



Cover Art By: Michael Tanamachi

OBJECT PASCAL THE LANGUAGE Comparisons and Reveries



ON THE COVER



8 Think Objects, Not Reuse — Richard Wagner
Code reuse is talked about a great deal, but seldom put into practice. Mr Wagner gets this month's issue off to a quick start by describing the contradictory disciplines of rapid application development and code reuse. He then goes on to resolve this dilemma through the use of inheritance and polymorphism. And manages to make it simple!

13 Power and Safety — John O'Connell
Using Delphi's powerful new Object Pascal language, we can create object-oriented programs that go head-to-head with anything created in C++. But Mr O'Connell wants us to slow down a bit and appreciate some robust language features that Pascal has had since its inception: subranges, enumerated types, and sets.

21 Cultural Differences — Richard Holmes
C++ and Object Pascal have a lot in common. For example, they're both OO hybrids of classic 3GL languages. And when it comes to raw power and capability, there is also little to separate them. Yet there are some important differences as well. Mr Holmes dissects the two languages and provides us with an extraordinarily detailed — and fascinating — comparison.

FEATURES



26 DBNavigator — Cary Jensen, Ph.D.
You've probably used the Table, DataSource, and Query components extensively (especially if you've followed this column). But what about the BatchMove component that shares the same Data Access page on the Component Palette? Chances are you haven't touched it — until now. In this month's DBNavigator, Dr Jensen shows us the many important tasks BatchMove can perform.



30 Visual Programming — Douglas Horn
Delphi's ObjectBrowser can provide unparalleled insight into the inner workings of Delphi and Windows. Unfortunately, its use is next-to-undocumented. This month, Mr Horn helps reveal the ObjectBrowser for the incredibly useful instrument that it is. It's a comprehensive introduction and visual tour of a tool you should become familiar with.



34 Informant Spotlight — Tom Costanza
The issues and protocols of serial communications are nearly as old as computers. In fact, in many respects, little has changed. And when Delphi is added to the mix, an interesting juxtaposition of old and new is created. From the middle of it all, Mr Costanza provides us with an outstanding introduction to serial communications and a Delphi implementation.

REVIEWS



42 RAD Pack for Delphi — Product review by Tim Feldman
Shortly after shipping Delphi, Borland issued an accompanying "RAD Pack" product. But just what is the RAD Pack? What does it contain? And what about the quality of those contents? It's not perfect, as Mr Feldman explains, but may contain some key tools for your Delphi development environment.

47 Delphi Developer's Guide

Book review by Tim Feldman

48 Mastering Delphi

Book review by Larry Clark

48 Delphi: A Developer's Guide

Book review by Richard Wagner

DEPARTMENTS

2 Editorial

4 Delphi Tools

6 Newline



Symposium

"I never think of the future. It comes soon enough."

— Albert Einstein

As we move too quickly into Camus' cool breeze from the future, I thought it might be a good idea to revisit some recurring themes surrounding *Delphi Informant* and its readers. With due respect to Dr Einstein, I'd like to keep us in sync. And that necessitates some planning, no matter how rudimentary.

So who reads *DI*? You come from diverse programming backgrounds — most notably Pascal, C/C++, Visual Basic, and Paradox for Windows. This letter is from a VB programmer:

Dear Editor;

One of the great things about Visual Basic is the attitude of the authors and programmers; There is none of the 'I'm smarter than the idiot that wrote...' letters that we see so often in the C++ magazines; When a VB author publishes stupid code, the other authors correct it with 'That's great, but maybe you should do it this way...'; I hope {and I am sure others will join me} that the Delphi community will continue with this supportive attitude; I thank your magazine for helping me make the transition from VB to Delphi; The only hard part has been to remember to end each statement with a damned semicolon;
Regards;
Dennis Pipes;

Yes, thankfully, I've yet to see the kind of acrimony that can become the "personality" of a programming language community. (A friend of mine once described a C++ forum he frequented as a "snake pit.") And there's that word — *community*. This may not be the first time I've seen the phrase "Delphi com-

munity," but it's the first time I've been struck by it. To observe that our profession changes quickly is passé, itself an indication of just how quickly things are moving. Still, it's remarkable that there is already a genuine community for this young product. The Borland and Informant CompuServe forums (GO DELPHI and GO ICGFORUM respectively) are evidence. And I heartily agree, Dennis — let's keep it friendly. Then there's your artful punctuation — too clever by half Mr Pipes.

Having shared the "snake pit" comment, I hasten to add that those from the C++ community I've run across have been perfectly civil, thank you. Take for example, this letter from down under:

Dear Mr Coffey:

... Do we need more articles that address Object Pascal basics? Probably; I'd appreciate something above the 'elementary' level that fills the big gap Borland left in explaining their component hierarchy. That may be in their *Component Writer's Guide*, but I haven't been able to get my hands on a copy yet!!
Do we want fewer database related articles? YES! although a couple on interfacing Delphi to dBASE and C++ would be welcome. ...

I'm not coming over from VB — never went there. I am using C++.
Make sure that as far as possible, and appropriate, each article leaves us with real, self-contained, working examples. ...
This is the most expensive programming magazine I subscribe to. I hope I resubscribe. That will depend on how I perceive its value, i.e. how much I learn from it. I certainly hope Delphi is a real success and I hope your magazine is too.
Regards, Fred Browne

Thanks for the great letter Fred. Your good wishes are very much appreciated and I hope we merit your patronage.

Change and its pace are also ineluctable themes of this business. And with the development landscape changing so rapidly, it's vital that *DI* be responsive and continue to serve your interests. What I'm hearing from you to-date is "So far, so good," but keep the comments coming. This message from Mr Rice is representative:

Dear Jerry:

... in reply to your appeal for what we would like to see in the *Informant*, for my money, you're already right on track. ... For the rest of us who slog out commercial code

for a living, the salient points of your magazine come through loud and clear. Much more of a beginner's bent, and you would lose some of us. However, the same goes for the other (guru) extreme. Most of us who are using Delphi are new to the product (but not necessarily OOP) because it just hasn't been on the market long enough to produce many experts ...

Sincerely,
David L. Rice

Thanks for the words of encouragement David. We'll continue to carefully ply a course between the neophyte and guru extremes.

And here's a letter that made my week:

Jerry:

As a long time Paradox Informant subscriber and new DI subscriber, I've grown accustomed to the layout, style and content of the Informant magazines. One thing is clear with regard to the question of technical content -- if you can't use it or completely understand it today, give yourself some time in the Delphi environment then re-read the articles. I've found at first pass I might be interested in using (or bothering to understand) snippets from 25-50% of the content, but on second pass (after heading up the learning curve) I end up "stealing" a good 80% or more of the ideas presented.

In what amounts to a one-person programming shop, the Informant magazines and CompuServe forums have "saved me" many times over, either from going down a dead-end path or allowing me to develop that much faster. As well, I've taken the time to contact the authors (giving fair opportunity to ignore me if too busy) to discuss/trade thoughts -- all-in-all, a very helpful, understanding group. Thanks for a fine magazine --

consider my subscription an investment in your efforts!

Paul Jordan

Thank you Paul! You have succinctly described *Delphi Informant's* raison d'être. Surveys have shown us that most *Paradox Informant* readers keep their magazines indefinitely and use them as reference. Our goal is to achieve the same with *DI*.

And now to one of my favorite topics — books. This month's issue closes with three book reviews that I've been impatient to read, and to share with you. Unfortunately it takes months to get these reviews into your hands. I've been especially impatient since, as Tim Feldman mentions in his review of Pacheco and Teixeira's *Delphi Developer's Guide* [Sams, 1995], we're now well into the "second wave" of third-party Delphi guides; books that took longer to get to press, but are more comprehensive and offer deeper insights into the product. The two other books reviewed in this issue are Marco Cantù's *Mastering Delphi* [Sybex, 1995], and Todd and Kellen's *Delphi: A Developer's Guide* [M&T, 1995].

These are three of what I think are the five must-have Delphi books available as of this early October writing. The others are Neil Rubenking's *Delphi for Dummies* [IDG, 1995], and Charlie Calvert's *Delphi Unleashed* [Sams, 1995]. These are the texts I return to again and again for general reference and to resolve specific programming challenges.

I'd like to sign off with a new topic and some questions for you. Here's another portion of that letter from David Rice (dated August 25):

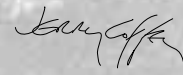
My shop has been utilizing Delphi Client/Server for some months now, and have, in the last three weeks, coupled it with the June Test Release

of Windows 95 and SQL 4.21. In the last week, we have finalized our installations with the production release of Win95 and SQL 6.0 running on a networked NT server. Our machines are Pentium/60s with 2MB of video RAM (DFI boards) and 16MB of on-board RAM. A little slow, but what the hell, right?

Nobody likes a braggart Mr Rice. <g> By the way, I'm writing this fabulous piece of editorial on a Pentium/100 I inherited from a guy in production. But I digress. The point is that this type of information is valuable to me; it lets me know where you are on the great software bell curve and helps me determine the article mix. It appears David is out toward the bleeding edge. He also shared his opinions regarding some previous letter writers, which, in the interest of keeping things friendly (see above), I'll keep to myself. However, I got a kick out of them David, and thank you very much for writing.

Which brings us to the aforementioned questions. With Delphi32 in the wings, where do you see yourself or your programming shop headed vis-à-vis Windows 95? Like Mr Rice, do you already have Windows 95 up and running? Or is it in your short-term plans? Or is it a year off? And most important, do you have clients clamoring for Windows 95 applications? Let me know what's going on in your part of the world and I'll share the results in this forum.

Until next time, thank you for reading.



Jerry Coffey, Editor-in-Chief

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



New Products
and Solutions



Delphi and Object Pascal Training in USA & Asia

The first Borland Training Center in the Connections Program, **GenoTechs, Inc.**, is providing Delphi and Object Pascal training internationally (i.e. Singapore, Indonesia, Malaysia, Thailand, India, and Hong Kong).

Developers can first learn the tool, OOP, and Pascal concepts in the Delphi course, then proceed to more advanced language methodology in the Object Pascal course.

The training is hands-on. On-site training is available, or users can attend a monthly five-day course in Phoenix, AZ. The Delphi Developer and Object Pascal in Delphi courses each start at US\$1340 (in Phoenix), and US\$1440 (on-site). Discounts apply for consecutive courses. For information or to register, contact GenoTechs, Inc. at (800) GENO-TEX; (602) 438-8647; or e-mail at 75374.2565@compuserve.com.

New VB Translator for Delphi

Eagle Research of San Francisco, CA is shipping **VB2D**, its new Visual Basic (VB) to Delphi translator. VB2D converts 90 to 100 percent of a VB software application to Delphi.

VB2D handles variants, control, form and re-dimensionable arrays, precedence adjustments, gotos, gosubs, type-1 VBXes, and virtually all intrinsic VB functions. VB2D Standard Edition includes a diagnostic report listing all project files, inserted typecases, precedence adjustments, renamed variables and procedures, substituted functions and classes, and detailed diagnostic messages cross-referenced to source code line numbers.

VB2D is designed for developers managing the translation and validation of medium to large systems. It creates a side-by-side listing of VB programs with the resulting Delphi code. The

VB2D Professional Edition also creates detailed analyses of VB and Delphi projects, including cross-references for variables, procedures, functions, and object properties and methods. The source code for all added program components is included with the Professional Edition.

Price: VB2D Professional Edition, US\$450; VB2D Standard Edition,

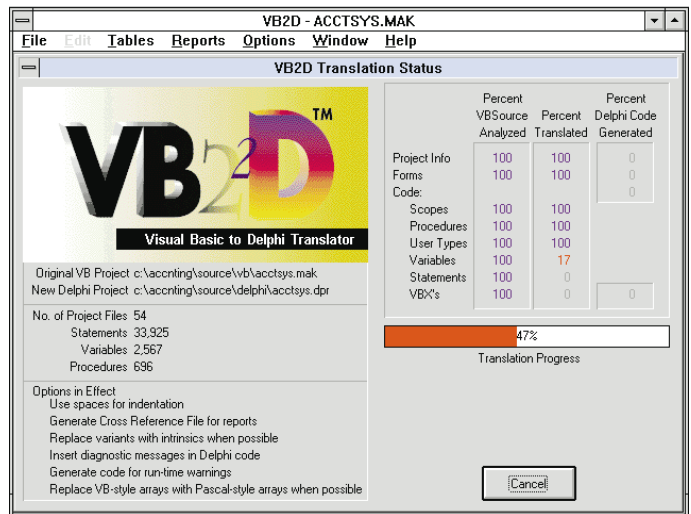
US\$150. Both versions include an unconditional 30-day, money-back guarantee.

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

Phone: (415) 495-3136

Fax: (415) 495-3638

E-Mail: Internet: vb2d@eri.uucp.net-com.com



ReportPrinter Version 1.1 for Delphi Released

Nevrona Designs of Chandler, AZ has released **ReportPrinter version 1.1**, a suite of native Delphi components that allow programmers to create reports compiled into applications without requiring

extra files, such as DLLs or VBXes. Therefore a separate installation program isn't necessary, and only 25 to 50K is added to the executable's size.

ReportPrinter features memo field or text stream printing with automatic word wrapping, justified text, boxes around text for table style listings, shaded fields or lines, and snaking columns. It also supports custom paper sizes, graphics, scaling, precise page positioning for pre-printed forms, direct printer output, printing to file, as well as standard or metric measurements.

ReportPrinter offers print preview with zooming, panning, and print-after-preview. It can be used without a database, or is compatible with a database

accessed from within Delphi. Using ReportPrinter's class library, professional-looking reports are created in Delphi with minimal coding.

Nevrona Designs will soon release an enhanced version of ReportPrinter with a visual interface and added functionality.

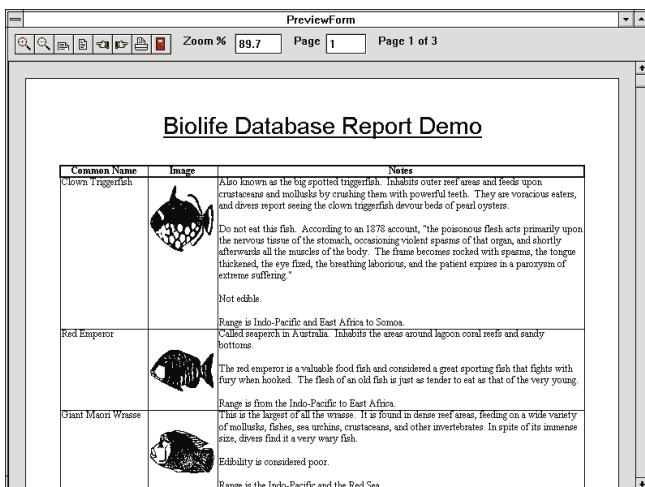
Price: US\$99 (includes complete source and printed documentation).

Contact: Nevrona Designs, 2581 E. Commonwealth Circle, Chandler, AZ 85225-6019

Phone: (602) 899-0794

Fax: (602) 530-4823

E-Mail: CIS: 70711,2020 or Internet: jgunkel@primenet.com



Delphi TOOLS

New Products
and Solutions



Starfish Announces Dashboard 95

Starfish Software has announced **Dashboard 95**, an upgraded 32-bit version of the popular utility. Dashboard provides a graphical front-end that operates consistently across all three primary Windows platforms: Windows 3.1, Windows 95, and Windows NT. It features a customizable selection of interactive controls, system activities gauges, and optional accessories.

Tabbed Quick Launch, Resource Gauges, AppOrganizer, Shortcut menus, Tool Tips, and a Panel manager are new features in Dashboard 95. Those retained from the earlier version include one-step drag-and-drop printing or faxing, and the ability to switch between full screen views of open applications.

Dashboard 95 is available for download from the Internet Shopping Network, ZiffNet, CompuServe, and software.net, and from retailers nationwide. Selling for US\$49.95, an upgraded version is available for US\$39.95. To order, call Starfish at (800) 765-7839.

SuccessWare's Apollo Pro Adds SDM to Apollo

SuccessWare International of Temecula, CA has released **Apollo Pro 1.0**, which adds a Source-code Documentation Module (SDM) and a customized version of Nevrona's ReportPrinter to the standard version of Apollo. Following Borland's lead with their RAD Pack product, the Apollo SDM is a VCL source file that can't be compiled; it is supplied for educational purposes only.

Apollo's Replaceable Database Engine (RDE) technology allows for record-based (Xbase) data-table navigation and management syntax during program development, with minimal concern for the database format. A "no-code" replacement for the BDE system, Apollo can be installed into an existing application with no source code changes. This allows immediate, multi-user access of Xbase files from legacy applications. Also, all concurrent-access record and

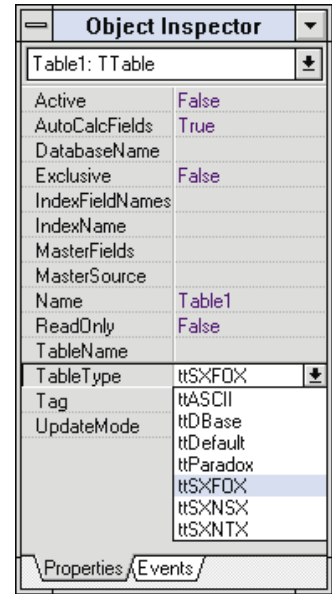
file locking is compatible with existing FoxPro and Clipper applications. Apollo supports CA-Clipper (NTX), FoxPro 2.x (IDX/CDX), and HiPer-SIx (NSX) systems.

Apollo's new features include Conditional Indexes, Index SCOPES, and Record-level Data Encryption. Also integrated into the RDE is MachSIx, SuccessWare's integrated query-optimizer that increases database retrieval performance.

Apollo is royalty-free and includes 30 days of free voice, BBS, Fax, or CompuServe technical support. A demonstration version of Apollo is available for immediate shipping.

Price: Apollo Pro, US\$239. Registered Apollo users can purchase SDM directly from SuccessWare for US\$99. There is a 30-day, money-back guarantee.

Contact: SuccessWare, 27349 Jefferson Avenue, Suite 101, Temecula, CA 92590



Phone: (800) 683-1657 or (909) 699-9657

Fax: (909) 695-5679

BBS: (909) 694-6891

E-Mail: Team SuccessWare 74774,2240

CIS Forum: GO SWARE

New Widgets Collection from Mobius

Mobius Ltd., of Hershey, PA has released its **Widget Collection**. This kit of components for Delphi includes TmoSticky, TmoShapedButton, TmoPicturePreviewDialog, TmoCards, TmoToolbox + TmoTool + TmoDockingPanel, and TmoTiler.

The TmoSticky component aligns all types of controls to fill the spaces on a re-sizable

form. The TmoShapedButton component provides buttons that aren't square, flat, or gray. With this feature developers can add color, rounded edges, or make buttons ellipsoid. TmoShapedButton also makes multiple-line captions easy to create.

Using TmoToolbox + TmoTool + TmoDockingPanel, tools can be placed in toolboxes and maintain "distant" radio button functionality. Simply drag the toolbox and drop it in space; the toolbox will float in its own window. The toolbars even re-size if the toolbox doesn't fit.

With TmoPicturePreviewDlg, developers can search for images in any storage medium. This component can be

used as-is or you can register a property editor and get the Mobius previewing dialog box.

The TmoCards component creates a card deck in the .DFM file, supports multiple card fronts and backs, and features several back designs. Mobius' Widget Collection ships with VCL source code and a hypertext help file.

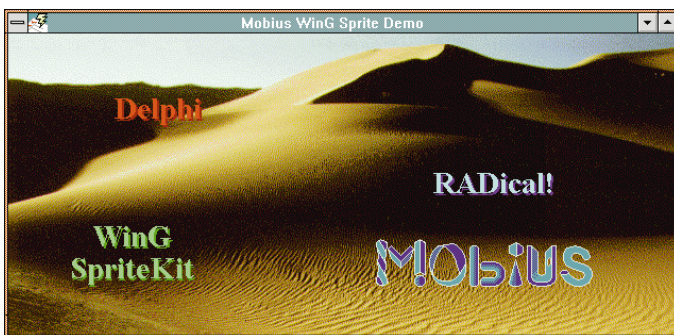
Price: US\$149

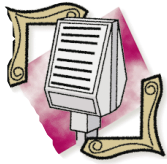
Contact: Mobius Ltd., P.O. Box 404, Hershey, PA 17033

Phone: (717) 944-8265

Fax: (717) 944-8265

E-Mail: CIS: 73563,533





Informant Moves from BBS to CompuServe

Elk Grove, CA — Beginning Dec. 1, 1995, Informant Communications Group, Inc. (ICG) will close their BBS and move its contents to the Informant CompuServe forum.

Operational since June, ICG created the Informant CompuServe forum to foster the exchange of technical information among developers who use Borland's Paradox and Delphi, as well as Oracle. This forum currently features code contained in past issues of *Delphi Informant* and *Paradox Informant*, in addition to shareware, company news, and much more.

The Informant CompuServe forum can be accessed by typing "GO ICGFORUM" at any CompuServe GO prompt. To join CompuServe and obtain a starter kit, including a US\$15 usage credit, call toll-free in the United States (800) 524-3388, and ask for REP Number 547.

Updates Available for Delphi and Delphi Client/Server

Scotts Valley, CA — Borland has released updates to their Delphi and Delphi Client/Server products.

For Delphi, this upgrade includes the latest versions of the Borland Database Engine, Local InterBase Server, and

ReportSmith, as well as the most recent technical information bulletins and frequently asked questions from Delphi Tech Support. It also features improved context-sensitive help in Delphi, Adobe Acrobat versions of the Delphi Language

Reference and VCL Reference, and maximum compatibility with Visual dBASE 5.5.

The Delphi Client/Server update adds support for Informix 5.x, Oracle synonyms, and the Sybase forced index feature.

To order, call Borland at 1-800-453-3375, extension 1327. An update CD is priced at US\$5.95, and an update disk set is priced at US\$19.95 (plus US\$5 shipping and handling for either update).

InterBase Workgroup Server for Unix Ships

Scotts Valley, CA — Borland International Inc. is now shipping the InterBase 4.0 Workgroup Server for Solaris, SunOS, HP-UX, AIX, SCO, and AT&T Unix platforms. InterBase 4.0 is designed for enterprise and workgroup computing environments and is available on Windows 3.1, Windows NT, Windows 95, and all popular Unix platforms.

InterBase offers superior performance for mission-critical operations including stock trading, pharmaceuticals, aerospace, and network management, while adhering to industry standards such as SQL 92 and ODBC.

The InterBase architecture offers a multi-client and multi-threaded server for speed and optimal use of resources. Its versioning engine ensures data availability for concurrent

transaction processing and decision-support users. InterBase provides lock-free transactions that require no additional programming, while providing a result for every query.

InterBase also features multi-dimensional arrays, two-phase commit and distributed recovery, stored procedures, event alerters, triggers, and BLOB filters. It supports ODBC, ANSI SQL 92 and UNICODE character sets.

Borland's Delphi, priced at US\$495, has a single-user copy of InterBase. Delphi Client/Server also includes a local Windows version of InterBase, along with the rights to deploy local InterBase applications, and costs US\$1,995. For more information call Borland at (408) 431-1000 or visit their Web Site at <http://www.borland.com>.

US Army to Use InterBase

Scotts Valley, CA — Borland International Inc. has announced that InterBase has been chosen by the US Army for its Advanced Field Artillery Tactical Data System (AFATDS). Magnavox Electronic Systems Company, the prime contractor, recommended InterBase because it's platform-independent, and includes unique, advanced distributed features.

"Operations on today's air-land battlefields are swift, intense and highly lethal.

The commander receives information from many sources over his extended battlefield. He then faces the complex problem of synchronizing his forces based on up-to-the minute information from remote field locations," explained John Williams, director of the AFATDS program at Magnavox. "Decision support of this nature requires a modular and flexible archi-

"US Army to Use InterBase"
continued on page 7

Supreme Court to Review Lotus vs. Borland Ruling

Washington, DC — The US Supreme Court has agreed to review an appellate court's ruling that Quattro and Quattro Pro spreadsheet products, formerly developed and marketed by Borland, did not infringe on the copyright of Lotus 1-2-3.

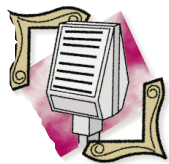
Recently acquired by IBM, Lotus appealed to the US Supreme court just months after the US Court of Appeals for the First Circuit reversed an earlier ruling in favor of Lotus. In their appeal to the Supreme Court, Lotus argued that Congress intended to treat computer programs as copyrightable literary works according to a copyright law enacted in 1976.

Lotus' decision to appeal was no surprise to Borland. "We are confident in the appellate decision," said Borland spokesman Steve Grady. "This will allow us to get a definitive answer and remove all questions surrounding the issue."

The first ruling in the Lotus vs. Borland case was announced in August 1992. At that time, the District Court ruled the

"Supreme Court"
continued on page 7

November 1995



Upcoming Client/Server Seminars

Andover, MA — The Client/Server Project Management Seminar comes to San Francisco, CA on Nov. 28-29, 1995. Seminar instructor Peter M. Storer, vice president of Client/Server Consulting for Atre Associates, will cover topics such as: What's Different About Managing Client/Server Projects; Common Pitfalls and Mistakes and How to Avoid Them; and The Breadth of Knowledge Required of the Client/Server Project Manager.

For more information, call DCI at (508) 470-3880 or visit their Web Site at <http://www.DCIexpo.com/>.

Objects and Relations Seminar

Author, columnist, and industry expert Chris Date will present the Objects and Relations seminar Nov. 7 - 9, 1995 in Palo Alto, CA.

This seminar reviews OODB systems and relational technology, but attendees are not required to have prior knowledge of object-oriented technology. For more information, call DCI at (508) 470-3880 or visit their Web Site at <http://www.DCIexpo.com/>.

DB/EXPO '95 Heads to New York

New York, NY — Offering three concurrent conferences, DB/EXPO '95 is slated for Dec. 5-7, 1995 at Javits Convention Center in New York, NY. With over 25,000 IT professionals and 150 vendors scheduled to attend, this year's DB/EXPO features the Database and Client/Server Development Conference, Data Warehousing and Parallel Computing Conference, and the DB/EXPO Executive Conference.

Participants in the Database and Client/Server Development Conference will discuss client/server issues such as building enterprise client/server applications, integrating client/server and legacy applications, and evaluating database servers and application development tools.

Supreme Court (cont.)

Command Hierarchy in Borland's spreadsheet products infringed the copyright of Lotus 1-2-3 and Borland voluntarily removed this feature. The court reaffirmed its decision in July 1993.

Then later in 1993, the Federal District Court ruled another compatibility feature in Quattro Pro and Quattro Pro for Windows infringed the copyright of Lotus 1-2-3. The court placed an injunction against Borland, barring further sales or distribution of the products. Borland shipped a new version of Quattro Pro without the infringing feature. In addition, Borland appealed the decision to the US Court of Appeals, and in their written opinions, all three appellate judges ruled in favor of Borland.

Topics for those attending the Data Warehousing and Parallel Computing Conference will include building a business case for a data warehousing project, selecting the right end-user tools, and listening to customer experiences with data warehousing.

The Executive Conference will examine IT trends including structuring an IT organization; business re-engineering; and maximizing the business benefits of data warehousing.

Keynote addresses will be presented by Colin White,

president of DataBase Associates International and DB/EXPO conference director; Don Haderie, IBM Fellow and director of Data Management Architecture and Technology, IBM Corporation; Dr Jerry Held, senior vice president of Server Technologies Division, Oracle Corporation; Dennis McEvoy, vice president of Products Group, Sybase, Inc.; and Mike Saranga, senior vice president of Management and Development, Informix Software.

For more information, call 1-800-2DB-EXPO.

Database and Client/Server World Nears

Andover, MA — DCI has announced the Database and Client/Server World Conference and Exposition, scheduled for Dec. 5-7, 1995 in Chicago, IL. This event will feature several concurrent conferences that cover client/server issues, data warehousing and repositories, parallel databases, middleware, object-oriented technologies, and groupware application development.

Keynote speakers scheduled to appear at the Database and Client/Server World include: Shaku Atre, president of Atre, Inc.; Dave Duffield, CEO and president of PeopleSoft, Inc.; Dr E.F. Codd, independent consultant; and Dennis McEvoy,

vice president of Products Group, Sybase, Inc.

Topics at this conference will address moving to finance, manufacturing, and human relations client/server applications; developing a practical framework for IT and end-user teaming on client/server application planning and implementation; and evaluating vendors and selecting software.

Database and Client/Server World is expected to attract over 25,000 MIS professionals and more than 800 exhibits. For more information, call DCI at (508) 470-3880 or visit their Web Site at <http://www.DCIexpo.com/>.

US Army to Use InterBase (cont.)

ture that would support both distributed processing and distributed databases."

AFATDS provides a singular fire support command

and control solution. It features commander's guidance, mission planning guidance, detailed asset control and status, and movement control.





ON THE COVER

DELPHI / OBJECT PASCAL



By *Richard Wagner*

Think Objects, Not Reuse

Using OOP Fundamentals to Achieve Code Reuse

Code reuse has always been a worthy objective for application developers. How many developers, however, have actually been successful in realizing this goal? Depending on the development environment you have used in the past, most forms of reuse typically center on code libraries, templates, and application frameworks. While Delphi supports these, it also provides native support for something much more powerful: object-orientation.

In this article, we will explore the notion that substantive code reuse is attainable only by introducing object-oriented programming (OOP) concepts into your application development process. In short, if you think in terms of objects, reuse will follow. And in contrast, if you try to achieve reuse using conventional approaches, you will ultimately fall short. Because most developers work in a highly-competitive market, an object-oriented strategy can make the difference between success and failure.

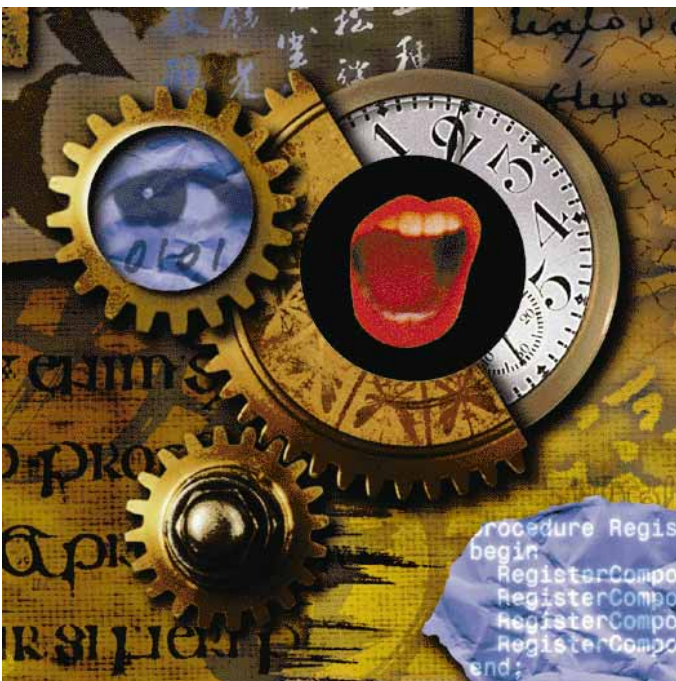
OOP in the Business World

Object-oriented programming has been around for a long time and is implemented in many languages. However, unless you have worked with C++ or Object Pascal, you may have never encountered a true OOP environment before working with Delphi. This is particularly true if you come from a client/server database application development background.

Tools such as PowerBuilder, Paradox for Windows, and Visual Basic tout themselves as being object-oriented, when in reality they are “object-based”. You may be able to work with objects in these environments, but support for such principles as abstraction and dynamic binding is altogether lacking.

I suppose one of the problems with object-oriented programming (OOP) is that it sounds so academic. Think of OOP’s principle concepts: *abstraction*, *inheritance*, *encapsulation*, and *polymorphism*. When first mentioning these to my wife, she remarked: “Have you joined a cult?”

On first take, such terms sound foreign and useless in a business world where *rapid application development* is the buzzword of choice. As you will see, however, these OOP principles can



be used in a practical manner to revolutionize the way you develop applications in Delphi.

The Reuse Battle

While nearly everyone agrees on the principle of reuse, the priority given to it by developers varies wildly. And this isn't the fault of the developer alone. After all, what developers usually hear from their managers, users, or clients is: "We want it now!" In this context, the developer is met with a dilemma: Do I develop a program for this specific problem or create a generic solution in twice the amount of time? The "tyranny of the urgent" typically comes into play and a developer will — unless provided with a sufficient incentive for reuse — develop a solution that is perhaps great for a specific need at a specific time, but useless for future needs.

Even the term *rapid application development* (RAD) seems inherently contradictory to the ideal of generic reusable code. While that is true in the short-term, a well thought-out code reuse strategy will facilitate RAD after an object component library has reached a certain degree of maturity.

Reuse Out of the Box

Reuse is certainly promoted in Delphi right out of the box — through components, VBXes, and templates. Prudent use of these resources will take you a long way to the "promised land" of reuse. Project templates serve as a good starting point for new applications, providing an application framework that you can then customize. Form templates offer a means of enforcing standards (font, button placement, and so on), but be careful of extending their functionality much beyond that.

Much of the reuse aspect of Delphi focuses on its component architecture. And for good reason — components are the "black box" objects you're striving for. However, don't make the mistake of thinking solely in terms of visual component libraries (VCLs). Components are a key element in a reuse strategy, but you can use VCLs in a non-object-oriented manner and, in so doing, miss much of what they can offer. As a result, you must first have a successful object development strategy in place *before* creating these components.

Think Objects

Inheritance and *polymorphism* are two basic OOP concepts that are fundamental to discussing code reuse in Delphi. First, inheritance defines the relationship between object classes. For example, a base class called *TCola* can have subclasses based on it called *TClassicCoke* and *TPepsi*. Both *TClassicCoke* and *TPepsi* inherit all the properties and methods of its ancestor (*TCola*), such as the *Caffeine* property or the *Drink* method. However, you can add new data or behavior to the descendant class and can even override methods of the ancestor at run-time.

To illustrate the importance of inheritance in your applications, let's think about a typical code reuse scenario. Suppose you have developed a nifty *CreatePrivTable* procedure and would like to

reuse it in future database applications. When you begin a new project (Project #2), you can then reference the library that the procedure is contained in and use it in the new application.

During the development of Project #2, however, you realize you must have specific table naming conventions that are not provided for in the original code. Therefore, you must modify the procedure to meet the needs of this second project. The dilemma becomes: Do you 1) modify the tried and true *CreatePrivTable* procedure to account for these new changes? or 2) copy the original code into a new procedure to avoid tampering with bug-free code?

This quandary demonstrates where the goal of reuse eludes us in traditional procedural languages. There is no architecture in place that can easily deal with change. Even small modifications to *CreatePrivTable* can have widespread implications throughout one or more applications.

The Inheritance Solution

Using object-oriented techniques in Delphi, you avoid this dilemma. Instead, you can convert the original procedure into a *TPrivTable* class. You could make it into an object because it has both properties (e.g. *TableName*, *TableType*, and *Structure*) and methods (e.g. *Execute*).

When you begin Project #2, you can create a descendant class called *TPrivTableCustomNames* and modify its properties and behavior to meet its specific needs. From a reuse standpoint, this has two advantages. First, in creating a subclass you avoid touching the bulletproof code of *TPrivTable*, yet at the same time, you have full access to this code. Second, you can maintain a link between the ancestor and descendant class. Let's say in six months you modify the *TPrivTable* class to take advantage of Windows 95 long filenames. *TPrivTableCustomNames* will be automatically updated as well.

Let's look at a second example of inheritance by creating standardized **OK**, **Cancel**, and **Help** buttons to use instead of constantly customizing *TButton* objects each time you need them. You have custom properties for each of them (e.g. *Caption*), and you want to make **OK**, **Cancel**, and **Help** the same width and height as standard Windows buttons. (*TButton* buttons are too large by default.)

Therefore, you can create a derivative of the *TButton* class called *TStandardButton* that overrides the size of the base class:

```
{ Within the interface section }
type
  TStandardButton = class(TButton)
  public
    constructor Create(AOwner: TComponent); override;
  end;

{ Within the implementation section }
constructor TStandardButton.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Height := 23;
  Width := 95;
end;
```

Next, you must create three derivatives under *TStandardButton* for the **OK**, **Cancel**, and **Help** buttons. This is accomplished with the code shown in [Figure 1](#). The entire .PAS file is shown in [Listing One](#) on page 11.

```
{ Within the interface section }
type
  TStandardOKButton = class(TStandardButton)
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
  end;

type
  TStandardCancelButton = class(TStandardButton)
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
  end;

type
  TStandardHelpButton = class(TStandardButton)
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
  end;

{ Within the implementation section }
constructor TStandardOKButton.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Caption      := 'OK';
  Default      := True;
  ModalResult  := mrOK;
end;

constructor TStandardCancelButton.Create(AOwner:
  TComponent);
begin
  inherited Create(AOwner);
  Cancel       := True;
  Caption      := 'Cancel';
  ModalResult  := mrCancel;
end;

constructor TStandardHelpButton.Create(AOwner:
  TComponent);
begin
  inherited Create(AOwner);
  Caption := 'Help';
end;
```

Figure 1: Creating three derivatives of *TStandardButton* for the **OK**, **Cancel**, and **Help** buttons.

The hierarchy of these buttons is shown in [Figure 2](#). Because of the relationship between *TStandardButton* and its descendants, if you decide to change the size of the three buttons in the future, you only need to make a single change to the *TStandardButton* class.

The Polymorphism Solution

The second OOP concept that is loosely integrated with reuse is *polymorphism*. You can think of polymorphism as a means by which objects can communicate freely with each other.

The importance of polymorphism is perhaps best explained with an example. Let's imagine three vehicles — auto, bicycle,



Figure 2: The *TButton* object hierarchy.

and plane — are available to take Bill from Los Angeles to Boston. When he arrives in LA, Bill wants to be able to tell any of the vehicles, “Go to Boston,” and have it take him across the country. Without an object hierarchy, these vehicles are independent creatures. That is, each has its own set of properties, methods, and events to respond to. In such an environment, how can Bill know the correct command to tell the vehicle? Maybe the directive for the plane is *FlyToBoston*; for the car, *DriveToBoston*; and for the bicycle, *RideToBoston*. Bill has no way of knowing for certain. To further complicate matters, suppose he encounters a new vehicle he's never seen before (like a spaceship) and is unsure of what it is. Since this object has no “common ground” with other vehicles he has previously used, Bill is unable to tell it what to do.

Without an object hierarchy, Bill is forced to know everything about each vehicle to ensure that it will get him to Boston. In addition, Bill needs to know about each possible vehicle *in advance* so he will know what command to tell it when he's ready for the cross-country trek.

Now, let's suppose Bill is in an object-oriented environment and knows that each of these belong to a base class named *TVehicle* and that all *TVehicle* objects understand a *GoToBoston* message. (In reality, we would want to separate the *GoTo* from the *Boston* since the *GoTo* is the behavior and *Boston* is a variable. But for our purposes, let's limit the *GoTo* method to a single location — *Boston*.)

Communication is simplified from Bill to all *TVehicle* objects and its descendants because he can issue a general message (*GoToBoston*) to whichever vehicle he decides to use. The vehicle is responsible for figuring out what to do with it. In fact, he doesn't even need to know the type of vehicle it is. This is polymorphism in action. In Object Pascal code, our example may resemble the code shown in [Listing Two](#).

Conclusion

By incorporating object-oriented principles into our applications, we can realize tangible benefits of reuse. (Remember that OOP has many advantages beyond reuse alone.) Inheritance enables us to account for change by maintaining a linkage

between a base class and its descendants. It also provides a “fire-wall” for a class, protecting it from being sullied when we work with a descendant class. Polymorphism allows us to create independent objects with general interfaces, making communication between objects easier to code and maintain. As a result, our objects become much more flexible and manageable.

If you intend to get serious about code reuse in Delphi, you will spend considerable time learning component development. However, make sure to approach components using an object-oriented strategy. And remember — think objects, not reuse. ▲

The demonstration source referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM95\NOV\RW9511.

Richard Wagner is a technical architect for IT Solutions in Boston, MA. He is author of several Paradox, Windows, and CompuServe/Internet books and is also a member of Team Borland on CompuServe. Richard can be reached on CompuServe at 71333,2031 or via Internet at richard_wagner@its.com.

Begin Listing One — TStandardButton .PAS File

```
unit Standbtn;

interface

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

type
  TStandardButton = class(TButton)
  public
    constructor Create(AOwner: TComponent); override;
  end;

type
  TStandardOKButton = class(TStandardButton)
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
  end;

type
  TStandardCancelButton = class(TStandardButton)
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
  end;

type
  TStandardHelpButton = class(TStandardButton)
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
  end;

procedure Register;

implementation
```

```
constructor TStandardButton.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Height := 23;
  Width := 95;
end;

constructor TStandardOKButton.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Caption := 'OK';
  Default := True;
  ModalResult := mrOK;
end;

constructor TStandardCancelButton.Create(
  AOwner: TComponent);
begin
  inherited Create(AOwner);
  Cancel := True;
  Caption := 'Cancel';
  ModalResult := mrCancel;
end;

constructor TStandardHelpButton.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Caption := 'Help';
end;

procedure Register;
begin
  RegisterComponents('Buttons', [TStandardButton]);
  RegisterComponents('Buttons', [TStandardOKButton]);
  RegisterComponents('Buttons', [TStandardCancelButton]);
  RegisterComponents('Buttons', [TStandardHelpButton]);
end;
end.
```

End Listing One

Begin Listing Two — Polymorphism in Action

```

unit Polymrph;

interface

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

type
TForm1 = class(TForm)
  Button1: TButton;
  Edit1: TEdit;
  procedure Button1Click(Sender: TObject);
end;

type
TVehicle = class(TObject)
  procedure GoToBoston; virtual;
end;

type
TPlane = class(TVehicle)
  procedure GoToBoston;
end;

type
TCar = class(TVehicle)
  procedure GoToBoston;
end;

type
TBicycle = class(TVehicle)
  procedure GoToBoston;
end;

var
  Form1: TForm1;

implementation

{ $R *.DFM }
procedure TVehicle.GoToBoston;
begin
end;

procedure TPlane.GoToBoston;
begin
  MessageDlg('Fly the skies.',mtInformation,[mbOK],0);
end;

procedure TCar.GoToBoston;
begin
  MessageDlg('Drive the Interstate.',
            mtInformation,[mbOK],0);
end;

procedure TBicycle.GoToBoston;
begin
  MessageDlg('Ride on backroads.',mtInformation,[mbOK],0);
end;

procedure TBill.Button1Click(Sender: TObject);
var
  Plane: TPlane;
  Car: TCar;
  Bicycle: TBicycle;
  AvailDays: Integer;
begin
  AvailDays := StrToInt(Edit1.Text);
  case AvailDays of
    1..3 : Plane.GotoBoston;
    4..6 : Car.GoToBoston;
    else  Bicycle.GotoBoston;
  end;
end;

end.

```

End Listing Two



ON THE COVER

DELPHI / OBJECT PASCAL



By *John O'Connell*

Power and Safety

Pascal Subranges, Enumerated Types, and Sets

Delphi's implementation of Pascal is a greatly enhanced dialect of standard Pascal. Pascal's roots lie in a language originally designed to help teach basic concepts of structured programming and top-down design. Indeed, Pascal was the language of choice in most academic institutions. For example, those of you who studied computers at colleges or universities in the early 1980s may have encountered Pascal on the school's mainframe computer.

This history leads some programmers to think of Pascal as an old-fashioned programming language, unsuitable for modern application development. This is especially true when Pascal is compared with languages such as C or C++. Indeed, quite a number of "less than well-informed" computer journalists hold this view of Pascal.

Pascal Emerges

However, this assertion about Pascal is — at least now — untrue. Borland has enhanced the language to the degree where there's little that programmers *cannot* achieve by using Pascal instead of the usual "serious" C/C++ development systems.

And compared with C/C++, Pascal code is easier to understand. This means that stable and bug-free applications are easier to build. In fact, the problems with difficult-to-understand C code led to the introduction of the short-lived Pascal-like systems language, Modula-2, marketed by companies such as Logitech and Jensen & Partners in the early 1980s. However, Modula-2 never really caught on, possibly because of the introduction of Turbo Pascal that went on to establish itself as the de-facto Pascal implementation on PCs.

In addition to the object-oriented (OOP) extensions that Borland added to Turbo Pascal (Object Pascal's ancestor) in version 5.5, a whole range of other useful language extensions essential for modern applications development were also added.

The aim of this article is not to discuss and compare Borland's current incarnation of Pascal with the version introduced by Niklaus Wirth in 1971. Instead, we'll discuss a few useful features of the original Pascal language that are often overlooked in the shadow of Object Pascal's extra language features.



Standard Pascal provided a number of user-definable data types such as the subrange, enumerated, and set type in addition to the usual numeric and character types. These user-definable types can be used to create flexible data constructs which, when used effectively, make it very easy to write clear, understandable Pascal code.

Let's find out more.

The Subrange Type

As its name implies, the *subrange* type allows you to define types that limit the possible values contained within a variable of this type to a user-defined range. The type is defined with the lower and upper bounds separated with two periods (`..`). For example:

```
type
  TNibble = 0..15;
```

This defines a type that can be used to store integers with values within the range 0 and 15. Any attempt to assign a value out of this range will cause a compile-time or run-time error (provided that **Range checking** is enabled on the Compiler page of Delphi's Project Options dialog box, or the `{SR+}` compiler switch is used in your code).

Moreover, the specified range need not be zero-based. So, we can make the following subrange type definitions:

```
type
  THiNibble = 16..255;
  TSignedByte = -128..127;
```

We can also declare a character-based subrange type to limit assignable values to numeric characters:

```
type
  TNumericChars = '0'..'9';
```

But how much space is used to store a variable of subrange type? In our examples above, only one byte is used. If the subrange was `0..65535`, then 2 bytes — the size of the `Word` type — are used for storage, the same as for a subrange of `256..65535`. If the range is `0..$FFFFFF`, the storage requirement is 4 bytes (not the expected 3 bytes, but the size of the `Longint` type). In terms of storage required, the subrange type is matched to the nearest ordinal type, thus `0..15` and `'0'..'9'` each occupy 1 byte of storage.

And just what is an *ordinal type*? An ordinal type is a sub-set of a standard type, such as any of the integer types (byte, word, integer, etc.) or the `Char` and `Boolean` types. Each possible value of an ordinal type has its own *ordinality* that is an integer value.

The various Real types, `String`, `Pointer`, and `Class` types are not ordinal types. The Integer types shown in [Figure 1](#) can be seen as equivalent subrange types. (The size and range of the `Integer` and `Cardinal` types will increase from 16 to 32 bits in Delphi32.)

The subrange type is a useful aid in debugging code where the possible values of a variable must be restricted to a certain range. As mentioned, any illegal assignment to a subrange type will cause

Integer Type	Subrange Type Equivalent
ShortInt	-128..127
Byte	0..255
SmallInt	-32768..32767
Integer	-32768..32767
Cardinal	0..65535
Word	0..65535
LongInt	-2147483648..2147483647

Figure 1: Integer types and their equivalent subrange types.

a compile-time or run-time error if **Range checking** is enabled. However, because assignments must always be checked to see if they are allowed, **Range checking** adds code and slows your program. Once your program is fully debugged (is there ever such a program?), you can turn off **Range checking** and re-compile your application to make it smaller and faster before delivering it to your customer.

Range checking can be useful for debugging Pascal applications. When using integer variables, choose the `Integer` type with a range that most closely fits the possible range of values your integer variable will encounter. Suppose, for example, that you're using an integer variable that should never be assigned a negative value — it's easy to be lazy and use a variable of type `Integer`. However, if you make the variable a `Byte` or `Word` type, **Range checking** will catch any attempt to assign a negative value to the variable and pinpoint a bug in your application.

The Enumerated Type

This user-defined type is frequently used throughout the VCL source code. Enumerated types are defined by a comma-separated list of "symbolic" ordinal values enclosed in brackets. For example:

```
type
  DOW = (Sunday, Monday, Tuesday, Wednesday,
        Thursday, Friday, Saturday);
```

declares a type representing a symbolic list of the days of the week. Declaring the variable, `DayOfWeek`, of this type, allows the following assignments:

```
DayOfWeek := Sunday;
```

or:

```
DayOfWeek := Friday;
```

but not:

```
DayOfWeek := 'Saturday';
```

nor:

```
DayOfWeek := 1;
```

Only the symbolic constants included in the enumerated type definition can be assigned to a variable of that type. (The most

fundamental enumerated type is Boolean which has just two possible values: *True* and *False*.)

Other examples of enumerated types used in the VCL include:

```
TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert,
                 dsSetKey, dsCalcFields);
```

```
TBorderIcon = (biSystemMenu, biMinimize, biMaximize);
```

```
TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast,
               nbInsert, nbDelete, nbEdit, nbPost,
               nbCancel, nbRefresh);
```

Examine those VCL properties with pre-defined single values selected from a drop-down list in the Object Inspector. Whenever a component property can be assigned one of a number of pre-defined values, that property is of an enumerated type.

The enumerated type can also be used as the basis for a subrange type:

```
type
  WorkingWeek = Monday..Friday;
```

We can even use the enumerated type as an array index:

```
const
  WorkDays = array [Monday..Friday] of string[3] =
    ('Mon', 'Tue', 'Wed', 'Thu', 'Fri');
```

or:

```
const
  WorkDays = array [WorkingWeek] of string[3] =
    ('Mon', 'Tue', 'Wed', 'Thu', 'Fri');
```

As I've stated, enumerated types are simply a symbolic representation of the list of possible ordinal values that a variable can have. The type *DOW* (defined earlier) is equivalent to the following hypothetical declaration:

```
type
  DOW = (0,1,2,3,4,5,6);
```

The type *DOW* is simply a list of zero-based ordered values. We can similarly interpret the Boolean type as (0,1) which is how *False* and *True* are interpreted in C/C++.

The code in [Figure 2](#) demonstrates this concept of underlying ordinal values. We see that with a little typecasting we can convert an enumerated type's current symbolic value to its underlying ordinality by using the standard function *Ord* or an integer typecast. We can also assign an integer that has been typecast to a variable of enumerated type.

An enumerated type can have a maximum of 65536 symbolic values that occupy 2 bytes of storage. Enumerated types with a maximum of 256 symbolic values occupy a single byte of storage.

The use of enumerated types makes code more readable and is certainly a better and safer alternative to defining several integer

```
type
  DOW = (Sunday, Monday, Tuesday, Wednesday, Thursday,
        Friday, Saturday);

var
  DayOfWeek: DOW;
  DayNum: Byte;
  ...

begin
  { Groan! }
  DayOfWeek := Monday;
  { Assigns 1 to DayNum }
  DayNum := Byte(DayOfWeek);
  { Also assigns 1 to DayNum }
  DayNum := Ord(DayOfWeek);
  DayNum := 5;
  { Assigns Friday to DayOfWeek }
  DayOfWeek := DOW(DayNum);
  { Causes a range-check error }
  DayOfWeek := DOW(10);
end;
```

Figure 2: Demonstrating underlying ordinal values.

constants (as you would have to with Visual Basic, for example). Standard ANSI C/C++ had the equivalent *enum* enumeration data type added only relatively recently.

The Set Type

The user-defined *set type* is one of the most useful language features of Pascal. A set type is a collection or list of values (also called *elements* or *set members*) of the same ordinal type (the base type) that is declared as a comma-separated list within brackets.

For example, a set of characters including the characters Y, y, N and n is declared as:

```
type
  YesNo = set of char;

var
  YN : YesNo;
  ...

begin
  YN := ['Y', 'y', 'N', 'n'];
```

So we've defined a set, but what use are sets in Pascal programs? Before continuing, we need to take a brief look at set theory.

Set Theory

We've all learned about sets at school and met with concepts such as *set union*, *set intersection*, and *set difference*. Set union is the set that includes elements from different sets; a set intersection is a set that includes only those elements common to different sets; and a set difference is the set containing those elements not present in another set. [Figure 3](#) illustrates these set operators diagrammatically. The shaded areas represent the result set of the operators applied between sets A and B.

Sets in Pascal can be used with set union, set intersection, set difference, and set equivalence operators, as well as the set membership operator. [Figure 4](#) lists the Pascal set operators.

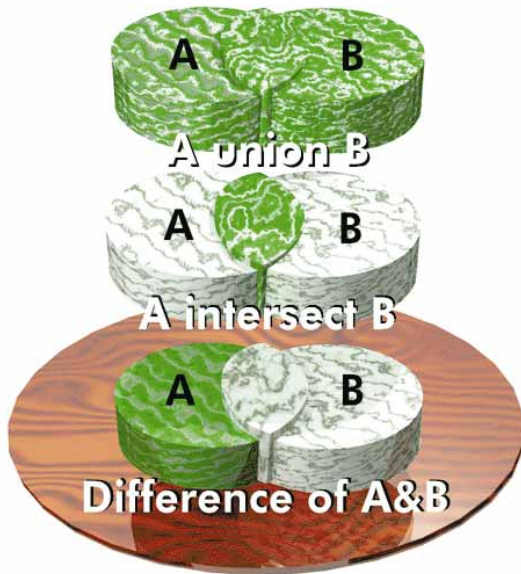


Figure 3: An illustration of Pascal's set operators.

The operands used in set union, set intersection, and set difference expressions are themselves sets. However, the left and right operand in set membership expressions are of ordinal and set type, respectively. Set equivalence and set membership expressions evaluate to a Boolean result. Expressions using the set union, set difference, or set intersection operators evaluate to a set result.

Obviously, the set operators that use set operands can only be used with sets of the same type. Therefore, for example, the following expression is illegal:

```
['Y','y','N','n'] + [1,0]
```

Sets can be declared in a number of ways (in addition to being declared in the standard comma-separated list). As we've seen, two periods can be used to define the range of values in a set. Thus, the set defined as:

```
['A'..'G']
```

is equivalent to the more verbose:

```
['A','B','C','D','E','F','G']
```

We can also combine the above techniques to define sets of non-consecutive values. For example:

```
['0'..'9','a'..'z','#']
```

defines a set of characters that includes the numeric characters, lower-case alphabetic, and the hash (or pound sign) character. Similarly, the set defined by:

```
[0..15, 255]
```

defines the set of integers between 0 and 15 and including 255.

Suppose we wanted to define a set of integers that includes all the values between 10 and 128 but excludes the values 15, 21, 29, 45, and 115. We could define the set by using the various ranges (0..14, 16..20, and so on). A far easier way would be to employ the set difference operator. It lets us define the required set as:

```
NumberSet := [10..128] - [15,21,29,45,115];
```

which is the equivalent of:

```
NumberSet := [10..14, 16..20, 22..28,
              30..44, 46..114, 116..128];
```

where *NumberSet* is of type Byte.

If we wanted our set of integers to also include 2, 4, 6, and 8, we could also use the set union operator:

```
NumberSet := [10..128] - [15,21,29,45,115] + [2,4,6,8];
```

And just to demonstrate the set intersection operator, the following set expressions are equivalent:

```
[10..20]
```

and

```
[0..20] * [10..100]
```

But what is the result of the

```
[0..255] - [0..255]
```

Operator	Meaning	Expression	Result
+	Set union	[1,2,3] + [4,5,6]	[1,2,3,4,5,6]
*	Set intersection	['A','B','C'] * ['B','C','D']	['B','C']
-	Set difference	['A','B','C'] - ['B','C','D']	['A']
=	Set equivalence	['A','B','C'] = ['A','B','C'] ['A','B','C'] = ['C','D','E']	True False
IN	Set membership	'A' IN ['A','B','C'] 'Z' IN ['U','V','W']	True False

Figure 4: Pascal's set operators.

expression? This results in the *empty set* that is denoted by empty brackets (`[]`). As the name suggests, the empty set is the set that contains no values.

So on to the set membership operator, **in**. This operator makes sets very useful in Delphi. Consider the following statement:

```
if (NumVal >= 0) and (NumVal <= 32) then
  DoSomething;
```

This can be shortened to:

```
if (NumVal in [0..32]) then
  DoSomething;
```

With a more complicated conditional test, the statement:

```
if (NumVal = 10) or (NumVal = 15) or
  (NumVal = 21) or (NumVal = 56) then
  DoSomething;
```

can be shortened to:

```
if (NumVal in [10,15,21,56]) then
  DoSomething;
```

With longer and more complicated conditional expressions, we can see a clear advantage to using sets and the **in** operator. However, the benefits are not just in terms of code clarity. Set expressions are more easily optimized by the compiler, and thus increase code speed and reduce code size.

Not Just Data Types

So far, the examples have involved characters and integers. But sets aren't restricted to these data types. Recall that a set is a collection of values of the same ordinal type. Therefore, a set's base type can be defined by an enumerated type. For instance, we could define a set of the days of the week:

```
DOWSet = set of DOW;
```

This line of code is equivalent to:

```
DOWSet = set of (Sunday, Monday, Tuesday, Wednesday,
  Thursday, Friday, Saturday);
```

A set's base type can also be a subrange type:

```
DOWNumbersSet = set of 0..6;
```

The only restriction on a set's members is that the maximum possible number of values the member type can have is 256. This effectively restricts you to sets of type byte, char, subrange, or enumerated. Therefore, the following set declaration is illegal:

```
BadSet: set of 0..120000;
```

Despite this restriction, sets are still useful and powerful tools in your code armory. Delphi's VCL uses sets in many places.

TBorderIcons, for instance, is a *TForm* property that is a set of *TBorderIcon*. Sets are also found when specifying the buttons to be displayed in the dialog box created by calling the *MessageDlg* function.

To check if a form has a maximize and minimize button, we could use the test:

```
if (BorderIcons = [biMaximize, biMinimize]) then
  DoSomething;
```

However, our test will fail if *BorderIcons* is set to *biSystemMenu*, *biMaximize*, or *biMinimize*. To avoid this potential problem, we should use the set intersection operator:

```
if ((BorderIcons * [biMaximize, biMinimize]) =
  [biMaximize, biMinimize]) then
  DoSomething;
```

Wherever the value of a component's property can take on a combination of values from a list, that property is of a set of enumerated type. If you closely examine the unexpanded *TForm.BorderIcons* property in the Object Inspector, you'll see the property values are enclosed in brackets as set elements.

Another use of sets is to test for a numeric character. This can be as short as:

```
IsNumeric := (NumChar in ['0'..'9']);
```

Pascal also provides the standard procedures, *Include* and *Exclude*, for use with sets. The *Include* procedure allows you to add an element to a set and takes the set and the element to be added as parameters. The *Exclude* procedure allows you to remove an element from a set and takes the set and the value of the element to be removed as parameters. Figure 5 shows the parameters used with these procedures.

Procedure	Example
Include (var s:set of T; e:T)	Include(CharSet, 'A');
Exclude (var s:set of T; e:T)	Exclude(CharSet, 'A');

Figure 5: The Object Pascal *Include* and *Exclude* procedures.

Using the *Include* and *Exclude* procedures is equivalent to using the set union and set difference operators. However, these Object Pascal procedures generate more efficient code. Note that these procedures can only add or remove one element at a time. On the other hand, using the union or difference operators can add or remove several elements in one statement. The choice is yours.

Set Storage

Okay. So, sets are useful for reducing complex Boolean expressions to a short and simple set expression. Surely there's a price to pay for this convenience, right? Sets are slow and occupy lots of storage space, right? Well, don't be

such a pessimist because this isn't necessarily so. To see why, let's take a look at how sets actually work.

A set is a bit array of a maximum 256 bits (or 32 bytes) with each bit representing a member of the set. If a particular member exists in the set, then the corresponding bit representation is set. The maximum size of the set, therefore, determines the amount of storage space required. For example, a set of Char or set of Byte always occupies 32 bytes.

However, a set of enumerated type requires only enough space to store its symbolic values. Therefore, the set *DOWSet* will occupy only 1 byte because the set contains 7 possible values. That equates to 7 bits that fit into a single byte. So, a set of enumerated type containing 9 possible values will occupy 2 bytes and so on, up to the maximum of 256 possible values that occupy 32 bytes.

The storage requirements of a set of subrange base type depend on the bounds of the subrange. The following set declarations will use one and two bytes of storage, respectively:

```
var
  { Has a maximum of 8 possible values and therefore
  needs one byte of storage }
  ByteSet: set of 0..7;
  { Has a maximum of 16 possible values and therefore
  needs two bytes of storage }
  WordSet: set of 0..15;
```

The number of bytes required to store a set of a particular base type is calculated with the following formula:

$$(\text{MaxValue} \text{ div } 8) - (\text{MinValue} \text{ div } 8) + 1$$

where *MaxValue* and *MinValue* are the largest and smallest values of the base type.

Set Internals

I've said that a set is simply a bit array. Let's support that with a few facts and examine the performance issues when using sets. Consider the following set declaration:

```
var
  ColourSet : set of (Red,Green,Blue,Cyan,Yellow,
                    Magenta,Black,White);
begin
  ColourSet := [Blue, Cyan, Black];
end;
```

Since there are 8 possible values for the enumerated type *ColourSet*, it occupies only a single byte (determined using `sizeof(ColourSet)`). If we examine the value of this byte (using `byte(ColourSet)` within the Evaluate/Modify dialog box of the IDE debugger) after assigning *ColourSet*, we see it's set to 76 (or binary 01001100). Because the set contains the third, fourth, and seventh member of the enumerated type, the third, fourth, and seventh bits are set in the byte representation of the set. (Remember that the lowest, or least significant, bit is the rightmost bit in this scheme.)

The byte position of a set member in the bit array can be determined with the formula:

$$(\text{SetElement} \text{ div } 8) - (\text{MinValue} \text{ div } 8)$$

where *SetElement* is a set element's position within the range of possible values of the set base type, and *MinValue* is the minimum value that the base type can store.

Narrowing down the position of the set element, we can determine the bit position (within the byte position) of a set member with the formula:

$$\text{OrdinalValueOfTheElement} \text{ mod } 8$$

So how do the set operators work? Considering that a set is simply a bit pattern, you may guess that set operators are essentially bitwise operators, and you'd be right. This makes set operations fast in practice. Let's examine each set operator in turn.

The set union operator is essentially a fancy wrapper for the bitwise OR operator. Consider these two sets:

```
a := [Red, Green, Blue];
b := [Black, White];
```

These are represented internally as the two bytes 7 and 192, or 00000111 and 11000000. OR'ing these two values gives us 11000111 which is the byte representation of the set:

```
[Red,Green,Blue,Black,White]
```

This is the result of set a added to set b.

The set intersection operator is based on the bitwise AND operator. For example, look at sets a and b:

```
a := [Red, Green, Blue];
b := [Black, White, Green];
```

The binary representation of these sets is 00000111 and 11000010 for a and b, respectively. Therefore, if we apply the AND operator we get:

```
00000111 AND
11000010
-----
00000010
```

This is the binary representation of the set [Green] which is the result of a multiplied by b.

The set difference operator is more complicated, and is based on the XOR and AND operators. Using sets a and b as declared above, we apply the XOR operator:

```
00000111 XOR
11000010
-----
11000101
```

This is the binary equivalent of the set [Red, Blue, Black, White]. Now we AND this result with the bit pattern of set a to complete the operation:

```
11000101 AND
00000111 (the bit pattern of set a)
-----
00000101
```

This is the binary form of the set [Red, Blue] which is the result of subtracting set b from set a.

The set membership operator uses the AND operator and compares the result with the value of the ordinal left operand. The following code is the equivalent of testing to see if White is a member of set b:

```
10000000 AND
11000010
-----
10000000
```

This is the binary value of White. It is then compared with the value of the left operand (White) to return True.

The inner workings of the set equality operator should be quite obvious!

A New Data Type?

Now that the inner workings of sets has been revealed, I will invent a useful “bit-field” data type. This can be achieved by using what we’ve learned so far in this article in combination with a feature that has been in Turbo Pascal for a long time: *absolute variables*.

Absolute variables are declared (using the keyword **absolute**) as being residents at a certain specified memory address. This address can be a segment and offset combination (e.g. 7FC0:FF00) or the address of another specified variable. For instance, the variables *AByte* and *AChar* overlay each other because both point to the same memory address:

```
var
  AChar: Char;
  AByte: Byte absolute AChar;

begin
  { Same as AByte := 65 }
  AChar := 'A';
  { Same as AChar := 'a' }
  AByte := 97;
end;
```

Therefore, you can carry out the code shown in Figure 6.

However, instead of messing around with bitwise operators to obtain or set a bit, we can use the far clearer set notation. Therefore, checking to see if the high bit of *MyByte* and *MyWord* is set is simply:

```
if (b7 in My8Bits) then
  ShowMessage('The high bit of MyByte is set!');
if (15 in My16Bits) then
  ShowMessage('The high bit of MyWord is set!');
```

```
type
  { Bits 0 to 7 of a byte }
  TByteBits = set of (b0,b1,b2,b3,b4,b5,b6,b7);
  { Bits 0 to 15 of a word }
  TWordBits = set of 0..15;

var
  MyByte: Byte;
  MyWord: Word;
  My8Bits: TByteBits absolute MyByte;
  My16Bits: TWordBits absolute MyWord;

begin
  { Sets My8Bits to [b7] }
  MyByte := 128;
  { Sets My8Bits to [b0,b1,b2, b3,b4] }
  MyByte := 31;
  { Sets My16bits to [0..7] }
  MyWord := 255;
  { Sets My16bits to [7] }
  MyWord := 128;
  { Sets My16bits to [0,15] }
  MyWord := 32769;
  { Sets MyByte to 3 }
  My8Bits := [b0,b1];
  { Sets MyWord to 7936 }
  My16Bits := [8..12];
  { Sets MyByte to 0! }
  My8Bits := [];
end;
```

Figure 6: Using absolute variables.

To clear the high bit use:

```
My8Bits := My8Bits - [b7];
My16Bits := My16Bits - [15];
```

To clear the high and low bits use:

```
My8Bits := My8Bits - [b0,b7];
My16Bits := My16Bits - [0, 15];
```

After this discussion, it’s obvious that the highly sophisticated set type is nothing more than a fancy wrapper around a bit-field used with the AND, OR, and XOR bitwise operators. While the set initially seems a sophisticated beast, once understood, it’s more like a sheep in wolf’s clothing.

Because of their internal workings, sets are fast and compact, especially those sets of ordinal types with up to 8 or 16 possible values that occupy a single byte or two bytes, respectively. Any number occupying 16 bits or less is a number that is easily handled by the CPU’s machine instructions. Because of this, such 1 or 2-byte sets can have their operators optimized by the compiler that generates in-line machine code instead of calls to run-time library routines. Similarly, the *Include* and *Exclude* standard procedures, when used on 8 or 16 bit sets, generate in-line machine code.

With Delphi32, you can expect to see all 32-bit set operations optimized as the CPU and operating system instructions make better use of 32-bit integers.

Nearing the End

Where would we be left with Pascal if we didn’t have the sub-

range, enumerated, and set types? Without enumerated types, we'd be forced to write code such as:

```
const
  Sunday   = 0;
  Monday   = 1;
  Tuesday  = 2;
  Wednesday = 3;
  Thursday = 4;
  Friday   = 5;
  Saturday = 6;
```

```
var
  DOW: byte;
```

```
begin
  DOW := Sunday;
end;
```

instead of:

```
var
  DOW: (Sunday, Monday, Tuesday, Wednesday, Thursday,
        Friday, Saturday);
```

```
begin
  Dow := Sunday;
end;
```

Also note that in the longer version of the code there is no built-in mechanism to prevent the assignment of values outside the range of 0 to 6 to the byte variable *DOW*.

Without sets we'd have to write:

```
begin
  if (Num = 1) or (Num >= 6 and Num <= 66) or
     (Num = 128) or (Num = 132) or
     (Num >= 149 and Num <= 155) or (Num = 170) then
    DoSomething;
end;
```

instead of:

```
begin
  if Num in [1,6..66,128,132,149..155,170) then
    DoSomething;
end;
```

and we'd be stuck with:

```
var
  DaysOffThisWeek: byte;

begin
  DaysOffThisWeek := Wednesday + Friday;
  ...

  { Check if Wednesday has been taken as holiday }
  if (DaysOffThisWeek and Wednesday) = Wednesday then
    ...
end;
```

instead of:

```
var
  DaysOffThisWeek = set of (Sunday, Monday, Tuesday,
                            Wednesday, Thursday, Friday,
                            Saturday);

begin
  DaysOffThisWeek := [Wednesday, Friday];
  ...

  { Check if Wednesday has been taken as holiday }
  if (Wednesday in DaysOffThisWeek) then
    ...
end;
```

Once again, the first version will be more prone to invalid assignments and is less easy to follow unless you understand bitwise operators.

Conclusion

So there you have it. Pascal makes it easier to write clear, readable code that gives you a better chance of catching those bugs that your customer may find first.

Who said Pascal is passé? ▲

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





ON THE COVER

DELPHI / OBJECT PASCAL / C++



By *Richard D. Holmes*

Cultural Differences

A Comparison of Pascal and C++ Language Features

Because of its capabilities as an easy-to-use Windows programming tool, Delphi is often compared to products such as PowerBuilder and Visual Basic. (Delphi is sometimes even described as a “Visual Basic killer”.) Yet, in many ways, the Object Pascal language that lies at the heart of Delphi is more akin to C++ than to Basic. Like C++, Object Pascal is a hybrid, object-oriented language with strong type checking that is enforced at compile-time. This article compares the language features of Delphi and C++, especially with regard to support for object-oriented programming. (In a future article the performance of identical applications written in Delphi and C++ will be compared.)

Note that throughout this article, the feature set of C++ is that of the proposed ANSI/ISO standard, as described by Bjarne Stroustrup in his book, *The Design and Evolution of C++* (Addison Wesley, 1994). Many current implementations of C++ differ significantly from this dialect. For details, consult your system’s language reference manual. Details of the Object Pascal language are given in the *Object Pascal Language Guide*.



The Object Model

An object is a data structure that packages some data fields together with the code that manipulates them. Objects allow the internal implementation of a data structure to be hidden. The users of an object can access only those fields and methods that the object designer chooses to expose. Objects allow the compiler to enforce the separation between interface and implementation. Through inheritance, an object can be extended and refined “by difference” — it need not be totally rewritten.

Hybrids. Both Object Pascal (OP) and C++ are hybrid languages in which object-oriented programming features have been grafted onto a non-object-oriented parent. The similarities between OP and C++ arise more from the similar way in which they have evolved to support object-oriented programming than from similarities in the parent languages.

As hybrids, both languages are required to maintain a large measure of backward compatibility. This results in a certain degree of schizophrenia as primitive data types are mixed with objects, and as calls to statically

bound procedures are mixed with calls to dynamically bound polymorphic methods. The requirement to support two programming models enlarges and complicates both languages, but also enhances their flexibility.

Classes. Both OP and C++ are class-based object languages. Individual objects are not defined directly. Instead, a class is defined and then objects are created that represent instances of that class. There is a one-to-many relationship between a class and its instances (objects). The class stores the code (*methods*) that is shared by the objects. However, each object stores its own data fields (*state*). In distinguishing between classes and objects, the nomenclature of C++ is more rigorous than that of OP, which often refers to both objects and classes as *objects*.

Fields. The fields within an object can be either primitive data types or objects. Both languages allow the construction of composite objects of unlimited complexity.

Normally, every object has its own copy of the data fields declared within its class. In the jargon of SmallTalk, these data fields are *instance variables*. It's also possible to have *class variables* that are created only once per class rather than once per object. In C++, a data field declared to be *static* is a class variable. OP has no direct support for class variables. However, they can easily be emulated using global variables within the unit's **implementation** section.

Properties. In OP, a *property* is a field that can be viewed and set at design time. The addition of properties, property sheets, and property sheet editors to OP represents a significant extension compared to other compiled object languages. There is no equivalent in C++.

Properties are also noteworthy because they are accessed indirectly. Read and write methods provide a layer of code that encapsulates a property's state. They can be either simple references to a data field or calls to functions that perform additional work, such as maintaining referential integrity. Read and write methods can easily be emulated in C++.

Methods. Both OP and C++ allow the overloading of method names. Two or more methods within a class are allowed to use the same name, provided that each method's signature is unique. A method's *signature* is determined by its name, the data type of its parameters, and the data type of its return value. C++ also allows operators, such as +, -, =, and [], to be redefined for each class. OP does not allow operator overloading.

The mechanisms for dispatching methods in OP and C++ clearly reflect their origins as hybrid languages. Both languages allow method calls to be either statically or dynamically bound. *Statically bound* methods are non-polymorphic, can be resolved at compile time, and correspond to traditional function calls. The address of a static method can be stored directly in the code segment by the linker. On the other hand, *dynamically bound* or *virtual* methods are polymorphic and can only be resolved at run-time, when the exact type of the object is known.

Calls to virtual methods must be dispatched by the program at run-time. Both OP and C++ use a *virtual method table* (VMT) to store the address of each virtual method. Each class has its own VMT that is nothing more than an array of function addresses. Dispatching virtual methods is fast, since it typically requires just de-referencing the pointer to the VMT and then indexing into the array to retrieve the method's address.

In addition to virtual methods, OP also allows the *dynamic* dispatching of methods. This is used primarily for handling Windows messages. Because there are hundreds of these messages, the space overhead of maintaining an entry for each of them in the VMT of every class that is a descendant of *TWindow* would be substantial. Therefore, the *dynamic method table* (DMT) for each class contains entries for only those methods (message handlers) that have been overridden within that class. If a message handler is not found in an object's DMT, then the DMT of each of its ancestors is searched. Since *TWindow* contains a default handler for every message, the search will be satisfied there, if not before. ANSI/ISO standard C++ has no equivalent to dynamic dispatching.

In both OP and C++, only methods that are explicitly declared as **virtual** can exhibit polymorphic behavior. This is probably an unavoidable tradeoff in a hybrid language that must maintain compatibility and competitive performance with its non-object-oriented parent. It is nonetheless a shortcoming compared to pure object languages such as SmallTalk or Eiffel. It limits the refinement and specialization of descendants to only those characteristics that the original class designer saw fit to make polymorphic.

C++ allows methods to be declared *in-line*. This allows simple methods, such as those that read or write a private data field, to be invoked without the overhead of a function call (that must create a stack frame and save the CPU's register values). In-line methods are essentially an optimized form of static binding.

OP allows a method to be declared to be a *class method*. Class methods, existing independently of any instance of the class, can be invoked even if no instances of the class have yet been created, and are not allowed to access any of the data fields within the class. In C++, a member function declared to be static is a class method. (Note that in OP a *static method* is statically bound, whereas in C++, it's a class method.)

Delegation. OP allows an event to be handled not by a method within the receiver's class, but by a method within an associated class. In essence, message handling is "delegated" to another class. The message receiver is usually a visual control, and the associated class is usually the form that contains the control. C++ has no direct support for delegation.

Access Control. Within an object's declaration, both OP and C++ use the access control modifiers — **private**, **protected**, and **public** — to control the visibility of the object's fields and methods. In C++, a public field or method has global visibility, a private field or method is visible only within that object's methods, and a protected field or method is visible only with-

in that object's methods or within the methods of its descendants. In OP, a public field or method has global visibility, a private field or method is visible to all procedures and functions contained within that object's unit, and a protected field or method is visible to all procedures contained within that object's unit or in the units that implement its descendants. C++ allows a class to grant methods in another class access to its private fields by declaring them to be "friends". OP does not have a **friend** directive, since all methods implemented within an object's unit are automatically treated as friends.

OP also adds the modifier **published** for properties that must be visible at design time. It's a kind of "super public". In C++, a field or method is **private** unless declared otherwise, whereas in OP, it's **published** unless declared otherwise. Although the terms are identical and the concepts are similar, the details of access control are somewhat different between OP and C++.

References. In C++, a reference is a synonym or alias for something. References act similarly to pointers, but without the need for explicit de-referencing. References are used mostly for passing arguments to functions, especially for operator overloading. OP does not have explicit references. However, in OP every object's identifier is implicitly a reference. Viewed another way, OP doesn't really have (local) objects, it has only references (to dynamic objects).

There are some differences between references in OP and C++. For instance, in OP a reference always points to an object (i.e. an instance of a class), whereas in C++, it may point to either an object or a built-in data type. In OP, a reference may be **nil**, but in C++, a reference must always point to something. In OP, a reference may be reassigned to point to a different object. However, a reference always points to the same object in C++.

Constructors and Destructors. Both OP and C++ use constructor methods to create and initialize new objects, and destructor methods to destroy them and perform any required clean up. Both languages support constructor overloading, and both require (C++) or recommend (OP) using virtual (polymorphic) destructors. Constructor overloading allows specialized constructors to be used in cases where the object requires specialized initialization.

Inheritance. Both OP and C++ allow a class to be specialized through inheritance. The descendant may add new data fields, but may not remove any existing fields. The descendant may add new methods and may replace (or override) existing methods.

Inheritance is implemented by appending any new fields to the end of the parent object's data record, appending any new polymorphic methods to the end of the parent's VMT, and replacing the address in the VMT of any inherited methods that have been overridden.

OP allows a descendant to inherit from only one ancestor. It uses a single-inheritance hierarchy that is rooted in *TObject*, the ancestor of all objects. C++, on the other hand, allows a descendant to inherit from multiple ancestors. The benefits of single

inheritance versus multiple inheritance are widely debated (and this controversy is beyond the scope of this article). Multiple inheritance introduces into C++ complications (name clashes) and language features (virtual base classes) not present in OP.

Both languages allow the declaration of abstract classes. Abstract classes allow an interface to be defined for a concept that cannot meaningfully exist as a concrete object. The classic example is a graphics program in which *TShape* defines the common interface for the concrete classes *TCircle*, *TRectangle*, and *TTriangle*. An abstract class is one whose declaration includes one or more abstract methods.

Methods in OP are made abstract by adding the **abstract** directive to the method declaration. In C++, methods are made abstract by adding the *pure virtual* qualifier (`= 0`) to the method declaration. In OP, abstract classes can be instantiated and used, provided the abstract (undefined) methods are not called. In C++, abstract classes can never be instantiated — they can only be used as a base for deriving other classes.

Templates. Templates are used in C++ to define generic classes that can be instantiated for a variety of specific types. Templates are especially useful for creating type-safe containers. For example, a linked list template could be used to create a list object that accepts only triangles. With templates, type checking can be performed at compile-time to ensure that every object added to *TList<TTriangle>* is either a *TTriangle* or one of its descendants. Delphi has no features for defining generic classes.

Run-Time Type Information. Both OP and C++ provide run-time type information (RTTI) on objects. OP uses the *ClassName*, *ClassType*, *ClassParent*, and *ClassInfo* methods to provide type information on an object. The **is** operator is used to determine whether an object is of a given type or one of its descendants. The **as** operator is used to perform type-safe casting. In C++, the **typeid** operator returns a reference to the **type_info** object that describes an object's class, including its name. The **dynamic_cast<>** operator is used both to determine whether an object is of a given type and to perform type-safe casting.

Memory Management

Both OP and C++ are designed to create objects that use memory efficiently. Except for the addition of a pointer to the VMT for its class, the size and memory layout of an object's data fields are identical to that of the equivalent **record** (OP) or **struct** (C++). (Classes that are not polymorphic avoid even this overhead.)

C++ supports three storage classes for allocating program variables: static, automatic, and dynamic. Static objects are created in the program's data segment, automatic objects are created on the stack, and dynamic objects are created on the heap. Static storage is used for global objects, automatic storage is used for local objects, and dynamic storage is used for objects created on-the-fly.

C++ also separates the object construction process into two steps: memory allocation and initialization. Memory allocation

is performed by the **new** method, which can be both overridden and overloaded. By allowing the use of customized memory allocation routines, this provides considerable flexibility, albeit at the cost of requiring the programmer to grapple with the low-level details of writing a custom memory manager.

OP, by contrast, supports only dynamic objects. Local and global objects can be declared, but in actuality, only the reference (pointer) is stored in the data segment or on the stack. The object itself must be manually created and is always stored on the heap. OP also does not allow custom memory allocation routines. All classes use the built-in heap manager that is optimized for both space and speed compared to the Windows *GlobalAlloc* function.

Neither language provides automatic garbage collection. The programmer is responsible for tracking and deleting any objects created dynamically.

Type Checking

Both C++ and OP use static type checking as the primary means of verifying that a program is internally consistent. All identifiers must be explicitly declared before they are used. All assignments, expressions, and function calls are checked for type compatibility. Type checking is done at compile-time rather than at run-time.

Both languages require that the identifiers used for variables and constants be explicitly declared. However, OP requires that all variables and constants be declared in special sections at the top of the sub-program (**const** and **var**). On the other hand, C++ allows them to be declared at the point in the code where they are first used. C++ allows a variable to be initialized as part of its declaration, while OP allows this only for “typed constants” (which really aren’t constant). If a variable or constant is an object, C++ will call the constructor specified by the initialization expression, or the default constructor if no initialization is provided. In OP, declaring an object variable only allocates space for the reference pointer. It neither creates nor initializes the object.

Both languages require that the identifiers for functions be explicitly declared and that the declarations include a list of the function’s parameters and their data type. (OP makes a distinction between functions and procedures, but a procedure is just a function with a return type of “void”.) The combination of a function’s name, returned data type, number of parameters, and sequence of parameter data types determines its “signature”. It is an error in either language to call a function without the correct number or sequence of parameters. In this respect, C++ is much stricter than its parent, which originally performed no parameter checking at all. For compatibility with C, both languages allow a function to be explicitly declared to accept a variable number of parameters. C++ also allows a default value to be specified for each parameter, which allows an abbreviated parameter list to be used for simple calls to a function that sometimes requires many parameters. OP does not support default parameter values.

Both languages allow parameters to be passed either by value or by reference. In both OP and C++, parameters can be declared **const** to prevent the function from modifying their value. This allows calls by reference to be used for efficiency without compromising safety. (In OP, passing a **const** parameter by value or by reference is decided by the compiler.) In C++, methods can also be declared **const**. This guarantees that they do not modify any object’s fields.

Both languages provide *casts* so that a variable of one type can be treated as if it were a different type. These casts can be used on either primitive or user-defined types. Unchecked casts place the burden of ensuring compatibility upon the programmer. In addition to ordinary casts, C++ also provides specialized casts that give the programmer finer grained control and make the intent of the cast more obvious. These specialized casts include **dynamic_cast<>**, **static_cast<>**, **reinterpret_cast<>**, and **const_cast<>**. In OP, the **is** and **as** operators must be used together to perform type-safe casting.

All in all, the type safety of C++ has improved to the point where it is now essentially equal to that of OP. This is a major change from the relative positions of C and Pascal.

Exception Handling

Both OP and C++ use exception handlers to separate the code that handles errors from the code that represents the normal flow of control. The code to be protected is enclosed within a **try** block and followed by an exception handling block. Functions that detect an error will either *raise* (OP) or *throw* (C++) an exception. If the caller is not prepared to handle (or *catch*) the exception, then it will be propagated upwards through the chain of exception handlers that may have been installed by higher-level callers. If the exception is not caught by a user-installed handler, then it will be caught by the default handler. In OP, the default handler displays a message box describing the error. In C++, the default handler terminates the program.

In addition to the **try...except** construct, OP also provides the **try...finally** construct. The difference is that the statements in the **finally** block do not constitute an exception block and are executed whether or not an error occurs. This is useful for ensuring that allocated resources are released regardless of the success or failure of the code in the **try** block. C++ has no structural equivalent to **try...finally**, but relies instead upon a strategy of “resource acquisition is initialization”, wherein the destructors of local objects (storage class “automatic”) are used to release any resources that were acquired by their constructors. The destructors will be called as the stack unwinds, regardless of whether it is being unwound because an exception was raised or because the function completed normally.

Both languages treat exceptions as objects and implement a hierarchy of exception types. This allows a generalized exception handler to catch any of a variety of specialized exceptions.

Modularity

Both OP and C++ allow the source code for a large program to be divided into individual files that can be separately compiled. In Delphi, the program is divided into *units* (these are similar to

modules in Modula-2 and *packages* in Ada). A unit consists of a **public** interface and a **private** implementation. The **public** part of a unit is imported into another unit through the **uses** directive. The explicit separation of **interface** and **implementation** allows the compiler to manage dependencies between units and to automatically recompile only those units that are affected by a change. **Implementation** sections also provide a level of enforced privacy that is not available in C++. Through the use of nested subprograms, OP also supports finer grained modularization than C++.

A C++ program consists of header files and source files. The header files typically contain the interface definitions, and the source files contain the implementation. However, this separation is not enforced by the language, and the contents of a source file have equal visibility to the contents of a header file. C++, like C, does not support nested functions. It does support nested classes, but these are generally less useful.

ANSI/ISO C++ provides *namespaces* to assist in the construction of large programs and in the use of libraries. This provides a powerful mechanism for managing namespace pollution and for ensuring that identifiers are not “silently” re-declared. Units in OP provide a similar form of encapsulation for identifiers declared within the **interface** part. (Identifiers declared within the **implementation** part are, of course, completely private to that unit.) However, the rules of unit scope mean that the resolution of clashing names is dependent upon the sequence of units listed in the **uses** clause. Changing the sequence of items in the **uses** clause can (silently) change the meaning of an OP program.

Lastly, OP has introduced the concept of **initialization** sections that are executed during program start up. This is useful for initializing library routines. A similar effect can be achieved in C++ by using constructors for static objects, but this can get tricky.

Portability

C++ is available on many platforms, while Delphi is currently available for only one. If portability is a primary concern, however, C is much more portable than either C++ or OP.

The limited portability of C++ is due to two factors. First, the language itself has undergone many revisions, some quite recently. Many implementations of C++ lag behind the feature set of the ANSI/ISO standard by as much as five years, lacking major features such as templates and exception handling, not to mention new-style casts, RTTI, and namespaces. Second, most object-oriented programs use third-party class libraries that are not portable. Even after the ANSI/ISO standard library for C++ is completed and becomes widely available, there will be problems with library portability. This is because the standard library does not cover such key areas as graphical user interfaces and database access.

Conclusion

In summary, Object Pascal provides most of the object-oriented programming features of C++. Both languages are hybrids that implement object-oriented programming through the use of virtual method tables for polymorphic methods. Both allow

objects to be specialized through inheritance, but allow only those methods originally declared as **virtual** to be made polymorphic. Both use the access control modifiers **private**, **protected**, and **public** to control the visibility of an object’s fields and methods. Both use strict type checking at compile time as the primary means of verifying that a program is internally consistent. Both treat exceptions as objects, use inheritance to implement a hierarchy of exception types, and pass exceptions from the innermost scope outwards until a handler is found. Lastly, both languages provide for the separate compilation of program modules and support reuse through object libraries.

The most significant features in C++ that are missing in Object Pascal are templates, multiple inheritance, operator overloading, in-lined functions, the ability to allocate objects on the stack (“automatic”) or in the data segment (“static”), and the ability to override the memory allocation operator (**new**). The features in Object Pascal that are missing in C++ are published properties that can be viewed and set at design time, delegation, nested subprograms, and modules (“units”). Neither language provides automatic garbage collection, and both are handicapped by gaps in their standard object libraries. C++ has the edge in portability, but for Windows programming, Delphi provides a development environment that is unsurpassed for convenience and speed.

Lastly, in spite of their similarity in features, there are significant cultural differences between Object Pascal and C++. Both languages are hybrids created by grafting object-oriented features onto popular 3GLs, but the philosophical foundations of their parents are quite different (see **Figure 1**). For

Pascal was designed:	C was designed:
for clarity, safety, and modularity.	for direct manipulation of raw memory with minimal overhead in space and time.
as a high-level language with direct support for multi-level modularization and data abstraction.	as a medium-level language that is close to the machine.
to protect the programmer.	to trust the programmer.
to teach applications programming.	for professional systems programmers.

Figure 1: Original design intentions for Pascal and C.

example, for nearly 20 years Pascal was the premier language for teaching structured programming. C was, and still is, the best “portable assembler”. Although their object-oriented offspring have converged significantly, these cultural differences are still quite evident. They should not be ignored. **Δ**

Richard Holmes is a senior programmer/analyst at NEC Electronics in Roseville, California, where he designs and develops client/server database applications. He can be reached on CompuServe at 72037,3236.





DB NAVIGATOR

DELPHI / OBJECT PASCAL



By Cary Jensen, Ph.D.

The BatchMove Component

Copy, Delete, Append, and Update Data
with this Versatile Component

It's often overlooked when the Data Access page on the Component Palette is mentioned, but contrary to general perception, the BatchMove component can play an important role in database applications created with Delphi.

The capabilities of the BatchMove component appear simple enough. It permits you to move or copy records from a DataSet to a table. However, this simplicity belies its usefulness. Like a SQL INSERT query, BatchMove can be used to insert records from a DataSet into an existing table. Unlike an INSERT query, however, the table you copy the records to doesn't need to exist.

But this is just the beginning. There are four major capabilities of the BatchMove component: creating a table and placing the current records of a DataSet in it, deleting records from a table that correspond to those in a DataSet, inserting records from a DataSet to a table, and updating existing records in a table based on those in a DataSet.

Before examples of these operations are highlighted, let's consider the essential properties and methods of this component.

Using BatchMove

There are three essential properties for using the BatchMove component: *Source*, *Destination*, and *Mode*.

First, the *Source* property can be assigned any DataSet component. Therefore, you can assign either a Table, Query, or StoredProc component name to this property. While any Table component is acceptable, you would only assign one of the other DataSet descendants to this property if they return a cursor to a DataSet. For example, it would be reasonable to assign a Query component containing a SQL SELECT statement to the *Source* property, but it wouldn't make sense to assign a Query containing an ALTER TABLE statement.

Second, the *Destination* property is always assigned a Table component. This table is where the records of the source DataSet are copied, deleted, inserted, or updated.

Third, the *Mode* property defines the type of operation performed by the BatchMove component. Figure 1 shows the five modes. It also shows whether a table assigned to the *Destination* property must exist before calling BatchMove's *Execute* method, and if the destination table must be indexed.

Mode	Destination Must Exist?	Must Be Indexed?
<i>batAppend</i> (default)	No	No
<i>batAppendUpdate</i>	Yes	Yes
<i>batCopy</i>	No	No
<i>batDelete</i>	Yes	Yes
<i>batUpdate</i>	Yes	Yes

Figure 1: The types of operations performed by BatchMove.

Creating a Table

One of the simplest and more common uses of BatchMove is to create a new table containing the records returned by a Query or StoredProc.

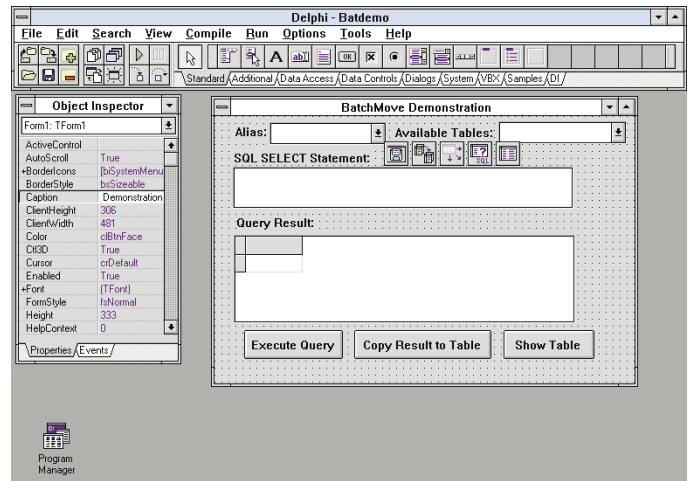
As an example, the form in Figure 2 provides a memo field where a user can enter a SQL SELECT statement. The results of this query — executed against the database selected in the **Alias** combobox when the **Execute Query** button is clicked — are displayed in the **Query Result** DBGrid.

After executing the query, the results can be written to a new table by clicking the **Copy Result to Table** button. The Object Pascal code associated with this button is shown in Figure 3. First, the code ensures the Query component is active. If so, it displays the Save File common dialog box using a SaveDialog component. If the user selects or enters the file name to copy the query result records to (indicated when the SaveDialog's *Execute* method returns *True*), the selected file name is assigned to the Table component *Table1*.

Next, BatchMove's *Source* property is set to *Query1*, its *Destination* property is set to *Table1*, and its *Mode* property to *batCopy*. Its *Execute* method is then called to initiate the copying. Finally, a message is displayed that indicates how many records were copied based on the BatchMove's *MovedCount* property.

When the *Mode* property is set to *batCopy*, BatchMove creates the destination table if it does not already exist. (This is also true when *Mode* is set to *batAppend*, despite what it says in the on-line help description.) If the destination table exists, it's replaced by the new table when *Mode* is set to *batCopy*, and added to it if *Mode* is set to *batAppend*. In each case where *Mode* is set to *batCopy*, the destination table is not keyed. To apply an index to this table, you must use the *AssignIndex* method of the *TTable* class.

Under normal conditions, all records in the source DataSet are copied to the destination (see the discussion of table ranges in this article). There may be times that you want to place an upper limit on the number of records that BatchMove can process. For example, if the source DataSet has the potential of containing a large DataSet (say, in the millions of records), and the user can request that these records are processed by BatchMove, you may want to limit the number of records to be copied. You can do this with the *RecordCount* property.



```

procedure TForm1.Button1Click(Sender: TObject);
begin
    if Query1.Active = False then
        Exit;
    if SaveDialog1.Execute then
        begin
            Table1.TableName := SaveDialog1.FileName;
            with BatchMove1 do
                begin
                    Source      := Query1;
                    Destination := Table1;
                    Mode        := batCopy;
                    Execute;
                    ShowMessage(IntToStr(MovedCount) +
                        ' records copied');
                end;
            end;
        end;
end;

```

Figure 2 (Top): The BATDEMO project. This project permits a user to enter a SQL SELECT statement, execute it, and then copy the results to a new table. Figure 3 (Bottom): The *OnClick* event handler for the **Copy Result to Table** button.

When *RecordCount* is set to its default value, 0 (zero), all records referred to by the source DataSet are processed by the BatchMove's *Execute* method. When you set this property to any positive integer, that number identifies the maximum amount of records BatchMove will process during any one call to its *Execute* method.

Using a Table as a Source

While the BatchMove *Source* property is often assigned a Query component, it is also useful to assign a Table component to this property. Several characteristics of such a table are important in BatchMove's execution, including index and range.

Either a primary or secondary index defines the ordering of records in a table component. For certain types of BatchMove operations, such as *batDelete*, *batUpdate*, and *batAppendUpdate*, it's necessary to assign an index to the source table, and a corresponding index to the destination table. This permits BatchMove to match the records in this table for deleting or updating the destination table records. Any index defined for the source table when *batCopy* is the mode, defines the order of records in the destination table.

A range defines a restriction on the records available in a table. For example, even though a table may contain all sales records for the past five years, it is possible to define a range that includes only those records from a single year. You can apply a range using either the Table method, *SetRange*, or the Table methods: *SetRangeStart*, *SetRangeEnd*, and *ApplyRange*.

When a range is defined for the source table of a BatchMove operation, only those records in the defined range are processed. If the *Mode* is *batCopy*, for instance, only the records in the defined range are copied to the destination table. Consider this example: If you have defined your Sales table as the source table and set a range to include only records from a given year, only records that lie within the range will be copied to the destination table.

The use of BatchMove where the source table has an applied range is demonstrated in the project BATRDEMO (see Figure 4). This table is the Employee table from the alias, DBDEMOS. This alias points to a directory of example data installed when a complete installation of Delphi is performed.

When a range is applied to this table and the **Move Records** button is clicked, only records in the currently selected range are copied to the destination table. The code that defines the range is shown in Figure 5, and the following code copies the records:

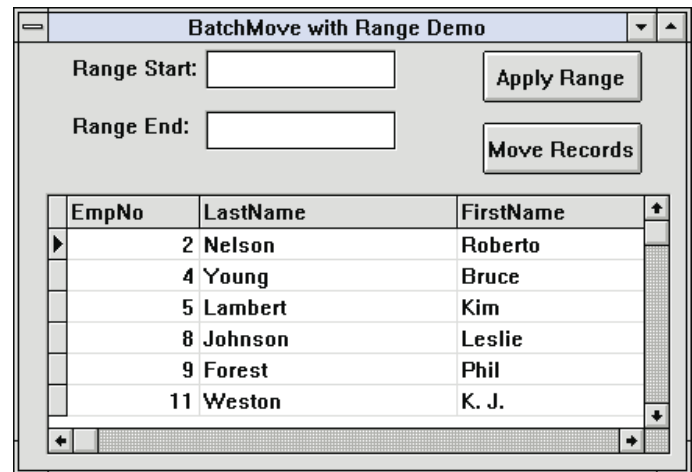
```
if SaveDialog1.Execute then
begin
  Table2.Tablename := SaveDialog1.FileName;
  BatchMove1.Execute;
end;
```

Deleting Records

Although not obvious by its name, BatchMove can delete records from the destination table. Both source and destination tables must use a compatible index, and the *Mode* property must be set to *batDelete*. Under these conditions, any record in the destination table with indexed field values matching those of the source table record will be deleted when BatchMove is executed.

This feature is useful for archiving records. For example, imagine you want to keep only the current year's sales in the Sales table. Once a year it's necessary to copy last year's records to an archive table, and then remove them from the Sales table. This is easily done by following these steps:

- 1) Assign a Table component to the Sales table.
- 2) Set a range on the Sales table to include only those records prior to the current year.
- 3) Assign the Sales table component to BatchMove's *Source* property.
- 4) Assign another Table component to the archive table.
- 5) Set the BatchMove *Mode* property to *batAppend*.
- 6) Call the BatchMove component's *Execute* method.
- 7) Remove the range from the Sales table component (calling *SetRangeStart*, *SetRangeEnd*, and *ApplyRange* without defining a range removes a range).



```
with Table1 do
begin
  EditRangeStart; { Set the beginning key }
  Fields[0].AsInteger := StrToInt(Edit1.Text);
  EditRangeEnd; { Set the ending key }
  Fields[0].AsInteger := StrToInt(Edit2.Text);
  ApplyRange; { Tell the dataset to establish
              the range }
end;
```

Figure 4 (Top): This project permits you to define a range on the employee number field to restrict which records are referred to by the table. After applying a range, using BatchMove to copy records from this table will copy only those records in the defined range. **Figure 5 (Bottom):** Defining the range.

- 8) Assign the Archive table component to the *Source* property.
- 9) Assign the Sales table component to the *Destination* table property.
- 10) Set the *Mode* property to *batDelete*.
- 11) Call BatchMove's *Execute* method.

Moving Records to an Existing Table

Using the *Mode*, *batAppend*, you can add records from a DataSet to an existing table. This operation is more complex than simply creating a new table, since the records added must conform to the rules defined for the destination table. For example, if the destination table is keyed, there may be one or more records in the source table with key field values that duplicate those in the destination table. Also, a source table record may contain values that conflict with table integrity rules (e.g. required fields), or other data rules (e.g. minimum or maximum acceptable values).

The BatchMove component has a number of methods that enable you to control what should happen if a problem occurs during execution. Using the *AbortOnKeyViol* and *AbortOnProblem* properties, you can instruct BatchMove to terminate operation if a corresponding problem is encountered. Using the properties, *KeyViolTableName* and *ProblemTableName*, you can enter the name of a Paradox table that BatchMove will create if a problem does occur.

Problem records encountered during BatchMove's execution are placed in one of the tables specified by the *KeyViolTableName*

and *ProblemTableName* properties. For example, any records that would generate a key violation (attempted duplication of destination key field values) are placed in the table specified by the *KeyViolTableName* property. If *AbortOnKeyViol* is set to *True*, one record appears in this table if a key violation is encountered. If *AbortOnKeyViol* is set to *False*, all records that would produce a key violation are placed into this table.

Updating Tables

While *batAppend* permits you to insert records into an existing table (but only if records using the same key do not already exist in the destination), *BatchMove* also provides two modes that permit you to update records in the destination table based on values stored in the source. These modes are *batUpdate* and *batAppendUpdate*. Using *batUpdate*, any records in the destination table whose current index values exactly match those same index values in the source table will have non-indexed field values updated by those corresponding fields in the source table. Using this mode, source table records with no corresponding records in the destination table are ignored. The index may be either primary or secondary.

The *batAppendUpdate* mode works similarly to the *batUpdate* mode, with one important difference. Records in the source table with no corresponding records in the destination table are appended to the destination. Therefore, when the mode is *batAppendUpdate*, all source table records either update existing records, or are added to the destination table.

Unlike the *batAppend* mode, there are no key violations when the *batAppendUpdate* or *batUpdate* modes are used. However, problem records are still possible.

Following the execution of *BatchMove* using either the *batAppendUpdate* or *batUpdate* modes, you can use the *ChangedCount* property to determine how many records were updated. Furthermore, if you assign a value to the *ChangedTableName* property, a copy of each record with a changed value is placed into a Paradox table of that name.

Mapping Tables

It is possible to use *BatchMove* when source and destination tables do not have identical structures. By default, *BatchMove* attempts to add or update records between the source and destination on a field-by-field basis. If the field types do not correspond exactly, a best-fit is attempted. For example, if the third field of a destination table is a seven-character text field, and the same field of the source table is 10 characters, values longer than seven characters are truncated when the move is executed.

BatchMove also allows you to move only a subset of fields or data between fields that are not in corresponding positions within the source and destination tables. This is done using the *Mappings* property.

Mappings is a *StringList* property. It can contain a list of text values that can take two forms. The first form is to include in each element of *Mappings* the field name you want to process as part

of the move. For example, regardless of how many fields there are in the source and destination tables, if you only want to move the values stored in the source table field named *Account_Number* to the destination table field named *Account_Number*, add the text *Account_Number* to the *Mappings* property. Here's an example:

```
with BatchMove1 do
begin
  Source      := Table1;
  Destination := Table2;
  Move        := batAppend;
  Mappings.Clear;
  Mappings.Add('Account_Number');
  Execute;
end;
```

The second form you can use with the elements of the *Mappings* list is to define field-name pairs. These field names appear separated by an equals sign (=). There is a destination field name on the left-hand side of the equals sign and a corresponding source table name on the right. Data is only transferred between fields when field-name pairs are provided. The code in [Figure 6](#) moves only those fields of the source table specified by the names on the right of the equals sign to the destination fields on the left.

```
with BatchMove1 do
begin
  Source      := Table1;
  Destination := Table2;
  Move        := batAppend;
  Mappings.Clear;
  Mappings.Add('Student_ID=ID');
  Mappings.Add('Last_Name=LNAME');
  Mappings.Add('First_Name=FNAME');
  Mappings.Add('Address=STREET');
  Mappings.Add('City_CITY');
  Mappings.Add('State/Prov=STATE');
  Mappings.Add('Country=ORIGIN');
  Mappings.Add('Postal_Code=ZIP');
  Execute;
end;
```

Figure 6: Using *Mappings* with field-name pairs.

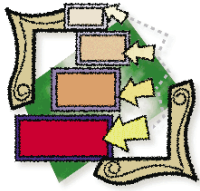
Conclusion

The *BatchMove* component is a remarkably flexible tool for transferring and updating data between tables. It's also an essential tool for creating permanent tables from the result sets returned by *SELECT* queries and stored procedures. ▲

The demonstration projects referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\NOV\CJ9511.ZIP.

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





VISUAL PROGRAMMING

DELPHI / OBJECT PASCAL



By *Douglas Horn*

The ObjectBrowser

An Introduction and Visual Tour

Imagine getting a fantastic deal on a new car. Then you discover that not only does it handle like a dream, it also sports a computerized map. The salesman didn't tell you it was there, the owner's manual only mentions it in passing, and you didn't pay extra for it. The only hitch is, no one tells you how to use it.

If this scenario sounds far-fetched, consider the Delphi ObjectBrowser: a powerful tool that enables you to inspect applications, as well as Delphi and Windows (see Figure 1). Unfortunately, Delphi's documentation on the ObjectBrowser is sparse. This article will demonstrate the ObjectBrowser's various features and show you how to use it.

Symbols

The ObjectBrowser works with symbols. For its purposes, a *symbol* is just about anything in a Delphi application. For example, a form, a button on that form, and a property of that button, are symbols the ObjectBrowser can examine. Anything Delphi can read, write, or execute — a constant, variable, types, property, procedure, etc. — is a symbol.

Developers can choose from three main categories of symbols to search with the ObjectBrowser: objects, units, and globals:

- Objects include those objects used within a Delphi application (e.g. forms, controls, procedures, and variables).
- Units consist of the class libraries that make up objects.
- Globals are symbols that exist throughout the current application, or outside of it, in Delphi or the Windows API.

The ObjectBrowser can explore these symbols, provided that certain conditions exist (discussed later in the article). This makes the ObjectBrowser a powerful tool for understanding the applications we develop with Delphi, as well as Delphi and Windows.

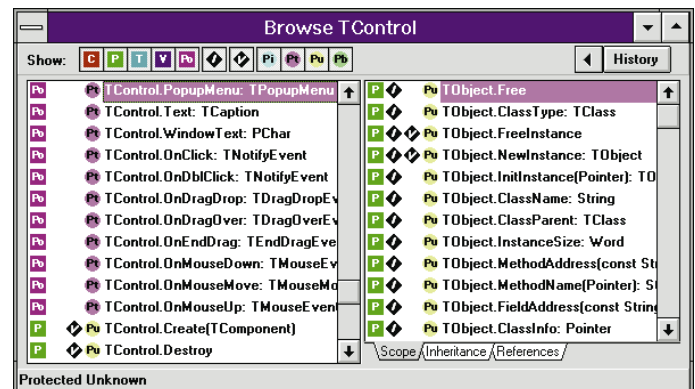


Figure 1: Delphi's ObjectBrowser allows developers to inspect objects from their applications, Delphi, and Windows.

The ObjectBrowser

Before exploring the ObjectBrowser, it's important to understand its parts. In many respects, the ObjectBrowser is a simple tool. The interface consists of a main screen, two dialog boxes, a SpeedMenu, and a few other controls.

The ObjectBrowser's primary control is the Inspector pane on the left side of the window (see Figure 2). It lists the symbols in the area the ObjectBrowser is currently inspecting. (The area's name appears on the Title bar.) Double-clicking on a symbol causes the ObjectBrowser to inspect the represented object, and moves the ObjectBrowser one step further down the inheritance tree.

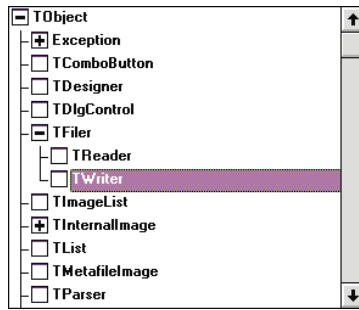


Figure 2: The Inspector pane lists information about the object being inspected.

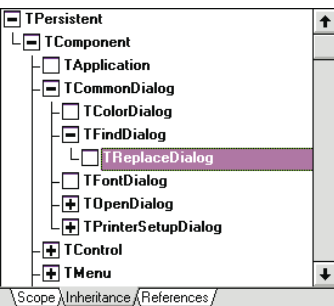
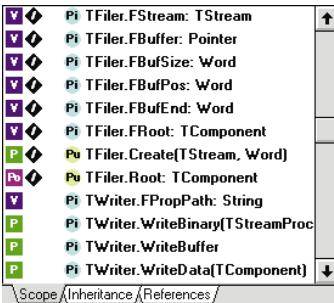


Figure 3 (Top): The Scope page of the Details pane shows all the symbols associated with the symbol in the Inspector pane. **Figure 4 (Bottom):** The Inheritance page of the Details pane displays a collapsible tree of object hierarchies.

compiled with the **Symbol info** parameter as we'll discuss). This page offers a direct link to the symbol within the code. Simply select the reference in the Details pane and press **[Space]** to highlight the corresponding line in the Code Editor, or double-click on the reference to move there directly.

The ObjectBrowser can show many types of symbols: Constants, Procedures, Types, Variables, and Properties. It can show statements from the Private, Protected, Public, and Published sections of the code (if this is enabled on the

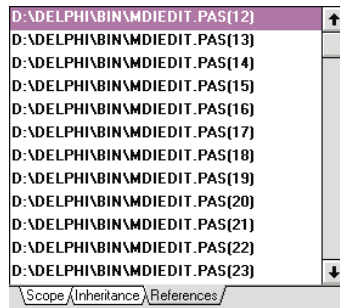


Figure 5: The References page of the Details pane shows where symbols appear in program code.

To the right of the **Show** buttons are navigation controls for returning to previous symbols: the **Back** and **History** buttons.



Figure 6: The 11 **Show** buttons control the types of symbols the ObjectBrowser will display. From left to right, they are: Constants, Procedures or Functions, Types, Variables, Properties, Inherited, Virtual, Private, Protected, Public, and Published.

Clicking the **Back** button (the left-pointing arrow) moves the focus of the Inspector pane to the last object inspected — usually one step back up the inheritance tree. Clicking the **History** button calls a list of previously inspected objects (see Figure 7). Double-clicking on any entry automatically moves the Inspector pane back to that object.

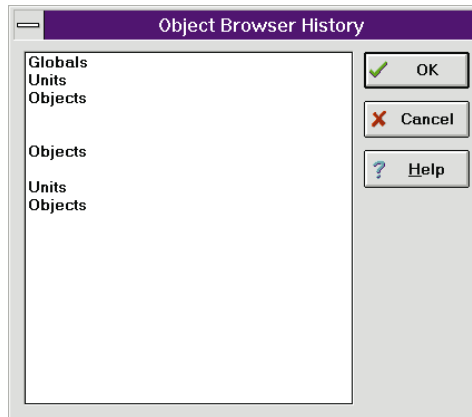


Figure 7: The **History** list allows the user to quickly return to previously browsed objects.

The status bar of the ObjectBrowser is referred to as the **Info Line**. It displays information about the current object in the Inspector pane or in the Scope page of the Details pane, depending on which is active.

The SpeedMenu

The ObjectBrowser has no main menu, but uses a simple SpeedMenu of eight commands (see Figure 8). Like most SpeedMenus, it is accessed by right-clicking anywhere within the ObjectBrowser window, or by pressing **[Alt][F10]**.

Compiler page of the Project Options dialog box).

The **Show** buttons at the top left of the ObjectBrowser control which symbols are shown (see Figure 6). By selecting or deselecting a button, a filter is applied to both the Inspector pane and the Scope page of the Details pane. This allows users to quickly search for the various types of symbols.

Objects	Ctrl+O
Units	Ctrl+U
Globals	Ctrl+G
Symbol...	Ctrl+Y
Qualified symbols	
Sort always	
<input checked="" type="checkbox"/> Show Hints	
<input checked="" type="checkbox"/> Info Line	

Figure 8: The ObjectBrowser's SpeedMenu.

The first three commands control the types of information displayed in the Inspector pane. As shown in Figure 9, **Objects** displays an inheritance tree for the application being browsed. The inheritance tree allows the developer to inspect the overall structure of the various components, procedures, and properties in an application.

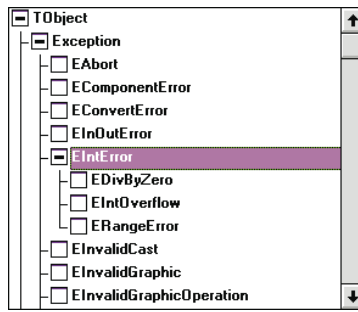


Figure 9: The **Objects** SpeedMenu item displays an inheritance tree in the Inspector pane.

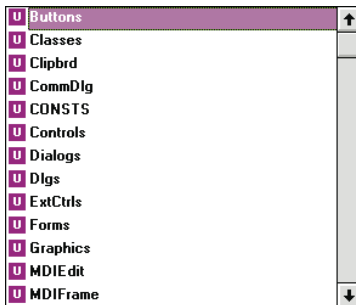


Figure 10: The **Units** SpeedMenu item lists all the application's units in the Inspector pane.

Selecting the **Units** command lists all the application's units in the Inspector pane (see Figure 10), while selecting the **Globals** command displays the global, Delphi, and Windows API symbols used in the current application (see Figure 11).

The **Symbol** command has the same effect as the **Search | Browse Symbol** command from the Delphi main menu. It allows the user to search for a specific symbol throughout the current application.

The next two SpeedMenu commands affect the appearance of items in the Inspector and Detail panes. When **Qualified symbols** is enabled, it displays each symbol with its qualified identifier (i.e. its more complete name in dot notation). For example, the symbol `ClassName:String` with its qualified identifier is `TObject.ClassName:String`. Although this notation occupies more space, it is easier to understand.

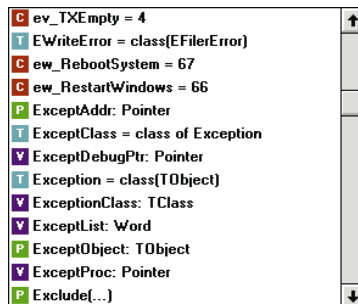


Figure 11: The **Globals** SpeedMenu item displays the global, Delphi, and Windows API symbols used in the current application.

When checked, the **Sort always** command sorts symbols alphabetically. If unchecked, the symbols are listed in the order they are declared in the application. If both **Qualified symbols** and **Sort always** are enabled, the symbols will be sorted by their names and not their qualified identifiers. Therefore, `TObject.ClassName:string` will come alphabetically before `Exception.Create(const string)` because the qualified identifiers `TObject` and `Exception` are ignored.

The last two SpeedMenu commands are **Show Hints** and **Info Line**. These control the behavior of ObjectBrowser's "helping hands". When **Show Hints** is activated, resting the mouse cursor

on one of the **Show** buttons causes a hint message to appear. When **Info Line** is selected, the Info Line is visible. Deselecting it causes it to disappear.

Enabling and Configuring the ObjectBrowser

For the ObjectBrowser to work properly, certain Delphi program options must be selected. These are found on the Project Options dialog box. To access them, select **Options | Project** from the Delphi menu, then select the Compiler page (see Figure 12).

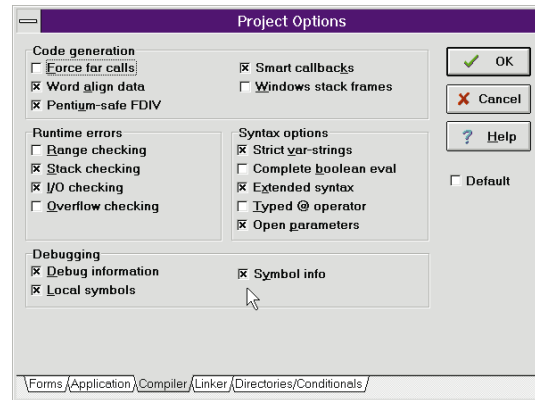


Figure 12: To enable the full use of ObjectBrowser, select **Debug Information**, **Local symbols**, and **Symbol info** on the Compiler page of the Project Options dialog box.

In the **Debugging** group there are three options: **Debug information**, **Local symbols**, and **Symbol info**. To get all the information the ObjectBrowser can provide, select all three options.

Debug information allows the ObjectBrowser to browse symbols that are declared in the **implementation** section of the program modules. If this option is not selected, ObjectBrowser only sees symbols from the **interface** part. **Local symbols** allows the ObjectBrowser to read those symbols declared locally within program routines (the **implementation** part).

When **Symbol info** is not selected, the ObjectBrowser sees only symbols declared in the **interface** part of the module. (It still sees other symbols in the **implementation** section, provided they are not declared there.) When enabled, the **Symbol info** option allows the ObjectBrowser to display line numbers in the References page of the Details pane, and jump directly to the selected line of code while browsing an object. However, neither **Local symbols** nor **Symbol info** have any effect unless the **Debug Information** option is selected.

Note: Delphi's on-line help is confusing in this regard. Under the topic "Enabling the ObjectBrowser," it mistakenly reverses the definitions for **Local Symbols** and **Symbol info**.

Using the debugging options increases the size of compiled .DCU files, but is almost always worthwhile for its debugging value. Turning the options off shrinks the re-compiled .DCU file. The debugging options have no effect on the size of executable (.EXE) program files.

Since the debugging information is stored in the project's .DCU file, once the debugging options are selected, the project must be compiled or re-compiled for them to take effect. The

ObjectBrowser receives its information from this file. Therefore, if the current project has not been compiled, the ObjectBrowser will not run, and the **View | Browser** Delphi menu option is disabled.

The ObjectBrowser options offered on the SpeedMenu can be configured via the Environment Options dialog box. From the menu, select **Options | Environment** and the Environment Options dialog box is displayed. Choose the Browser page (see **Figure 13**). These settings control the default appearance and behavior of the ObjectBrowser.

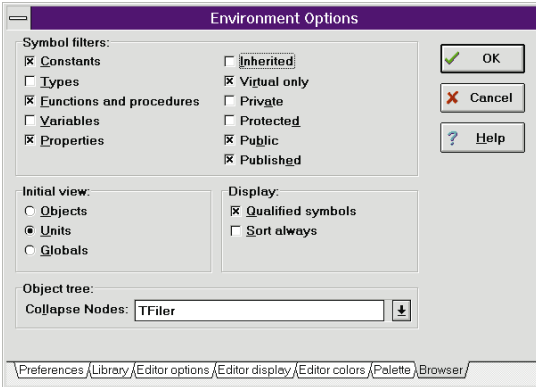


Figure 13: Use the **Browser** page of the **Environment Options** dialog box to configure the **ObjectBrowser**'s default appearance and behavior.

The **Symbol filters** group contains 11 check boxes that correspond to the 11 **Show** buttons. The check boxes selected here will appear as depressed **Show** buttons in the ObjectBrowser. The **Initial view** group of radio buttons controls what will be shown in the Inspector pane at opening: **Objects**, **Units**, or **Globals**. Likewise, the **Qualified symbols** and **Sort always** options in the **Display** group have SpeedMenu commands.

Finally, there is the **Object tree** group. Entering an object name (or series of object names separated by semi-colons) into the **Collapse Nodes** field instructs the ObjectBrowser to initially collapse those nodes on the object hierarchy tree.

The options selected from the Environment Options dialog box take effect only after the ObjectBrowser is re-opened. The exception is the **Display** group — **Qualified symbols** and **Sort always**. These commands take effect immediately when the ObjectBrowser is already running.

Using the ObjectBrowser

This brings us to the fun part — using the ObjectBrowser. Its usefulness is nearly limitless and becomes more helpful as your developer skills increase.

The most common way to use the ObjectBrowser is to start at the Code Editor and select **Search | Browse Symbol** from the menu. This is useful for finding all other incidences of a particular constant, variable, or procedure. For example, when working with a variable, it can be very convenient to find all the lines dealing with that variable (from the time it was declared to the present).

By searching for the symbol and then selecting the Detail pane's References page, the developer can view all files and line numbers where the variable appears. This even allows direct access to

those lines. Note however, that if a reference is modified to no longer occur on the same program line, the ObjectBrowser cannot move directly to it again until the project is re-compiled.

Here's another possibility: If you want to figure out exactly what's going on with some aspect of your program — say the Popup Menus — select **Objects** from the ObjectBrowser's SpeedMenu to get an object hierarchy tree. Now move through the levels from *TObject* to *TPersistent* to *TComponent*, *TMenu* — all the way to *TPopupMenu*. With *TPopupMenu* selected, choose the References page of the Details pane and go to the first reference listed. This should be the declaration statement in the **type** section of the code. If so, it lists the name corresponding symbol (e.g. `PopupMenu1: TPopupMenu;`). From here, browse the symbol directly and repeat it for each *TPopupMenu* reference until you find what you're looking for.

The ObjectBrowser is also an excellent self-guided learning tool that takes you as deep into the inner-workings of Delphi and the Windows API as you dare go. For example, this allows a developer to find the values of all the Delphi and Windows API constants. Such knowledge can be used for silly things, like rendering source code practically unreadable to other programmers by replacing commonly-used constants with their actual numerical values. (It works, but why bother?)

It's useful for more practical purposes, as well. As an example, select **Units** from the SpeedMenu and select **Controls** in the Inspector pane. Then search for the values of the cursor constants. (Typing `CR` takes you right there.) The various cursor constants are listed (e.g. `crArrow`, `crDefault`, `crNone`, etc.). Funny, `crNone` never appears in the Object Inspector as a possible listing.

Now try this: Create a one-form project and instead of selecting one of the Object Inspector's listed constants for the *Cursor* property, enter -1, the value found for `crNone` using the ObjectBrowser. (Typing `crNone` returns an error.)

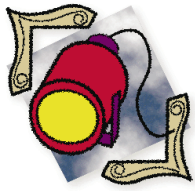
When you run the form, the cursor is invisible. This capability isn't apparent from the Object Inspector, and who's to say you won't need it some day?

Conclusion

This is only a small example of what you can find using the ObjectBrowser. (`crNone` is a documented constant, but difficult to find in the documentation.) Who knows what other hidden pearls lurk in the depths of Delphi? One thing's for sure — the ObjectBrowser is the way to find them. ▲

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





INFORMANT SPOTLIGHT

DELPHI / OBJECT PASCAL / WINDOWS API



By *Tom Costanza*

One Bit at a Time

A Serial Communications Primer and Delphi Implementation Guide

A serial port is a cheap and easy way to connect two or more electronic devices. Every PC comes with at least one serial port. Most printers allow for serial input, and an infinite number of bulletin board services are accessible via a serial port.

The serial port is also used to transfer data from computer to computer with programs such as LapLink. In an industrial setting, a PC can communicate with dedicated industrial automation equipment.

Support for Serial Communications: DOS vs. Windows

When Microsoft wrote DOS, they chose to include only minimal support for the asynchronous serial port. Calls to DOS and the BIOS are provided to initialize the port, and read and write single characters. However, there is no support for interrupt-driven buffered communication. Programmers are forced to buy a third-party library, or write their own interrupt service routines. And having written interrupt service routines, I can tell you that it's no day at the beach. (For more information about interrupts, see the sidebar "[What Is an Interrupt?](#)" on page 36.)

Microsoft mercifully corrected this omission with Windows' feature-rich API (which we'll discuss later). We'll also look at a simple technique that enables the programmer to read (on demand) any characters in the receive buffer, as well as send a string of characters.

In addition, we'll cover a slightly more complicated technique that enables the programmer to write an event handler to respond to interrupts from the serial port. These interrupts can be programmed to occur for a number of reasons, including: a character has been received and needs to be processed; a modem status signal has changed state (e.g. the modem has answered the phone); and, the transmit buffer is empty.

To understand the Windows API for communications functions, you must first understand how serial communications is accomplished. Therefore, we'll discuss generic serial communications, the PC's implementation of asynchronous serial communications, and the Windows API. Finally, we'll talk about how to implement all this in Delphi.

Serial Communications: A Primer

There are three generic types common to all forms of communications: simplex, half-duplex, and full-duplex:

- *Simplex* communication is one-way only. An example of simplex communication is television.

While you can see and hear the characters on a television show, they can't hear or see you.

- *Half-duplex* is two-way communication, but in only one direction at a time. An example of half-duplex communication is the two-way radios used by police departments. The dispatcher talks to an officer in a patrol car. Then, the dispatcher waits for the officer's response.
- *Full-duplex* means both ends of the conversation can talk at the same time. A telephone is an example of full-duplex communication.

Serial communications is accomplished by varying the voltage on a wire. The voltage can be one of two levels. The level for a "1" or "MARK" is defined as anything between -5 and -15 volts, with -12 volts being nominal. A "0" or "SPACE" is defined as anything between +5 and +15 volts, with +12 being nominal.

If a transmitter were to just start varying the voltage on the line, the receiver would soon become confused about where one character ends and the next character begins. What is needed is a way to synchronize the receiver with the transmitter. There are two techniques for this: synchronous and asynchronous communication. Synchronous communications is outside the scope of this article. *Asynchronous communications* is accomplished by re-synchronizing the receiver each time a character is received. This is accomplished by framing each character with *start* and *stop* bits.

Start, Stop, and Data Bits

With asynchronous communications, a character is sent by making the wire sending data from the transmitter to the receiver change from a 1 (MARK or idle state), to a 0 (zero or SPACE). This is known as the *start bit*. Every character sent begins with a one-to-zero transition. Data bits are then sent, followed by one or more stop (MARK) bits. A *stop bit* is created by letting the line remain idle, or quiescent, for 1, 1.5, or 2 bit times. A *bit time* is the time it takes to transmit one bit.

By letting the line remain idle for a period, and since a start bit is defined as the transition from idle to a MARK state, it is readily apparent to the receiver when a start bit has been received. When the receiver sees the one-to-zero transition, it waits for half a bit time and looks at the data line again. If the line is still zero, the receiver assumes a start bit has been received. If not, it assumes noise caused the transition, and waits for another one-to-zero transition. If a start bit has been received, the receiver samples the line once every bit time.

The data line's state at the time of the sample is the next bit's value. The receiver can maintain synchronization for a short period of time (the time it takes to receive one character) by sampling the data line once every bit time. If the sampled voltage is positive, the bit is a zero. The bit is a one if the sampled voltage is negative.

Figure 1 illustrates the transmission of a single character. For simplicity, logic levels rather than voltage levels are shown. Note that

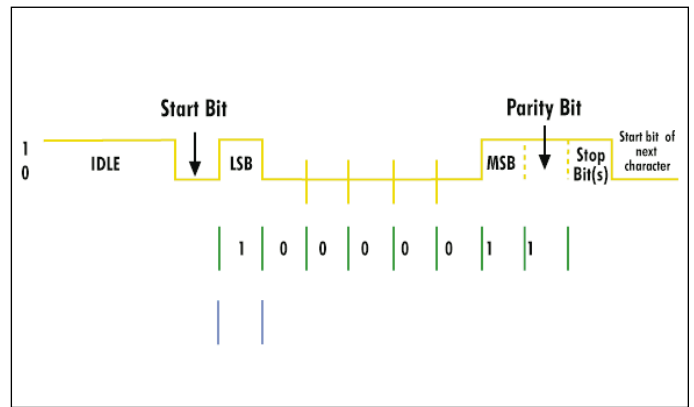


Figure 1: The character "A" with odd parity.

the least significant bit is sent first and the parity bit is sent last. Each time a new start bit is received, the receiver re-synchronizes.

Baud Rate

Baud rate is loosely defined as the transmission facility's signaling rate. For the purposes of this article, baud rate and bits-per-second represent the same thing. Therefore, this article will use the term baud rate to refer to the number of bits-per-second transmitted from, or received by, the computer's serial port.

A common baud rate today is 9600. This means that bits are transmitted at a rate of 9600 bits-per-second. Note that this does not mean that every second 9600 bits are transmitted. It means that one bit takes 1/9600 seconds (about 104 microseconds) to transmit.

For the receiver to sample the data line at the correct times, the receiver and transmitter must agree on the baud rate. Therefore, if 9600 baud is used, the receiver will sample the data line every 104 microseconds.

Parity

The ASCII character set contains 128 different characters. To have 128 unique codes, seven bits must be used ($2^7 = 128$). When this data is being transmitted over phone lines with modems, noise on the phone line can often change a 1 to a 0, or vice versa. While the noise can't be eliminated, the errors caused by the noise can be detected.

One technique for detecting these errors is to send an eighth bit, called a *parity bit*. The transmitter and receiver must agree as to what this parity bit will be. (This is why BBS systems say you must set your communications port to a specific parity.) The agreement is this: the transmitter will send an even or odd number of 1's per character. It's inconsequential if the parity is even or odd, as long as both sides agree.

Let's assume odd parity is used. When a character is sent by the transmitter, the transmitter sets the eighth bit, the parity bit, as appropriate, to force an odd number of 1's to be sent. For example, the character "A" has a binary value of 1000001. This value has an even number of 1's, so the transmitter sends a 1 for the

parity bit, making the total number of 1's odd. The character "C" has a binary code of 1000011. This code has an odd number of 1's, so the transmitter sends a 0 for the parity bit — leaving the total number of 1's odd.

At the receiver, the number of 1 bits are counted and compared to the agreed parity. Suppose that noise on the phone line changes the code for the character "C" from 1000011 to 1010011 (i.e. the third bit has changed from 0 to 1). As stated in the previous paragraph, the parity bit is 0. The receiver has received four bits that are set to 1. Since four is an even number, and there's agreement the number of bits set to 1 *must be odd*, the receiver knows an error has occurred.

If the number of 1 bits is ever even (when an odd parity was agreed upon), the receiver knows something is wrong, and generates a parity error. It should be obvious by now that this technique works only if no errors occur, or the number of bits that are inverted is an odd number (i.e. one bit is inverted, or three bits are inverted, etc.).

If an even number of bits are inverted, the parity will still be correct, even though the character received is not the character sent. It should also be apparent that it makes no difference whether a data bit is inverted, or the parity bit itself is inverted in transmission. This form of error detection is crude at best. For reliable data exchange, a more sophisticated method can be used.

Parity is only an option (on PC-style hardware) when seven data bits (or fewer) are used. When anything but text (e.g. a program, or a Paradox data file) is sent serially, eight data bits are needed, because each byte in the data can contain a value from 0 to 255.

The serial port in PC-style hardware will not let you send more than eight bits per character. If you use all eight bits for data, there are no bits left to use for parity. So, if eight data bits are specified, parity is set to "NONE". This is not much of a loss since, as discussed earlier, character-by-character parity checks are not of much value.

For completeness, note that there are two other types of parity: mark and space. With *mark* parity, the parity bit is always a mark (1), and with *space* parity, the parity bit is always a space (0).

Flow Control and Handshaking

Different serial devices operate at different speeds. For example, a computer can send characters to a printer at a rate far greater than the printer's ability to print those characters. Therefore, it's usually necessary to provide some way of telling the transmitter that the receiver cannot currently receive any more characters, and the transmitter should suspend transmission until further notice.

There are two commonly used techniques for this. *Hardware* flow control uses voltage levels on one or more lines of the RS-232 interface. For example, the serial port can be configured so the *clear-to-send* (CTS) line is monitored to determine if it's acceptable to transmit characters. If this line is set to one, the

What Is an Interrupt?

An interrupt is a piece of hardware's request for attention from the processor. Without interrupts, the programmer must poll the device to determine if the device needs attention. Here's an analogy: Suppose you're at home reading a book and you want to know if anyone comes to the front door. One way would be to put down the book every minute or so, and open the door to see if anyone is there (polling). This obviously wastes time, since usually no one is there. A more efficient way would be to install a door bell. Now you can keep reading your book until you hear the bell ring (interrupt). You can then mark your place in the book and answer the door only when needed.

When an interrupt occurs, program execution is suspended, and the processor is directed to an *interrupt service routine*, a special subroutine written to deal with the interrupt. When this routine has completed, the normal program execution is resumed.

Anyone who has installed an interface card in a PC has had to deal with interrupts. For serial ports, an interrupt may indicate that a character has been received and must be processed, the transmit buffer is empty, or some modem status signal has changed state. The programmer usually has control over whether an interrupt will occur. If the programmer has enabled interrupts, then the next question to be answered is this: What is allowed to cause the interrupt? For example, a program might want to be interrupted when a character is received, but not if the transmit buffer is empty. This also is programmable.

In the PC there are *interrupt levels* that denote the priority of an interrupt. The higher the number, the lower the priority. Serial port interrupts have a fairly high priority. While reading from a disk can be postponed for a second or two, a character received at the serial port must be retrieved before the next character is received or the second character will overwrite the first. At 9600 baud, a character can be received approximately every 960 microseconds.

transmitter can send characters. If not, the transmitter should suspend transmission.

This technique is sometimes used for half-duplex communication, where one device tells another device, "I have finished transmitting. Now I will receive and you can transmit." It's also used when the two devices are connected by a "hard" cable (a continuous piece of copper from one end of the cable to the other). Modems and phone lines don't qualify. It can also be used with simplex communication. For example, the connection between a computer and a serial printer can use this type of flow control.

On the other hand, if you are sending data over a modem to another computer, and if full-duplex communication is possible, another form of flow control is usually employed. With this technique, when computer A's receiver can't accept any more characters from computer B, computer A's *transmitter* sends a special charac-

ter (called *XOff*) to computer B. When computer B sees this character, it suspends transmission. Later, when computer A can again accept characters, its transmitter sends another special character (called *XOn*) to computer B. When computer B sees the XOn character, it knows to begin sending data again. This is called *XOn/XOff* flow control (the X is an abbreviation for *transmit*).

The PC's Serial Port

The original PC had an optional internal card known as the IBM Asynchronous Communications Adapter that enabled connection to modems and other serial devices. Although this serial interface has evolved with the PC, and is now usually included on the motherboard, the software interface has remained the same for compatibility reasons.

There is a block of I/O address space reserved for the asynchronous communications adapters. The addresses range from \$3F8 to \$3FF for COM1, and \$2F8 to \$2FF for COM2. A set of registers in this address space allow the programmer to configure the port, send and receive characters, and read and write modem control signals and status bits. (A *register* can be thought of as a single memory location, although it's separate from the computer's RAM.)

Most of these control and status bits are available through the Windows API. For example, the programmer can tell when the DSR line changes state, but cannot read its current value. (*DSR* is an acronym for Data Set Ready. A complete listing of signals used in serial communications is shown in [Figure 2](#).) The DSR signal is sent by a modem to its attached computer to indicate it's ready to operate.

To obtain this signal's current value, the programmer can access the modem status register directly, and many DOS programmers will be tempted to do just that. This is a bad idea, since one of Windows' strengths is its device independence. If the vendor of a new serial port uses some new hardware, that vendor can supply a new device driver for the port, and your old program won't need to be changed. If however,

Signal	Description	Pin Number on 25-Pin Connector	Pin Number on 9-Pin Connector
TxD	Transmit Data	2	3
RxD	Receive Data	3	2
RTS	Request To Send	4	7
CTS	Clear To Send	5	8
DSR	Data Set Ready	6	6
SG	Signal Ground	7	5
DCD	Carrier Detect	8	1
DTR	Data Terminal Ready	20	4
RI	Ring Indicator	22	9

Figure 2: Signals used in serial communications with cable-connector pin numbers.

you let your program talk directly to the hardware, you may need to change it if the hardware changes.

In Object Pascal (as well as earlier versions of Pascal) the command to read or write to an I/O register is *Port[x]* for byte values, and *PortW[x]* for 16-bit (word) values, where *x* is the port address to be read or written. (This syntax is well-noted in the *Turbo Pascal User Guide*, but I have not found it in the Delphi documentation.) The communications adapter registers are 8-bits (1 byte) wide, so the command to read the modem status register at address \$3FE is:

```
val = Port[$3FE]
```

The DSR bit is bit 5. To isolate that bit, we must perform an AND operation with the value read from the register with a mask that has a 1 at bit 5, and 0's at all other bit locations. In binary, this is 00100000; in hexadecimal it's \$20. Thus the statements:

```
val := Port[$3FE];
val := val AND $20;
```

will set *val* to 0 if DSR is low, or 32 (\$20) if DSR is high. (For more information about AND operations, see the sidebar "[Masking Bits with the AND Operator](#)" on page 38.)

A better way (that is undocumented in the Windows Software Development Kit) makes use of an extended *device control block* (or DCB). Within this extended DCB is a field called *MSRShadow* (modem status register shadow). This is a copy of the modem status register on the UART. MSRShadow is located 35 bytes past the event word for the desired port. The event word's address is returned by the *SetCommEventMask* function (which we'll discuss later).

The following Object Pascal code reads the MSRShadow word:

```
procedure TForm1.GetMSR;
var x: PWord;
    z: byte;
begin
    { Get pointer to event word. }
    x := SetCommEventMask(hCommPort,constCommEvents);
    { MSRShadow byte is 35 bytes past this pointer. }
    word(x) := word(x) + 35;
    { Get MSRShadow byte and display it. }
    z := x^;
    Edit2.Text := intToHex(z,2);
end;
```

Again, this code isolates the DSR bit:

```
z := z AND $20;
```

(Additional information on the asynchronous communications adapters and register addresses can be found in the *IBM Personal Computer Technical Reference Manual*.)

Windows API Functions

Delphi can communicate with the serial port through the

Masking Bits with the AND Operator

Good programmers won't waste a whole byte of memory to store a logical (yes/no, on/off, etc.) value. Only one bit is required. This means a programmer can store eight logical values (sometimes called *flags*) in one byte of memory.

It's clear that you need a way to isolate one particular flag in a byte. This is done with the AND operator. The truth table for the AND operator is shown in **Figure A**. The programmer masks off every bit except the one of interest, and then compares the result to zero. If the result is zero, the bit in question is 0. If the result is non-zero, the bit in question is 1. In **Figure B**, bit 3 is the one we are interested in.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Figure A: The truth table for the AND operator.

Bit number		7	6	5	4	3	2	1	0
Memory byte holding flags	var1	1	0	0	1	1	0	1	1
Mask	var2	0	0	0	0	1	0	0	0
var1 AND'ed with var2		0	0	0	0	1	0	0	0

Figure B: Masking off bit number 3.

Windows API. The API communicates with the serial port via a DCB. The actual DCB is a component of the device driver (COMM.DRV). You edit this DCB by declaring a Delphi variable, setting the fields of the variable, and setting the actual DCB with your copy of the DCB using the **SetCommState** function.

The variable declared in the Delphi program is of type TDCB. A TDCB is a Pascal record (or a structure in the C language) that contains several fields (baud rate, parity, etc.) describing how the serial port is configured. The Windows API help that ships with Delphi contains excellent documentation on each field — just search on TDCB. The following Windows API functions are used in the demonstration form, Communications Demo #1. (Again, for more information about these functions, use Windows API Help.)

OpenComm. This function opens a COM port and allocates memory for the buffers. The COM port to be opened is one parameter, and the other two parameters are the sizes of the input and output buffers. **OpenComm** returns an integer identifying the opened port. This identifier is used by many other communications functions listed here.

BuildCommDCB. This function is a quick way to initialize a TDCB variable. It takes two arguments: A null-terminated string of the form:

`'COMx,BaudRate,Parity,DataBits,StopBits'`

where *x* is the port number (e.g. 'COM1,9600,N,8,1'), and the TDCB variable that is passed by reference. Any fields in the DCB that are not specified in the parameter string are set to their default values. This function does *not* set the COM port to the settings specified in the TDCB variable. **SetCommState** does that. **BuildCommDCB** returns zero if successful. Otherwise, it returns less than zero.

SetCommState. This function sets the COM port to the settings specified in the TDCB variable. The TDCB variable is the only parameter for this function. The API knows which COM port to set because the COM port is specified in the TDCB variable itself. This function returns zero if successful. Otherwise it returns less than zero.

SetCommEventMask. This function enables selected events for the specified COM port. The two parameters are the ID of the port (returned by **OpenComm**) and a mask for the appropriate events. The mask should contain a bit set to 1 for each event to be enabled. Enabled events are recorded in the *event word*. **GetCommEventMask** will retrieve the event word.

EnableCommNotification. This function enables or disables a Windows message (WM_COMMNOTIFY) being posted to a window. Messages are disabled by default. The function takes four parameters:

- 1) The communications device in question.
- 2) The window to which messages are to be posted.
- 3) A parameter indicating the number of bytes the input buffer must contain before a message is sent to the application. A message is sent if the number of characters in the input buffer exceeds this number. It's a notice that the application must read characters from the input buffer or there will soon be no more room in the buffer. If this parameter is set to -1, the message for this event is disabled.
- 4) A parameter indicating the minimum number of bytes the transmit buffer must maintain without sending a message to the application. A message is sent if the number of bytes in the transmit buffer falls below this number. It's a notice to the application that the transmit buffer will soon be empty. If this parameter is set to -1, the message for this event is disabled.

GetCommEventMask. This retrieves the event word for the desired port, and then clears those bits in the event word that are specified by the event mask. The two parameters are the port's ID (returned by **OpenComm**) and a mask for the appropriate events. This function returns the entire event word for the specified port, regardless of the event mask.

GetCommState. This function retrieves the device control block for the specified port. It takes two parameters: the port's ID (returned by **OpenComm**), and a reference to the TDCB variable.

ReadComm. This reads characters from the device driver's internal serial buffer. It takes three parameters: the serial port's ID

(returned by **OpenComm**); a pointer to the buffer to receive the bytes read; and the number of characters to read. If the function is successful, it returns the number of bytes read. If unsuccessful, it returns less than zero, the absolute value of which is the number of bytes actually read.

WriteComm. This function sends a buffer of information to the serial port for transmission. It takes three parameters: the serial port's ID (returned by **OpenComm**); a pointer to the buffer holding the characters to be written; and the number of characters to write. If the function is successful, it returns the number of bytes written. If unsuccessful, it returns a value less than zero, the absolute value of which is the number of bytes actually written.

GetCommError. When a communication error occurs, Windows locks the COM port until the programmer calls this function. The **ReadComm** and **WriteComm** functions will return an error if called before this function. However the port will, if able, continue to receive characters, and transmit whatever characters are currently in its transmit buffer.

This function takes as parameters the COM port to be checked (the **OpenComm** function returns this value), and a record (structure) of the type TCOMSTAT. After the call to the **GetCommError** function, the TCOMSTAT structure will contain status information for the COM port. Information such as "Transmission has been suspended because of reception of an XOff character" can be obtained from the TCOMSTAT record. **GetCommError** returns an integer. Each bit of the integer represents one type of error. The return value can be a combination of these bits. The programmer can then decode these bits and find the reason for the error.

EscapeCommFunction. This function directs the specified communication device to carry out an extended function. The first of two parameters is the COM port that will carry out the function (**OpenComm** returns this value). The second parameter is an integer specifying which function to carry out. **EscapeCommFunction** is used in sample form, Communications Demo #2, to set and clear the RTS (Request To Send) and DTR (Data Terminal Ready) signals. It returns zero if successful, less than zero if unsuccessful.

CloseComm. In the demonstration form, this function is called by the *FormDestroy* event, and transmits any characters left in the transmit buffer, closes the COM port, and then releases any memory allocated for the transmit and receive buffers. It returns zero if successful, less than zero if unsuccessful.

Basic Serial Communications with Delphi

The Windows device driver for the COM ports is interrupt-driven. This means your Delphi program does not need to respond to each character received at the port. Windows is interrupted when each character is received. Windows then takes the character received at the port and places it in Windows' own input

buffer. You only need to ensure that you remove characters from this buffer before the buffer becomes full.

The first demonstration program, Communications Demo #1, relies on the operator to click a button to get characters from the buffer. It also relies on the operator to click a button to send whatever is in the transmit window. Communications Demo #1 has four buttons and two memo fields on the main form (see Figure 3).

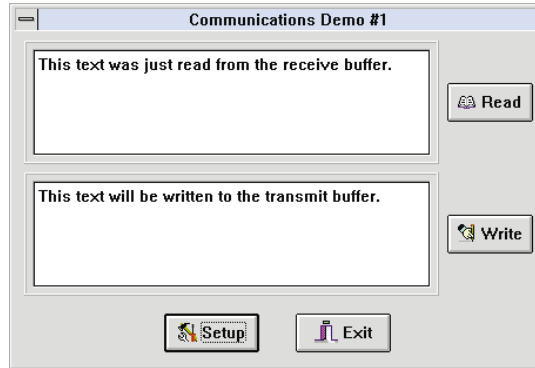


Figure 3: The example Communications Demo #1 application in action.

The *OnClick* event handler for the **Read** button takes the characters in the receive buffer and adds them to the adjacent memo field:

```
if ReadComm(hCommPort,inBuf,RecBufSize) > 0 then
    Memo1.Text := Memo1.Text + StrPas(inBuf);
```

The *OnClick* event handler for the **Write** button takes the characters in the lower memo field and copies them to the transmit buffer.

```
StrPCopy(outBuf,Memo2.Text);
rVal := WriteComm(hCommPort,outBuf,StrLen(outBuf));
```

The Windows communications device driver (COMM.DRV) then sends these characters through the serial port. The baud rate, parity, data bits, and stop bits can be set-up by selecting **Setup** from the main menu (see Figure 4).

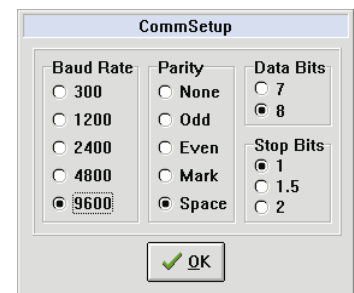


Figure 4: By selecting the **Setup** button on either the Communications Demo #1 or #2, this CommSetup dialog box appears.

Windows serial communications is interrupt-driven since Windows (without any intervention from any application program) places received characters in a receive buffer, and transmits characters from a transmit buffer. This is accomplished by the Windows communications device driver COMM.DRV. If you want Windows to interrupt your Delphi program, you must write the appropriate code to intercept Windows messages.

Programmers new to Windows may be unfamiliar with the interaction between the application program and the Windows operating system. In the DOS world, the application takes

total control of the computer, and calls the DOS operating system for various services. In Windows, while the application still calls the operating system for services, the operating system can make calls to the application. In Windows terminology, Windows sends a *message* to the application.

Windows responds to events by sending *messages* to an application. Most of these messages are handled behind the scenes by Delphi. Delphi converts these messages into appropriate *events* (*OnClick*, etc.) for each component. Delphi programs respond to these events using event handlers — procedures that process the keystroke, mouse click, etc. However, what is needed is a way to respond to serial port events.

By default, each Delphi application declares a variable, *Application*, of the type *TApplication*. Using this variable, you can write event handlers, not for individual components, but for the application itself.

First, write an event handler that is called for, but does not need to process, *all* Windows messages. In the second sample application, this handler is called *MsgHndlr*. You then tell Delphi about this event handler. This is done in the *FormCreate* event handler:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnMessage := MsgHndlr;
end;
```

Now, whenever Windows sends a message, *MsgHndlr* will be called, and will have an opportunity to intercept the message. *MsgHndlr* is called with two parameters. The first is a variable of the type *TMsg*, a structure in which Windows places information about the message, including the type of message. The message for communications events is called *WM_COMMNOTIFY*. You can test for communication events with the following code:

```
procedure TMainForm.MsgHndlr(var Msg: TMsg;
                             var Handled: Boolean);
begin
  if Msg.message = WM_COMMNOTIFY then
  begin
    { Process the communication event here. }
  end;
end;
```

The second parameter tells Windows if it should handle the event. If you set *Handled* to *True*, then Windows/Delphi will not do anything with the message. Conversely, if *Handled* is set to *False*, then Windows/Delphi performs the default processing.

The Communications Demo #2 (see [Figure 5](#)) responds to received characters by displaying them in the Memo component at the top of the screen. This is in contrast to the first sample form that required some user event (i.e. a button click) to read the receive buffer and display the characters.

The *MsgHndlr* procedure requires a test to determine why the *WM_COMMNOTIFY* message was sent. If it was sent

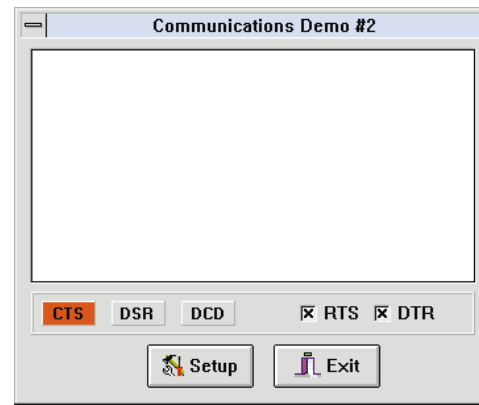


Figure 5: The example Communications Demo #2 application displays the character it received in the Memo component.

because a character was received, the program should get the character and place it in the Memo component's window. The code for the message handler accomplishes this by testing the right-most bit in the event word (returned from the *GetCommEventMask* function). If the bit is set, a character has been received. Otherwise, the message was sent for some other communications event:

```
if Msg.message = WM_COMMNOTIFY then
begin
  if (GetCommEventMask(hCommPort, constCommEvents)
      and 1) = 1 then
    ReadCommPort;
  GetMSR; { Examine modem status register. }
end;
```

How does Windows know for what events to send the *WM_COMMNOTIFY* message? Your program must tell Windows the events you're interested in. You do this with the *SetCommEventMask* function which takes two parameters.

The first is a handle to the port (the value returned by the *OpenComm* function). The second is a mask with bits set for each event in which you are interested. (These bits are described in Windows API help.) Communications Demo #2 is configured so a *WM_COMMNOTIFY* message is sent if one or more characters are in the receive buffer, or if the CTS, DSR, or DCD lines change state. *SetCommEventMask* is called in the *FormCreate* event handler for the main form.

Communications Demo #2 also displays the state of the CTS, DSR, and DCD lines by changing the color of the labels to red if the signal is asserted. It does this by examining the modem status register and updating the display each time a *WM_COMMNOTIFY* message is received for any reason.

The output signals RTS and DSR are set or cleared using the *EscapeCommFunction* function:

```
procedure TMainForm.btnDTRClick(Sender: TObject);
begin
  { SetDTR & ClrDTR are constants defined in WINTYPES.PAS }
  if btnDTR.Checked then
    EscapeCommFunction(hCommPort, SetDTR)
  else
    EscapeCommFunction(hCommPort, ClrDTR);
end;
```

Conclusion

Learning to program a computer is similar to learning to play the piano. You can't learn how to do it by just reading about it. The documentation for writing serial communications programs is so thin, there isn't much to read anyway.

Feel free to experiment with the two sample applications. It's possible to connect two PCs together with serial cables and a null modem adapter (available at most computer stores and electronic distributors). A null modem adapter makes each computer think it is connected to a modem. Then run the demonstration programs on both computers and send characters back and forth. Make small modifications to the program and observe the results. A few hours of creative playing should have you on your way to using the serial port in your programs. ▲

The demonstration programs referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\NOV\TC9511.

Tom Costanza is the founder of Costanza & Associates in Langhorne, PA, providing development tools, programming, and training since 1989. He can be reached at (215) 752-5115, or on CompuServe at 76615,2154.



NEW & USED

BY TIM FELDMAN



Borland RAD Pack for Delphi

What It Is, Why You Need It

Even if you never use the VBX controls in Borland's RAD Pack for Delphi, its other elements make the package well worth its price. Besides the Visual Basic controls, the RAD Pack contains Delphi-compatible versions of Borland's classic Turbo Debugger and Resource Workshop tools; the source code of the Visual Component Library (VCL); a new Resource Expert for the Delphi IDE; and a patch kit for the first release of Delphi. There are also hardcopy manuals for the VBXes and the Turbo Debugger, and a hardcopy version of *Object Pascal Language Guide*. About the only thing missing is hardcopy of the *Visual Component Library Reference*.

It's no secret that Borland has fallen back upon its core strength — development tools — to survive in today's PC software business. Delphi is critical to Borland's future. The first release of Delphi earlier this year must have involved tough marketing decisions. The package had to be small enough to be irresistible to independent developers, but complete enough to avoid being rejected as a "toy language." Borland couldn't throw the kitchen sink into its initial release of Delphi or it would have been too expensive. By releasing Desktop and Client/Server versions of Delphi, Borland seems to have made the right marketing decisions. Delphi appears to be well on its way to success. Now Borland has released the Delphi RAD Pack, which rounds out the IDE with tools that most serious developers will want to have.

Installation

The RAD Pack software comes on a single CD-ROM and eats up about 25MB of hard disk. A four-page printed

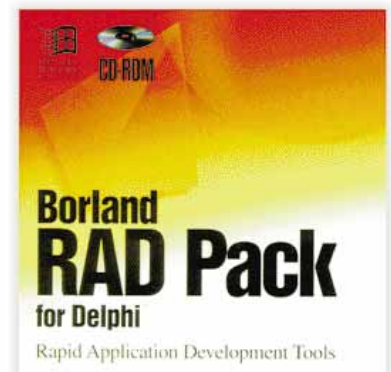
"Roadmap" explains what's in the RAD Pack and gets you started on the installation. The installation program had one small problem — it tried to write the VCL source code to my CD-ROM — but I overrode it easily.

After I finished the Windows installation, I followed the README.TXT instructions and used DOS to patch the Delphi IDE from the RAD Pack's CD-ROM. There was no need to patch the VCL source code or rebuild the IDE's component library.

The VBXes

The RAD Pack includes volume 1, version 1.1 of Borland's Visual Solutions Pack for Windows. This is a collection of VBXes written by commercial vendors. Perhaps the key word here is "commercial" — many of the VBXes are less-powerful versions of other packages from the vendors. That the controls are limited versions is not mentioned in the RAD Pack's documentation, but if you activate their "About" property at design time, many of the controls display dialog boxes that read like magazine advertisements for the full versions.

The About boxes also reveal that the VBXes are of 1993/94 vintage. Older VBXes that predate Delphi, such as those in the Visual Solutions Pack, are more likely to suffer in comparison to Delphi's object-oriented VCLs. For example, they may not be data-aware nor do a good job of wrapping complexity in an easy-to-use set of properties and methods. In addition,



using VBXes in commercial software requires you to redistribute and properly install .VBX and .DLL files.

Despite these limitations, if you buy the RAD Pack, you'll be curious about the VBXes. You'll want to install them in the IDE and try them. I installed all the VBXes easily — there are clear instructions in the RAD Pack Roadmap and several other places.

Using them, however, was a different story. The first VBX I tried was AniButton (for "Animated Button") from DesaWare. Drop this simple little gadget on a form, give it an event to handle, and it works much like a normal button. The only typical button events it doesn't handle are *OnMouseMove*, *OnMouseDown*, and *OnMouseUp*.

The difference with AniButton is its appearance. If you don't give its *Picture* property a value, it looks like a standard Edit control (a plain white rectangle). The fun comes after you use its *Picture* and *Frame* properties to specify a set of bitmaps for it to animate. The AniButton will play the pictures back on an event, showing a little cartoon movie on its surface. Other properties let you stretch the bitmaps, change their playback rate, and modify the animation so that some bitmaps play when the button is pushed, and the rest play when the button is released. The AniButton is cute, easy to use, and quite flexible. It's data-aware and similar to a true Delphi VCL. I had fun playing with it.

Heartened, I tried another, more complex VBX: the ImageKnife component from Media Architects, Inc. The ImageKnife is billed as a super-charged version of the IDE's standard Image component. It can display advanced image formats, perform simple image processing operations, and manipulate the image's color palette. Or rather, the ImageKnife should be able to do all those things. I don't know if it can or not, because I could not make it work.

Perhaps the ImageKnife's Delphi interface isn't complete, or its documentation skips some crucial step. I had no problems installing the ImageKnife VBX or dropping its visual control, PicBuf, onto a form. PicBuf's design-time properties worked, as did some of its run-time-only properties and methods. Some of the run-time methods compiled without problems, but many of them — including the ones that load an image file into PicBuf — were unrecognized by the compiler. The interface file to the VBX (KNIFE.PAS) appears to be valid, but omits references to many of the properties and methods described in the *Visual Solutions Pack Reference Guide*.

I was curious about this quiet failure. Knowing that the Visual Solutions Pack also supports C++ and dBASE (the hardcopy user guides for both languages are included), I looked at the ImageKnife example programs in C++ and dBASE on the RAD Pack CD-ROM. Those examples do support the image load function. Unfortunately, the only Delphi examples on the CD-ROM are for the simpler VBXes and the TX Word Processing

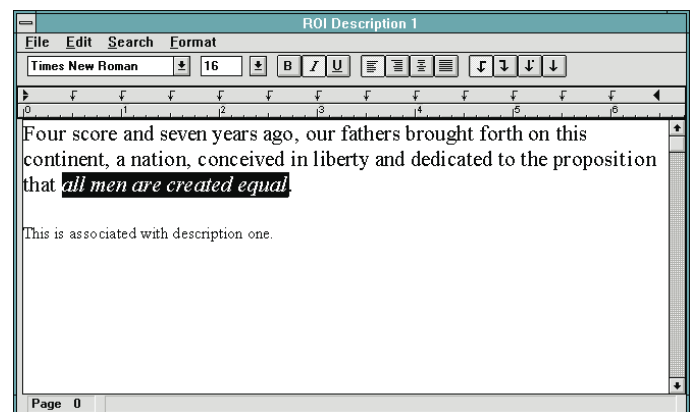
controls. There are no examples for the ImageKnife control. The ImageKnife documentation shipped with the VBXes is no help. The massive hardcopy *Reference Guide* (nearly 1,000 pages) and the ImageKnife on-line Help file both predate Delphi and lack programming examples. The *Reference Guide* has several chapters about the ImageKnife control (also with no examples).

A thin (36-page) *Visual Solutions Pack User's Guide for Delphi* contains what appears to be an excellent tutorial — on one of the other VBXes. It has nothing about the ImageKnife VBX. In fact, the only VBX it discusses in detail is the tutorial's set of TX Word Processor components. While this makes some sense (re-writing the entire *Reference Guide* for Delphi would probably have been prohibitively expensive), it's an indication that mastering the more complex VBXes may be a "learning experience."

In the end, I abandoned efforts to make the ImageKnife VBX work since I had no real-world need to use it. If I did need to work with images, I would probably look for a true Delphi VCL rather than a VBX.

The VBXes in the Visual Solutions Pack include:

- AniButton Animated Buttons by Desaware
- Chart Controls by Kansmen
- Formula One Spreadsheet Controls by Visual Tools
- Gadgets by MicroHelp
- ImageKnife Image Editor Control by Media Architects
- Integra Visual Database Builder by Coromandel Industries
- SaxComm Communications Tools and SaxTabs Notebook Tab Controls by Sax Software
- TX Word Processor Controls by European Software Connection



From the Borland Visual Solutions Pack, the TX work processor VBX in action.

The VCL Source Code

Disappointing as the VBXes are, the VCL source code inspires the opposite feeling. For experienced developers, the source code is an exciting treasure. I have always found that no matter how well a development environment is documented, there are questions that can only be answered by "looking under the hood." For that, you need the source code to the library routines. And,

as you try to master the new environment, you'll never find enough example programs. The source code helps fill that void. Installing it adds hundreds of files under \DELPHI\SOURCE in three subdirectory trees.

The IDE displays special dialog boxes to help you set certain design-time component properties. For example, to set the *Picture* property for an Image component, the IDE displays a Picture Editor dialog box. The \DELPHI\SOURCE\LIB tree contains the source code for those special dialog boxes.

The \DELPHI\SOURCE\RTL tree contains the Delphi run-time library (RTL) source code. The \RTL\SYS subdirectory is where the string, math, and other "primitive" routines are implemented in .ASM and .PAS files. Do you need to see how random numbers really work in Object Pascal? Check the code in RAND.ASM. Are you trying to figure out what the real differences are between *New* and *GetMem*, and are you confused by the printed documentation and on-line help? Search for "NewPtr" in SYSTEM.PAS and WMEM.ASM. You might be surprised by what you find!

The \RTL\WIN subdirectory holds the interface .PAS files that join Delphi to the Windows 3.1 APIs. Here, you can discover how Delphi connects to Pen Windows, to the Windows Common Dialogs, and to other APIs that aren't even mentioned in the Delphi documentation.

If you want to translate Delphi's run-time error messages into another language, you'll want the English, French, and German subdirectories in the \RTL\SYS tree. Example files in all three languages show how to recompile the RTL to change the messages.

Finally, the \DELPHI\SOURCE\VCL tree contains the Delphi source code to the VCLs in the IDE's Component Palette (except for the sample VBXes — their source code is not given — and the Sample VCLs, whose source code is shipped with Delphi in \DELPHI\SOURCE\SAMPLES). This source code is an absolute must-have for any developer creating custom visual components. Foreign-language developers will also want the English, French, and German resource files, which define the error message strings in all the VCLs.

Note that the source code package does not supply the source code for other parts of Delphi, such as the Image Editor or the IDE's text editor. And, of course, if you have the Client/Server version of Delphi, you already have the source code. It's also available separately from Borland for approximately US\$100. Since that is almost as much as the RAD Pack's street price, it makes sense to get the RAD Pack. Its additional tools are easily worth the price difference.

The Turbo Debugger

The Delphi IDE's Integrated Debugger does a very good job of debugging at the Object Pascal level. But when you need to grab the CPU by its registers and shake it, you need the Turbo Debugger.

The Turbo Debugger does things the Integrated Debugger can't do. It logs debugging sessions to a disk file, records your key-

strokes, steps backward through your code, and examines the CPU and system memory directly. It runs on a second video display, or on a second computer over a network or serial link. It contains its own assembler, disassembler, and expression evaluator. It supports C, C++, and Object Pascal, and understands objects, exceptions, and properties. It debugs DLLs, monitors Windows messages, and even performs hardware debugging by taking advantage of the debug registers in 386 and higher processors. This lets the Turbo Debugger detect CPU accesses to specific memory and I/O addresses, or with specified ranges of data values. In short, it handles industrial-strength bugs.

The version in the RAD Pack is Turbo Debugger 4.6 for Windows only. Unlike separate Turbo Debugger packages from Borland, the RAD Pack doesn't include DOS and Win32 versions of it (nor Borland's Turbo Assembler). Those won't be serious shortcomings for most Delphi developers.

A 144-page manual specifically for Delphi/Object Pascal users is included. Along with several text files on disk, it covers the basics of configuring and using the Turbo Debugger. While the manual is well-written, it falls short in one area: it does not contain any tutorials. There are no sample Delphi programs with subtle captive bugs for you to practice on. Because of this, you will probably defer learning to use the Debugger until the worst possible time: when a real bug bites you and your code needs to ship in a hurry. But at least when that does happen, there's an excellent tool to help fix your code.

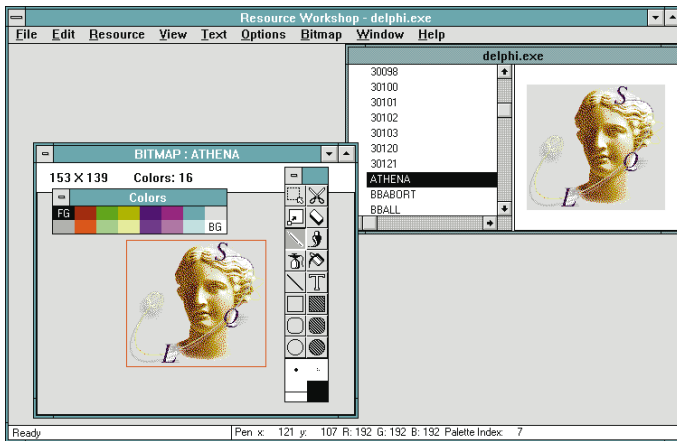
The Resource Workshop

The last major tool in the RAD Pack is Borland's Resource Workshop, version 4.5. It lets you edit the Windows resources used in your program.

The most common editable resources are icons, bitmaps, and menus. The Delphi IDE contains built-in support for editing those resource types. For example, IDE's Image Editor can be used to work on icons, bitmaps, and several other resource types. However, just as the Turbo Debugger is more powerful than the Integrated Debugger, the Resource Workshop is more powerful than the Image Editor. The Workshop has more tools and options. More importantly, the Resource Editor is more stable than IDE's Image Editor, which has been known to crash on occasion. I found the Resource Workshop's stability well worth the price.

If you want to define custom fonts for your applications, you'll want the Workshop. If you want to internationalize your programs by having them load their strings at run-time, you'll find putting the strings into a resource file created by the Workshop a good method. You can even use the Workshop to extract the resources from a .DLL or .EXE file, modify them, and then bind them back into the same executable file.

If your application needs a custom resource — perhaps special binary data, or a large block of text to be displayed at run-time — the Workshop will help you define, edit, and manage it.



Borland's Resource Workshop.

With all these capabilities, it's a shame the Resource Workshop isn't well documented. Borland did not include a hardcopy manual in the RAD Pack. However, there are several language-independent Windows help files that do an average job of making up for the missing manual. A single short README.TXT file discusses installing and using the Workshop with Delphi. Beyond that, you'll have to learn about the Workshop by tinkering with it. It's worth the time.

The Resource Expert

If you have developed Windows applications in other languages, you've probably worked with .RC files. These are the files that define the Windows resources created with the Resource Workshop or similar tools. They're a kind of source code or script file written in a language defined by Microsoft. Delphi supports .RC files indirectly. After they have been compiled into .RES or .DFM files, Delphi can link them into your application. That's what the mysterious little `{$R *.DFM}` compiler directive in the **implementation** part of a Delphi form's unit is for. It tells the compiler that any .DFM resource files in the project's directory should eventually be linked into the application.

Normally, you use the IDE to compile Delphi components into .DFM files or the Resource Workshop to compile .RC files into .RES files. Then, you manually rename the .RES files to .DFM files so Delphi will link them into your application. But, if you have developed a large number of .RC files containing menus or dialog boxes, you may want to convert them into Delphi menus and dialog box forms. That is what the Resource Expert does. After they have been converted, you can manipulate them within the Delphi IDE as if you had defined them using the IDE's usual tools.

The Resource Expert is a new plug-in for the IDE. After installing it and configuring the IDE, the Resource Expert appears alongside the Database Form and Dialog Experts in the Experts page of the Browse Gallery whenever you create a new form. If you activate the Resource Expert, it asks for the location of the .RC and support files you wish to convert. Then it processes those files and displays the new form on

your screen. The form contains the menu or dialog box defined in the .RC file. From that point on, the new form behaves exactly as if you had created it by hand, and you can discard the old .RC and support files.

I had no problems converting old .RC scripts with the Expert, even though its only documentation is a Windows Help file. Installing the Resource Expert hooks it into the IDE's **Help** menu so the Expert starts when you select **Help | Resource Expert**. The Database Form Expert and Interactive Tutors are also wired into the **Help** menu the same way. It's a startling effect — you're expecting a Windows Help file to pop up, not a separate application — but after working with it for a while, I grew to like it.

Even though the Resource Expert was released after Delphi shipped, it fits into the IDE perfectly. It's a nice example of the forward thinking in Delphi's design. I wonder how quickly other vendors will start developing their own plug-in experts for the Delphi IDE. Most of the source code for the Resource Expert is provided with the RAD Pack.

The Patches

A few months after Delphi's official release, Borland issued free patches to the VCL and its source code, and to both the Desktop and Client/Server versions of the Delphi IDE. These version 1.01 patches apply to the Delphi files dated 2/15/95 8:00 am. The patches are available from Borland's CompuServe and ftp sites, and are also part of the RAD Pack.

The patches fix a number of relatively minor bugs in Delphi. I never noticed the differences after installing them. A complete list of the bugs is included with the patch kit.

The Object Pascal Language Guide

When Delphi was released, users were quick to praise it. They were also quick to complain about the skimpy documentation that shipped with the Desktop version of Delphi. I recently had the opportunity to ask Borland's Phillippe Kahn if Borland had decided to change their future policy on hardcopy documentation because of all the loud complaints. He replied that they were "satisfied" with their practices.

Still, Borland responded to the complaints by making an Adobe Acrobat version of the *Object Pascal Language Guide* available at no cost on their CompuServe and ftp sites. They have also announced the 32-bit version of Delphi will include a hardcopy version of the *Guide*, and a copy is part of the RAD Pack.

At 290 pages, the *Guide* goes into some depth about the language Delphi is built upon. It's a reference, not a tutorial, but contains plenty of code snippets. Its 20 chapters and three appendices cover all the details of the Object Pascal language, the compiler/linker, and the built-in assembler.

The *Guide* also covers the basic parts of the Object Pascal RTL, including strings, I/O, and memory issues. An interesting point is that Object Pascal and Delphi are not synonymous. Object

Pascal is the core upon which Delphi's IDE and the VCL are built. By itself, Object Pascal could compile DOS applications (if DOS RTLs were supplied; they are not). Therefore, the *Object Pascal Language Guide* says little about Windows or Delphi. It does, however, include a short chapter about Windows DLLs and a section on using the WinCrt unit to produce text-oriented Windows applications. The *Guide's* main purpose is to explain the Object Pascal language succinctly. It does that quite well.

The (Missing) VCL Reference

If Object Pascal is the hidden foundation upon which Delphi is built, the VCL is the elegant structure that we think of as Delphi. The definitive guide to that structure is Borland's *Visual Component Library Reference*, a 1,000 page hardcopy manual. Unfortunately, it's not part of the RAD Pack. It would have been better to include the *Reference* than the entire Visual Solutions Pack of VBXes.

The hardcopy *Reference* is available separately from Borland. An Adobe Acrobat version of the *Reference* is also available. Curiously, it's not included in the RAD Pack. It can be downloaded from Borland's CompuServe and ftp sites. The file is about 5MB in size when expanded. I find it essential for serious development work.

Conclusion

If you are a Delphi developer, you should get the RAD Pack for its VCL and RTL source code, its tools, and the hardcopy *Object Pascal Language Guide*. Use those parts because they are excellent, and ignore the VBXes of the Visual Solutions Pack because they are not. Add a hardcopy or Adobe Acrobat version of the *Visual Component Library Reference*, and you'll have the essential tools to do serious work with Delphi. ▲

INFORMANT
FACT FILE

Borland's RAD Pack for Delphi is a collection of tools, source code, and documentation that enhances the Delphi IDE. It includes version 1.1, Volume 1 of the Visual Solutions Pack; Delphi-compatible releases of Turbo Debugger and Resource Workshop; a new Resource Expert for the IDE; patches for the initial release of Delphi; the source code for Delphi's Run-Time Library and Visual Component Library; and a printed copy of the *Object Pascal Language Reference*. The Visual Solutions Pack is the only weak aspect of an otherwise must-have package for serious Delphi developers.

Borland International
100 Borland Way
Scotts Valley, CA 95066-3249
Phone: 1-800-336-6464
Internet: <http://www.borland.com>
CompuServe: GO BORLAND
Price: \$US189.95

Tim Feldman has more than fifteen years of experience in hardware and software development. His most recent large project was designing and coding an event-driven GUI, debugging, and calibration package for a high-speed almond sorting system. He can be reached via the Internet at tfeldman@wheel.dcn.davis.ca.us.



TEXT FILE



Sometimes Good Things Take a Little Longer

The first wave of Delphi books has passed. Now come the more thorough books; the ones that took a little longer to get to market. These are the books that professional developers will use as standard references — the ones that will end up with dog-eared pages and smudged CD-ROMs. The *Delphi Developer's Guide* by Xavier Pacheco and Steve Teixeira is one of those standards. And if you're serious about Delphi, you need this book.

Steve Teixeira is a regular contributor to Delphi groups, and has written a number of excellent articles for various Delphi magazines including *Delphi Informant*. Both Teixeira and Pacheco worked in Borland's Technical Support department. Teixeira is still there, and Pacheco is now with TurboPower Software, makers of VCL add-in components for Delphi. Their combined experience is what makes this book so valuable. Manning the front lines at Borland, they've answered Delphi questions that most of us haven't even thought of. Their book is packed with valuable source code, discussions, step-by-step procedures, cautions, and tips.

As its name states, this guide is for developers, not begin-

ners. Its 900 pages are divided into three major sections. The first section alone is worth the price of the book. In fact, it contains more material than many other complete Delphi books.

In twenty-two chapters, it covers every major area of Windows and DPMI programming using Delphi including: MDI applications, GDI programming, DLLs, OLE, DDE, the Windows multimedia API, SQL, Windows messages and hooks, building custom components, and testing and debugging.

The authors use short example programs to illustrate each topic and offer useful notes and tips. They cover advanced operations such as creating custom components in a clear, step-by-step fashion that's just right.

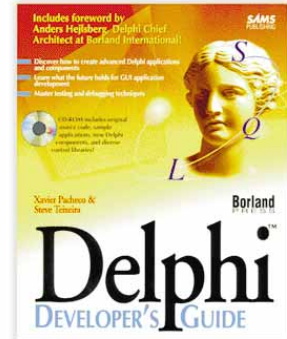
First, they explain the method they will use and then present an example project that explicitly illustrates the method. In many places, they point out the easiest way to accomplish something. Then they bring up advanced alternatives and explain why you would need to use them. And always, there are examples — lots of them — that are well-documented and ready to run.

In the second section, the authors move beyond short examples to begin teaching the techniques used to build real-world applications. In six chapters, they create a system resource monitor, an address book, a calendar/scheduler/ alarm application, a phone dialer/terminal application, a file manager, and a time-tracker application.

With these projects, you learn development techniques as well as the details of various Delphi components. The authors explain each project completely and deftly insert advanced concepts, such as deriving descendant components.

In the final section, Pacheco and Teixeira devote another six chapters to developing two large applications: an inventory manager and a personal information manager. As with the other examples, they build each application with an emphasis on real-world development techniques. They cover preliminary design issues and decisions, develop the application's user interface, and then add finishing touches.

In a brief Appendix, the authors cover Delphi error handling and error messages. As usual, the material has a practical emphasis, describing common causes of errors



and how to resolve them.

The text is well written. The authors have a comfortably direct style, informal without being chatty. It's neither academic nor simplistic — they assume you understand computers and programming, and spend their effort in providing lots of examples and useful tips. Their experience in working with developers gives them a refreshingly practical tone.

Despite the book's professional orientation, it's accessible to serious amateurs. It contains excellent chapters on programming in Pascal and Object Pascal, and it succinctly explains important aspects of Windows programming, with an emphasis on providing useful examples.

The CD-ROM included with the book is also valuable. In addition to the source code,

*"Sometimes Good Things
Take a Little Longer"*
continued on page 49

A *Developer's Guide* Targets Database Developers

Surveys indicate that most developers use Delphi as a tool to create database applications, and authors Bill Todd and Vince Kellen cater to this audience in *Delphi: A Developer's Guide*. This new title from M&T Books provides the most comprehensive guide to-date in chronicling the complexity of designing client/server applications in Delphi.

Todd and Kellen are well-respected in the Paradox community and have strong backgrounds in database applications. Their expertise definitely comes through in the pages of this book.

Beginning and intermediate database developers will appreciate the comprehensive coverage of the issues surrounding database application development. For example, it shows that developing a database application involves more than working with *TQuery* and *TDataSource* components. It might well be necessary to deal with issues such as database integrity, data security, SQL-92 isolation levels, and implicit vs. explicit transactions.

Of the book's 33 chapters, 15 focus on database-related topics. Although that makes for a strong book on database application development, it also detracts to an extent from Object Pascal programming.

For example, Chapters 23-28 and 33 concentrate on database development concerns (e.g. database integrity, Microsoft SQL Server, InterBase, and the Database Desktop). Although crucial database concerns, these are not Delphi topics per se, and

will have little appeal to developers not interested in developing database applications. Despite its intense coverage of database issues however, *Delphi: A Developer's Guide* provides more information about Object Pascal than most third-party Delphi books. Several chapters study Object Pascal program structure, data types, expressions, statements, procedures and functions, and units.

However, the discussions of some issues (e.g. linked lists and PChar variables) are not as detailed as those in Charles Calvert's, *Delphi Unleashed* [Sams, 1995]. In fact, *Delphi: A Developer's Guide* can be seen as a good companion to Calvert's *Unleashed*.

Mastering Delphi is Complete and Insightful

If you intend to buy just one book on Delphi, Marco Cantù's *Mastering Delphi* deserves a place on your short list of candidates. This encyclopedic text weighs in at 1500 pages, covering the Delphi spectrum from introductory concepts to advanced programming techniques.

Cantù uses his unusually high page count to great advantage, providing more complete treatments of the topics than any other text I've seen. Many discussions show several ways to accomplish a particular task, with welcome comments on the tradeoffs.

Despite its length, the text is highly readable and about as entertaining as the material allows. The author's enthusiasm for Delphi is apparent and infectious (as if Delphi users

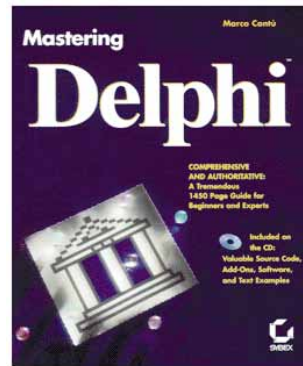
Delphi: A Developer's Guide also tackles other topics important to advanced developers, including creating DLLs, exception handling, and calls to the Borland Database Engine. The book and accompanying CD are filled with useful examples and sample code. And there's no doubt that you'll find code (e.g. string manipulation procedures), that can easily be used in your applications.

The book is well-written and interesting. However, it could have been better organized. For example, information on error handling is curiously sandwiched between chapters that describe the Query component and "Building a



Database Application". Also, the advanced topic of creating DLLs is discussed before an introductory chapter on working with VCL components. In addition, object-oriented programming is examined in three chapters when one would have been sufficient. Therefore, a cover-

"A Developer's Guide Targets Database Developer"
continued on page 49



needed much encouragement). The first three chapters provide a general introduction to Delphi. Chapter 4 offers a concise overview of the Pascal language, emphasizing the motivation for various elements of the language, rather than just their syntax. Chapter 5, "Object Pascal as an OOP Language," devotes 76 pages to object-oriented programming concepts, and includes the best treatment of Delphi's exception handling I've seen.

An overview of the Visual Component Library is followed by 12 chapters (Part II of the book, which fills 700 pages) that explore the individual components in detail.

This same material is covered in most other Delphi texts, but *Mastering Delphi* goes far beyond the standard enumerations of properties and methods. Examples are especially well chosen. One example often serves to illustrate several key concepts that are woven together seamlessly.

Mastering devotes more than a hundred pages to database programming. A separate chapter describes how to write client/server applications, covering topics such as the Local

"Mastering Delphi is Complete and Insightful"
continued on page 49

Sometimes Good Things Take a Little Longer (cont.)

project, data, and resource files for all the book's examples, it includes trial versions of many Delphi-related commercial utilities and VCLs. Some of the VCLs are "crippled" because they only work while the Delphi IDE is running. A few of the advanced projects in the text use these special VCLs, but most of the projects do not. There are also several directories of extra "bonus" source code for

projects not described in the text. I had some difficulties with some of this extra source code — missing classes and project files — but only with code that was written by third parties. All of Pacheco and Teixeira's code that I tried worked properly.

An excellent setup program serves as a hypertext guide to the CD-ROM, and lets you install the source code and utili-

ties to your hard disk. Installing the text's source code took about 14MB; the extra source code took another 1.4MB.

To reiterate: If you are a serious Delphi developer or want to become one, you'll find Pacheco and Teixeira's *Delphi Developer's Guide* indispensable. And well worth the wait.

— Tim Feldman

Delphi Developer's Guide

by Xavier Pacheco and Steve Teixeira,
Sams Publishing/Borland Press,
201 West 103rd Street,
Indianapolis, IN 46290-1097;
phone: (800) 428-5331;
fax: (800) 882-8583.

ISBN: 0-672-30704-9

Price: US\$49.99
907 pages, CD-ROM

Mastering Delphi is Complete and Insightful (cont.)

InterBase Server, InterBase server tools, and the Visual Query Builder.

Part III, "Advanced Delphi Programming", goes well beyond the scope of most other Delphi books. One chapter covers techniques such as timers, painting methods, and background computing. Another provides excellent treatment of debugging methodologies. Still others discuss the use of Windows resources, printing techniques (including ReportSmith, which Cantù disparages for its limitations), file support, data exchange (Clipboard and DDE), OLE, and multimedia devices.

Mastering concludes with a well constructed chapter on creating components and a shorter one on using DLLs from Delphi. Appendices provide a somewhat formal discussion of OOP principals and a too-brief introduction to SQL.

An accompanying CD-ROM provides the code for all the examples in *Mastering*. In addition, it offers a useful set of components, including free-

ware, shareware, and demonstration versions of commercial products.

There is also a directory containing tools, and another with the text of several Delphi-oriented magazines (including the Premiere issue of *Delphi Informant*).

The book is not without minor flaws. Some of the introductory material seems to switch too rapidly from elementary to difficult material. A few terms that may be unfamiliar to new users (e.g. "heap" on page 111 and "call stack" on page 213) are used without definition. I noted minor inconsistencies among variable names in one set of sample code fragments (page 109).

Finally, I felt that some important warnings, notably one about memory leakage (on page 217) receive less emphasis than they deserve.

Flaws notwithstanding, this is by far the best Delphi book I have seen. No single book will transform a novice into a Delphi guru overnight, but one or two careful readings of

Mastering Delphi will give you a good head start on the road to guruhood.

— Larry Clark

Mastering Delphi by Marco Cantù, SYBEX Inc., 2021

Challenger Drive, Alameda, CA 94501; (800) 227-2346 or (510) 523-8233.

ISBN: 0-7821-1739-2

Price: US\$49.99
1,503 pages, CD-ROM

A Developer's Guide Targets Database Developers (cont.)

to-cover read may not be the best approach.

Delphi: A Developer's Guide includes an obligatory chapter on using ReportSmith that is weak and perhaps the least useful. ReportSmith integration with Delphi, for example, is only superficially discussed. Finally, the book offers just 16 pages to component design, a subject important to Delphi developers. Although, to be fair, none of the current third-party books I've seen have covered this subject adequately.

In conclusion, despite its shortcomings, I recommend *Delphi: A Developer's Guide* for developers interested in designing database applica-

tions — especially client/server applications. There is simply no other book like it on the market when it comes to database programming. Experienced Delphi developers will also find the Object Pascal discussions and advanced chapters complementary to Delphi books they already have.

— Richard Wagner

Delphi: A Developer's Guide by Bill Todd and Vince Kellen, M&T Books, 4375 W 1980 S; Salt Lake City, Utah, 84104 (800) 488-5233.

ISBN: 1-55851-455-4

Price: US\$44.95
820 pages,
CD-ROM

