capabilities, incremental searching, expanding memos, editable lookup combo-boxes, and a set of standard and user-defined search dialog boxes.**

Users familiar with the capabilities of Paradox for Windows' filters and TableFrame object will appreciate this software because it brings similar capabilities to Delphi. InfoPower's components are also optimized to account for client/server operations so you can access remote data with minimal overhead. In addition, applications you create with InfoPower can be distributed royalty-free.

### Installing InfoPower

Installing InfoPower on your system is easy if you follow the instructions in the user manual. InfoPower comes with the typical Windows-hosted installation program that copies the necessary files to your hard disk. Once installed, you're instructed to complete a series of manual steps to create an alias for the optional demonstration programs, install the components into Delphi's component palette, and merge the InfoPower on-line help into Delphi.

For those tempted to skip the user manual and dive right into the sample applications, be sure to follow these steps or you might have trouble running the demonstration files. These files require you to create an alias called InfoDemo that points to the directory containing the sample tables.

After you are finished installing InfoPower, you can immediately begin using its components to build a database application.

### Enhanced DataSet and DataSource Controls

At the core of InfoPower are three new data access components: *TwwDataSource*, *TwwTable*, and *TwwQuery*. (There are actually four if you include a new *TwwQBE* component, but we'll ignore it for now.) These new components are enhanced versions of the built-in *TDataSource*, *TTable*, and *TQuery* components. They build upon the basic VCL versions by adding new properties and methods. Many of InfoPower's visual and non-visual components rely on the extended functionality provided by these new components to work. The *TwwDataSource*, *TwwTable*, and *TwwQuery* components are derived directly from their Delphi counterparts and are 100% backward compatible. Because of this, you can substitute the InfoPower versions directly into your existing forms, including the ones created by the Form Expert.

While *TwwDataSource* and *TwwQuery* serve mainly to interface with other InfoPower components, *TwwTable* adds important new functionality. A new *Filter* property has been added to support the ability to filter tables (explained below). A new *wwFindKey* method has also been added to permit optimal searching against SQL tables. This new method replaces Delphi's *FindKey* method that is slow when used against large remote data sets. Finally a new *Pack* method lets you remove dead space from Paradox and dBASE tables.

### Access to BDE Filters

By default, Delphi lets you use ranges and queries to select a subset of a table. The Borland Database Engine (BDE) also supports the concept of *filters*. Unlike ranges, filters are not restricted to indexed fields. A filter offers the flexibility of a query because a filter lets you use an expression to specify a criteria for any or all the fields in a table. A filter also provides you with an editable result set, which may not always be possible with a query.

Delphi lets you use filters, but only if you are willing to do some engine-level programming. To implement filters in Delphi, you must program complex data structures and make BDE calls. If the idea of using pointers scares you, you're better off using the filtering capability in InfoPower. InfoPower makes using filters a breeze.

To create a filter, simply use the *TwwTable* component's *Filter* property and *FilterActivate* method. The *Filter* property is a *TStringList* so use the *Add* method to assign criteria to the filter. Valid criteria statements can contain relational operators (< , > , <= , >= , = ,<>) and the AND/OR logical operators. After specifying the filter, call *FilterActivate* to apply the criteria against your table. The following Object Pascal code illustrates how to create a filter that shows orders that have amounts exceeding 100 dollars:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with TwwOrdersTable1 do
    begin
      Filter.Clear;
      Filter.Add('Amount > 100');
      FilterActivate;
    end;
end;
```

The source table doesn't need to be indexed to use a filter. If an index does exist, however, you can select it for any purpose, such as to change the display order of the records. The availability of an index in the result set is one of the key advantages that filters have over queries.

Filters also have some disadvantages. They tend to be slower than ranges because each record is evaluated as it is seen. Also, when you apply a filter on a table, the *RecordCount* property reflects the total number of records in the table, not the number of records matching the filter. To obtain an accurate record count, you can use Object Pascal to iterate through the result set and count the records.

## New QBE Component

Besides *TwwTable* and *TwwQuery*, InfoPower offers a third data access component: *TwwQBE*. This useful component enables you to use Paradox-style query-by-example (QBE) statements to query tables. This is a welcome addition to Paradox for Windows developers who are familiar with QBE and want to make the transition to Delphi. Not only does *TwwQBE* let you run existing QBE statements unmodified, it also permits the creation of some queries not possible when you use SQL syntax against local tables (i.e. the dreaded "Capability Not Supported" message).

To use the *TwwQBE* component, Woll2Woll recommends you create the QBE statement from the Database Desktop or Paradox for Windows, then load it into the *QBE* property of the component. You can also use the *Add* method to build the QBE statement at run-time since the QBE property is a *TStringList*. Finally, the *AnswerTable* and *AuxiliaryTables* properties are available so you can define the result set and any auxiliary tables arising from INSERT, DELETE, or CHANGETO operations.

## Sophisticated Database Grid

The centerpiece of InfoPower's enhanced data controls collection is the *TwwDBGrid* component. This component extends upon Delphi's built-in *TDBGrid* by adding significant new functionality, such as the ability to attach combo-boxes, checkboxes, and memos to the grid. Fields from multiple tables in a one-to-one relationship can be displayed on the same row without coding. In addition, you can define fixed, non-scrollable columns in the grid so they always appear as the user pans through the fields. You can also alter the default appearance of the grid, including word-wrapping the titles and changing individual cell colors. Another important, but subtle, improvement is the accurate display of the vertical scrollbar's thumb when scrolling through Paradox or dBASE tables. Delphi's *TDBGrid* leaves the thumb position "stuck" in the center of the scrollbar. Many of InfoPower's customers will be purchasing the software just to obtain this component.

Back Orders

| Order No | Sale Date | Ship VIA | Terms | Payment Method | Amount Paid | Runr Balai |
|---|---|---|---|---|---|---|
| 1003 | 4/12/88 | UPS | FOB | Credit | $0.00 | |
| 1004 | 11/26/95 | DHL | FOB | Check | $7,885.00 | $ |
| 1005 | 4/20/88 | UPS | FOB | Visa | $4,807.00 | $7 |
| 1006 | 11/6/94 | Emery | FOB | Credit Visa | $31,987.00 | $4 |
| 1007 | 5/1/88 | US Mail | FOB | COD Check | $6,500.00 | $5 |
| 1008 | 5/3/88 | US Mail | Net 30 | Mastercard | $1,449.50 | $5 |
| 1009 | 5/11/88 | US Mail | Net 30 | COD | $5,587.00 | $5 |
| 1010 | 5/11/88 | UPS | Net 30 | COD | $4,996.00 | $8 |
| 1011 | 5/18/88 | UPS | Net 30 | COD | $2,679.85 | $8 |

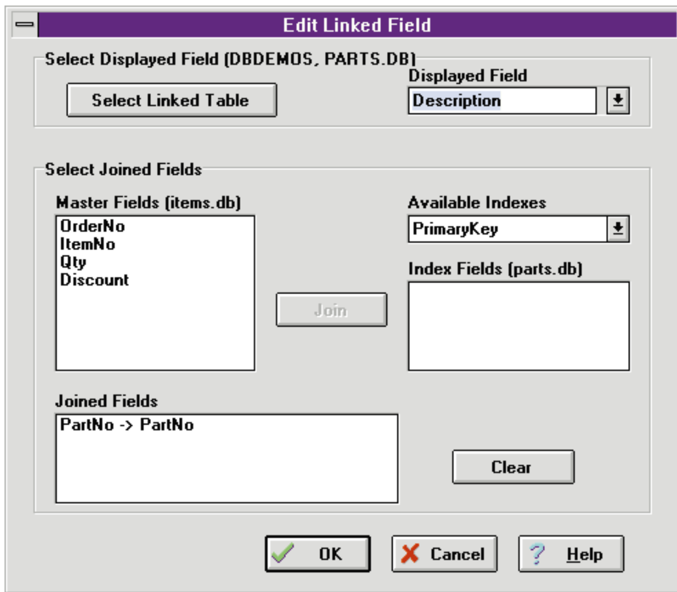
A *TwwDBGrid* component with customized titles and a combo box.

*TwwDBGrid* inherits from *TDBGrid*, so it works just like the built-in grid. To use it, you assign a *TwwDataSource* to the grid's *DataSource* property. In turn, the *TwwDataSource* component's *DataSet* property must refer to a *TwwTable*, *TwwQuery*, or *TwwQBE*. The grid has several properties and events that you program to alter its behavior and appearance.

## Enhanced CheckBoxes, Combo-boxes, etc.

InfoPower makes it convenient to display a field in the grid as a checkbox, combo-box, or lookup combo. Each field in a *TwwDBGrid* has a Control Type setting. You can access this setting from the Select Fields dialog box. This dialog box appears when you click on the grid's *Selected* property or the DataSet's *ControlType* property. The field's control type determines the appearance of its cell. This can be the default edit box, a checkbox, or one of InfoPower's several combo-box or lookup components. If what you want is an edit box or checkbox, you set the control type to *Field* or *CheckBox*, respectively, and you're finished.

To display a combo-box or lookup, you must first bind one of InfoPower's combo-box or lookup components to the field, then leave it on the form with its *Visible* property set to *False*. Next you use the Select Fields dialog box to set the field's *ControlType* property to *Combo* or *LookupCombo*, depending on the component type you want to use. After you do this, a second drop-down list appears showing the names of all the components that




Setting up a linked field.

can be attached to the cell. When you choose one, the grid will display it each time the cell has focus.

Two InfoPower components you can use with the grid are *TwwDBCombobox* and *TwwDBLookupCombo*. *TwwDBCombobox* replaces Delphi's built-in *TDBCombobox* component. Like the *TwwDBLookupCombo*, this control can be used inside the grid or by itself. Woll2Woll found keystrokes — such as Tab, Enter, End, and Shift Tab — didn't function as expected when using the regular *TDBCombobox* with the InfoPower grid. You should use this replacement component with *TwwDBGrid* in place of *TDBCombobox* whenever you need to use a combo-box with a fixed list of choices. Aside from this adjustment, nothing else is new about *TDBCombobox*.

*TwwDBLookupCombo* is an enhanced *TDBLookupCombo* that adds several new features. *TwwDBLookupCombo* lets you select any number of fields from the lookup table to display. In addition, you can control their display width and title descriptions. When the drop-down list is displayed, the user can incrementally search the list by typing into the edit box. It also lets you determine the alignment of the drop-down list relative to the component's edge. Finally, you do not have to fill in the *DataField* and *DataSource* properties to bind the field. The component can be used without editing an underlying table, so it is useful in situations other than in the grid.

## Creating Linked Fields

*TwwGrid* lets you display a linked field from another table by setting properties. To do this with the built-in grid, you would have to write code in the *OnCalcField* event to reference the appropriate value from a lookup table, then display it in a calculated field. InfoPower simplifies this by removing the need to write code.

To display a linked field, you use the Edit Linked Field dialog box of the *TwwTable* to select a display field. This dialog box lets you choose the table used in the one-to-one relationship, the names of the joined fields, the index used, and the field that is displayed. When you run the form, any linked fields are automatically shown as read-only fields.

## Controlling the Grid's Appearance

*TwwDBGrid* offers much more control over the appearance of the grid than Delphi's *TDBGrid*. The *FixedCols* property lets you define non-scrollable columns on the grid's left side. The *Ctrl3D* property gives the whole grid a 3-D look instead of only the titles as *TDBGrid* permits. The *TitleAlignment*, *TitleColor*, and *TitleLines* properties let you control the alignment, color, and number of lines occupied by the titles. When you use the latter property to create multi-line titles, you embed "~" symbols in your titles to force the text onto the next line.

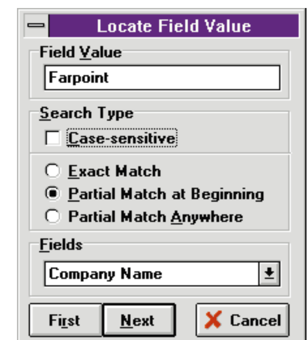
*TwwDBGrid* adds the *OnCalcCellColors* event to let you set the appearance of *individual* cells in the grid. You can use this event to highlight important fields or to convey status to the end-user. The following code sample turns the text color of the OnHand field to *clRed* if its value falls below 30 items:

```
procedure TForm1.OnCalcCellColors(Field:TField;
            State:TGridDrawState; Highlight:Boolean;
            AFont:TFont; ABrush:TBrush);
begin
  if Field.FieldName = 'OnHand' then
     if Field.Value < 30 then
        AFont.Color := clRed;
end;
```

## Searching, Sorting, and Locating

InfoPower contains two controls, *TwwKeyCombo* and *TwwIncrementalSearch*, that let you give the end-user the ability to "sort" a table and incrementally search for values. The *TwwKeyCombo* doesn't actually sort a table, but lets you change its display order. This component is a drop-down list that is populated with the names of all the indexes for a table. When a selection is made, the table is ordered by the fields in the index. This mimics the capability of Paradox for Windows' **Table | Filter | Order By** menu command.

The *TwwIncrementalSearch* component lets the user incrementally search for values in a table. As the user enters a value in the control's visible edit box, the selection is narrowed to the closest record. Internally, *TwwIncrementalSearch* searches the first field in the table's active index. This control is best used in conjunction with *TwwKeyCombo* and a data grid to allow the user to select the index and view the results of the search.




The Locate Field Value dialog box enables you to search the fields of a table for a specific topic.

InfoPower provides two components, *TwwSearchDialog* and *TwwLocateDialog*, that prompt the user when searching for a record.

*TwwSearchDialog* combines the features of *TwwIncrementalSearch* and *TwwKeyCombo* in a dialog box so that users can incrementally search for a record. The dialog box also contains up to two user-definable buttons that you can program responses to. *TwwLocateDialog* displays a dialog box that lets the user search for a record by entering a value and a field to match it against. The dialog box contains options to allow case-sensitive, and exact or partial matches. This component also contains two methods, *FindFirst* and *FindNext*, that let you perform a behind-the-scenes search without displaying the dialog box at all. Both *TwwSearchDialog* and *TwwLocateDialog* are used by calling their *Execute* methods.

## User Defined Combo-boxes

*TwwDBComboDlg* is one of the more versatile visible components in the InfoPower collection. This visible control is bound to a field and works very much like a combo-box, except that you control what happens when the user clicks the component's ellipsis (...) button. InfoPower ships with an example that uses a *TwwDBComboDlg* component with a date field. When you click on the ellipsis button, a calendar is displayed that lets you pick a date and fill in the field.

To program the ellipsis button, you add code to respond to the *TwwDBComboDlg* component's *OnCustomDlg* event. The following Object Pascal code fragment displays a calendar in response to this event:

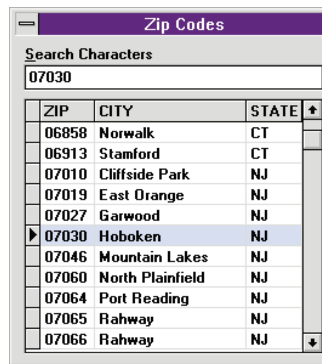
```
procedure TForm1.wwDBComboDlg1CustomDlg(Sender: TObject);
begin
  wwCalendarComboDlg(Sender as TwwDBComboDlg);
end;
```

## Additional Lookup Dialogs

To round off its collection of lookup components, InfoPower includes *TwwDBLookupComboDlg* and *TwwLookupDialog*. These two components offer functionality similar to *TwwDBLookupCombo* because they let you display choices from a lookup table. What's different is they let you present the lookup table as a searchable grid in a dialog box instead of a drop-down list. The dialog box can contain an index selection combo-box and up to two optional command buttons that you can program responses to when they are pressed. *TwwDBLookupComboDlg* is bound to a data field, just as *TwwDBLookupCombo* is. When the user clicks on the ellipsis button, the dialog box containing the grid appears.

This wwDBLookupComboDlg component displays a calendar.

*TwwDBLookupDialog* displays a similar dialog box, but is not bound to a data field. It's a non-visual component just as Delphi's common dialog components, so you control it using code. To display the dialog box, you call the component's *Execute* method. When the user is finished with the dialog box, you can query the component's *LookupTable* property to determine what record was selected. *TwwDBLookupDialog* is very flexible because you can control exactly when the lookup table is displayed and what to do with the selected value.




The wwDBLookupComboDlg component enables you to search for a record.

## Deployment, Documentation, and Sample Code

Registered users are given a royalty-free license to distribute executable applications they build using the InfoPower components. Since InfoPower is a VCL library, there is no need to distribute additional files with your application. If you buy the source code, you can modify it and/or derive your own components from InfoPower's basic set. You can then use these customized components in applications provided you do not distribute them outside your company.

Woll2Woll's documentation for InfoPower consists of 115 pages. This user's manual contains instructions on how to install the software, descriptions of each component, and useful examples of how to use each one. Since many of the InfoPower components are inherited from VCL components, it is important to understand where and why the newer component diverges from the pre-defined behavior of its ancestor. To this end, the section for each component lists any new, modified, or obsolete properties.

Also in the documentation is a tips and techniques section that describes how to achieve common functionality. This information is also available in the on-line help system. Since the on-line help is integrated into Delphi's IDE, the reference is always conveniently available. A final valuable reference is InfoPower's source code that is available as a separate purchase. If you are interested in learning about how InfoPower works, or how to make direct BDE calls, you should buy this option.

InfoPower ships with a series of small Delphi sample programs that illustrate the features of each component. Although installation of the sample programs is optional, I recommend running through each one to get a quick overview of InfoPower's capabilities. By studying each example, you can learn how to use InfoPower most effectively in your application. In fact, prospective customers can obtain a demonstration version, InfoDemo.ZIP, from CompuServe

**INFORMANT FACT FILE**

InfoPower is a collection of data-aware components that enhance Delphi's existing VCL components, including an enhanced data grid, database filtering, lookup combo-boxes, expanding memo dialogs, and incremental search components. The package has a well-written user manual and a comprehensive collection of sample code. Source code is available separately.

**Woll2Woll Software**
1032 Summerplace Drive
San Jose, CA 95122
**Phone:** (800) WOL2WOL, or (408) 293-9369

**Price:** Introductory price US$149 expires August 31, 1995; source code US$79; after August 31, US$199, with source code at US$99 (30-day money-back guarantee).

**Free technical support is provided via:**
**CompuServe:** 76207,2541
**Internet:** wol2wol@webcom.com
**Fax:** (408) 287-9374
**Voice:** (408) 293-9369

(Informant Forum, Library 14) or at Woll2Woll's World-Wide Web address (http://www.webcom.com/~wol2wol/). This demonstration version contains the same examples that ship with InfoPower.

## Conclusion

InfoPower provides capabilities essential to every professional database developer. The components are complete and well thought out, significantly enhancing the development process. The documentation, help system, and sample code serve as a very good tutorial in helping new users master the product.

Developers who want to build full-featured front-ends for their database applications should take a serious look at this product. The quality of the total package rivals or exceeds many of the more established same-niche VBX products that I've used to develop Visual Basic database applications. Woll2Woll plans for future versions include many grid enhancements, such as multi-selection rows, editable fixed columns and linked fields, and spin edit support. ∆

Joseph C. Fung is a principal of Farpoint Systems Corporation, a NY/NJ-based consulting firm specializing in developing traditional and client/server database applications with Delphi, ObjectPAL, and Visual Basic. He writes for *Paradox Informant,* and is the author of *Paradox for Windows Essential Power Programming.* Mr Fung is also the architect of ScriptView and AppExpert, two award-winning development tools for Paradox. Currently, Mr Fung chairs an advisory board for the Borland Developer Conference '95. He can be reached at (201) 656-6561, on CompuServe at 71121,2331, or via Internet at jfung@panix.com.

# TEXTFILE

## *Delphi Unleashed*: Worth the Wait

The first books to appear after the release of a new software product typically whet readers' appetites, but leave them wanting more. Only later do the major reference works begin to appear. From Sams Publishing, Charles Calvert's *Delphi Unleased* is in this second category, and well worth the wait. *Unleashed* is a large, satisfying "must read" for Delphi developers. It is well-written, detailed, and contains insights into Delphi that you will find nowhere else.

While *Unleased* has something for nearly everybody, Parts II and III of this book will be particularly welcome for those new to the Pascal language, or returning to it after a long absence. *Unleashed* spends more than a third of its 900 pages covering the basics of Object Pascal. These sections begin with a discussion of variables and types, continue into control structures, and end with a detailed look at pointers, *PChars*, and linked lists. The information contained here will be especially appreciated by anyone who has tried to read the practically indecipherable *Object Pascal Language Reference*.

Part IV of *Unleashed* contains a serviceable introduction to using Delphi for database applications. Here readers learn the relationship between DataSource and DataSet components, and how to use these to create forms that provide access to databases. Included in this section is a nice, but limited introduction to client/server topics. The remainder of *Unleashed* covers a wide variety of topics including OOP (object-oriented programming), component creation, exception handling, and a short primer on good programming techniques.

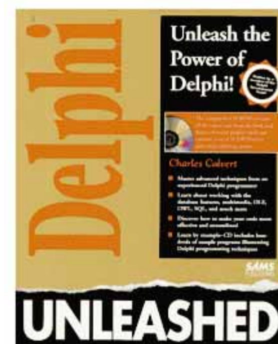
In general, there are many things to like about this book. Calvert does an excellent job of explaining the sometimes complex issues, making extensive use of analogies to assist the reader. Calvert also shows consideration for the more advanced reader, suggesting which parts of the book may be skipped, skimmed, or reviewed based on the reader's experience level.

The topics covered in *Unleashed* are as varied as the book is long. Programmers with experience with Borland Pascal will appreciate the comparisons to Delphi, including how their object models differ. Those who rely heavily on Windows API calls will learn how to easily include them in their Delphi applications. And those fairly new to programming will benefit from the detailed explanations of control structures and type declarations.

Another plus for *Unleashed* is its many code examples included on the CD-ROM accompanying the book.

In short, *Delphi Unleashed* is an important new resource for Delphi developers. It covers a wide variety of topics, and contains valuable insights, hints, and tips that make it a great addition to any programmer's library.

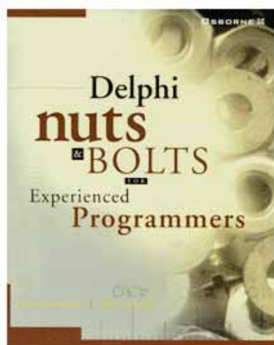
— *Cary Jensen, Ph.D.*




***Delphi Unleashed*** by Charles Calvert, Sams Publishing (Macmilian Publishing) 201 West 103rd Street, Indianapolis, IN 46290; (317) 581-3500

**ISBN:** 0-672-30449-6
**Price:** US$45.00
930 pages, CD-ROM

## Putting it Together with *Nuts & Bolts*




Experienced programmers looking for a quick jump start into Delphi will welcome *Delphi Nuts & Bolts for Experienced Programmers* by Gary Cornell and Troy Strain (Osborne McGraw-Hill). I looked forward to seeing this title, having often recommended Cornell's earlier books on Visual Basic. In *Nuts & Bolts*, Cornell, a mathematics professor, teams up with Strain, a former testing manager for Visual Basic (VB) at Microsoft. They assume the reader has a working knowledge of programming (in any language) that enables them to skip such topics as how to use loops and other common constructs. They still cover the syntax and the unexpected idiosyncrasies, but they don't waste pages dwelling on material programmers already know.

The book offers numerous short tips. Icons in the margins classify the tips as being of general interest, for VB, or Pascal programmers. Pointing out how Delphi differs from VB and Pascal is especially valuable for the many programmers coming to Delphi from one of those backgrounds. Unfortunately, I found the typographic presentation of the tips interrupted the flow of text. Each tip is preceded by a column-wide dotted rule and followed by a page-wide solid rule. The rules create too much visual separation between the main-body paragraphs that precede and follow the tip. After reading a tip, I consistently was surprised to return to the original subject rather than a whole new topic.

The book begins with the expected basics: the Delphi environment, form design, components, menus, etc. By concentrating on the big picture, on easily missed features of Delphi, and on unexpected "gotchas", the early chapters

*"Nuts & Bolts", continued on page 51*

## A Delphi Adventure in Three Easy Parts

*Delphi Programming EXplorer* by Jeff Duntemann, Jim Mischel, and Don Taylor is a great way to learn Delphi programming. At the same time, it's a lot of fun to read. Jeff Duntemann is an excellent author, unsurpassed at explaining programming subjects in an easy-to-follow manner. His editorials are the first thing I read each month when *PC Techniques* magazine arrives. Mischel and Taylor also write for *PC Techniques*. When I learned they had teamed up to write *EXplorer*, I ordered a copy, sight unseen. I wasn't disappointed — it's a winner.

*EXplorer* is written in three parts, and includes a diskette of source code and utilities. In Part 1, Duntemann and Mischel alternate writing chapters. They use a "hands-on" approach that produces results quickly. Chapters by Mischel guide you through the mechanics of using Delphi to create simple, interesting applications. In alternating chapters, Duntemann reviews the theory behind the code. Both authors use personal anecdotes to tie the code and discussions to the real world. After several chapters, you'll have enough experience and confidence to experiment with Delphi on your own.

Duntemann and Mischel continue their team approach in Part 2. They don't hold your hand as much, and the pace quickens. The theory and practice start to mix together, and there are some examples that are not part of complete projects. The subject matter is more advanced: objects, files, printing, graphics, and the beginnings of database programming. The examples include a file viewing utility, a mortgage calculator, a graphics toy like the "Spirograph", and several simple databases. This section has a few editing errors, and some printed listings have bugs that have been corrected on the diskette, but they are minor problems. By the end of the section you're ready to start working on significant projects.

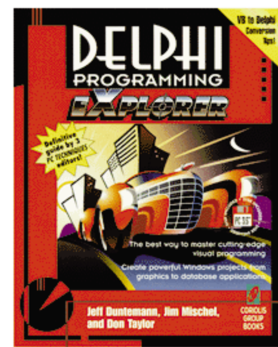
Part 3 of *EXplorer* is where the fun really starts. In "Ace Breakpoint's Database Adventure" Don Taylor presents a database design study. In the form of a hard-boiled detective novel! Ace is an ex-private investigator turned programmer, learning Delphi while trying to become sensitive and politically correct (a sort of Sam Spade for the 90s). As you follow Ace in his struggles to deliver a demo to an important customer, defeat his scornful competitor, and regain his lost love, you'll actually learn important steps in the Rapid Application Design (RAD) process. You'll write a sophisticated application using Delphi database components. You'll also learn how to add help to a project using the supplied HelpGen utility, and find out what happens in Ace's love life.

The book includes an appendix on Delphi for Visual Basic users. It contains a brief review of the differences between Delphi and Visual Basic, instructions for installing and using an evaluation version of EarthTrek's Delphi Conversion Assistant, and a short example of the conversion process.

With *Delphi Programming EXplorer*, Duntemann, Mischel, and Taylor have created a rare combination of valuable content and refreshing presentation. Palatable for experienced developers as well as newcomers — amusing without being frivolous — it's a great way to start your adventures in Delphi. Sign up for the trip. And be prepared for a good time!

— *Tim Feldman*

***Delphi Programming EXplorer*** by Jeff Duntemann, Jim Mischel, and Don Taylor, Coriolis Group Books, 7339 East Acoma Drive, Suite 7, Scottsdale, AZ 85260; (800) 410-0192, or (602) 483-0192.




**ISBN:** 1-883577-25-X
**Price:** US$39.99
627 pages, diskette

### "Nuts & Bolts" (cont. from page 50)

provide a wealth of information very quickly. The authors devote little space to examples, but refer readers to Delphi's sample programs. Similarly, they urge readers to consult Delphi's Help files for details of less important component properties.

The middle chapters cover basic and advanced aspects of the Delphi language. I was pleased by the emphasis on data structures, including arrays, string lists, records, pointers, etc. Objects are covered in a separate chapter, although perhaps more briefly than object-oriented programming (OOP) newcomers may wish.

The final portion of the book skims over a variety of topics. Most of the material is helpful, and provides a good starting point for personal exploration. However, these chapters seemed less complete than the earlier sections. It appeared as if the authors were working under a page-count limit or a severe time deadline. The too-brief chapters include discussions of error and exception handling (nine pages), testing and debugging (11), and working with files (18). Other short chapters deal with inter-application communication (Clipboard, DDE, and OLE), graphics, and "advanced user-interface features" (toolbars, status bars, common dialog boxes, MDI forms, drag-and-drop operations, and help systems). I suspect that many readers could benefit from more depth on these topics.

The biggest disappointment was the final chapter "A Survey of Database Features," that runs just eight pages. The authors acknowledge the brevity, but plead that it would take a book twice the size of *Nuts & Bolts* to explain "any substantial part" of Delphi's database programming power. Perhaps so, but such an important topic surely deserves more than eight pages.

If you already know how to program and want to learn what's new in Delphi, *Nuts & Bolts* will take you farther, faster than most other books. This is especially true if your background is in VB or Pascal. The quality of what's provided is excellent, but the brevity leaves you wishing for even more.

— *Larry Clark*

***Delphi Nuts & Bolts for Experienced Programmers*** by Gary Cornell and Troy Strain, Osborne McGraw-Hill, 2600 Tenth Street, Berkeley, CA 94710; (800) 227-0900.

**ISBN:** 0-07-882136-3
**Price:** US$24.95,
307 pages