



Delphi Revealed

Power, Speed, and Scalability

FEATURES



- 8 Visual Programming** - by Doug Horn
We kick off our premiere issue with a quick visual tour of the Delphi landscape. If you haven't been exposed to Delphi yet, let Mr Horn introduce you to the Component Palette, Object Inspector, Browse Gallery, Code Editor, and much more.



- 15 The Way of Delphi** - by Gary Entsminger
Delphi development = component development. It's the mantra of Delphi programming, so it deserves your immediate attention. Mr Entsminger provides a solid introduction to properties, beginning with the simplest component.



- 21 DataSource1** - by Dan Ehrmann
Mr Ehrmann's demonstration program is a *tour de force* of Delphi features, including modifying a Query component at run-time, sharing an event handler among multiple components, and creating a dynamic tabbed interface to filter database information.



- 26 Sights & Sounds** - by Ken Nesbitt
Like to listen to a few tunes while you're debugging an application? The Delphi MediaPlayer component makes setting up a CD player short work, as Mr Nesbitt explains. Didn't you know that Windows API calls could be fun?



- 31 Delphi C/S** - by Sundar Rajan
Delphi was designed for client/server programming. Using Gupta's SQLBase as an example, Mr Rajan gets us started by demonstrating how to set up a C/S development environment — from ODBC, to the BDE, to the Delphi Database Form Expert.



- 39 DBNavigator** - by Cary Jensen Ph.D.
The Borland Database Engine is the common denominator to working with Borland databases — InterBase, Paradox for Windows, and dBASE for Windows. And the BDE uses aliases to make it all easy, as Dr Jensen points out.



- 44 From the Palette** - by Jim Allen and Steve Teixeira
We just can't say enough about components! Misters Allen and Teixeira launch their component series with an introduction to component design, including property and method basics, and the proper use of class directives.



- 48 At Your Fingertips** - by David Rippy
Mr Rippy kicks off his regular "tips & tricks" column of brief techniques to help get you quickly up to speed with Delphi and Object Pascal. This month's features include: implementing a status bar, locating values in a table, and filtering records in a DBGrid.



- 50 API Calls** - by Alistair Ramsey
Delphi makes it easy to call a DLL function, but what if you don't know the name of the DLL until run-time? It's not a problem. Mr Ramsey demonstrates a technique for calling DLL functions on-the-fly.

DEPARTMENTS

- 2 Delphi Tools**
4 News Line



Delphi TOOLS

New Products
and Solutions



Delphi Training

The DSW Group, Ltd. of Atlanta, GA now offers **Delphi training**. The five-day **Delphi Basics** course is scheduled for May, June, and July of 1995. For US\$1500, developers will learn about Delphi architecture, forms, database design, the Local InterBase Server, and ReportSmith. The course provides an overview of Object Pascal, and instruction regarding Help and OLE components. Call (800) 356-9644 to receive more information. The DSW Group is a Borland Training Connections Member.

Conversion Assistant Moves VB Applications to Delphi

EarthTrek of Burlington, MA is shipping its *Conversion Assistant*, a tool that allows developers to migrate Microsoft Visual Basic (VB) applications to Delphi. It reads VB project and program files and creates a mapping to the equivalent files that can be read, modified, and executed in Delphi. The Conversion Assistant can handle VB projects of varying size and complexity allowing Delphi developers to preserve their investment in previous development work.

The Conversion Assistant reads VB MAK, FRM, and BAS files and provides a "best fit" translation to the Delphi DPR, DFM, and PAS format files. Users can port an entire application or portions of a VB project to Delphi units.

Conversion Assistant features; VB to Object Pascal conversion; mapping of VBX controls to corresponding Delphi controls;

VB project placement and sizing form conversion; and automatic creation of Delphi project files. It also accounts for differences between the two products, including multiple object instances in VB (at design and run time), and fonts and colors.

Future versions of Conversion Assistant will convert

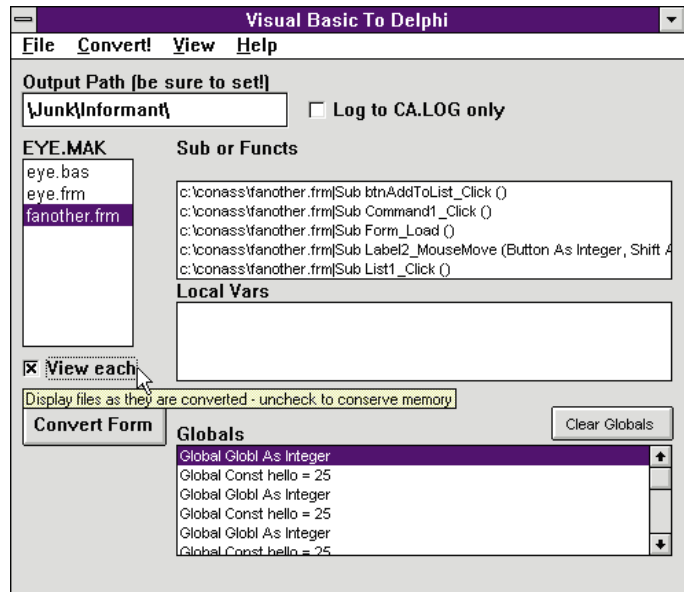
Powersoft's PowerBuilder applications to Delphi.

Price: US\$79

Contact: EarthTrek, 7 Mountain Road, Burlington, MA 01803

Phone: (617) 273-0308

Fax: (617) 270-4437



New Windows Data Transfer Tool Available

Conceptual Software, Inc., of Houston, TX has announced the release of *DBMS/COPY for Windows*. This tool transfers data between most spreadsheets, databases, statistical analysis packages, ODBC, ASCII (fixed and free formats), time series, and other

packages. DBMS/COPY lets the user view the data, control which variables are written, select the records written, and compute new variables. The built-in batch processing enables users to save transfer specifications for future use.

DBMS/COPY for Windows supports data transfer between over 80 software packages. These include: Excel, Lotus 1-2-3, Quattro Pro, Alpha Four, Clipper, DataEase to 4.53, dBASE II III and IV, FoxBase and FoxPro, Paradox to 4.5, PRODAT to 4.0, Rbase to 3.5, Reflex 1 & 2, BMDB-Dynamic, BMDB/New System, EpiInfo to 6, Minitab Extended to 10, S-Plus, SAS, SigmaStat, SPSS, Stata to 3.1,

StatGraphics, Statistica, Systat, Forecast Pro, and ACT!

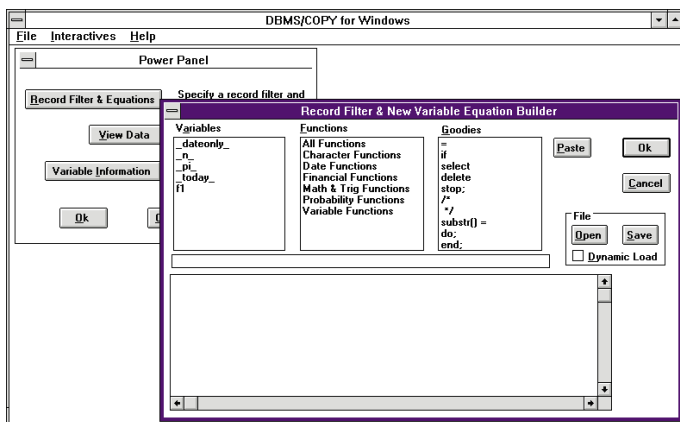
DBMS/COPY has a built-in query builder for reading packages with ODBC drivers. Complex SQL queries can also be created with the integrated query builder.

Price: DBMS/COPY for Windows US\$295; DBMS/COPY for Windows and DOS (new version 5) US\$395; Windows upgrade US\$149; Windows and DOS upgrade US\$199. Prices do not include shipping charges.

Contact: Conceptual Software, Inc., 9660 Hillcroft #510, Houston, TX 77096

Phone: (800) 328-2686 or (713) 721-4200

Fax: (713) 721-4298



Delphi TOOLS

New Products
and Solutions



New Delphi Books

The Tao of Objects

By Gary Entsminger
M&T Books

(ISBN: 1-55851-412-0)

The Tao of Objects is an introduction to object-based software design and object-oriented programming. It uses examples from Delphi, C++, Visual Basic, and dBASE.

Price: US\$27.95 (245 pages)

Instant Delphi Programming

By Dave Jewell
Wrox Press

(ISBN: 1-874416-57-5)

Instant Delphi Programming shows experienced developers how to program in Windows using Delphi. It's equipped with three appendices: Delphi for Pascal Developers, Delphi for Visual Basic Developers, and Delphi Database Connectivity.

Price: US\$24.95 (400 pages)

VisualPros: Developer and Video Tools for Delphi

Shoreline Software of Vernon, CT is shipping *VisualPROS for Delphi*, two sets of tools written and optimized entirely in Delphi for Delphi: *Visual Application Enhancement Widgets* and *Visual Video Widgets*. The sets include application development and full motion video tools for Delphi application developers.

The Visual Application Enhancement Widgets package includes: VisualProgress, a customizable progress bar component; a VisualTileBack component for placing bitmaps (tiled, centered, stretched) and gradient fills into forms and dialog boxes; a VisualHelpCloud component for bubble help support including links to the WINHELP Engine; a VisualRegINI component that provides an interface to the flat-file INI structure and the advanced Registry structure of Microsoft Windows and Windows NT; VisualPrompt-Edit, an enhanced data-entry component; VisualLED, an LED display component;

VisualButtonExt, an enhanced bitmap button component; and VisualMenuExt, a tear-and-tack menu component.

The Visual Video Widgets package includes: a Visual-Capture component for capturing video from Delphi; a VisualPlayBack component for viewing video clips; a VisualAnnotate component for adding voice and/or text descriptions to video clips; and a VisualQuery component for managing video or

frames stored in a relational database.

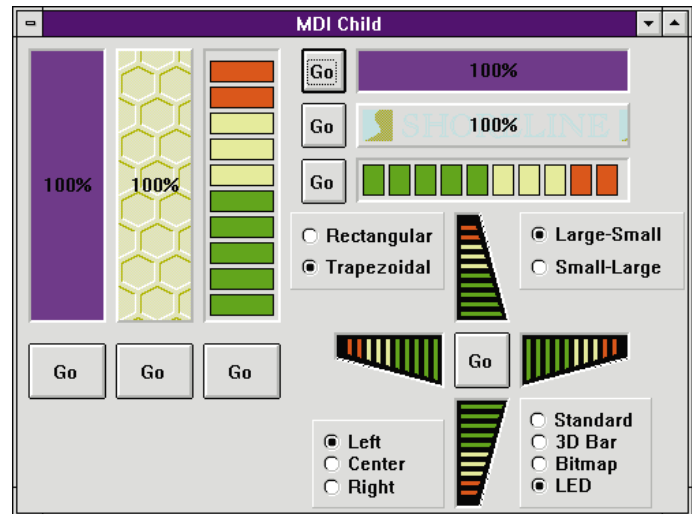
Price: Visual Application Enhancement Widgets, US\$79.99; Visual Video Widgets, US\$129.99.

Contact: Shoreline Software, A Transdominion Company, 35-31 Talcottville Road, #123, Vernon, CT 06066

Phone: (800) 261-9198 or (203) 870-5707

Fax: (203) 870-5727

CompuServe: 70541,2436



MicroHelp Ships Communications Library 3

MicroHelp Inc. of Marietta, GA has announced the release of *Communications Library, Version 3*. The new version includes a VBX control, 16 and 32-bit OLE controls, and an additional DLL

optimized for 386 enhanced-mode performance.

Comm Library 3 is a complete set of communication routines that offer a wide variety of terminal emulations and file transfer protocols to help programmers add data communications to applications. Written in C, Comm Library's file transfer routines work in the background allowing other programs to continue foreground tasks. The following terminal emulations and file transfer protocols are supported: ANSI color/graphics, TTY, VT52, VT100, VT220,

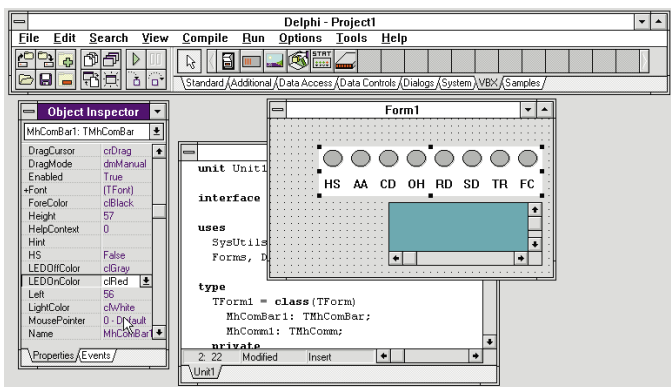
CompuServe B+, Kermit, Xmodem/CRC/1K, Ymodem/Ymodem-G, and Zmodem (with AutoRecovery).

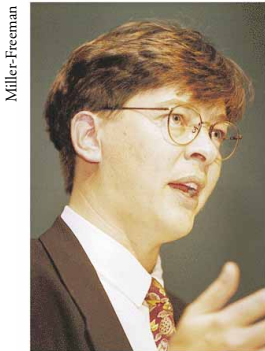
Comm Library 3 includes over 150 of the most popular modem initialization strings, and supports speeds up to 256,000 bps.

Price: US\$149; Comm Library 2 upgrade, US\$69.

Contact: MicroHelp Inc., 4359 Shallowford Industrial Parkway, Marietta, GA 30068

Phone: (800) 777-3322





Miller-Freeman

Borland's Anders Hejlsberg, the Principal Engineer of Delphi.

Alistair Ramsey and Ian Sharp of **Dunstan Thomas**, in association with **Borland International**, took second place in the **EXE Developers Challenge** with an application written in **Delphi**. Two-time world champion, Bruce Lomasky (an American) won the event, held annually in Sandown Park, England. The competition benefited the Royal National Institute for the Blind and was sponsored by EXE magazine.

SQA Inc. and Borland International have announced an agreement to enhance the integration of their **SQA TeamTest** and **Delphi** products. SQA TeamTest is built on a network repository that integrates six major testing areas: test planning, test development, test execution, results analysis, defect tracking, and summary reporting and analysis.

Delphi Unveiled, Main Attraction at Software Development 95

San Francisco, CA — February 14-16, 15,000 attendees packed San Francisco's Moscone Center for Software Development '95. And while over 200 companies participated, the undeniable stand-out was Borland International powered by the popularity of its new Rapid Application Development environment, Delphi, which was officially unveiled at the conference.



Miller-Freeman

The Delphi roll-out event occurred on the evening of the 14th and featured: Starfish Software's CEO Philippe Kahn [see associated news items on [page 7](#)]; Borland's VP of Research and Development, Paul Gross; and Borland's Client/Server Group Product manager, Zack Urlocker. Beta testers also demonstrated a number of impressive Delphi applications that have already been deployed.

The star of the roll-out was the Principal Engineer of Delphi, Anders Hejlsberg, who gave a brilliant demonstration of the new product. As a finalé, Hejlsberg loaded the Delphi project into Delphi, commenting "As you can see, it's a rather large project." The estimated 1,500 attendees responded with a standing ovation.

The launch was followed with a dinner party at the San Francisco Exploratorium, with live music provided by a steel-drum band. According to Cindy Blair of Miller Freeman, over 35 busses were used to shuttle attendees to the Delphi party. The party also featured a battery of computers connected to Borland's World Wide Web site where Delphi was undergoing a "virtual launch".

Software Development 95: Pandemonium reigns at the Borland booth.

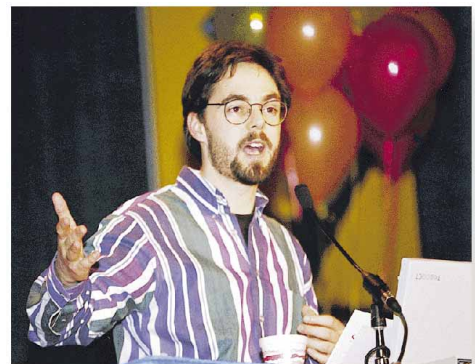
The Borland booth overflowed all three days of SD 95. When the doors opened each morning, attendees literally ran to the Borland booth to see Delphi demonstrated. At the booth, it was standing room only for the continuous Delphi demonstrations.

A staff of over 20 Borland employees managed the booth, with six Delphi demonstrators keeping the 60 amphitheater seats full. Borland's Database Public Relations Manager Tami Casey said "We were really pleased. There was a lot of traffic at our booth and there was a lot of excitement. We actually had people running to the Delphi booth."

Ian Robinson, Borland's Senior Product Marketing Manager for Delphi, said Borland had expected Delphi to be well received, but the launch event was better than they anticipated: "Delphi certainly was the key launch of the show." Robinson said

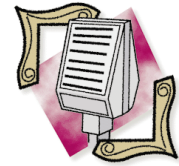
Borland gave away 3,000 pre-release copies of Delphi to launch attendees and those attending Delphi demonstrations. Over half were given away the first day.

Shoreline Software's Glenn Field said the response to their Delphi-compatible Visual Pros at SD '95 was outstanding. He said there was "tremendous momentum from the crowd, it was very intense." With two hours of demonstrations each day, the Shoreline Software booth attracted at least 100 people per demonstration. And since the launch, Field says he's been swamped. "Every time I get on CompuServe [in the Borland Delphi Forum] I'm getting buried in messages."



Miller-Freeman

Borland's Client/Server Group Product Manager, Zack Urlocker.



MicroHelp Inc. has announced that three of its Windows development tools — **Compression Plus 4**, **SpellPro 2**, and **Communications Library 3** — now support **Borland International's Delphi**. In addition, MicroHelp's **VBTools 4** and **Fax Plus** products will add Delphi examples when upgraded for **Windows95**. For more information, call MicroHelp at (800) 777-3322, or (404) 516-0899.

Borland International has announced the resignation of **David Liddle**, 49, from its board of directors. Liddle cited a conflict of interest due to his position on **Sybase, Inc.'s** board of directors. Sybase recently finalized its acquisition of **Powersoft Corporation**, a company that competes with Borland. Liddle's experience includes 10 years at **Xerox Palo Alto Research Center**. He is also on the board of directors of **Broderbund Software** and **NetFrame Systems**.

Borland Wins Lotus Case

Scotts Valley, CA — The United States Court of Appeals for the First Circuit has reversed a District Court ruling that stated the Quattro and Quattro Pro spreadsheet products, formerly developed and marketed by Borland International Inc., infringed the copyright of Lotus 1-2-3.

In written opinions, all three appellate judges held in favor of Borland, overturning previous decisions by the district court. (The lower court had held that a single element of Borland's Quattro and Quattro Pro spreadsheets — the menu command hierarchy — infringed the copyright of Lotus 1-2-3.) In its opinion, the court concluded, "Because we hold that the Lotus menu command hierarchy is an uncopyrightable subject matter, we further hold that Borland did not infringe the copyright by copying it."

Philippe Kahn, chairman of Borland, was pleased with the ruling. He said it represented a clear victory for consumers, computer programmers, and open systems. Kahn also said he believed it will foster a more competitive software industry and promote innovation. Borland's President, Gary Wetsel agreed and said the decision removes a huge cloud of uncertainty from Borland and will allow them to move forward with their business plans.

In regard to legal guidelines, Robert Kohn, senior vice president of Corporate Affairs and general counsel for Borland, said the First Circuit's decision established clear guidelines that reflect Congress' intent for what is and is not copyrightable in a

computer program. In his opinion, this decision is consistent with other federal circuit courts' rulings and will now provide the software industry with a clear precedence. Kohn also said Borland may request Lotus to pay their attorney fees.

Although Lotus can appeal this decision, Kohn doubts they will. An appeal by Lotus would require several steps and a large financial commitment.

When the District Court ruled that the Command Hierarchy in Borland's spreadsheet products infringed the copyright of Lotus 1-2-3 in August 1992, Borland voluntarily removed this feature from shipping versions of the product. The

court reaffirmed its decision in July 1993.

Then in August 1993, the Federal District Court ruled that another compatibility feature in Quattro Pro and Quattro Pro for Windows infringed the copyright of Lotus 1-2-3. The court subsequently entered an injunction against Borland, barring further sales or distribution of then current versions of Borland's spreadsheet products. In response, Borland shipped new versions of Quattro Pro that didn't include the features found to be infringing, and announced it would seek an immediate appeal. Borland sold its Quattro Pro spreadsheet to Novell Inc. in March 1994.

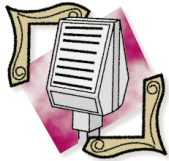
InstallSHIELD Now Supports Delphi

Schaumburg, IL — Stirling Technologies, Inc. has announced that its InstallSHIELD installation toolkit will support Delphi. As part of their relationship with Borland, Stirling will create a customized template for Delphi applications, similar to the template available in the Borland dBASE Developer's Kit. This template will be included on the Delphi CD ROM. Developers will be able to make changes to the template and create a customized Delphi installation. ISVs will benefit from InstallSHIELD's ready-to-use installation script.

Stirling will also create a visual tool based on InstallSHIELD for creating Delphi installations. It will provide a point-and-click environment. According to the company, they have encapsulated all the

specialized installation requirements for developing Delphi applications — developers point to the project file for the application and the installation toolkit does the rest.

InstallSHIELD features three customizable templates, 50 pre-built dialog boxes, and is capable of detecting over 50 parameters on the end-user's system. In addition, InstallSHIELD can automatically add program folder icons; modify WIN.INI, CONFIG.SYS, and AUTOEXEC.BAT files; call any DLLs regardless of format; provide compression/decompression for your application files better than or equal to PKZIP; and keep the user informed with built-in graphical feedback objects. For more information, call Stirling at (708) 240-9111.



Asymetrix Corp. is adding **Delphi** support to its database design CASE tools, **InfoModeler 1.5 Desktop Edition** and **InfoModeler 1.5 Client/Server Edition**, expected to ship in the second quarter of 1995. InfoModeler uses sample data and English statements to create a model from an information system, then validates and transforms the model into a database design.

The Chameleon Group has established a clearing house for **Delphi third-party software** for developers in the Central European German-language countries. For more information contact The Chameleon group at 01149 211 379057, or send a message to Joe Williams via CompuServe at 74372,3060.

Mortice Kern Systems Announces Alliance with Borland

Waterloo, ON — Mortice Kern Systems has announced a new strategic alliance with Borland International Inc. MKS has developed hooks to seamlessly integrate MKS Source Integrity into Delphi. MKS Source Integrity's fully-documented open API will give

Delphi users all the benefits of MKS' configuration management system from within the Delphi environment.

According to MKS, Source Integrity increases the opportunities for VARs to provide a complete development solu-

tion to their customers. With the growing popularity of configuration management in today's development industry, more companies are seeing the need for a truly open configuration management system such as MKS Source Integrity.

LBMS Announces Alliance with Borland

San Francisco, CA — LBMS, Inc., a provider of Windows-based client/server CASE and process management tools, has announced a cooperative development and marketing agreement with Borland International Inc.

Under the agreement, LBMS will provide the tools for Borland to develop SE/Open for Delphi, a bi-directional interface between LBMS' client/server application development tool, Systems Engineer (SE) and Borland's Delphi.

SE/Open for Delphi will provide developers with a seamless environment for the complete analysis, design, and development of both the client and server portions of LAN-based applications. SE/Open for Delphi will facilitate object management, re-use management, database design, and prototyping management. According to Borland, SE/Open for Delphi will combine Delphi's visual development environment with LBMS' CASE modeling and repository capabilities.

Specifically targeting team-based application development, SE is a comprehensive set of open, integrated client/server CASE tools that address the full development cycle. It includes graphical

user-interface (GUI) modeling facilities, as well as server design and generation capabilities based on client/server development techniques created by LBMS.

SE/Open for Delphi is scheduled to be available during the first half of 1995 and sold jointly by LBMS and Borland. It is priced at US\$10,000 per server.

Novell's Tuxedo Products Support Delphi

Summit, NJ — Novell, Inc. has announced a business relationship between its Tuxedo product line and Borland International, Inc., enabling Tuxedo products to support Delphi. Tuxedo is an application development and run-time architecture for implementing client/server applications and components of the enterprise computing solution set, including NetWare and UnixWare.

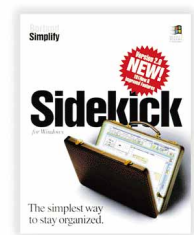
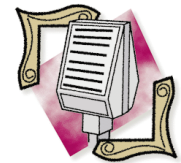
With Tuxedo, programmers can currently choose from 35 platforms and can operate with all clients on today's market. Tuxedo and Delphi together provide rapid application development and client/server scalability to create a software development environment for programmers producing line of business applications.

Tuxedo System 5, the latest

MKS Source Integrity features include complete project management facilities, Visual Merge, new reporting capabilities, event triggers, a new configuration language, integration into Visual C++ and Borland C++, an automated building process, file promotion, and NetWare-specific functionality. MKS Source Integrity is available for DOS, OS/2, Windows, Windows NT, and UNIX. For more information, call MKS at (800) 265-2797.

release of Tuxedo, gives programmers the flexibility to deploy large distributed systems and easily expand or change their system with one of Tuxedo's newest features, /Domains. With /Domains, programmers can manage Tuxedo servers in administratively autonomous groups called domains, and set parameters for the interactions between domains. Tuxedo System 5 also extends Tuxedo interoperability to include DCE and other non-Tuxedo environments.

Tuxedo system provides an architecture for implementing business applications in distributed, client/server environments. Tuxedo is also a transaction processing environment that provides electronic commerce for the banking, telecommunications, finance and retail industries.



Starfish Software began shipping Sidekick 2.0 on March 1.

Borland Launches World Wide Web Site

Scotts Valley, CA — Borland International Inc. has established an Internet World Wide Web (WWW) site, Borland Online. Currently up and running, it is designed to grow into a comprehensive resource for Borland developers, providing access to everything from the

latest product information and technical documents to information on seminars and training locations. Borland Online can be accessed on the Internet WWW at <http://www.borland.com>.

Borland Online includes links to Borland's existing Internet

FTP (File Transfer Protocol) site. Borland Online services include: news, company information, technical information, product information, program services, feedback, and files. For more information about Borland Online, contact Cindy Martin at webmanager@borland.com.

TurboPower Acquired by Casino Data Systems, Plans Delphi Tools

Colorado Springs, CO — TurboPower Software has announced that it was acquired by Las Vegas-based Casino Data Systems. TurboPower will continue to develop and market programming tools and components for Pascal and C++ programmers. It will operate in Colorado Springs, Colorado as TurboPower Software Company, a wholly-owned subsidiary of Casino Data Systems. In addition, it will retain its existing staff and immediately seek to hire additional programmers and support staff, particularly in the area of Delphi components.

Mr. Kim Kokkonen, founder of TurboPower and current President of TurboPower Software Company, said the market for Windows programming components has grown faster than their ability to

support it, and that the acquisition provides new opportunities for TurboPower Software. Specifically, he said Borland's Delphi compiler will expand the need even further and their experience with Pascal will enable TurboPower to

produce Delphi components.

Casino Data Systems is using Borland's Delphi as the development platform for its new Windows-based OASIS system scheduled for general release during the third quarter of 1995.

Borland Sells Sidekick, Dashboard to Starfish

Scotts Valley, CA — Borland International Inc. has announced the sale of its consumer software products, Sidekick for Windows and Dashboard for Windows, to Starfish Software [see [separate news item](#) on this page]. Borland also announced it has discontinued its Simplify consumer products division. According to Borland president Gary Wetsel, the company will focus on the software developer market, and the Borland Simplify division was not consistent with these plans.

As part of the transaction, Borland will receive a minority equity position in Starfish Software, but will not be involved in the management or operation of the venture. In addition, a majority of Borland Simplify's 35 employees will be offered positions with Starfish Software.

Starfish Software will now support Sidekick and Dashboard customers, as well as contractual obligations. As part of the agreement, Borland retains the rights to bundle Sidekick and Dashboard with its other products through September 1995.

Kahn Announces Starfish Software

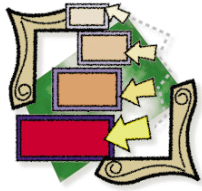
Santa Cruz, CA — Philippe Kahn has announced a new software venture, Starfish Software, to be located in Santa Cruz, California.

According to Kahn, Starfish Software was created to develop simple, high-quality tools for PC users worldwide. Specifically, Kahn noted the increasing interest in the

Internet and said Starfish products will "pioneer electronic commerce across the Internet." Kahn will serve as Chairman of Starfish Software, and remain Chairman of Borland International. He will also continue to serve in an advisory role in Borland's long-term strategic planning and technology division.

Kahn also announced that Starfish Software has purchased Sidekick for Windows and Dashboard for Windows from Borland International.

Although the terms of the purchase were not released, Borland International will receive a minority equity position in Starfish Software.



VISUAL PROGRAMMING

DELPHI / OBJECT PASCAL | BEGINNER

By *Doug Horn*

Delphi: A Visual Tour

Getting to Know the Neighborhood

Know thyself. — Inscription at Delphi

Twenty-five hundred years ago, Delphi was a city in Greece, famous for its enigmatic oracle of Apollo. Today, Delphi lends its name to a new and robust software development system. Because it's so new, the Delphi of today is nearly as mysterious as the city of legend.

The power of Delphi adds new muscle to visual programming. It includes full database and SQL compatibility with simplified programming, yet Delphi also allows developers to create custom DLLs and components — something never possible with Visual Basic.

But before developers can take advantage of Delphi, they must first understand it. Although some components of the Delphi interface may be familiar to many users, others may be completely unknown. This article will walk new users through Delphi, point out some of its more important elements, and explain how they work.

Essential Delphi Elements

All tours of Delphi must begin at its desktop, shown in [Figure 1](#). This is Delphi's main interface, and the only portion of the program that must always be running. It includes the menu, SpeedBar, Component Palette, Object Inspector, and a blank form.

Delphi's menu is similar to most Windows applications. It controls generic functions such as opening files, searching for text strings, and viewing program windows. The menu also performs many tasks distinctive to Delphi. For example, you can compile, debug, and run Delphi applications, customize the Delphi environment, and launch Delphi-related tools.

Delphi also offers SpeedMenus (pop-up menus) that include commonly used design commands. To access a SpeedMenu, you can right-click on the SpeedBar, Component Palette, Code Editor, or on any component.

The SpeedBar

The SpeedBar is below and to the left of the menu. This group of buttons allows instant access to a number of program functions. SpeedBar buttons (and Component Palette buttons) have bitmapped icon images instead of labels. If a button function is unclear from its picture, resting the mouse pointer over the button will call up a Help Hint with a short description. (*Help Hint* is the Delphi term for what is commonly known as “balloon help”.)

The SpeedBar contains buttons for simple file management — including creating, opening, or saving a new project or other file — and other basic functions. The SpeedBar is fully configurable. To add, remove, or reposition command buttons on the SpeedBar, right-click anywhere on the SpeedBar and select **Configure** from the SpeedMenu. This calls the Speedbar Editor dialog box (see [Figure 2](#)) and puts the SpeedBar in configure mode.

Now you can reposition buttons by dragging them to another location on the SpeedBar, or remove them by dragging them off the SpeedBar. To add new command buttons, select the appropriate category and command from the Speedbar Editor list boxes, then drag the command icon to a position on the SpeedBar. You can return the SpeedBar to its shipping defaults by clicking the **Reset Defaults** button.

You can also resize the SpeedBar. Place the mouse pointer over the separator bar between the SpeedBar and the Component Palette, and the pointer becomes a bi-directional arrow. Click and drag the separator bar to expand or shrink the SpeedBar as needed. You can remove the SpeedBar altogether by selecting

Hide from its SpeedMenu. To replace it, select **View | Speedbar** from the Delphi menu.

The Component Palette

Another essential element of Delphi is the Component Palette, located below the menu and next to the SpeedBar. The Component Palette has eight pages that are selected by clicking on their respective tabs. The first page — named **Standard** — is shown in [Figure 3](#). It contains standard Windows controls that you can simply select and apply to a form using the mouse.

Other pages of the Component Palette take the idea further. The **Additional** page for example, contains components for bitmap buttons, speedbar buttons, graphic images, shapes, a scrollbar, etc. Also of interest are the **TabSet**, **Notebook**, and **TabbedNotebook** components used to create a tabbed, paged interface like that of the Component Palette itself.

The **System** page has components for a timer, file list box, directory list box, drive combo box, multimedia control, and OLE and DDE components. Among other things, these components make it short work to create a multimedia application, or add OLE and DDE capabilities to your applications. [An example of a multimedia application created with Delphi is presented in Kenn Nesbitt's article "[A PC Sound Garden](#)" beginning on page 26.]

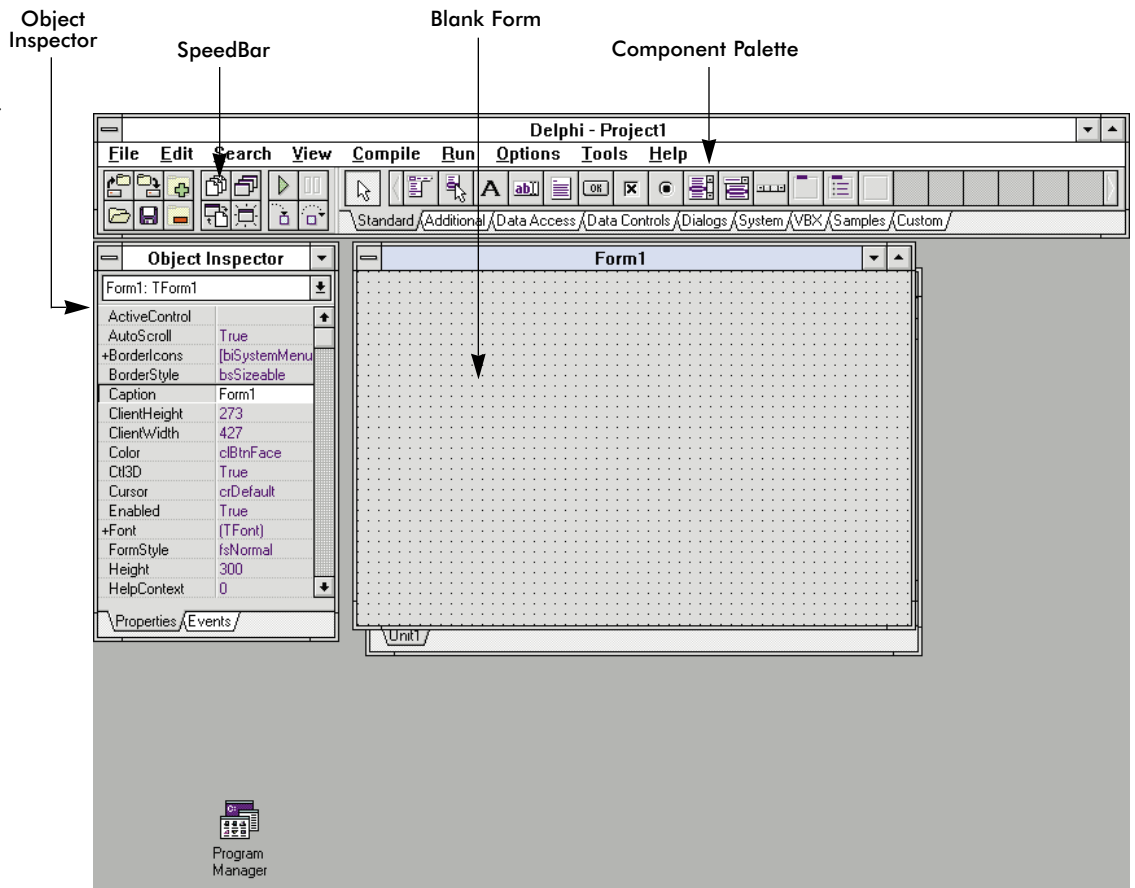


Figure 1: Delphi as it appears when first opened in its default aspect.

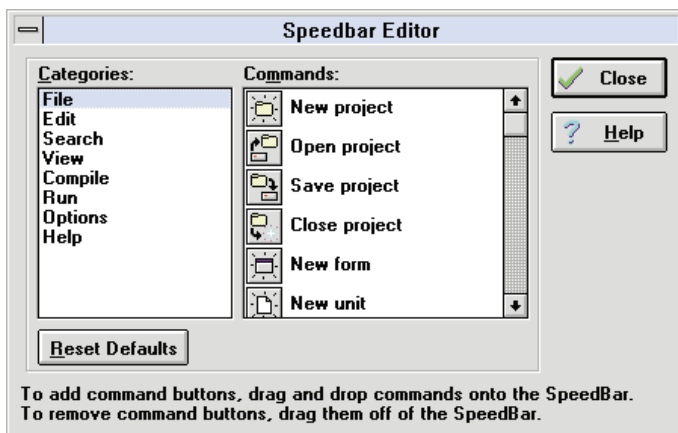


Figure 2: In the Speedbar Editor dialog box, you can add, remove, or reposition SpeedBar buttons.

Database developers will be particularly interested in the Component Palette's Data Access and Data Controls pages (see Figures 4 and 5 respectively). The Data Access page contains components that describe the source of data: Database, Table, and Query. It also contains a special component, DataSource, that acts as a conduit between these data repositories and a form's user-interface components.

The Data Controls page contains the data-aware equivalents of many of the components on the Standard page. For example: DBLabel allows you to place a table-driven label on a form; DBEdit is a standard edit field that is tied to a particular field in a table or view; DBNavigator is a VCR-style table navigation control; and DBImage displays a graphic stored in a table. [An example of these components at work is presented in Dan Ehrmann's article "Data-Aware Delphi" beginning on page 21.]

The Samples page contains custom components (e.g. gauges, a spinbutton, a calendar, etc.). The Dialogs page contains ready-made dialog boxes for opening a file, saving a file, searching for a file, font selection, printer setup, etc. A VBX page is available for adding your favorite VBX controls. (Four are provided to get you started.)

Like the SpeedBar, the Component Palette can be configured. You can add and remove components from each page, rearrange their order, and even add, remove, or rename pages. To configure the Component Palette, right-click anywhere on the Component Palette and select **Configure** from the SpeedMenu. The Environment Options dialog box will appear with its Palette page displayed (as shown in Figure 6).

Notice the **Pages** and **Components** list boxes. The page order can be changed by selecting a page and dragging it to a new position in the **Pages** list box. You can also reorder the pages by highlighting a page and clicking the up or down arrow buttons.

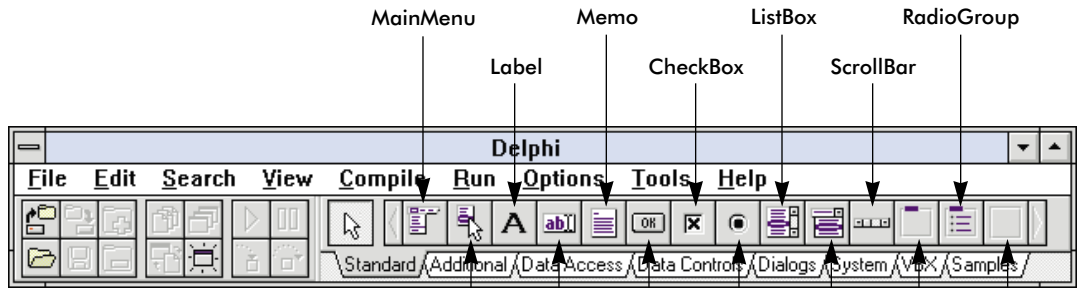


Figure 3: The Standard page of Delphi's Component Palette.

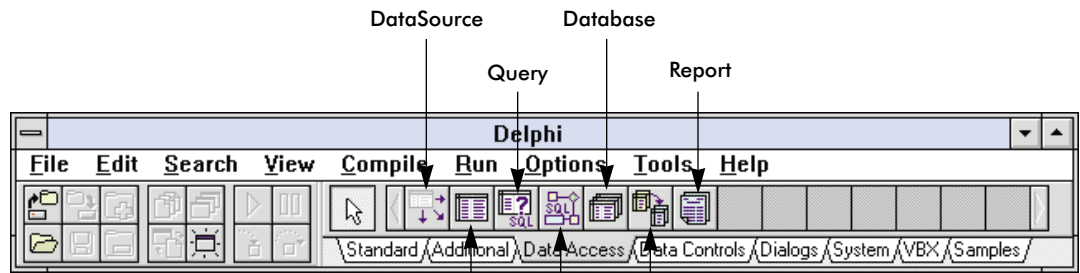


Figure 4: The Data Access page of the Component Palette.

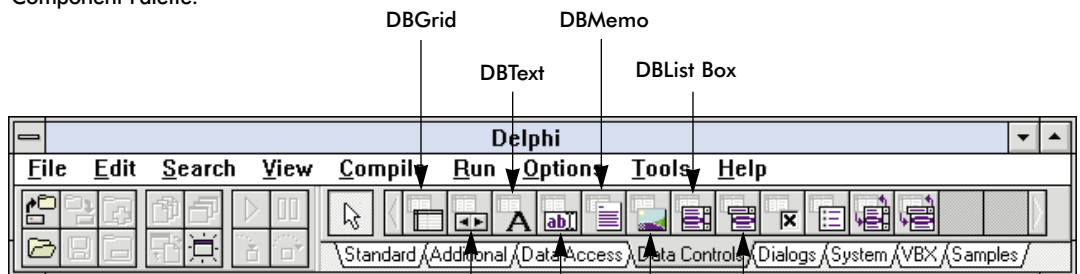


Figure 5: The Data Controls page of the Component Palette.

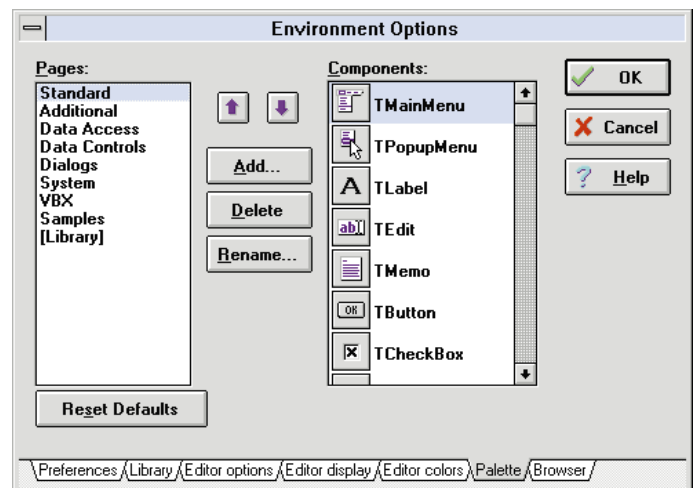


Figure 6: The Component Palette can be configured via the Palette page of the Environment Options dialog box.

To rename a page, simply click the **Rename** button and the Rename page dialog box will prompt you for a new name. Use the **Add** and **Delete** buttons to create or remove tab pages. Tab pages must be emptied of all components before they can be deleted.

You can also remove a component by selecting it in the **Components** list and clicking the **Delete** button. Components deleted in this fashion are still available to be added to any page of the Component Palette. To replace a deleted component, click [Library] in the **Pages** list box. All components in the component library will be displayed in the **Components** list box. To re-add a deleted component, drag it from the **Components** list box and drop it on the desired page in the **Pages** list box.

A component can only be on one page at a time. Components can be reordered on the page (in the same manner as pages are reordered), and dragged from one page to another. To do this, select a component in the **Components** list box and drag it over the name of the destination page in the **Pages** list box. Click the **OK** button to accept the changes, or the **Cancel** button to reject the changes. The **Reset Defaults** button restores the Component Palette to its “factory settings”.

Installing a component is an altogether different operation and outside the scope of this article. [For a step-by-step description, see Gary Entsminger’s article “[Approaching Components](#)” beginning on page 15.] Installing a Visual Basic (VBX) component is a similar process.

Forms

Delphi is a visual programming environment, and characteristically, visual programming is based on forms. Delphi can create programs without forms (e.g. DLLs), but most Delphi applications use forms as their main interface.

A form is a canvas containing all the components an application uses. It is therefore, the primary portion of the programming interface. When Delphi starts, it automatically opens a blank form (as shown in [Figure 1](#)). You can start building your application immediately by dropping components on the form.

To place a component on a form, click the corresponding component on the Component Palette. For example, to place a Button component on a form, click the Button component on the Component Palette and then click again anywhere on the form. Or, simply double-click a component to place it in the center of the form. Press **Shift** before clicking on a component to place multiple instances of the same component on the form.

Once placed, user-interface components (i.e. controls) like a Button can be easily moved and resized with the mouse. Non-UI components, such as a DataSource component, cannot be resized.

You can create a new form by selecting **File | New Form** from the menu or by clicking on the New Form button on the SpeedBar.

However, rather than opening blank forms and designing them from scratch, Delphi users can use form templates.

Form Templates

Many forms have standard functionality and contain common user-interface elements. For example, many simple dialog boxes display a message and feature **OK**, **Cancel**, and **Help** buttons.

Form templates are pre-built forms made available in a Browse Gallery dialog box (as shown in [Figure 7](#)). The “Standard dialog box template” for example (seen in the middle of the dialog box), creates the dialog box just described. To complete this form, a developer only has to add the appropriate text to it at design or run-time.

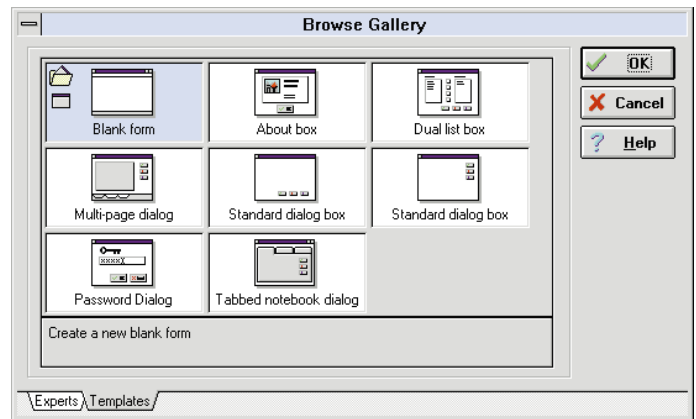


Figure 7: Selecting a form template from the Browse Gallery.

To use form templates, the Browse Gallery must be enabled. It’s enabled by default when Delphi is shipped. That is, the Browse Gallery is displayed whenever **File | New Form** is selected from the menu. This is controlled on the Preferences page of the Environment Options dialog box. Within the **Gallery** group box, select the **Use on New Form** check box.

You can also reorder, edit, delete, and add form templates that appear in the Browse Gallery. To do this, select **Options | Gallery** from the menu to display the Gallery Options dialog box. The selection of Delphi form templates will be displayed on the Form Templates page (as shown in [Figure 8](#)). You can designate which form will be the default by selecting it and clicking the **Default Main Form** button.

You can also modify an existing form template or create a new form and add it to the Browse Gallery. After changing or creating a form, simply save it by selecting **Save As Template** from the SpeedMenu or **Edit | Save As Template** from the menu.

The Object Inspector: Properties

A Delphi application consists of components. Buttons, labels, combo boxes, queries, etc., are all components. The Component Palette contains the blueprints or *classes* of components. When you select a component on the Component Palette, Delphi creates an object based on that component’s class.

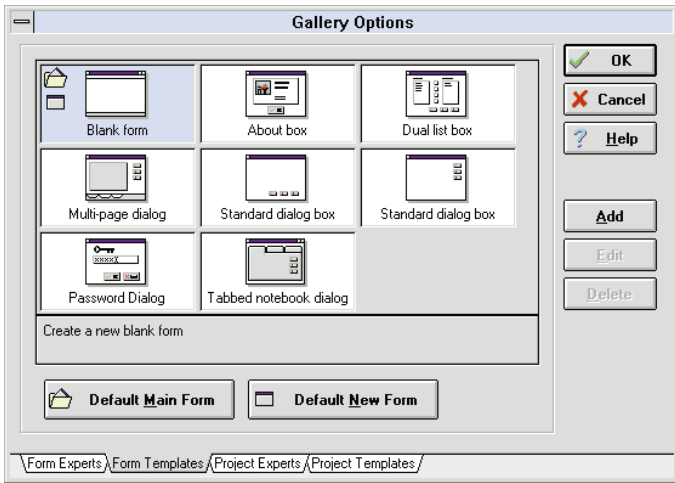


Figure 8: The Form Templates page of the Gallery Options dialog box.

Each component in a Delphi application has properties that describe its appearance and functionality. For example, a Button has properties that affect its appearance (e.g. *Height*, *Width*, and *Caption*). Other properties, such as the button's *Name* (e.g. *Button1*) are transparent to the user, but provide important information to the application.

Components are defined by their properties and controlled by their methods. The bulk of a Delphi application's program code manipulates component properties. At design time, a selected object's properties are displayed in the Object Inspector (shown in Figure 9). Here, developers can modify properties that immediately affect the component's appearance or behavior.

For example, you can change a button's *Caption* property by simply clicking on that property in the Object Inspector and typing a new caption in the Value column. The new value will be reflected in the button as it is typed. Properties with a finite list of choices, such as *Cursor* (which determines the shape of the mouse pointer), provide a drop-down list of options (e.g. *crIBeam*, *crHourGlass*, and *crArrow*).

Other properties (e.g. *Font*) themselves contain properties, called nested properties (e.g. *Name*, *Pitch*, and *Size* for the *Font* properties). Properties with nested properties are indicated by a plus sign (+) in the Property column. Double-click on such a component to expand the Property column to display the nested properties. You can

also double-click the Value column to display the appropriate dialog box.

To select a component for inspection, click on it. It then has focus in the Object Inspector (i.e. the name of the component is shown in the Object selector). Another way to select a component is to click the Object selector's down arrow and select the component from the provided drop-down list.

You can select several components at once by holding down **Shift** and clicking on each component. When multiple components are selected, the Object Inspector displays only the shared properties. If all components are the same (e.g. all buttons) the Object Inspector will show all properties (except for *Name* and *TabStop* which must be unique to each component).

However, if the components are different — say a Label and an Edit component — the Object Inspector will omit properties that are unique to each component (such as *ReadOnly* and *Text* which apply only to the Edit component). When multiple components are selected, changing any properties will set those properties for all components selected. For example, changing the *Top* property for several components aligns the top of each component to the same line.

It's also important to remember that Delphi is a two-way tool. That is, just as changing a property on the Object Inspector — such as *Top* or *Width* — changes the position or size of the component on the form, dragging the compo-

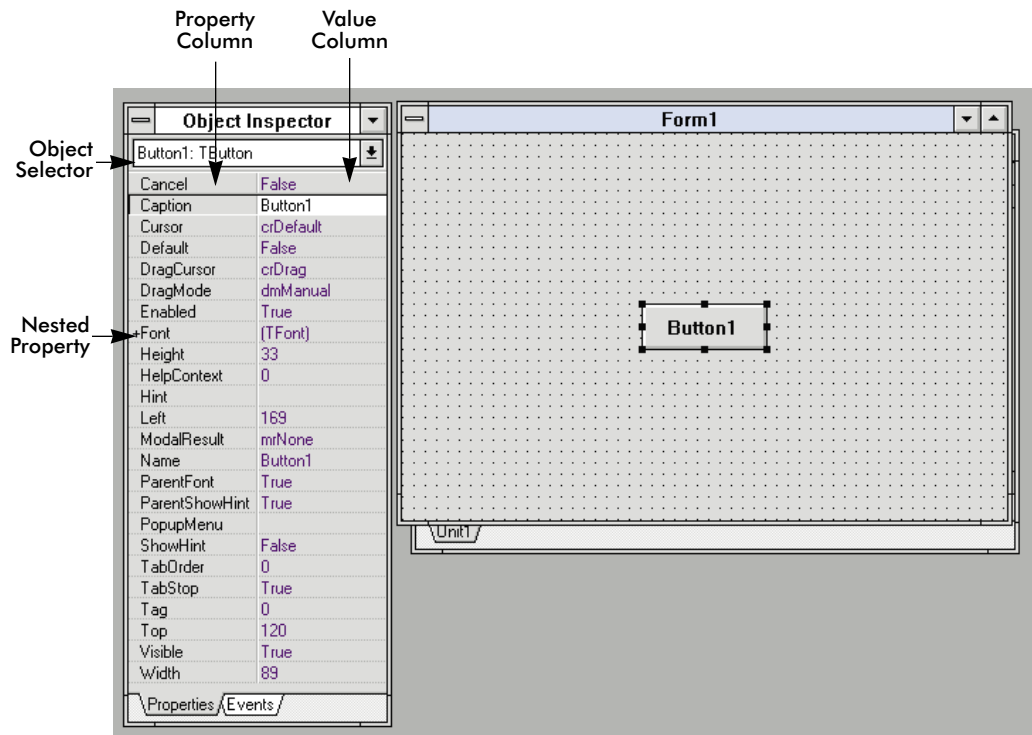


Figure 9: The Object Inspector allows component properties and events to be controlled during project design time. As you can see, the Object Inspector has two tab pages: Properties and Events. Use the Properties page to alter the characteristics or appearance of a component. Use the Events page (shown in Figure 10) to alter a component's associated events.

ment with the mouse will update new properties in the Object Inspector.

The Object Inspector: Events

In Delphi, components are only half the story. The other half are events. An *event* is anything that can happen to a component — for example, the mouse pointer entering a component, a button being pushed, a form opening or closing, or editing the value in an edit box. Every component has certain events associated with it. Some, such as *OnClick*, are common to all components, while others may be specific to a group of components or even a single component.

The Events page of the Object Inspector shows the events associated with each component (see Figure 10). This page is much like the Properties page, in that it allows you to select multiple components and displays only the events common to the selected components.

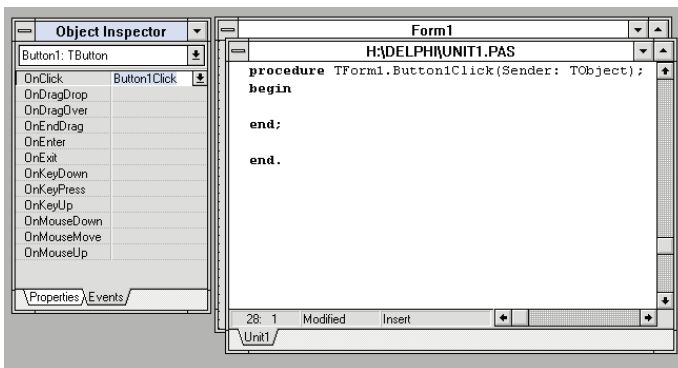


Figure 10: Event handler code is generated automatically by double-clicking the desired event in the Value column in the Object Inspector.

Initially, the values of all events are blank. This indicates that no event handler has been attached to any portion of the form. An *event handler* is a block of Object Pascal code that describes how a component will respond to an event.

To create a Delphi event handler, select a component. Next, choose an event and double-click on the Value column. The Code Editor will appear as shown in Figure 10. Alternatively, double-clicking a component creates an event handler for the most common events to occur for that type of component (e.g. *OnClick* for buttons).

Once an event handler is created, Delphi places the preliminary statements into the program's code listing. Figure 10 shows the code for a button's *OnClick* event with the cursor on the line between **begin** and **end**. Now the developer only needs to write a few lines of code that actually performs the event.

When an event handler is created for multiple components, it applies to all of them. However, rather than creating an event handler for *Button1*, *Button2*, and so on, Delphi can reference multiple events to the same event handler. These are called shared events. Events may be shared by multiple components, such as the *OnClick* event of both a button and menu item call-

ing the form to close. Events can also be shared within a single component, such as the *OnEnter* and *OnExit* events of a field triggering the same block of program code. [A shared events example is presented in Dan Ehrmann's article "Data-Aware Delphi" beginning on page 21.]

The Code Editor

Program code is edited via the Code Editor (Figure 11). The Code Editor lists program code from the unit file of each open form. This unit file contains all the information — from **variable** and **uses** declarations to event handler code — necessary to build and perform the functions on a form. The code for each open form is listed on a separate page within the Code Editor.

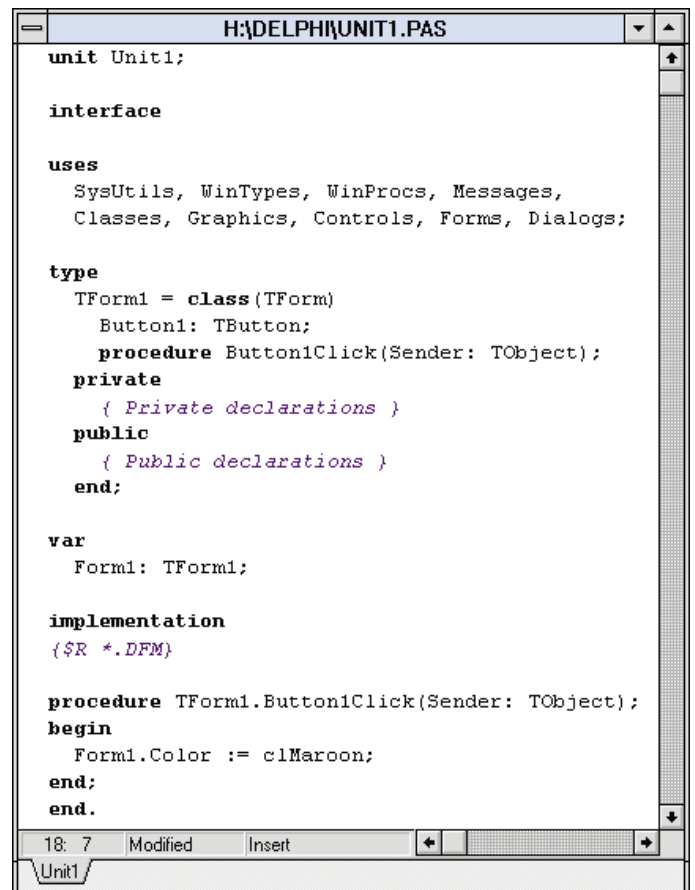


Figure 11: The Code Editor acts as a traditional text editing program, with several smart capabilities added for good measure.

When you create a new form, the Code Editor also creates a new unit file with a .PAS extension, containing the Object Pascal code for the form. As new components and events are added to the form, the Code Editor tracks these and adds additional code to the unit file. This code, along with code added by the user, creates the program. New units can also be created independent of forms when no forms are necessary (e.g. creating customized components, etc.) using the **File | New Unit** command from the menu. In many ways, the Code Editor acts as a traditional text editor, but it also has several advanced capabilities. First, of course, the Code Editor adds program code automatically to declare new types as new components and event handlers are added.

The Code Editor also recognizes reserved words and other types of programming structures, and uses different colors and font attributes to make the code easier to read and understand.

In addition, the Code Editor is fully configurable. As with configuring the Component Palette, selecting **Options | Environment** from the menu calls the Environment Editor. The Environment Editor has three pages pertaining to the Code Editor: Editor options, Editor display, and Editor colors (shown in Figure 12). Using the options on these pages allows you to control and customize a variety of functions to make the Code Editor as easy to use as possible.

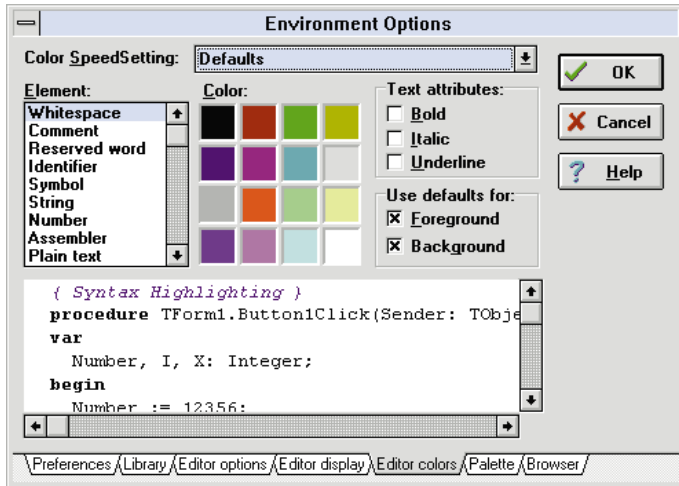


Figure 12: The Code Editor is highly configurable, as evidenced by the Editor colors page of the Environment Options dialog box.

The Project Manager

Delphi groups every application as a project. A *project* can consist of anything from dozens of form, unit, and database files, to a single unit file. However, every application will be a single project and only one project can be open at a time.

The Project Manager (Figure 13) tracks all the files in a project making it easy to open a form or unit file. It also allows users to add or remove files from an existing project. This is a strength, because it allows developers to create form and unit files they can reuse in multiple applications without additional programming.

To view the Project Manager, select **View | Project Manager** from the Delphi menu. The Project Manager consists of a

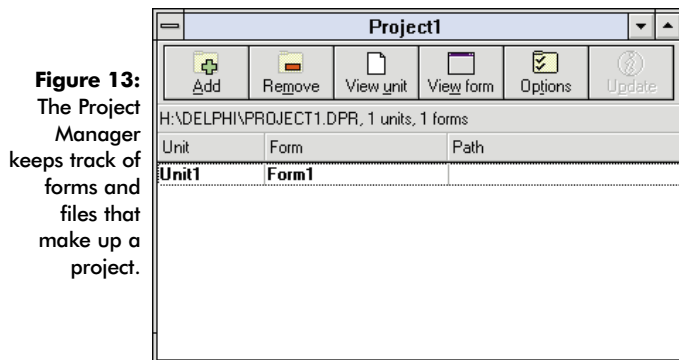


Figure 13: The Project Manager keeps track of forms and files that make up a project.

SpeedBar at the top, project files in the middle, and a unit list at the bottom. The Project Manager SpeedBar buttons allow for project file management.

New or existing projects can be opened by selecting **File | New Project** or **File | Open Project** from the menu, or using the Project Manager SpeedBar buttons. Because only one project can be open at any time, the existing project will be closed before another is opened. Each project has a project file (with the extension .DPR), listing the forms included in the project, and which form opens first (i.e. the *main form*). When a project is created or edited, this code is generated and updated automatically by the Project Manager.

Another important function of the Project Manager is its ability to track the myriad options for configuring project options. Clicking on the Project Manager's check box, SpeedBar button, or selecting **Options | Project** displays the Compiler page of the Project Options dialog box (shown in Figure 14). Here, the developer can configure run-time error checks, enable specific syntactical checks, and customize debugging functionality.

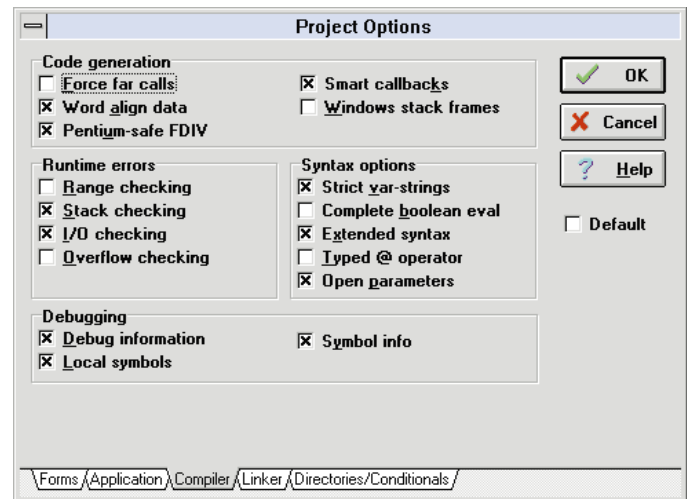


Figure 14: The Compiler page of the Project Options dialog box. You have great freedom to configure how to compile code, link source code, debug files, and much more.

Conclusion

Delphi is a remarkable programming environment that promises to leave its mark on application development. It combines the power of a language compiler, the ease of a visual programming system, and the data control of a desktop database.

And now it's time to begin your Delphi adventure. ▲

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





THE WAY OF DELPHI

DELPHI / OBJECT PASCAL | INTERMEDIATE



By *Gary Entsminger*

Approaching Components

An Introduction to Delphi Component Creation at Run-time

But that, Socrates, Critias said, is impossible, and therefore if this is, as you imply, the necessary consequence of any of my previous admissions, I will withdraw them and will not be ashamed to acknowledge that I made a mistake, rather than admit that a man can be temperate or wise who does not know himself. For I would almost say that self-knowledge is the very essence of temperance, and in this I agree with him who dedicated the inscription “Know thyself!” at Delphi. — Plato

During a recent lecture, I explained how this epigraph from Plato’s *Charmides* foreshadowed Windows programming 1995-style. I used *necessary consequences* (programming logic), *previous admissions* (ancestry and program hierarchies in OOP), *not being ashamed to make mistakes* (revising objects), and *Delphi* (today’s state-of-the-art development environment) as evidence.

Later, I suggested an example of Delphi’s component creation capability. The example would illuminate a few key issues of basic component development, such as how to:

- create properties and methods
- control access to properties
- read and write component properties
- invoke (or send messages to) component methods

In this article, we’ll do exactly that. We’ll create several very simple components, and learn a few component development basics along the way.

Creating Components

Components are the building blocks of Delphi applications. You build an application by adding components to forms and using the properties, methods, and events of the components. All components have the *Name* and *Tag* properties in common. Most components are available from the Component Palette. Some components (*TApplication*, *TMenu*, *TMenuItem*, and *TScreen*) are available only through code.

You create a new component using these three general steps:

- 1) Derive a new component object from an existing component
- 2) Modify the component
- 3) Register the component

Although each component in this project resides in a separate unit, you can put more than one component in a unit. After you create a component, compile and install it onto the Delphi Component Palette. To use the component, select it from the Component Palette and add it to a form.

Let's begin by creating a new project by selecting **File | New Project** from the menu. Delphi will automatically create a form and unit. We don't want them for our task, so close them without saving them. Then save the project as PropDemo.DPR.

The next step will be to create a component. But first, let's discuss components in general and their place in Delphi development.

The TComponent Class

You can create a new component directly in Object Pascal code using the Code Editor, or you can use the Component Expert. Since you will always derive a new component from an existing one, the Component Expert saves time by providing a template based on the selected ancestor component. The Component Expert uses this ancestor component to determine the basic behavior of the new component.

For example, if you want a bare-bones component, derive the new component from *TComponent*. The *TComponent* type is the abstract starting point for all components. Every item used in the Form Designer is a descendant of *TComponent*. It gives its descendants the basic properties and behavior necessary to interact gracefully with Windows.

If you want a new component to have more than basic *TComponent* behavior, select an ancestor at a different level in the *TComponent* hierarchy. A *TControl* component, for example, is the abstract base type for all controls. *Controls* are visual components, meaning the user can see and manipulate them at run-time.

The *TControl* class supplies all the properties, methods, and events that any control needs. Properties, methods, and events required by all controls have been declared as public and appear in all controls. Other items that might be useful to controls have been declared as protected. This means that you can publish them in the components you derive from *TControl* or one of *TControl*'s descendant types. [For more information about the **public** and **protected** directives, see Jim Allen and Steve Teixeira's article "[Component Basics](#)" beginning on page 44.]

Delphi supplies over 70 pre-defined descendants of *TControl*, for example — *TBitBtn*, *TButton*, *TCheckBox*, *TColorDialog*, *TComboBox*, *TFontDialog*, *TForm*, *TGroupBox*, *THeader*, *TImage*, *TLabel*, *TListBox*, *TMainMenu*, and *TMediaPlayer*. You can derive as many more new *TControl* descendants as you want.

Delphi makes it easy to start at the most convenient level you need. As already mentioned, if you need a bare-bones component, start with *TComponent*. If you need a more complex component that's similar to an existing component, begin with it. For example, if you need a new kind of label, begin

with *TLabel*. If you need a new kind of memo, begin with *TMemo*, and so on. A rule of thumb is to take as much behavior as you can from an ancestor, but don't take anything you don't want. It's awkward to remove behavior once you've derived a new component.

The Simplest Component

In this project, to make it easier to focus on how a component works, we'll start with the most basic component — *TComponent*. Open the Component Expert by selecting **File | New Component** from the menu and derive the first component. As shown in [Figure 1](#), enter the following parameters in Component Expert dialog box:

- **Class Name** — CustComp1
- **Ancestor Type** — TComponent
- **Palette Page** — Custom

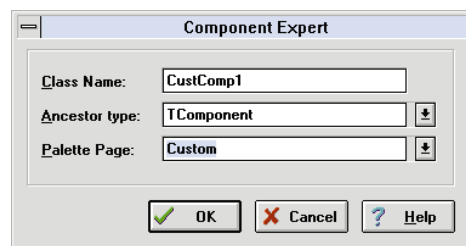


Figure 1: Using the Component Expert to create a new component of type *TComponent*.

Click on the **OK** button to accept these entries. The Component Expert automatically creates the Object Pascal code shown in [Figure 2](#) in a unit called Unit1. Every support file that a new *TComponent* type needs is added to the **uses** clause.

The component file — Unit1.PAS — even has the code necessary to register itself in the component library. Pretty slick, *Socrates*.

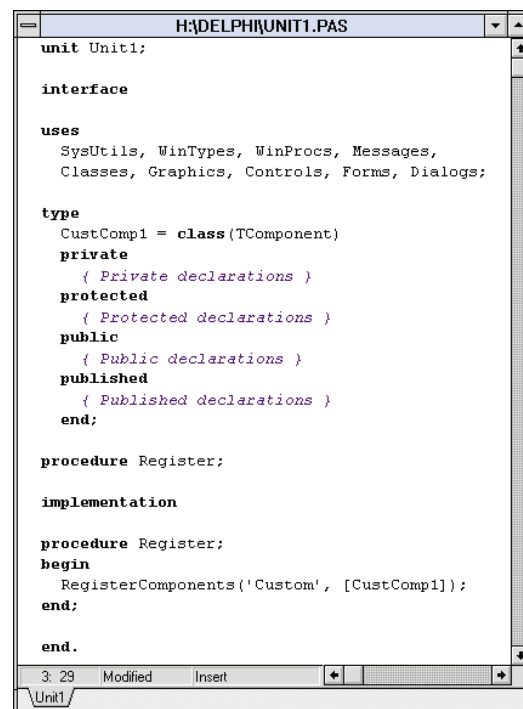


Figure 2: The PAS file created by the Component Expert in [Figure 1](#). This Object Pascal code will create a component named *CustComp1* and place it on the Custom page of the Component Palette.

Adding a Component to the Palette

If you add the new component, as is, to the Component Palette, it will have two default properties: *Name* and *Tag*. Let's go through the steps.

First, rename the PAS file to Cust1.PAS by selecting **File | Save File As** from the menu to display the Save As dialog box. The next step is to add the component to the Component Palette. This is accomplished at the Install Components dialog box accessed by selecting **Options | Install Components** from the menu. At the Install Components dialog box, click on the **Add** button to display the Add Module dialog box. Then use the Browse feature to locate the PAS file, as shown in **Figure 3**. *Cust1* will be added at the bottom of the **Installed Units** list box and its path will be added to the **Search path** edit box in the Install Components dialog box.

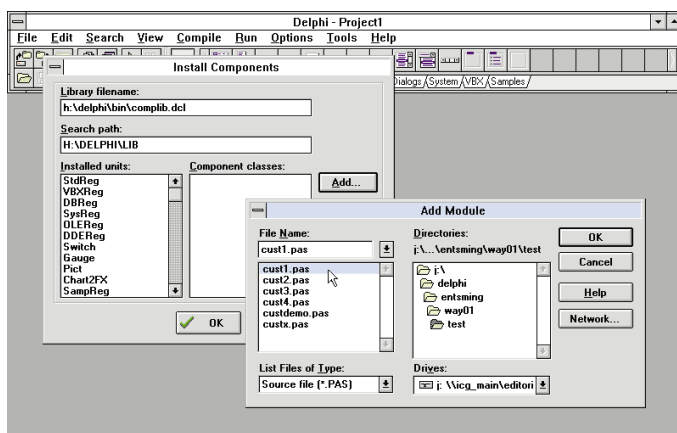


Figure 3: Installing a component on the Component Palette.

Finally, click **OK** to recompile the Component Palette. After a quick compile, a new page (Custom) will be added to the Component Palette, and the new component (CustComp1) will be on the page (see **Figure 4**). That's it! You've placed a custom — albeit primitive — component on the Component Palette.

You can try out the new component by creating a new form and placing the CustComp1 component on the form (as shown in **Figure 4**). As you can see in the Object Inspector, CustComp1 has just two (the minimum) properties: *Name* and *Tag*.

This component — real as it is — doesn't do much. You can think of it as a rather dim-witted component, capable of telling you its name and the value of one integer held in its *Tag* property. However, it does know how to interact gracefully with Windows, and that's worth the price of admission. From this humble starting point, you can easily give it the properties and behaviors it needs to be more useful.

Creating a Property

Let's modify this generic component by adding a property. To declare a property, specify the:

- property's name,
- property's type,
- and object fields for reading and/or writing the property.

Although Delphi doesn't restrict how you store the data for a property, it's a good idea to store the property data in an object field (variable) that you declare in the **private** declarations part of the class. We'll use an object field of type Integer named *FDemoProp*:

```
private
{ Private declarations }
FDemoProp: Integer;
```

The following Object Pascal statement creates a property (*DemoProp*) with a value that can be read from and written to the *FDemoProp* field:

```
property DemoProp: Integer read FDemoProp
write FDemoProp;
```

Note that while the property's name is *DemoProp*, the object field that will hold the property's value is named *FDemoProp*. The initial "F" conforms to the naming convention for property object fields.

Although you can declare fields and read and write methods for a property as **private** or **public**, you should declare them as **private**. Private declarations force descendant components to use the inherited property for access, instead of using the access method directly. Here's the revised class declaration:

```
type
Comp2 = class(TComponent)
private
{ Private declarations }
FDemoProp: Integer;
protected
{ Protected declarations }
public
{ Public declarations }
property DemoProp: Integer read FDemoProp
write FDemoProp;
published
{ Published declarations }
end;
```

The entire listing is shown in **Figure 5**. This is the simplest way to declare a property, known as *direct access*. That is, no method is used

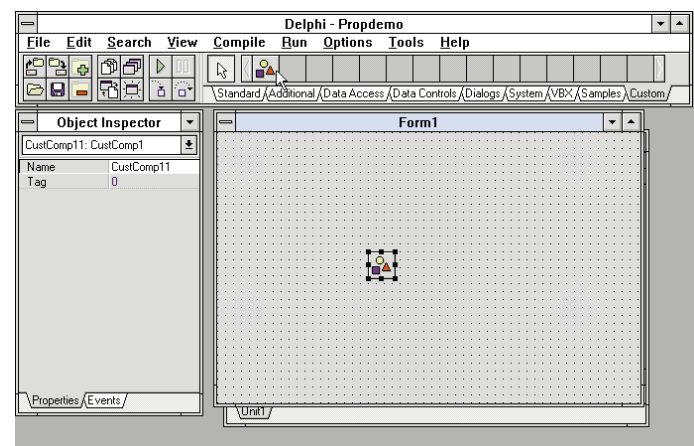


Figure 4: The properties of a new CustComp1 component are shown in the Object Inspector.

```

unit Cust2;

interface

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

type
  CustComp2 = class(TComponent)
  private
    { Private declarations }
    FDemoProp: Integer;
  protected
    { Protected declarations }
  public
    { Public declarations }
    property DemoProp: Integer read FDemoProp
                                     write FDemoProp;

  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Custom', [CustComp2]);
end;

end.

```

Figure 5: The Object Pascal code for the CustComp2 component (Cust2.PAS). The component will now have a property, *DemoProp*.

to read or write the property's value. All that's required are two statements — a type declaration for the object variable, *FDemoProp*, and a **property** statement to describe how the property is accessed.

Save the modified unit as Cust2.PAS and follow the steps described above to add the component to the Component Palette. Then create a new form and place the new CustComp2 component on the form (as shown in [Figure 6](#)).

But what's wrong? The new property (*DemoProp*) doesn't appear in the Object Inspector. This is because we made the property declaration in the **public** part of the component type declaration. Such a property is available only at run-time. That is, it can be accessed only through Object Pascal code.

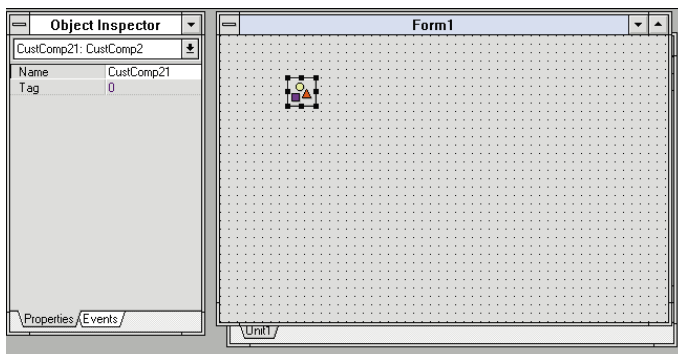


Figure 6 (Left): The CustComp2 component on a form at design-time. The new property isn't displayed in the Object Inspector because it was not published. **Figure 7 (Right):** The CustComp3 component at design-time. The published property — *DemoProp* — now appears in the Object Inspector.

The **public** declarations part of a class corresponds to its run-time interface. In other words, public declarations are available to all code in a project at run-time. If you only want a property or method to be available at run-time, declare it in the public part of the class. If you want a property to be available in design mode as well, you must publish it.

A **published** declaration is just like a public declaration, except other applications can get information about published properties, methods, and events through the published interface. The Delphi Object Inspector uses the published interface of objects installed in the Component Palette to determine the properties and events it displays.

Therefore, to make the property available during design-time (i.e. in the Object Inspector), it must be *published*. This is easily done by moving the property declaration into the published portion of the component type declaration. The class declaration now looks like this:

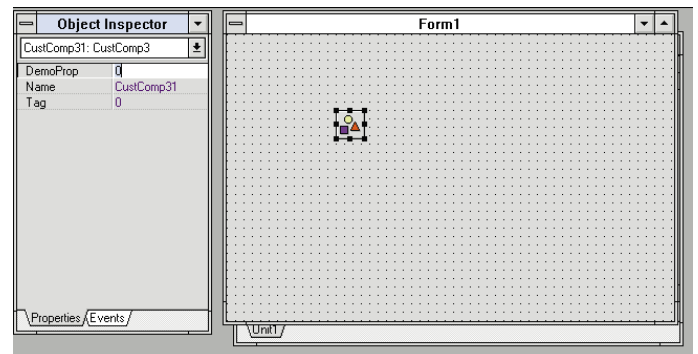
```

type
  Comp3 = class(TComponent)
  private
    { Private declarations }
    FDemoProp: Integer;
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
    property DemoProp: Integer read FDemoProp
                                     write FDemoProp;
  end;

```

Let's take a look. Save the modified unit as Cust3.PAS and add the component to the Component Palette. Then create a new form and place the new CustComp3 component on the form (as shown in [Figure 7](#)). There's the new property, *DemoProp*, available for modification in the Object Inspector.

As stated earlier, this is the simplest way to declare and access a property. It's also possible to read and write a property's value using functions and procedures known as *access methods*.



Access Methods

Instead of accessing a property's value directly, as just demonstrated, it's also possible to use a method. This is useful when you need to perform some other programming task while reading or writing a property value (for example, setting a valid range of values for the property).

First, let's write the method (a function) to read the property, named *GetDemoProp* by convention:

```
function Comp4.GetDemoProp;
begin
  { Return the value of the FDemoProp field }
  { In a more complex example, add other code here }
  Result := FDemoProp;
end;
```

Here's the method (a procedure) to write a new value to the property, named *SetDemoProp* by convention:

```
procedure Comp4.SetDemoProp(Val: Integer);
begin
  { Set a new value for the FDemoProp field }
  FDemoProp := Val;
end;
```

The complete unit listing (Cust4.PAS) is shown in [Figure 8](#). Notice that all it takes to have Delphi use the access methods is to name them in the **property** statement:

```
property DemoProp: Integer read GetDemoProp
  write SetDemoProp;
```

We've declared *DemoProp*, *GetDemoProp*, and *SetDemoProp* in the **private** section of the class declaration. Making these private protects them from unintentional modification and forces descendent components to use the inherited property, *DemoProp*, for access.

Use **Options | Install Components** to install this new component (CustComp4), and then place it on a form. Note that this new component behaves just like the component in [Figure 7](#) (CustComp3). The fact that access methods are now used to access the *DemoProp* property is transparent to the component user.

Invoking Component Methods

To complete this introduction to components, let's create a form that puts one of them to work. Select **File | New Form** to create a new blank form. Then add two buttons, a label, and the CustComp3 component to the form. Then, change the captions of the buttons to **Read DemoProp** and **Write DemoProp**. Remove the *Label* caption. [Figure 9](#) shows this form in design mode.

Now implement the *OnClick* events for the two buttons. The first button click event reads the *DemoProp* property:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  DemoProp: Integer;
  S: string;
begin
  DemoProp := CustComp31.DemoProp; {Read the property}
  Str(DemoProp,S);
  Label1.Caption := 'DemoProp Value Read: ' + S;
end;
```

```
unit Cust4;

interface

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

type
  CustComp4 = class(TComponent)
  private
    { Private declarations }
    FDemoProp: Integer;
    function GetDemoProp: Integer;
    procedure SetDemoProp(Val: Integer);
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
    property DemoProp: Integer read GetDemoProp
      write SetDemoProp;
  end;

  procedure Register;

implementation

function CustComp4.GetDemoProp;
begin
  { Return the value of the FDemoProp field }
  { In a more complex example, add other code here }
  Result := FDemoProp;
end;

procedure CustComp4.SetDemoProp(Val: Integer);
begin
  { Set a new value for the FDemoProp field }
  FDemoProp := Val;
end;

procedure Register;
begin
  RegisterComponents('Custom', [CustComp4]);
end.

end.
```

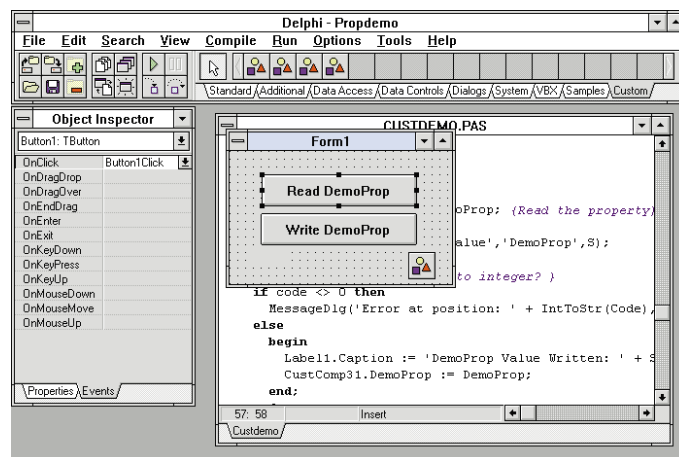


Figure 8 (Top): The Cust4.PAS file with access methods for reading and writing the *DemoProp* property. **Figure 9 (Bottom):** The demonstration form in design mode.

The *OnClick* event for *Button2* uses an *InputBox* to get a new *DemoProp* property value from a user at run-time.

```

procedure TForm1.Button2Click(Sender: TObject);
var
  DemoProp, Code: Integer;
  S: string;
begin
  DemoProp := CustComp31.DemoProp; { Read the property }
  Str(DemoProp,S);
  S := InputBox(Enter New Value',DemoProp',S);
  Val(S,DemoProp,Code);
  { Error during conversion to integer? }
  if code <> 0 then
    MessageDlg('Error at position: ' + IntToStr(Code),
      mtWarning, [mbOK],0)
  else
    begin
      Label1.Caption := 'DemoProp Value Written: ' + S;
      CustComp31.DemoProp := DemoProp;
    end;
  end;

```

Note the little bit of error checking to insure that a user has entered a number before attempting to change the property value. This isn't really necessary. Delphi actually handles this bit of error checking for you. But in other situations you might want to filter input, for example to ensure that a number falls within a range of numbers. The code for the form unit is shown in [Figure 10](#).

[Figure 11](#) shows the form at run-time. This form uses *CustComp3*, but it could have been constructed using *CustComp2* or *CustComp4* as well. The run-time behavior will be the same.

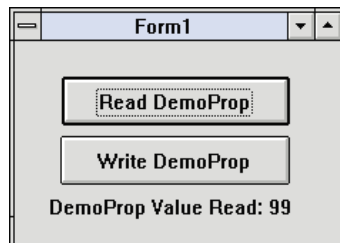


Figure 11: The demonstration program at run-time.

Conclusion

That's component creation in a nutshell. We've seen how easy it is to create new components based on the *TComponent* provided for us by Delphi. We've also seen how easy it is to add a property to a custom component, and to implement access methods to get to the property's value.

And, by the way, nice foreshadowing *Critias*. Δ

The demonstration files referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\APRIL\GE9504.

Gary Entsminger is the author of *The Tao of Objects, an Introduction to Object-oriented Programming*, 2nd ed. (M&T 1995) and *Secrets of the Visual Basic Masters, 2nd ed.* (Sams, 1994). He is currently working on an advanced Delphi book for Prentice Hall and is the technical editor for *Delphi Informant*.

```

unit Custdemo;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls, Cust3;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    CustComp31: CustComp3;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{ $R *.DFM }

procedure TForm1.Button1Click(Sender: TObject);

var
  DemoProp: Integer;
  S: string;
begin
  DemoProp := CustComp31.DemoProp; { Read the property }
  Str(DemoProp,S);
  Label1.Caption := 'DemoProp Value Read: ' + S;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  DemoProp,Code: Integer;
  S: string;
begin
  DemoProp := CustComp31.DemoProp; { Read the property }
  Str(DemoProp,S);
  S := InputBox('Enter New Value', 'DemoProp',S);
  Val(S,DemoProp,Code);
  { Error during conversion to integer? }
  if code <> 0 then
    MessageDlg('Error at position: ' + IntToStr(Code),
      mtWarning, [mbOK],0)
  else
    begin
      Label1.Caption := 'DemoProp Value Written: ' + S;
      CustComp31.DemoProp := DemoProp;
    end;
  end;

end.

```

Figure 10: The complete program code for CustDemo.PAS.





DATA SOURCE 1

DELPHI / OBJECT PASCAL | BEGINNER / INTERMEDIATE



By *Dan Ehrmann*

Data-Aware Delphi

Modifying a Query Component via a Tabbed Interface at Runtime

One of the delights of programming in Borland's Delphi is the small amount of code you have to write to implement powerful data-based applications. For many common tasks, you just place components from the Visual Component Library (VCL) on a form, modify a few properties, link the data-bound components to an underlying data source, and compile. If any code is necessary, it's usually short and relatively simple.

For example, consider the program shown in [Figure 1](#). This application displays the Customer table that ships with Delphi (in the \Delphi\Demos\Data directory by default). Using the provided DBGrid and DBNavigator components, you can freely scroll through the table horizontally and vertically.

In addition to these standard navigational controls, the sample application provides a tabbed interface. If you click one of the lettered tabs along the bottom of the window, you can filter the view of the Customer table to only those customers beginning with that letter. If you click on the **Filter Field** combo box, you can select one of four fields within the table to use as the filter criterion field.

You can create this application using Delphi's Database Form Expert. It will place components on the form for you and link them automatically. For this article, however, you'll place and configure the components yourself so you don't have to change the design and properties the Expert creates.

Along the way, you'll learn about a number of sophisticated components included in Delphi's component library, how to modify their properties, and how to attach the same code to more than one event.

Placing the Database Components

As with most Delphi applications that reference data in tables, you begin by placing data components on a form to manage this data. Delphi reads data from a table using either a Query or a

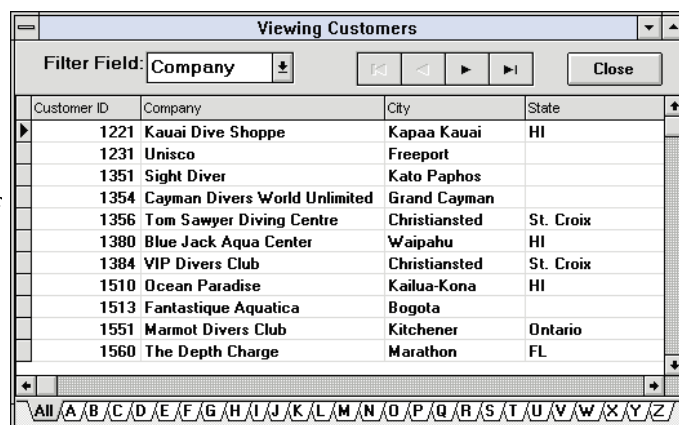


Figure 1: This simple application provides a view of a Customer table. The user can filter on the Customer ID, Company, City, and State fields using the first letter of the field.

Table component. The Table component provides a static channel to data and allows you to open a table, but limits your ability to filter or control what is accessed from that table. The Query component allows you to open a table using a SQL query, either as a static or a live view of the data.

For this example, you should place a Query component anywhere on the form; it will be invisible at runtime. Figure 2 lists the important properties you should set for this component. (That is, confirm the default settings for the *Active*, *RequestLive*, and *UniDirectional* properties, assign the *DatabaseName* property to the DBDATA alias, and enter the proper SQL statement for the *SQL* property.)

Property	Value
Active	False (the default)
DatabaseName	DBDATA, a predefined alias that points to the \Delphi\Demos\Data directory.
RequestLive	False (the default)
SQL	The SQL statement shown in Figure 3.
UniDirectional	False (the default)

Figure 2: Set these properties for the Query component.

The SQL SELECT statement shown in Figure 3 generates a result set containing all records in the table. However, one of Delphi’s powerful features is its ability to modify this expression under program control to generate different results. As you’ll see below, whenever the user selects a different tab, a dynamic WHERE clause is added to the query.

You should also place a DataSource component on the form, and link it to the Query component using the *DataSet* property. Delphi’s DataSource component serves as a “traffic cop”, managing the requests and activity between data-bound com-



Figure 3: Using the String list editor to define the SQL property of the Query component.

ponents such as DBGrid and DBNavigator, and data components such as Query.

Finally, select a DBGrid component from the Data Controls page of the Component Palette and place it on the Panel as shown in Figure 1. Using the *DataSource* property, attach this component to the DataSource component you just placed on the form. You can also set any other properties you need. For example, you might want to disable the editing capability by setting the *ReadOnly* property to *False*. You can also disable the user’s ability to resize columns by expanding the *Options* property (i.e. double-click *Options*), and setting the *dgColumnResize* property to *False*.

At this point, you can return to the Query component and change its *Active* property to *True*. This has the effect of running the query and creating a view. The DataSource then picks up this data set and passes it to the DBGrid which in turn displays the result; all records from the Customer table are displayed.

Components on the Top Panel

Begin by placing a Panel component on the form. (It’s located at the end of the Standard page.) Panel components are used simply as containers for grouping user-interface elements. For example, you can place components in a Panel, then move the entire Panel around the form without disturbing the relationship of the components within the Panel.

On the Panel’s far right side, place a standard Button component and change its *Caption* property to indicate the function it will perform — in this instance, *Close*. When you double-click the button, Delphi displays the *OnClick* procedure, the one you are most likely to modify. The Object Pascal code to place in this procedure is as simple as it gets. The single method call:

```
Close;
```

tells the program to close the window.

Next, place a DBNavigator component from the Data Controls page of the Component Palette. You should also connect this component to the DataSource so the navigator buttons can be used to move through the query’s result set.

Then, expand the *VisibleButtons* property by double-clicking on it. Set all these properties to *False* except for *nbFirst*, *nbPrior*, *nbNext*, and *nbLast*. No code is required! By attaching the DBNavigator component to a DataSource, you automatically enable each navigational button to perform its appropriate function.

Place a ComboBox component (also on the Standard page of the Component Palette) to the left of the DBNavigator buttons. Notice how the entry field and drop-down arrow are separated by a narrow gray area. This is the Windows convention indicating that you can select from the list or type a value into the entry field. Now, change the *Style* property to *csDropDownList*. Notice

how the entry field is now attached to the drop-down arrow. This is also a Windows convention that indicates you must select an item from the list.

Select the *Items* property and click on the ellipsis button (or simply double-click the *Items* property) to display the String list editor dialog box. Then type in the four entries as shown in Figure 4. These are the names of the four fields from the Customer table that will appear when the user clicks the combo box to filter records.

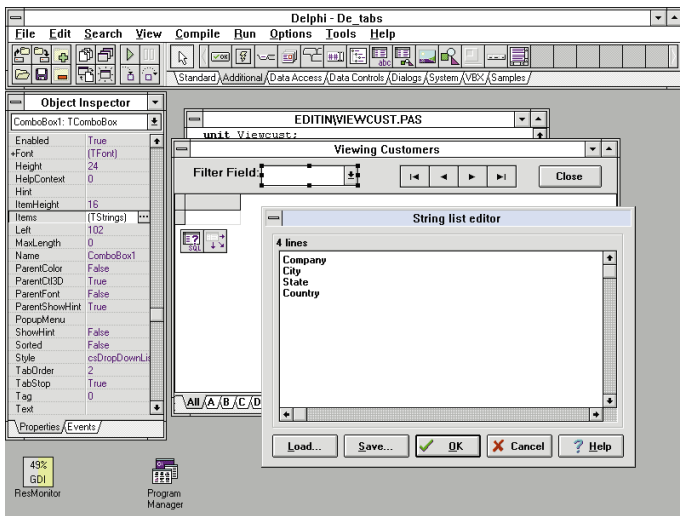


Figure 4: Using the String list editor to specify the entries for the Filter Field’s drop-down list.

You can also add a Label component to the left of the combo box. Modify its *Caption* property to indicate the field’s purpose, `Filter Field` in this example. We’ll associate Object Pascal code with the ComboBox below.

The Tab Component

It seems that every Windows application contains a Tab component as part of its interface. Some of them display pages within a dialog box, such as the dialog box that is displayed when you select **Options | Environment** from Delphi’s menu. Others are used as selectors, to modify the information displayed in one or more components of the window.

Delphi provides both types of components. The TabSet and TabbedNotebook components are located on the Additional page of the Component Palette. For this example, select the TabSet component and place it along the bottom edge of the form. To properly align TabSet, set its *Align* property to *alBottom*.

Select the *Tabs* property and click on the ellipsis to display the String list editor. The first entry should be `All`. Then enter each letter of the alphabet on successive lines, until you have 27 entries. Click **OK** and the TabSet component is defined.

Adding Event-Handling Code

For ComboBox components, the *OnChange* event is triggered when the user makes an entry in the combo box entry field or

selects an item from the drop-down list. Therefore, this is the appropriate event for redefining the SQL expression.

However, the same event isn’t appropriate for TabSet components. *OnChange* is triggered before the tab changes. In effect, it asks permission to make the change and allows you to deny the event. A more appropriate event for TabSet components is *OnClick*. This is triggered when the user clicks on a tab.

Since both of these events have the same procedure template, it’s a good idea to centralize the query update in a single procedure that is called for both events. To add this template, select the TabSet component and display the Events page of the Object Inspector. In the *OnClick* event, type `DefineAndRunQuery`) and press **Enter**. Delphi will add a procedure template for this event to the `.PAS` file.

Now select the ComboBox and display its Events page. In the *OnChange* event, click on the drop-down arrow and select `DefineAndRunQuery` from the drop-down list. Now the “change” events for each component are connected to the same procedure.

Reviewing the Source Code

The complete `ViewCust.PAS` file is shown in Listing One on page 24. Here’s how it works.

QueryDef is a defined string constant, containing the basic SQL query without a WHERE clause. You can copy and paste the shell of this query from the *SQL* property of the Query component. When the user changes the filter field or selects a new tab, a new WHERE clause is calculated and concatenated with the constant. This string is then assigned to the *SQL* property of the Query component and the query is rerun, displaying a new set of records.

Code must also be added to the *OnCreate* method for the form, to ensure the ComboBox and TabSet are correctly initialized, and to run the query the first time.

QueryWhereField is a global variable that tracks the field used in the WHERE clause. It’s first initialized to the value `Company`. When the user selects a new entry, the value of this variable is changed at the beginning of the *DefineAndRunQuery* procedure.

QueryWhereClause is a complete SQL WHERE clause built up in the correct format. If the current TabSet is zero — equivalent to selecting “All” — this string is set to null, so all records are displayed. However, if another tab is highlighted, this string is assigned to the highlighted letter using the expression:

```
TabSet1.Tabs[TabSet1.TabIndex];
```

TabIndex returns the number of the highlighted tab, starting at zero. *Tabs* is a *String List* property, enabling you to index to the current tab using the same number. For example, when the user

selects the City field and clicks on the “C” tab, the following WHERE condition is constructed:

```
WHERE Customer.City LIKE "C%"
```

This SQL expression establishes a selection criterion on the City field to find all cities where the first letter is “C”.

To redefine the query, it must first be closed and the *SQL* property cleared. If this isn't done, subsequent operations on the *SQL* property will be added to existing code, resulting in a meaningless expression. The *Add* method of the *SQL* property allows you to create a new SQL expression. In the example, this is a con-

catenation of the basic query — previously defined as a constant — and the updated WHERE clause.

(Delphi provides another feature called *parameterized queries* for creating queries that are variable. However, parameters can only be used for selection criteria, not for changing the field being queried. Since this example allows the user to filter on different fields, parameterized queries cannot be used.)

Once the *SQL* property has been redefined, the query is opened again. It runs and allows the DataSource component to read a new result set and pass the set to the DBGrid.

Begin Listing One: Viewcust.PAS

```
unit Viewcust;

interface

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

type
  TForm1 = class(TForm)
    DBGrid1: TDBGrid;
    Panel1: TPanel;
    CustDS: TDataSource;
    CustQuery: TQuery;
    DBNavigator: TDBNavigator;
    TabSet1: TTabSet;
    CustQueryCustNo: TFloatField;
    CustQueryCompany: TStringField;
    CustQueryCity: TStringField;
    CustQueryState: TStringField;
    CustQueryZip: TStringField;
    CustQueryCountry: TStringField;
    CustQueryPhone: TStringField;
    Button1: TButton;
    ComboBox1: TComboBox;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure DefineAndRunQuery(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;

const
  { Base query, without the WHERE clause }
  QueryDef = 'Select ' +
    'customer."CustNo", ' +
    'customer."Company", ' +
    'customer."City", ' +
    'customer."State", ' +
    'customer."Zip", ' +
    'customer."Country", ' +
    'customer."Phone" ' +
    'From "customer.db" ' +
    'As customer ' ;

var
  Form1: TForm1;

  QueryWhereClause,      {Where clause to be built}
  QueryWhereField: String; {Field for selection criteria}
```

implementation

```
{ $R *.DFM }

procedure TForm1.FormCreate(Sender: TObject);
begin
  {Initialize field and tabs}
  ComboBox1.ItemIndex := 0;
  QueryWhereField := 'Company';
  TabSet1.TabIndex := 0;
  QueryWhereClause := '';

  {Run query the first time with these settings}
  DefineAndRunQuery(Sender);
end;

procedure TForm1.DefineAndRunQuery(Sender: TObject);
begin
  {Set field name based on value in combo box}
  case ComboBox1.ItemIndex of
    0: QueryWhereField := 'Company';
    1: QueryWhereField := 'City';
    2: QueryWhereField := 'State';
    3: QueryWhereField := 'Country';
  end;

  {If TabIndex = 0 then "All" option is active}
  if TabSet1.TabIndex = 0 then
    QueryWhereClause := ''
  else begin

    {Build WHERE clause based on field and tab index}
    QueryWhereClause := 'Where customer.'" +
      QueryWhereField +
      '" LIKE "' +
      TabSet1.Tabs[TabSet1.TabIndex] +
      '%"';
  end;

  {Redline and run the query}
  CustQuery.Close;
  CustQuery.SQL.Clear;
  CustQuery.SQL.Add( QueryDef + QueryWhereClause );
  CustQuery.Open;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Close;
end;

end.
```

End Listing One

Compiling and Running the Form

After you have built the .PAS file to match the one shown in the listing, compile it. If there are no errors, the form will appear on-screen and you can navigate the Customer table (see Figure 5). Try it out by clicking on one of the tabs to filter on the Company field. Then click on the **Filter Field** combo box to select another field to use as a filter. Finally, click on the **Close** button to exit.

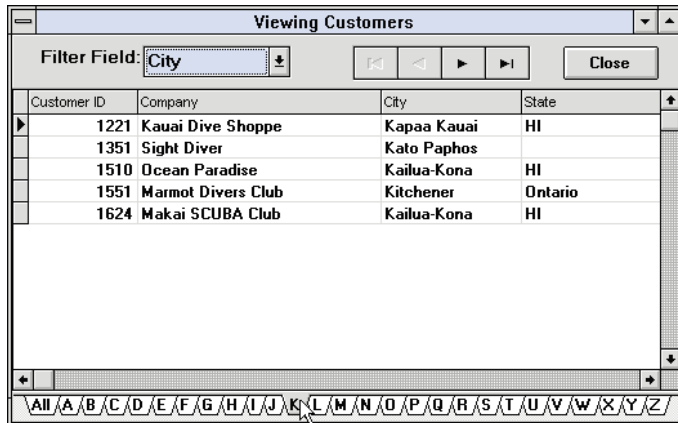


Figure 5: The sample application in action. Here, the filter feature is used to show only those records with cities beginning with the letter "K".

Conclusion

This simple example demonstrates how the various Delphi components can work together to create powerful data-based applications. Delphi provides a rich collection of components to place on your forms, and it's a simple matter to change the properties of these components. Data access is managed through components that are invisible at runtime, but have properties you can modify at design time and at runtime. Delphi also allows more than one event to share the same event handler, making it easy for you to centralize behavior resulting from different user actions.

What is amazing about this small example is how little code had to be written. Two short methods encompass most of the functionality in this application. This clearly demonstrates the elegance of the Delphi environment.▲

The demonstration project referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\APRIL\DE9504.

Dan Ehrmann is the founder and President of Kallista, Inc. a desktop database consulting firm based in Chicago. Dan is well known for his active involvement in the Paradox world, writing books and articles, supporting Paradox on CompuServe, and speaking at conferences and users group meetings. Kallista is already working on Delphi-based applications for a number of clients.





By *Kenn Nesbitt*

A PC Sound Garden

Writing a Custom CD Player in Delphi

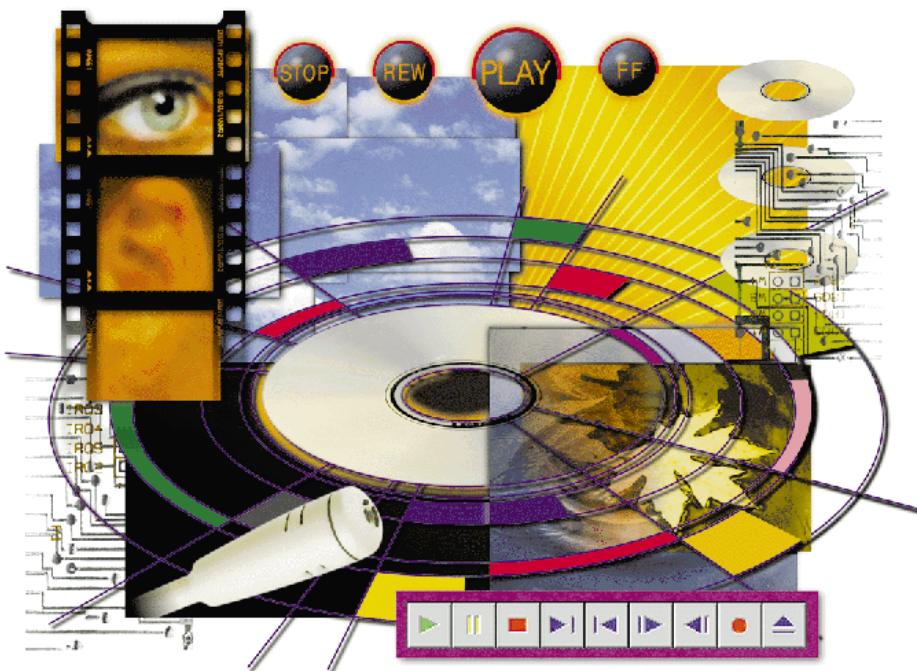
You have a CD-ROM drive in your computer, but it goes to waste 90 percent of the time. Why not use it to play music CDs when it's not otherwise occupied?

Of course, there are plenty of programs available on CompuServe or the Internet to coax audio out of your CD-ROM drive. But why use one of those, when you can have fun writing a CD player application in Delphi?

The TMediaPlayer Component

Delphi has a component named *TMediaPlayer* that allows you to control multimedia devices such as VCRs, videodisc players, scanners, DAT recorders, CD-ROM drives, and more. In addition, the *TMediaPlayer* component can play WAV audio files, MIDI files, and even AVI video.

Delphi makes all of these unbelievably simple to implement. The truth is, you can write controllers for most of these devices with little or no code. However, the simplest device to implement is the CD audio player. This is because the CD audio player does not require a filename — it simply plays whatever CD is in the player. The CD audio player is probably the most useful device as well. This is because while you may not have lots of MIDI or AVI files laying around, chances are, you have some CDs you'd like to listen to while you work.



The Minimal CD Player Project

To create a working CD player, here's all you have to do:

- Create a new project.
- Select the *TMediaPlayer* component from the System page of the Component Palette and drop it on the form.
- In the Object Inspector, set the *DeviceType* property to *dtCDAudio*.
- Change the *VisibleButtons* property to remove the *Step*, *Record*, and *Back* buttons (i.e. set *btStep*, *btRecord*, and *btBack* to *False*).

That's it! Your program should look similar to the one in [Figure 1](#). Now pop a CD in the player, run your new project, and press the **Play** button. If you don't hear anything, try plugging your speakers into the audio-out port on the front of your CD-ROM drive (instead of your sound card). If you hear it



Figure 1: A minimal CD player at runtime.

now, you may need an extra cable to connect the audio-out connector on the back of your CD-ROM drive to the audio-in connector on your sound card.

Now save your project. In this example, the main unit is saved as CD.PAS and the project is saved as CDPLAYER.DPR. [The code for CD.PAS is in [Listing Two](#) on page 28; the code for CDPLAYER.DPR is in [Listing Three](#) on page 30. The CD player program also features an About box. The ABOUT.PAS file is in [Listing Four](#) on page 30.] Also, since *MediaPlayer1* is a long name to type repeatedly in code, you'll probably want to change the name of the *TMediaPlayer* component to something brief, like "CD".

Adding features to your CD player project will involve adding some Object Pascal code. You'll also need to understand the properties of the *TMediaPlayer* component and some of the multimedia macros available in the Windows API. You will find documentation for Windows multimedia macros in the file MMSYSTEM.HLP (by default this file is in the \DELPHI\BIN directory). Don't worry — it's not very difficult and this article will demonstrate everything you need to know.

Tracking the CD

Probably the first feature you want to add is a display to show what track is playing and how many minutes and seconds have elapsed. All this information is available in the *Position* property, but it must be decoded. The *Position* property is a Longint that contains one piece of information in each of its four bytes. For *dtCDAudio* devices, it stores the information in a format called *TMSF* (for tracks, minutes, seconds, and frames). So you don't have to break out each byte manually, the *MMSYSTEM* unit provides a number of macros that do the work for you. To get the track, minutes, and seconds, these are: *mci_TMSF_Track*, *mci_TMSF_Minute*, and *mci_TMSF_Second*, respectively.

The easiest way to track the current position of the CD is by adding a *TTimer* component to your form, setting its *Interval* property to one second or less, and adding some Object Pascal code, as shown in [Listing Two](#) on page 28.

The first thing you'll notice is the *InitCD* method loading the length of each track into a global array. This happens because there's a lot of overhead involved in reading properties from a *TMediaPlayer* component. When you read a property such as *TrackLength* or *Pos*, Delphi calls a method of the component that calls a Windows API call, that in turn calls the device driver for the CD-ROM drive. This is not something you want to do more often than absolutely necessary. Therefore, the sample project reads the length of each track only once when the CD is loaded.

Note that the *Timer* event has code that looks at the CD player once per second. If the player is stopped, it initializes the CD if necessary. If the player is open, it resets a global variable to indicate the CD's tracks need to be loaded. If the player is playing, it

reads and decodes the *Position* property, and updates the *Caption* properties of two labels to display the current track and time.

Also, the *Timer* event code only reads the *Position* property once, storing it in the variable *Pos*, instead of once for the track, once for minutes, and once for seconds. Just as with the *TrackLength* property, we want to call the *Position* property as few times as possible. Rather than repeat all this overhead three times, we can simply save the property on the first read.

In addition, you'll see some code in the *OnPostClick* event of the *CD* component. When a user clicks a button to complete a process, *OnPostClick* is triggered. For example, if the user clicks the next button, any code in *OnPostClick* runs when the CD-ROM drive positions itself to the next track. If the user clicks the play button, *OnPostClick* runs right after the CD begins playing and so on. The code in this procedure turns *Timer1* on and off, depending on which button the user clicks. There is no need to have the timer waste processor cycles when the CD is not playing.

Adding More Features

Next, you will probably need a menu to implement your new features, so drop a *TMenu* component on your form and add menu items for whatever features you want your CD player to have. The sample project implements continuous play mode, that causes the CD to start again when it has finished, and repeat mode, which repeats the current track. [Figure 2](#) shows the CD Player project at runtime.

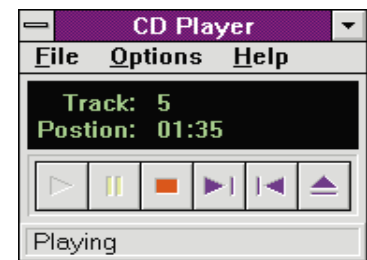


Figure 2: The example CD player at runtime. The program displays the current track and time, the state of the CD-ROM drive, and offers special features on the Options menu.

You may want to add random mode to play tracks in random sequence, a database to store the names of the CD, artist, tracks, etc., or set the *ApplicationTitle* property or *CDForm's Caption* property to indicate the CD or track name.

For features that are turned on or off, you can toggle the *Checked* property of a menu item, and then add a little more code to the *Timer* event.

To implement continuous play mode, all you need to do is check the status of *mnuOptionsContinuous*. Check it in the *Timer* event when the *MediaPlayer's Mode* property is *mpStopped*.

Adding repeat mode involves storing the end of the current track (as an absolute position, in seconds, from the beginning of the CD), and then checking in the *Timer* event to see if you have passed this point on the CD. The *CDPos* function shows how to calculate the end of the current track. Notice that the *TrackLength* property (and the *TrackLen* array in this project) returns the length in MSF format, *not* in TMSF format. To

decode the track length, you will need to use the MCI macros `mci_MSF_minute` and `mci_MSF_second` (as opposed to the TMSF macros).

Happy Listening

Of course, this is just a single use for the *TMediaPlayer* component. This component can also be used for playing audio and video files, controlling other peripherals, and so on. And the concepts you learn by implementing a CD player project will also help you write other types of multimedia applications. Try adding some custom features, or take what you've learned here and start creating your own multimedia programs. ▲

Begin Listing Two — CD.PAS

```
unit Cd;

interface

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

type
  TCDForm = class(TForm)
    CD: TMediaPlayer;
    MainMenu1: TMainMenu;
    mnuFile: TMenuItem;
    mnuFileExit: TMenuItem;
    mnuHelp: TMenuItem;
    mnuHelpAbout: TMenuItem;
    Options1: TMenuItem;
    mnuOptionsContinuous: TMenuItem;
    mnuOptionsRepeat: TMenuItem;
    Timer1: TTimer;
    Panel1: TPanel;
    lblTrackLabel: TLabel;
    lblPositionLabel: TLabel;
    lblTime: TLabel;
    lblTrack: TLabel;
    Bevel1: TBevel;
    pnlStatus: TPanel;
    procedure mnuFileExitClick(Sender: TObject);
    procedure mnuHelpAboutClick(Sender: TObject);
    procedure mnuOptionsContinuousClick(Sender: TObject);
    procedure mnuOptionsRepeatClick(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure CDPostClick(Sender: TObject;
      Button: TMPBtnType);
    procedure CDNotify(Sender: TObject);
    function CDPos(Sender: TObject; Track,
      Min, Sec: byte): Longint;
    procedure InitCD(Sender: TObject);
    procedure ResetCD(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
    CurrentTrack: byte;
    EndCurrentTrack: Longint;
    CDTracksLoaded: boolean;
    CDPlaying: boolean;
    CDPaused: boolean;
    TrackLen: array[1..100] of longint;
  public
    { Public declarations }
  end;

var
  TCDForm: TCDForm;

implementation
```

The demonstration CD player project referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\APRIL\KN9504.

Kenn Nesbitt is an independent Windows and database consultant, formerly with Microsoft Consulting Services. He is a Contributing Writer to *Data Based Advisor*, the Visual Basic columnist for *Access/VB Advisor* magazine, a regular contributor to the German magazine *Office and Database*, and co-author of *Power Shortcuts: Paradox for Windows*. You can e-mail Kenn at kenn@netcom.com, or CompuServe 76100,57.

```
uses About;

const
  ModeStr: array[TMPModes] of string[10] =
    ('Not ready', 'Stopped', 'Playing', 'Recording',
     'Seeking', 'Paused', 'Open');

{ $R *.DFM }

function TCDForm.CDPos(Sender: TObject;
  Track, Min, Sec: byte): Longint;
var
  i: integer;
begin
  { Total up the number of minutes and
    seconds from the beginning of the CD
    specified position }
  Result := 0;

  for i := 1 to Track - 1 do
  begin
    Inc(Result, mci_MSF_Second(TrackLen[i]));
    Inc(Result, mci_MSF_Minute(TrackLen[i]) * 60);
  end;

  Inc(Result, Sec);
  Inc(Result, Min * 60);
end;

procedure TCDForm.InitCD(Sender: TObject);
var
  i: integer;
begin
  { Store track information in an array
    so that we don't have to retrieve it
    from the device driver more than once }
  if not CDTracksLoaded then
  begin
    pnlStatus.Caption := 'Loading tracks';
    pnlStatus.Update;
    for i := 1 to CD.Tracks do
    begin;
      TrackLen[i] := CD.TrackLength[i];
    end;
    CDTracksLoaded := True;
  end;
end;

procedure TCDForm.ResetCD(Sender: TObject);
begin
  { Reset everything to 0 }
  CurrentTrack := 1;
  lblTrack.Caption := '0';
  lblTime.Caption := '00:00';

  { Change display color to red }
  lblTrack.Font.Color := clRed;
  lblTime.Font.Color := clRed;
```



```

lblTrackLabel.Font.Color := clRed;
lblPositionLabel.Font.Color := clRed;

{ Stop the CD player }
CD.Stop;

{ Set the start position to the
beginning of the first track }
CD.StartPos := mci_Make_TMSF(1,0,0,0);

CDPlaying := False;
CDPaused := False;
end;

procedure TCDForm.mnuFileExitClick(Sender: TObject);
begin
  Close;
end;

procedure TCDForm.mnuHelpAboutClick(Sender: TObject);
begin
  AboutBox.ShowModal
end;

procedure TCDForm.mnuOptionsContinuousClick(Sender:
TObject);
begin
  mnuOptionsContinuous.Checked :=
  not mnuOptionsContinuous.Checked;
end;

procedure TCDForm.mnuOptionsRepeatClick(Sender: TObject);
var
  Track, EndMin, EndSec, EndFrame: byte;
begin
  mnuOptionsRepeat.Checked :=
  not mnuOptionsRepeat.Checked;
  if mnuOptionsRepeat.Checked then
    { Find 2 seconds before end of track }
    With CD do
      begin
        Track := mci_TMSF_Track(Position);
        EndMin := mci_MSX_Minute(TrackLen[Track]);
        EndSec := mci_MSX_Second(TrackLen[Track]) - 2;
        EndCurrentTrack := CDPos(CD,Track,EndMin,EndSec);
      end;
end;

procedure TCDForm.Timer1Timer(Sender: TObject);
var
  Track, Minutes, Seconds: byte;
  strMinutes, strSeconds: string;
  Pos: Longint;
begin
  if CDPaused then
    pnlStatus.Caption := ' Paused'
  else
    pnlStatus.Caption := ' ' + ModeStr[CD.Mode];

  case CD.Mode of
    mpStopped:
      begin
        { Initialize the CD as soon as it is loaded }
        if not CDTracksLoaded then InitCD(CD);

        { In case CD ends when continuous play is
enabled }
        if CDPlaying and
(not CDPaused) and
mnuOptionsContinuous.Checked then
          begin
            CD.StartPos := mci_Make_TMSF(1,0,0,0);
            CD.Play;
            CD.EnabledButtons :=
[btPause, btStop, btNext, btPrev, btEject];

```

```

      end;
    end;

mpOpen:
  { In case user manually ejects CD }
  begin
    CDTracksLoaded := False;
    ResetCD(CD);
  end;

  { mpPaused, mpOpen, mpSeeking: }
  { Do nothing }

mpPlaying:
  begin
    { Get the current track, minutes and seconds }
    { Add the MMSystem unit to the Uses
clause to make these functions available }
    Pos := CD.Position;
    Track := mci_TMSF_Track(Pos);
    Minutes := mci_TMSF_Minute(Pos);
    Seconds := mci_TMSF_Second(Pos);

    { Code for Repeat }
    if mnuOptionsRepeat.Checked then
      begin
        if CDPos(CD,Track,Minutes,Seconds) >
EndCurrentTrack then
          begin
            Timer1.Enabled := False;
            CD.Notify := True;
            CD.Previous;
          end;
        end
      else
        CurrentTrack := Track;

    { Build a minutes and seconds display string }
    strMinutes := IntToStr(Minutes);
    strSeconds := IntToStr(Seconds);
    if length(strMinutes) < 2 then
      strMinutes := '0' + strMinutes;
    if length(strSeconds) < 2 then
      strSeconds := '0' + strSeconds;
    { Display the current track, minutes,
and seconds }
    lblTrack.Caption := IntToStr(Track);
    lblTime.Caption := strMinutes+'-'+strSeconds;
  end;
end;

procedure TCDForm.CDPostClick(Sender: TObject;
Button: TMPBtnType);
begin
  case Button of
    btPlay:
      begin
        { Change display color to lime }
        lblTrack.Font.Color := clLime;
        lblTime.Font.Color := clLime;
        lblTrackLabel.Font.Color := clLime;
        lblPositionLabel.Font.Color := clLime;

        CDPlaying := True;
        CDPaused := False;
        Update;
      end;
    btPause:
      begin
        { Change display color to yellow }
        lblTrack.Font.Color := clYellow;
        lblTime.Font.Color := clYellow;
        lblTrackLabel.Font.Color := clYellow;
        lblPositionLabel.Font.Color := clYellow;

```

```

        CDPlaying := False;
        CDPaused := True;
    end;
    btStop:
        ResetCD(CD);
    btEject:
        begin
            CDTracksLoaded := False;
            ResetCD(CD);
        end;
    end;
    Timer1Timer(Timer1);
end;

procedure TCDForm.CDNotify(Sender: TObject);
begin
    Timer1Timer(Timer1);
end;

procedure TCDForm.FormCreate(Sender: TObject);
begin
    CDPlaying := False;
    CDPaused := False;
    CDTracksLoaded := False;
    Application.Title := 'CD Player';
end;

```

end.
End Listing Two

Begin Listing Three — CDPLAYER.DPR

```

program Cdplayer;

uses
    Forms,
    Cd in 'CD.PAS' { CDForm },
    About in 'ABOUT.PAS' { AboutBox };
{ $R *.RES }
begin
    Application.CreateForm(TCDForm, CDForm);
    Application.CreateForm(TAboutBox, AboutBox);
    Application.Run;
end.

```

end.
End Listing Three

Begin Listing Four — ABOUT.PAS

```

unit About;

interface

uses WinTypes, WinProcs, Classes, Graphics, Forms,
    Controls, StdCtrls, Buttons, ExtCtrls;

type
    TAboutBox = class(TForm)
        Panel1: TPanel;
        OKButton: TBitBtn;
        ProgramIcon: TImage;
        ProductName: TLabel;
        Version: TLabel;
        Copyright: TLabel;
        Comments: TLabel;
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    AboutBox: TAboutBox;
implementation

{ $R *.DFM }
end.

```

end.
End Listing Four





DELPHI C/S

DELPHI / ODBC / BDE / GUPTA SQLBASE | ALL LEVELS

By *Sundar Rajan*

From ODBC to the BDE

Setting Up a Delphi Client/Server Development Environment

Delphi is a highly versatile rapid application development tool that caters to a potentially huge audience. With Delphi, you can revert to low-level assembly language and write device-drivers and the like, or develop sophisticated client/server database applications without writing a single line of code. Delphi is also a next-generation application development environment that combines the well-known Pascal language with the familiar Windows user interface and access to the entire Windows programming environment. This set of abilities is unmatched by any other Windows development tool.

Client/server is one of today's buzzwords. The client/server model is quickly becoming the *de facto* standard for developing new database applications in most corporate MIS departments. From e-mail systems such as Lotus Notes to component models like OLE, client/server is making its mark on all facets of the software world. If you depend upon developing any kind of software for a living, you can scarcely afford to ignore it.

Why is client/server so important? Before we discuss this in detail, let's briefly review client/server architecture.

Client/Server Architecture

Client/server architecture splits the workload between client computers that request services — such as printing, information retrieval, or updating of a customer account — and server computers that process the request. Client/server does not rely exclusively on the expensive resources of a centralized mainframe or try to do everything on desktop PCs the way some LAN-based multi-user programs do. Rather, it takes advantage of the computing power of desktop PCs by letting them share some of the processing burden (see [Figure 1](#)).

Layer	Contents
Front-end	Desktop GUI application
Back-end	Relational SQL database

Figure 1: Client/server application architecture.

Most modern client/server configurations use a two-tiered model, consisting of a client that invokes services from a server. While the client application interacts with and presents information to end-users using a graphical user-interface (GUI), the database server perform high-speed data manipulation, protect data integrity, and enforce business rules.

(Client/server is a logical concept. It doesn't necessarily require the client to have a GUI front-end, nor does the back-end have to be a SQL database server. Client/server technology is merely a *paradigm* or *model*, for the interaction between concurrently running software processes that may or may not run on separate machines. However, the popular model for client/server architecture is the one described above.)

At first glance the term client/server seems to imply there are only two components. Although not mandatory, as we shall see later, there is usually a third component — the network (and even a fourth if you count the human component). The network component provides the plumbing, the pipeline through which data moves.

The Progression to the Client/Server Model

The interest in the client/server model didn't arise overnight. It was a gradual progression from the mainframe era, primarily motivated by an effort to make better use of desktop processing power and easier access to information.

The Gartner Group developed a chart to describe the gradations of client/server computing (see Figure 2), especially as they relate to IBM's terminology for different degrees of client/server. At one end of the spectrum are mainframe applications, where all processing is done on the mainframe and the terminal is simply a vehicle for data entry. As you move towards client/server, display and significant data manipulation are accomplished on local PCs. Client/server models harness processing power at the workstations.

A practical example of the client/server model is shown in Figure 3. It uses Sybase SQL Server architecture as its example. Distributed logic occurs when the logic of the application is split across the network. The design goal of distributed logic applications is to minimize messages between components. The protocol "stack" in Figure 3 represent the OSI 7-layer model. This model is the basis for SQL servers such as Sybase/Microsoft, Oracle 7, InterBase, etc.

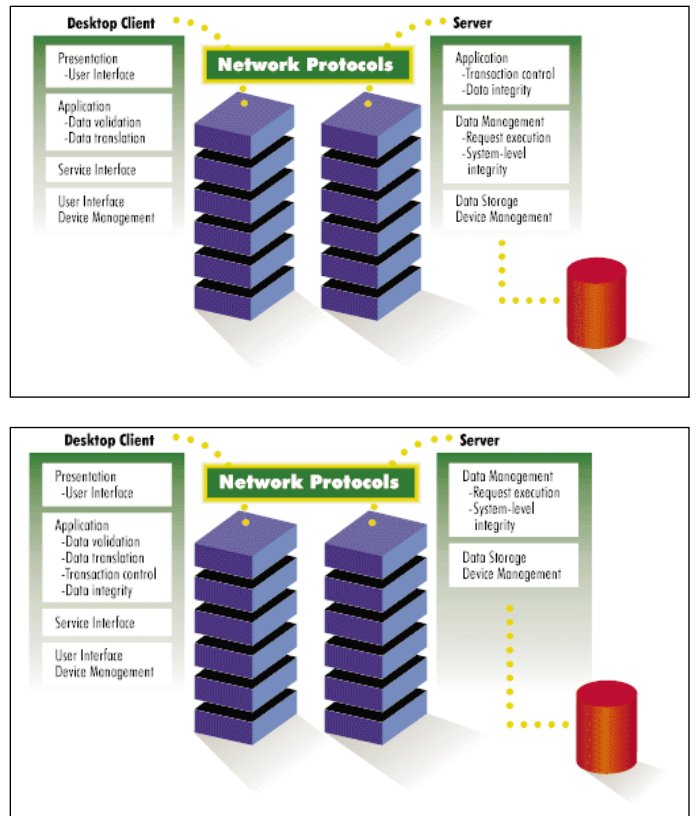


Figure 3 (Top): A practical example of the client/server model. Figure 4 (Bottom): The Remote Presentation client/server model.

Oracle 6.3 supports the Remote Presentation model shown in Figure 4. There are no commercially available databases that support the distributed processing model yet.

Downsizing: Advantages of Client/Server Architecture

In mainframe environments, the terminal on the desktop provides the presentation services, and the host provides the remaining functions. The host determines every response to the user's queries, completes the business logic, and retrieves data from the database.

A mainframe environment has its disadvantages. First the user-interface is terminal-based, so you cannot create GUI interfaces when all the processing is on a central host. Second, each additional user increases the load on the system. Even the network load is higher than necessary.

Downsizing to client/server architecture offers several key advantages over the typical IBM mainframe environment:

- Programming effort can be more efficient, through the use of better tools and languages.
- Hardware/system software cost per transaction can be dramatically reduced up to 80 percent.

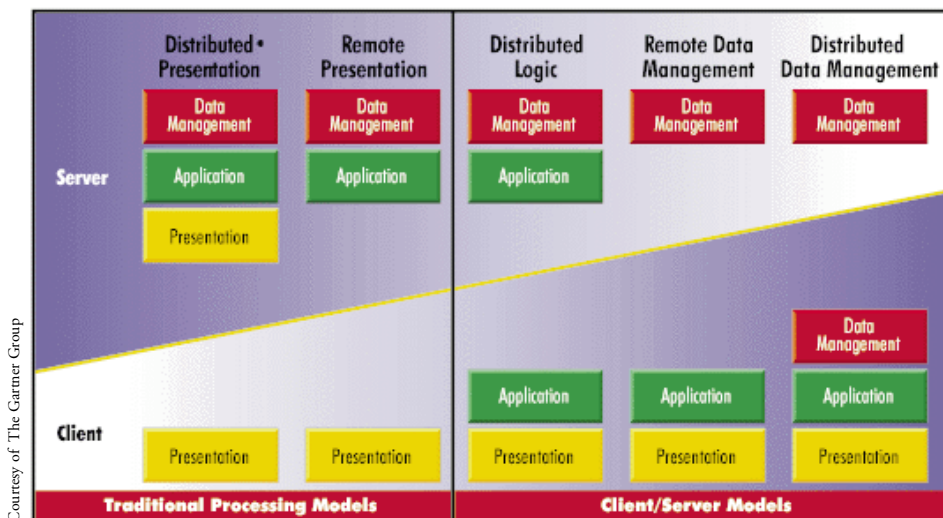


Figure 2: Degrees of client/server computing. The amount of processing accomplished at the workstation increases as you move from left to right.

Courtesy of The Gartner Group

- Average response time is usually less than one second, compared to a typical mainframe response time of three to five seconds. Response time improvement is even more dramatic when the user is on a dial-up line, or in a location where wide-area network links are slow.
- Client/server systems are scaleable. They can begin as a small departmental system, and grow to support the entire company — without a significant change in the programming.

Why Upsize to Client/Server from File Servers?

For PC-based programmers, the best way to understand client/server architecture is to contrast it against the file server architecture most of us are familiar with. Both architectures are designed to allow users to store and access data in common databases.

In file server architecture, workstations on the network running a database (e.g. Paradox) access common data on the network file server. The file server — however large and fast — is treated as a large, shared hard disk by each user's copy of Paradox.

The desktop client handles all the presentation, business logic, and database functions. The file server merely retrieves and passes the necessary files to the desktop. With no network lag between presentation logic and services, using the desktop to handle all displays makes it easier to create a good GUI for applications. Unlike mainframes (where each additional user is an increased burden), each additional user on a file server actually brings more processing power to the network.

However, there are two drawbacks to the file server model: demands on the desktop are too high, and there is an increased network load. Suppose you have an order entry database that contains a customer, orders, parts, and items table. A perfectly reasonable request would be to list customers, each part ordered by the customers, and the total value of each part. In a file server situation, the contents of all four tables must cross the network to the client, regardless of how much data is involved, thus bogging down the network.

Also, in a file server model, locking of records and tables is left to program-specific methods instead of the database, and data access is controlled by the code within the application.

Some of the disadvantages with this environment include:

- When querying and accessing data, much of the data and all the indexes travel across the network to the program running on the workstation.
- Each workstation must have significant computing power to execute queries, perform presentations, and manipulate data.

- Most file server databases have no data recovery capabilities in case of a software or hardware problem.
- Each application must enforce referential integrity and business rules (for example, not allowing users to delete an account if there are open customers for that account).
- Since the data on the file server is usually stored in a proprietary format, other applications can only access the data by exporting from the database (unless they support the same proprietary format). This creates redundant data, and introduces major concurrency and accuracy problems.

As we'll see, the simple client/server model (client = PC, server = database) addresses these and other limitations effectively by splitting the responsibilities between the two. The client application — typically a GUI application running on the user's workstation — takes care of the user interface (including display and data presentation), user interactions with the application, local validity checks, and communications with the server.

The database server — usually a multi-tasking program running on a dedicated PC on the LAN, a mini-computer, or even a mainframe — handles all database updates, data security and recovery, query optimization and processing, transaction management, referential integrity, enforcement of business rules, and communication to the client.

By providing all access to the data using an open, non-proprietary SQL standard, client/server systems allow multiple applications to access the same database.

Borland's Upsizing Strategy and BDE

From the above discussions, it's clear there is a strong case for upsizing. Borland rates upsizing to be an even more important trend than downsizing, and contends that upsizing will occur when PC databases become business-critical at the workgroup level. Borland's upsizing strategy for Delphi (and its other products including dBASE and Paradox) is based entirely upon the

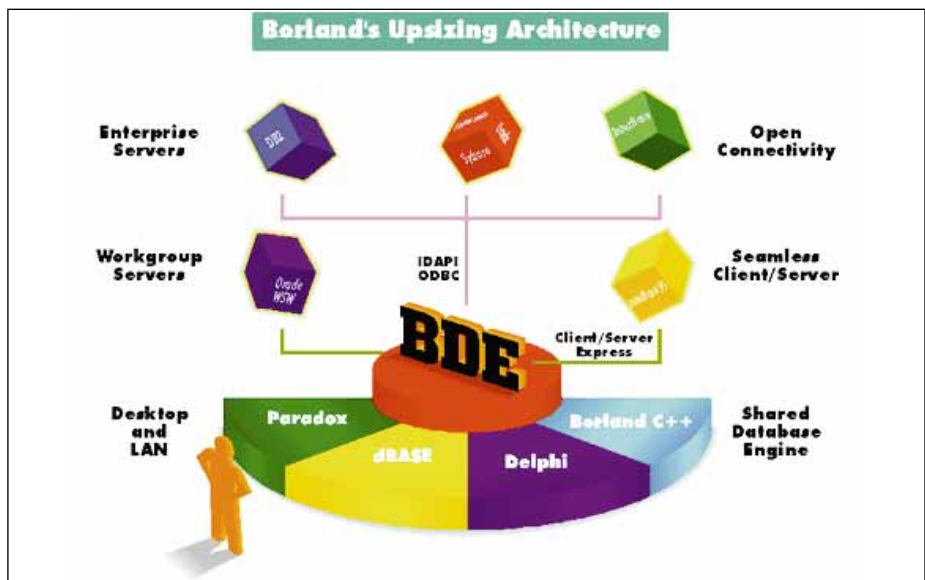


Figure 5: Borland's upsizing strategy is built on the Borland Database Engine (BDE).

powerful Borland Database Engine (see [Figure 5](#)). The BDE's goal is to provide full access to all types of data, regardless of where they reside. This means that you'll be able to create applications that not only use desktop data, but SQL data from ODBC data sources and SQL servers.

Borland's product developers have used IDAPI and BDE as components in dBASE and Paradox for Windows. Paradox 5.0 for Windows was the first product to show off the advantages of Borland's underlying database technology. Because all data access in the new Borland desktop products occurs via BDE/IDAPI, switching from desktop databases to SQL servers should be transparent.

The BDE has an impressive list of features including bi-directional cursors, linked cursors, bookmarks, navigational access to SQL data sources, and support for either SQL or QBE queries against all data sources. While the BDE handles dBASE, Paradox, and ODBC data sources, SQL Link provides high-performance native access to SQL servers. It also supports database aliases that provide a powerful metaphor for pointing to databases.

The key points to Borland's up-sizing strategy for Delphi are:

Scaleable Applications. By enabling developers to reference tables by aliases created using the IDAPI utility, Delphi makes it easier to develop an application using test data in local tables and later deploy it using a SQL database server. Theoretically, applications can be ported by merely changing the aliases. However, although this is true for simple decision support systems, most OLTP applications require considerable planning and design to be scaleable. Nevertheless, aliases do ease the migration process.

Concurrent and Transparent Access to ODBC, SQL, dBASE, and Paradox. Delphi is built on the BDE, which enables Delphi to concurrently connect to and join dBASE IV, Paradox, and ODBC data sources as well as Oracle, Sybase, and InterBase SQL servers. Borland SQL Link 2.0 is required for connecting to InterBase, Informix, Oracle, and Sybase/Microsoft SQL Server. It provides high-performance, native access to data residing on these servers.

Delphi has key advantages for SQL database users, such as: bi-directional navigation, data ordering by index, bookmark reusability, and dynamic manipulation of SQL data via "live" data source access. You can use ODBC to connect with everything else (DB2, AS/400, Btrieve etc.). Users can access ODBC by using the BDE and third-party ODBC drivers such as the Intersolv ODBC Pack or those shipped with Microsoft's Office, Excel, Visual C++, etc.

SQL Passthrough. If you need to make database requests in SQL instead of with PC-style navigational commands, Delphi's *TQuery* component is very useful. Using *TQuery*, SQL statements (including stored procedures) can be executed directly. Data controls, such as DBGrids, can be populated from the results. [For an example of the *TQuery* component in use, see Dan Ehrmann's article "Data-Aware Delphi" beginning on page 21.]

Loading the ODBC Driver

Let's look at some practical examples of this up-sizing architecture. First, create an ODBC data source for a popular PC-based SQL database, Gupta's SQLBase. Then create a Delphi form to maintain data.

Using an ODBC data source in Delphi involves a four-step process:

- Have an ODBC driver for the data source you wish to connect to. Note that the BDE (also known as IDAPI) provides the means to connect to ODBC but does not supply the ODBC drivers. ODBC drivers for a variety of databases can be purchased from sources such as Intersolv or Microsoft. Intersolv's Data Direct Pack has more than 35 ODBC drivers. However, if you have Excel, Word, Office, or Visual C++ installed, you already have the drivers for Access, Oracle, and Microsoft SQL Server.
- Create a data source by using the ODBC Administrator (ODBCADM.EXE) from the Windows Control Panel. (If ODBC is installed on your system, then ODBCADM.EXE should be in the \Windows\System directory.)
- Create a new IDAPI driver for the ODBC data source using the IDAPI Alias Manager.
- Create a database alias using the IDAPI Alias Manager.

This article's example uses Gupta's SQLBase and Intersolv's ODBC driver for that product. To begin, create the SQL Base ODBC data source, and follow these steps:

- Launch the ODBC Administration Application from the Windows Control Panel or by running ODBCADMN.EXE.
- Click the **Add** button of the Data Sources dialog box to display the Add Data Source dialog box.
- Double-click **Gupta SQLBase Driver** in the **Installed Drivers** list box to display the ODBC SQLBase Driver Setup dialog box. Click the **Options** button to display the additional choices available (see [Figure 6](#)).
- Enter Gupta SQLBase in the **Data Source Name** edit box and a description such as Gupta SQLBase ODBC Driver in

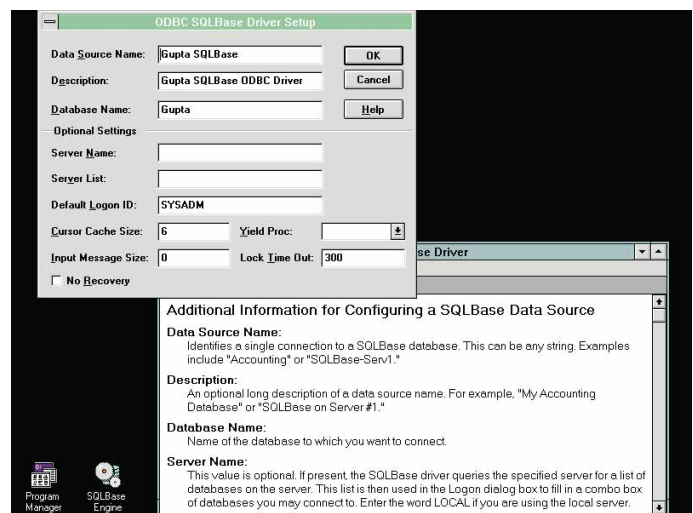


Figure 6: Setting the Database Name to Gupta in the ODBC SQLBase Driver Setup dialog box.

the **Description** edit box. When SQLBase is installed, it installs a sample database named Gupta as well. Since we are going to use tables from this database, set the **Database Name** to Gupta and **Default Logon ID** to SYSADM. (As shown in the [Figure 6](#), all ODBC drivers are accompanied by an on-line help file for additional information.)

- Click the **OK** button to close the dialog box, and click the **Close** button of the Data Source dialog box to exit the ODBC Administration application.

Now, we'll create a new IDAPI driver. Launch the BDE Configuration Utility and select the **New Driver** option from the Drivers page to display the Add ODBC Driver dialog box (see [Figure 7](#)). Enter the name of the **SQL Link Driver**, for example SQLBase (the Utility automatically prefixes it with ODBC_). Then select Gupta SQLBase Driver as the **Default ODBC Driver**, and Gupta SQLBase as the **Default Data Source Name**.

Creating a BDE Alias

Before using the ODBC data source, you need to define an alias for the data source. Select the Aliases page of the IDAPI Configuration Utility, and click the **New Alias** button to display the Add New Alias dialog box (see [Figure 8](#)).

Enter a name in the **New alias name** edit box (SQLBase for example) and select the correct **Alias type** from its list box. That is, specify the driver name of the ODBC data source entered earlier in the ODBC Administrator, ODBC_SQLBase in this example. Click **OK** to accept the changes to the dialog box.

Back at the Aliases page, highlight the SQLBase alias and enter the appropriate path to the SQLBase sample database (c:\sqlbase\gupta by default). Then save the changes to the IDAPI configuration file (IDAPI.CFG by default) by selecting **File | Save** from the menu. Finally, exit the BDE Configuration Utility.

Once you have set up an alias, the ODBC database can be used just as a local Paradox or dBASE database. It's important to note that although we're using a Gupta SQLBase database in this example, the steps indicated above apply equally to all desktop or client/server databases (e.g. Paradox, dBASE, Access, Oracle, Sybase, or IBM AS/400) provided you have access to IDAPI/ODBC drivers and connectivity software for those back-ends.

Creating a Delphi Client/Server Application

Delphi's database features combine the best of both the desktop and client/server worlds. It lets you navigate data (PC style) using the *TTable* component. It also allows you interact with a back-end database directly using pass-through SQL statements in the *TQuery* component.

Even if you're not a client/server guru, you can sample Delphi's RAD capabilities almost immediately. Let's explore some of Delphi's unique features by creating a simple master/detail form for our SQLBase database.

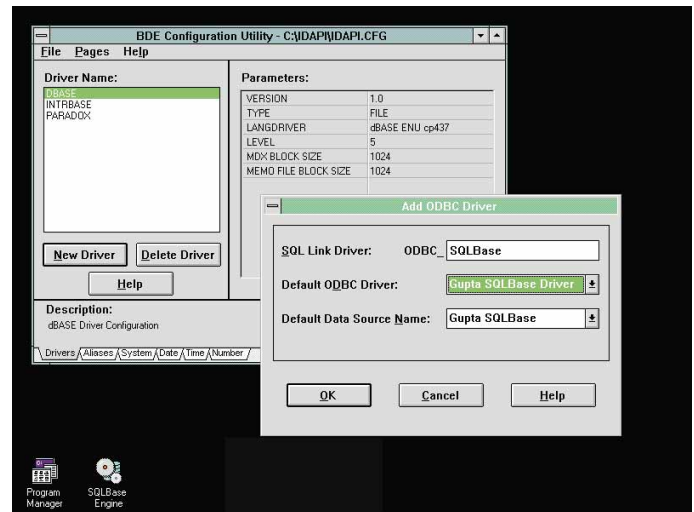
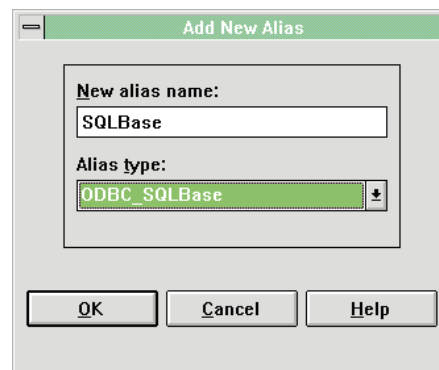


Figure 7 (Top): Using the BDE Configuration Utility's Add ODBC Driver dialog box.
Figure 8 (Left): Defining a BDE alias at the Add New Alias dialog box.



Start Delphi and close the default project (Project1.DPR) and Pascal unit (Unit1.PAS) that Delphi automatically generates. Choose **No** in the dialog box asking if you want to save changes to Unit1.PAS. We're now ready to use Delphi's Database Form Expert.

To do this, select **Help | Database Form Expert** from the Delphi menu. The first screen of the Expert will be displayed (see [Figure 9](#)). For **Form Options**, select **Create a master/detail form**. Note that the **DataSet Options** is set to *TTable* components. Accept this default option and click on the **Next** button.

The second Expert screen prompts you to **Choose a table to use with the master query**. Use the **Drive or Alias name** control to specify SQLBase as the database alias. The Expert will display the Database Login dialog box for SQLBase (as shown in [Figure 10](#)) for you to enter the appropriate password. Enter the default password, SYSADM, and click on **OK**.

The **Table Name** list box will be filled with SQLBase tables. Specify ORDER_MASTER as the master table (as shown in [Figure 11](#)) and click on the **Next** button. At the next screen, select all fields from the **Available Fields** list for the master table (see [Figure 12](#)) and click on **Next**. Then choose **Horizontal layout** on the next screen (not shown) and click on **Next**.

The next Expert screen prompts you for the detail table (see [Figure 13](#)). Specify ORDER_DETAIL as the detail table and click on **Next**. At the next screen (not shown), select all the

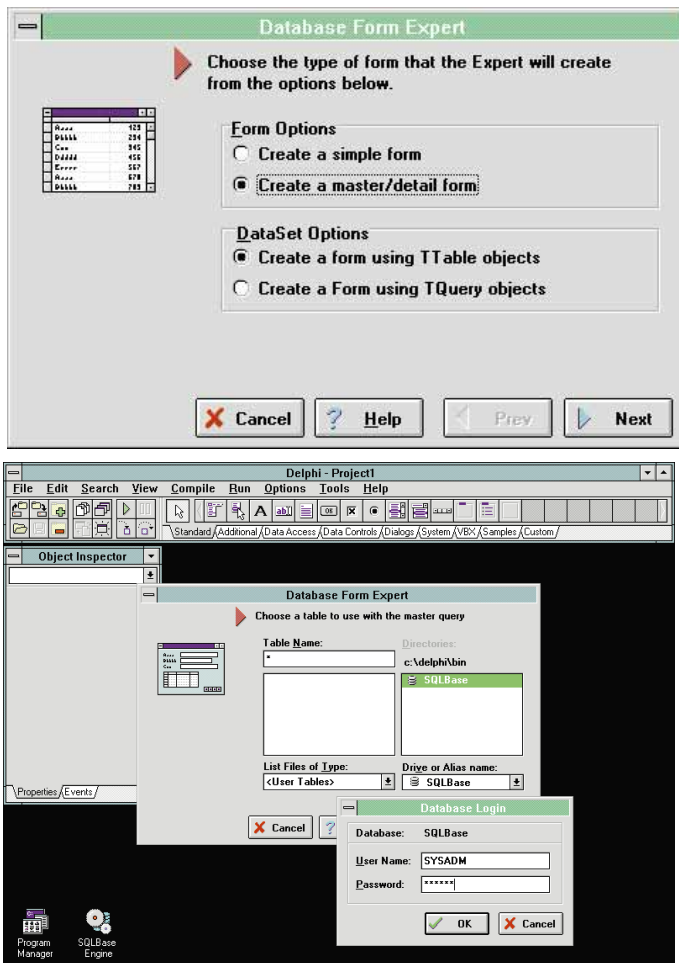


Figure 9 (Top): The first screen of Delphi's Database Form Expert.
Figure 10 (Bottom): Selecting the SQLBase alias.

Available Fields and click on **Next**. The subsequent screen (not shown) prompts you for the layout option. Accept the default (**Grid**) option by clicking on **Next**.

The next Expert screen asks you to specify the relationship between the master and detail tables. The tables are joined by the `ORDER_NUM` field, so select `ORDER_NUM` from both list boxes and click the **Add** button to create the join (as shown in [Figure 14](#)) and click the **Next** button.

The final Database Form Expert screen (not shown) asks if you want to generate a main form for the new application. Do so by clicking the **Create** button. The new form and its associated unit file will be displayed (as shown in [Figure 15](#)).

Delphi's Unique Features

It's a "Two-Way Tool". Note the way Delphi automatically generates all the Object Pascal code for the form. This two-way facility generates code automatically from form components and vice-versa, and is the first of Delphi's unique features.

Live Data at Design time. When you set the `Active` property to `True` for the `TTable` component generated by the Database Form Expert, you will immediately see the *live data* displayed for the first master record and corresponding detail record (see [Figure 15](#)).

Using SQL servers to execute *stored procedures* (collections of procedural code that typically include SQL for accessing the database) with the `TQuery` component is an interesting way to use this live data facility. To see this, create a form using the Database Form Expert and specify the `TQuery` component as the source and a Sybase/MS SQL Server as the alias. After the form is generated, modify the `TString` property to `exec sp_who`. When you press **OK** to return from the String list editor, you will be prompted to enter the **User Name** and **Password** for the SQL Server (as shown in [Figure 16](#)).

Next, even in design mode, you can see the results of the executed stored procedure (see [Figure 17](#)). Furthermore, you can replace `sp_who` with any other Sybase stored procedure and execute the results.

"Client/Server" on Your Desktop. There is a common misconception that client/server means elaborate network configurations. Actually a client/server system doesn't require the database to physically reside on a different PC. Gupta's SQLBase (like Borland's InterBase) is a true SQL database server that can still run on your desktop. The little icon titled SQLBase Engine in [Figure 15](#) represents the SQLBase engine process running simultaneously on your PC.

True High-Performance EXEs. If you've used Visual Basic or PowerBuilder to develop applications, you will immediately realize the strengths of Delphi. Delphi's native code compiler generates true high-performance machine-code unlike the p-code generated by PowerBuilder and VB. Once you have compiled the application by selecting **Compile | Build All** or **Run | Run** from the menu, the resulting EXE can be run as a stand-alone application.

Less Code Than Visual Basic. In VB, the same master-detail functionality would have required several pages of code. Why? VB provides very little support for database work since it's primarily a general-purpose Windows programming tool. VB's native grid and other controls are not data-aware. The programmer must write unique routines to process data to and from the control (such as a grid) to the database table, or resort to third-party controls. In Delphi, the same functionality is achieved with *no code*.

Client/Server Tips

Needless to say, front-end tools can only go so far in the application building process. Building successful client/server applications requires complex multi-disciplinary skills — knowledge of the client operating system and front-end tools, a firm grounding in client/server architecture, SQL literacy, middleware knowledge, and back-end server skills.

To go with the new skills, there are a few general rules for creating client/server systems. The first is "Divide and rule". One of client/server architecture's major benefits is its ability to provide a more efficient division of labor between client applications and database servers. Try to make use of this architecture — split your application into presentation (UI) and business logic components, and off-load business logic to the back-end server where possible.

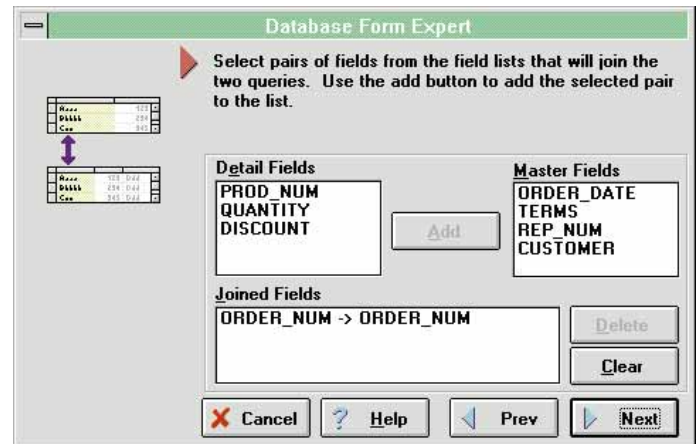
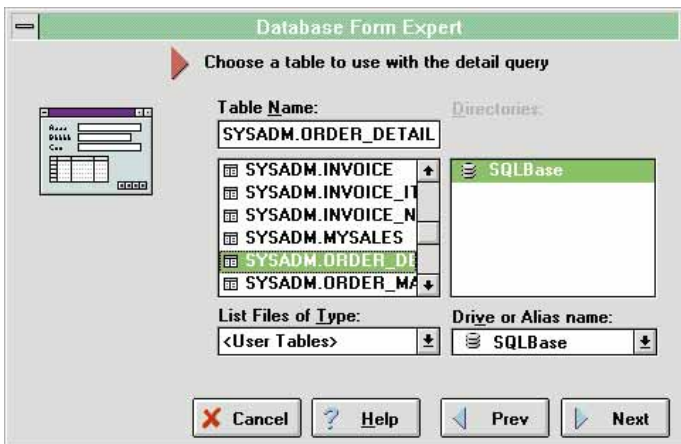
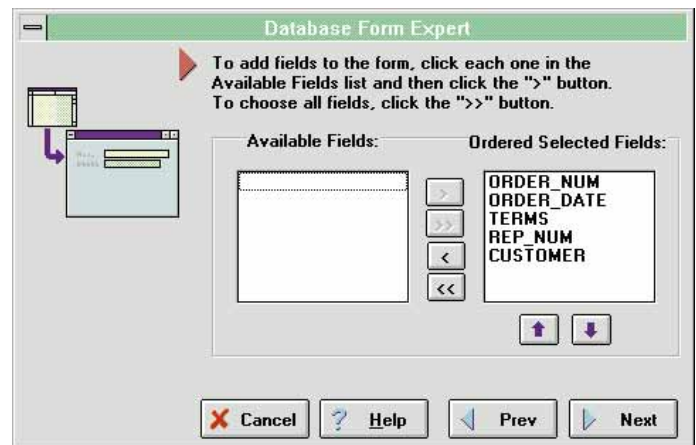
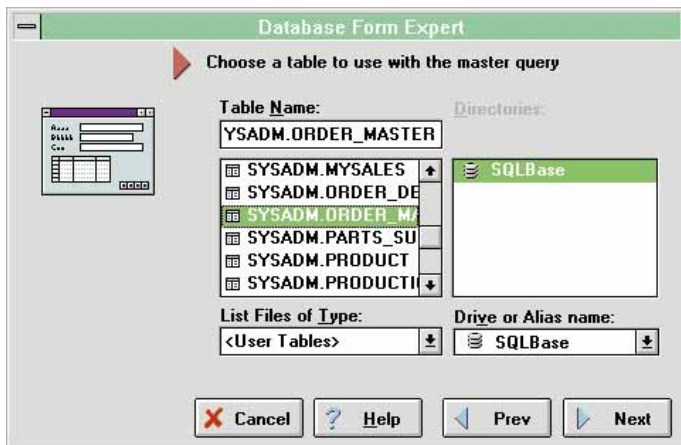


Figure 11 (Top Left): Selecting ORDER_MASTER as the master table. **Figure 12 (Top Right):** Selecting all fields from the master table. **Figure 13 (Bottom Left):** Selecting ORDER_DETAIL as the detail table. **Figure 14 (Bottom Right):** Defining the link between the master and detail tables.

Since the database could be accessed from outside the application you are building (e.g. a query tool such MS-Query may be updating your tables), it makes sense to implement the business rules using triggers on the back-end server. You can reduce the load on both the client and network by moving the business logic to the server, leaving just the presentation services and logic on the client. The mechanism most commonly used for putting business logic on the servers is stored procedures.

The second rule is “Minimize network traffic between the client and server”. Follow the general rule for client/server front-ends — deal with fewer records at a time. Any time a set of records has to be retrieved from the server you will incur the transmission time for the records to move from the server to the front-end. Large numbers of records retrieved from the server increase network traffic — watch out for this especially in QBEs. (If you are using Sybase, you can make effective use of temporary tables [#tables] to hold intermediate results on a pass-through SQL query or write stored procedures. Stored procedures offer significant benefits over regular SQL statements — improved performance and reduced network traffic. In addition, they are both shared and cached.)

If accessing dBASE or Paradox files on your file server is akin to making local phone calls, doing SQL queries against SQL servers is a like making long-distance phone calls. As with any long-distance phone call, you will have to plan your conversations to minimize

the amount of time (thereby the cost) you spend on the network.

The third rule is “Use stored procedures to fine tune performance”. Because they are compiled, the judicious use of stored procedures can improve performance and reduce server load (compared to direct QBE's or uncompiled embedded SQL sent from the client). Another aspect of performance is network load. Instead of sending

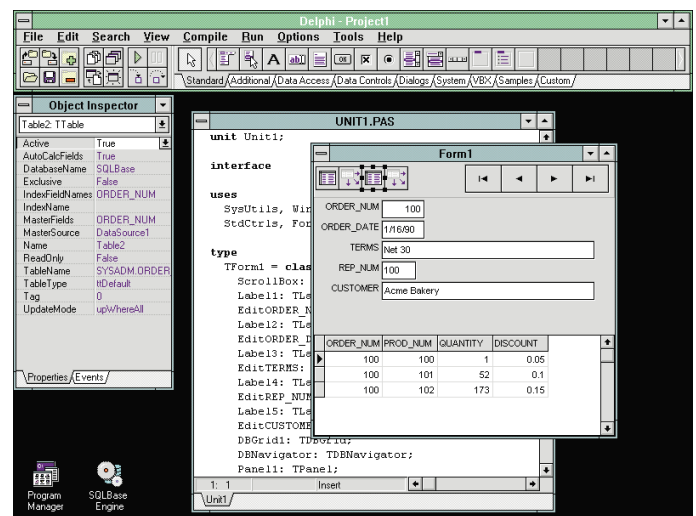
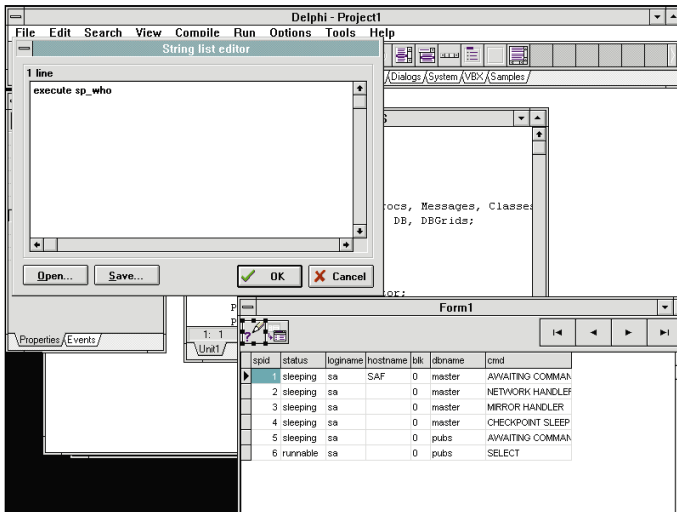
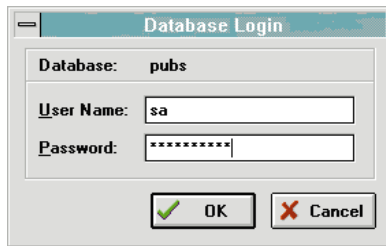


Figure 15: The result of the Database Form Expert. It has generated a complete form and the underlying Object Pascal code. Note that the SQLBase Engine is running.

Figure 16 (Top): The Database Login dialog box for SQL Server.
Figure 17 (Bottom): Executing a stored procedure in design mode.



SQL statements from the client to the database server and returning intermediate results to the client, all the processing and decisions can happen on the server with a simple call to the stored procedure.

For example, you could use a stored procedure to update a Part table so that the average price of all the parts would increase by an amount specified in the parameter @avg. Here's an example of what such a stored procedure might look like:

```
CREATE PROCEDURE price_increase @avg AS
  WHILE (SELECT AVG(Price) FROM Part) <= @avg
    UPDATE Part SET price = 1.1 * price
```

Stored procedures improve database and application integrity. And because they are shared, they ensure the consistency of operations and calculations.

Conclusion

We've taken a quick tour of client/server and Delphi's database capabilities. Delphi provides an attractive framework for making the transition to client/server. While the opportunities are immense, several new skills have to be acquired especially in the areas of application design, SQL server know-how, and the SQL language.

There are exciting times ahead for Delphi developers willing to take on the client/server challenge. ▲

Sundar Rajan is a Consultant with On-Line Resources Inc., a Longwood, Florida based consulting firm specializing in client/server development. He is currently developing a number of client/server applications for a large Japanese semi-conductor manufacturer in Northern California. Before moving to the US in 1993, Sundar founded and operated SunSoft Systems (NZ), a software consulting firm in Wellington, New Zealand. He can be reached on CompuServe at 72774,1030.





DB NAVIGATOR

DELPHI / BDE CONFIGURATION UTILITY | ALL LEVELS



By Cary Jensen Ph.D.

A Programmer's Compass

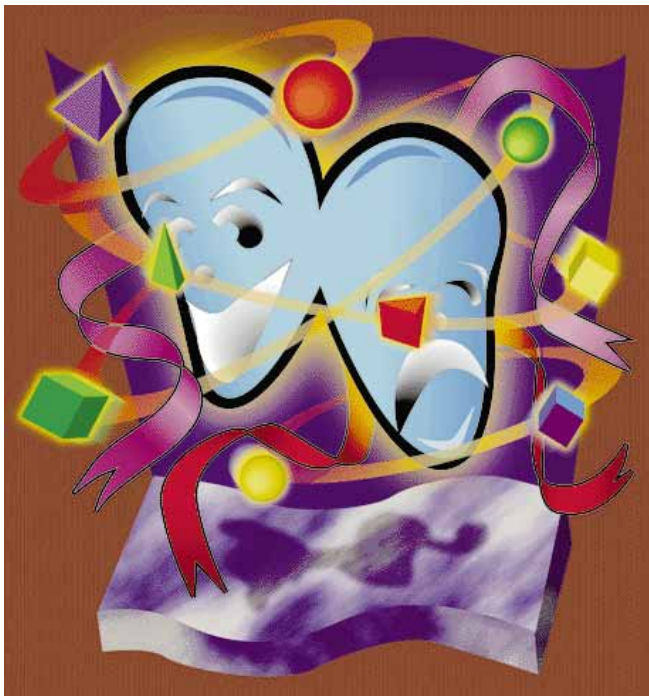
Using Borland Database Aliases in Delphi

Delphi database developers are discovering what Paradox for Windows and dBASE for Windows users have known for a while: Aliases are hot.

An alias is a pointer to data. On a stand-alone machine or a local area network, an alias refers to a subdirectory. When used in a client/server environment, an alias points to a remote database.

The advantage provided by an alias is quite simple. An alias permits you to refer to data without an explicit reference to where the data is stored. For example, the *DatabaseName* property of a *TTable* component can be set to an alias instead of a DOS path. (A particular table stored in the directory specified by the alias is defined by the *TableName* property of the *TTable* component.)

If the location of the table specified in the *TableName* property is later changed, it's necessary only to change the directory referred to by the alias. This can easily be done without having to re-compile. In fact, using an alias permits changes to the directory even when the source code is unavailable.



The Types

There are two types of aliases. Usually when we think of aliases we think of those specified in the IDAPI configuration file (IDAPI.CFG by default). (*IDAPI* stands for Integrated Database Application Programming Interface.) These aliases can be thought of as global to whoever shares the .CFG file. That is, all applications that use a particular IDAPI configuration file share access to all the aliases defined in that file. These aliases are called *BDE aliases*. (BDE is an acronym for Borland Database Engine.)

The second type of alias is defined from within a project. These alias definitions are available to that project only, and are called *local aliases*.

Let's begin by taking a look at adding a BDE alias to IDAPI.CFG, and then using that alias in a form. Later in this article, we'll discuss how to create a local alias.

Creating BDE Aliases

The most common way to add an alias is through the BDE Configuration Utility (BDECFG.EXE), shown in [Figure 1](#).

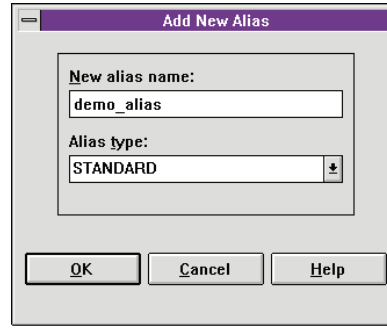
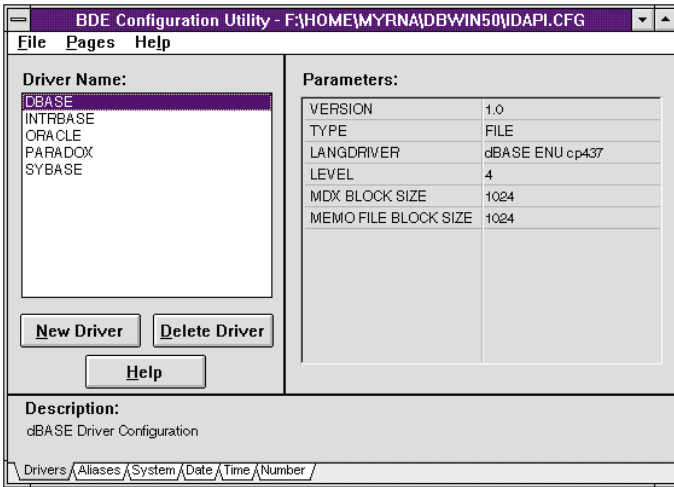


Figure 3: Use the Add New Alias dialog box to define the name and driver of an alias to add to IDAPI.CFG.

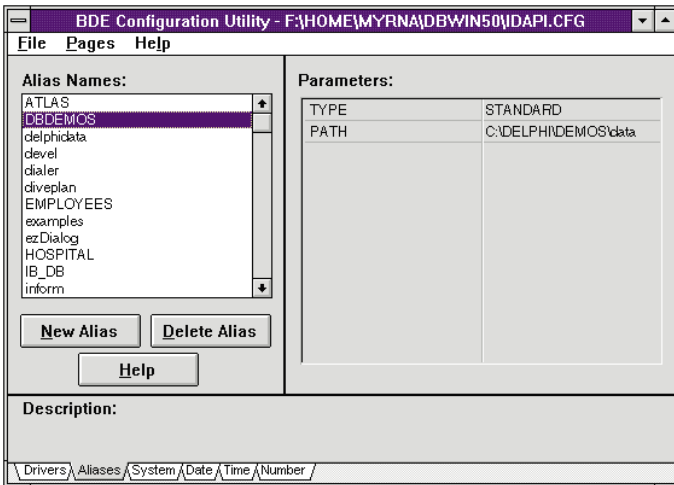


Figure 1 (Top): The BDE Configuration Utility. **Figure 2 (Bottom):** The Aliases page of the BDE Configuration Utility.

(Paradox for Windows and dBASE for Windows users will recognize this applet as the IDAPI Configuration Utility.) To access it, double-click on the Database Engine Configuration icon in your Delphi program group.

Next, click on the Alias tab to display the Aliases page, shown in Figure 2. From here you can add, delete, and modify your BDE alias definitions.

To add an alias, click the **New Alias** button. The BDE Configuration Utility displays the Add New Alias dialog box, where you can specify both a name and a driver type for the alias (see Figure 3). Your alias name can be up to 31 characters in length and can consist of letters, numbers, and the underscore character. Most developers try to keep their alias names informative, yet short.

The driver type you select depends on the types of tables you want to access using this new alias. If you want to use the alias to open ASCII, dBASE, or Paradox tables, select the STANDARD driver. If you are establishing your alias to attach to a database server, select the driver corresponding to that server. To attach to an Oracle server for example, select ORACLE from the **Alias type** drop-down list. By default, Delphi includes the STANDARD and INTRBASE dri-

vers. Other drivers are available if you have also installed the Borland SQL Link. [To see an example of creating an alias for a Gupta SQLBase database, see Sundar Rajan’s article “From ODBC to the BDE” beginning on page 31.]

After you specify the alias name and driver type, click the **OK** button. The BDE Configuration Utility will return to the Aliases page, and the new alias name will appear in the **Alias Names** list.

You’re still not finished, however. You must now define the parameters of the alias. Which parameters an alias has depends on the driver type. For example, aliases created using the STANDARD driver have two parameters — the path and the default driver. Aliases created to refer to a database server will have several more parameters. This can be seen in Figure 4, which shows the **Parameters** list for an alias specified for use with the INTRBASE driver.

You can also use the Aliases page of the BDE Configuration Utility to modify or delete existing aliases. How you modify an alias depends on whether you need to change the driver type. If you don’t need to change the driver type, highlight the alias name and change its parameters displayed in the **Parameters** list. To delete an alias, highlight the alias name and click the **Delete Alias** button.

Changing the driver type is more involved. You must first delete the alias, and then use **New Alias** to add a new alias using the old alias name. The most common reason for changing the driver type is changing an application from running against local data to running against data stored on a database server.

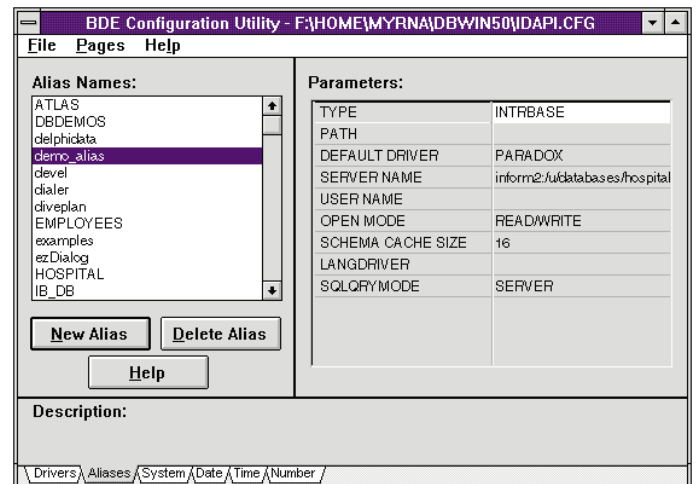


Figure 4: The parameters for an alias based on the INTRBASE driver.

Once you have added, deleted, or modified your aliases, you need to update IDAPI.CFG. From the BDE Configuration Utility menu select **File | Save** to update the current version of IDAPI, or **File | Save As** to create a new IDAPI.CFG. (Remember that when Delphi loads, it knows which IDAPI.CFG to use based on the [IDAPI] section of the WIN.INI file.)

(You can also create, modify, and delete BDE aliases using the Database Desktop, which is also available in the Delphi group window. Select **File | Aliases** from the Database Desktop menu to display the Aliases Manager dialog box.)

Using Aliases

Let's now turn our attention to using a BDE alias to refer to data on a form. To use an alias you must set the appropriate property of a Data Access component. Examples used here will include Table (*TTable*) and Database components (*TDatabase*). Keep in mind, however, that you can use aliases in other areas of Delphi, such as local SQL as well as other descendants of *TDataSet* objects, such as *TQuery*. (*TTable* is itself a *TDataSet* descendent.)

The following steps demonstrate the use of the alias DBDEMOS, which is created automatically when you install Delphi. If this alias is not available, you can use the BDE Configuration Utility described earlier to create it. This alias should point to the database demonstration files stored in a Delphi subdirectory named \DEMOS\DATA (C:\DELPHI\DEMOS\DATA by default). If both these files and alias are not available, then reinstall your Database Desktop from your Delphi installation disks.

First, create a new project by selecting **File | New Project** from the Delphi menu. Then add the following components from the Component Palette: a *DataSource* and *Table* from the Data Access page, and a *DBGrid* from the Data Controls page.

Select the *TDataSource* component (named *DataSource1*) and set its *DataSet* property to *Table1* (the *TTable* object's name). Select *DBGrid1* (the *TDBGrid* component) and set its *DataSource* property to *DataSource1*.

You are now prepared to use the alias. Select the *TTable* object (named *Table1*) and click on its *DatabaseName* property in the Object Inspector. The drop-down list displayed will contain all aliases currently defined in IDAPI.CFG (as shown in Figure 5). Select the BDE alias named DBDEMOS. Next, select the *TableName* property of *Table1*. Click on the down arrow and select the table named EMPLOYEE.DB.

Start the database connection by changing the *Active* property of *Table1* to *True*. The easiest way to do this is to double-click in the Value column of the *Active* property. With the database connection now active, the contents of the Employee table are displayed in the *DBGrid* component, as shown in Figure 6.

The important characteristic of this demonstration is that the path to the EMPLOYEE.DB file is not stored in the project.

Only the alias is stored as a property of the *TTable* component. The path is stored in IDAPI.CFG. If you later move a copy of EMPLOYEE.DB to another directory, and modify the path of the alias DBDEMOS using the BDE Configuration Utility, this form will automatically display the contents of the table copy — not the original.

Now is a good time to save this project. Select **File | Save Project** and save Unit1 as APAGE1.PAS, and the project as ALIAS.DPR.

As mentioned earlier, you can also create an alias that is local to an application. This alias can be a completely new alias, or based on an existing one, overriding one or more of the existing alias parameters.

Creating Local Aliases

The following steps will demonstrate several interesting capabilities. First, you'll learn how to establish an alias local to an application. Second, you'll learn how to change the alias that a *TTable* component points to during run-time.

Before continuing, you need to make a copy of the EMPLOYEE.DB table. Using the File Manager (or the DOS COPY command) copy EMPLOYEE.DB to another directory. Make sure to remember which directory you are placing this copy in. In the following demonstration, a copy of EMPLOYEE.DB is placed in the subdirectory C:\TEMP.

We'll continue to modify the example form. From the Data Access page of the Component Palette, double-click on a Database component to add it to the form. Next, double-click on this component to display the Database dialog box (see Figure 7). (Alternatively, you can right-click the Database

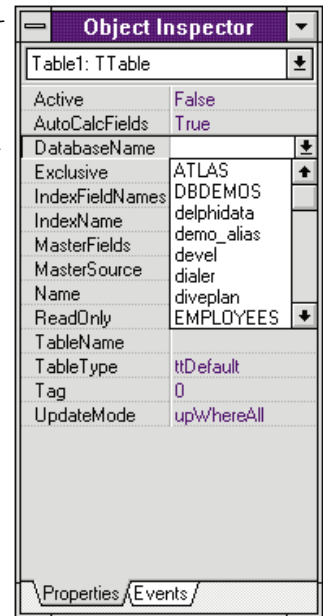


Figure 5: The *DatabaseName* property drop-down list includes all aliases defined in IDAPI.CFG.

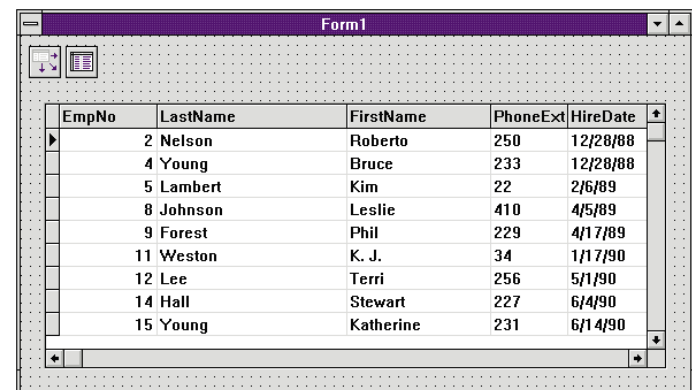


Figure 6: Live data is displayed in the *DBGrid* component once the database connection is activated.

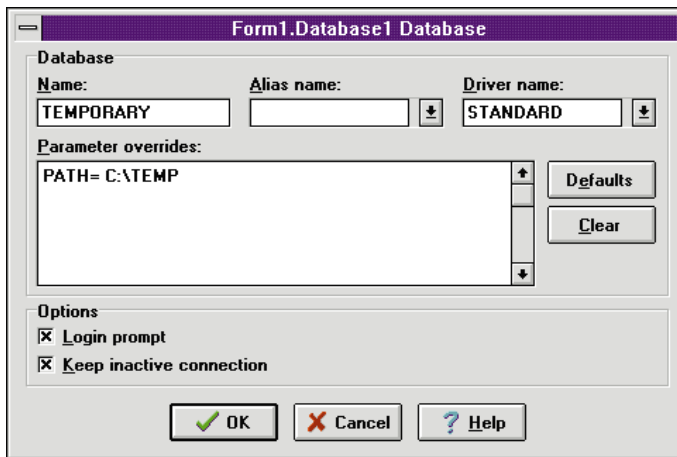


Figure 7: Use the Database dialog box to define a local alias.

component and select **Database editor** from the SpeedMenu.) You can use this dialog box to define either a new local alias, or one that overrides the parameters of an existing alias.

Begin by entering a name in the **Name** text box. For this example, type in the name TEMPORARY. If you want to base this new alias on an existing one, you can use the **Alias name** drop-down list to select the existing alias. However, when you are creating a new alias, leave this drop-down list empty.

When you create a local alias based on an existing one, you do not, and cannot, specify a driver name. The driver name will automatically be based on the driver used by the existing alias. When your local alias is not based on an existing one, you must use the **Driver name** drop-down list to define the driver that the alias will use. For this example, select the STANDARD driver.

The **Parameter overrides** box is used for specifying the parameters of the local alias. If your alias is based on an existing one, enter only those parameters you want to override. All other parameters will be based on those of the existing alias. You can click the **Defaults** button to load all parameters of the existing alias in the **Parameters overrides** box. Modify those you want to override and delete the remainder.

If your alias is new, you must supply all parameters required by that alias driver. Click **Defaults** to easily prepare these parameters. If your alias is not based on an existing one, the **Defaults** button places blank parameters based on the selected driver name in the **Parameter overrides** box. Click the **Defaults** button. Because the STANDARD driver is selected, the text "PATH=" appears in the **Parameter overrides** box. Complete this path by typing C:\TEMP, the subdirectory where you placed the copy of EMPLOYEE.DB. The completed Database dialog box appears in [Figure 7](#).

The Database dialog box contains several check box options to use when your local alias points to a remote database. Enable the **Login prompt** when you want Delphi to display a dialog box to the user when attempting to make a connection to the

database. If you do not enable the **Login prompt**, you must supply a user ID and password programmatically. You can also enable the **Keep inactive connection** option to prevent Delphi from dropping an inactive connection. Since the alias in this example does not refer to a remote database, you can ignore these check boxes.

Click the **OK** button to close the Database dialog box. When you close the Database dialog box, its settings are used to modify the properties of the Database component. Using the Object Inspector, you'll notice that the *DatabaseName* property is the local alias name, *DriverName* is the name of the specified driver, and the *Params* property is the parameter list. If your local alias was based on an existing alias, the *AliasName* property would contain the name of the existing alias, and *Params* would contain the parameter overrides. (Note that instead of using the Database dialog box, you could have also defined these properties directly by using the Object Inspector.)

You can now select this local alias for use by the *TTable* component, *Table1*. Select *Table1* from the Object selector. Disconnect from the current database by toggling the *Active* property from *True* to *False*. Next, use the *DatabaseName* property drop-down list to display the available aliases. As shown in [Figure 8](#), this list now contains both the BDE aliases defined in IDAPI.CFG, as well as the local alias defined by the *TDatabase* object. If your project contained more than one *TDatabase* object, more than one local alias would be available.

Select the local alias, TEMPORARY. Now make the connection active again by toggling the *Active* property from *False* to *True*.

Switching Aliases at Run-Time

Let's now add a button to this form. The button will toggle between the aliases that the *TTable* component points to during run-time. From the Standard page of the Component Palette, double-click the Button component to add it to the form and place it beneath the DBGrid. Resize the button to about double its length (we're going to define a pretty long caption for it). Now, double-click the button to display the *Button1Click* procedure. Enter the Object Pascal code shown in [Figure 9](#).

Press **[F9]** to compile and run the form. If you click the button, you will notice that the DBGrid is momentarily blank while it's inactive. When it becomes active again, the table displayed is one from a directory pointed to by a different alias. Because of the code added to the button's *OnClick* event, successive clicks toggle between the two tables, as well

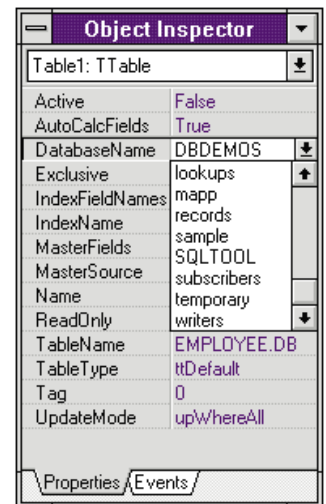


Figure 8: Local aliases appear with BDE aliases in the *DatabaseName* property drop-down list.


```

procedure TForm1.Button1Click(Sender: TObject);
begin
  if Table1.DatabaseName = 'DBDEMOS' then
    begin
      Table1.Active      := False;
      Table1.DatabaseName := 'temporary';
      Table1.Active      := True;
      Button1.Caption    := 'Switch to Local Alias';
    end
  else
    begin
      Table1.Active      := False;
      Table1.DatabaseName := 'DBDEMOS';
      Table1.Active      := True;
      Button1.Caption    := 'Switch to DBDEMOS';
    end;
end;

```

Figure 9: The *OnClick* event handler for *Button1*. This Object Pascal code toggles between a BDE and local alias, and the *Caption* for *Button1*.

as toggle the button's captions. The form is shown in [Figure 10](#) at run-time displaying data pointed to by the local alias, TEMPORARY.

EmpNo	LastName	FirstName	PhoneExt	HireDate
2	Nelson	Roberto	250	12/28/88
4	Young	Bruce	233	12/28/88
5	Lambert	Kim	22	2/6/89
8	Johnson	Leslie	410	4/5/89
9	Forest	Phil	229	4/17/89
11	Weston	K. J.	34	1/17/90
12	Lee	Terri	256	5/1/90
14	Hall	Stewart	227	6/4/90
15	Young	Katherine	231	6/14/90

Switch to DBDEMOS

Figure 10: The demonstration form at run-time. The form displays data stored in a table pointed to by a local alias. Clicking the button will switch to a table pointed to by a BDE alias.

Conclusion

Aliases permit you to point to data without having to hard-code directory paths in your applications. The aliases you use can either be those stored in IDAPI.CFG, or private to a project. When your applications use tables, aliases make your applications more flexible. ▲

The demonstration project referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\APRIL\CJ9504.

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.





FROM THE PALETTE

DELPHI / OBJECT PASCAL | BEGINNER / INTERMEDIATE

By *Jim Allen & Steve Teixeira*

Component Basics

An Introduction to Delphi Custom Components

If you haven't guessed it by now, Delphi is a component-based application development environment. So what does that mean? It means with Delphi you develop applications in part by dragging components off a palette and dropping them onto a form. A component encapsulates data and functionality. The component user doesn't have to worry about its implementation — unless, of course, you're the one designing the component.

As a component designer, you must anticipate the needs of the component user in your design and implementation. This is no easy task. You must not only anticipate their obvious needs such as properties, events, and methods, but the not-so-obvious ones such as error checking (exception handling), resource management, and implementation methods — the elements that make components “dummy proof”.

A component that is simple to use and makes a difficult or complicated task (e.g. serial communications, bitmap manipulation, complex data calculations) easy to perform, should be the goal of every component designer.

Before you begin to design custom components, you should have a thorough understanding of the components in Delphi's Visual Component Library (VCL) and custom component design. In this article, we'll present the basic information you need to know to write custom Delphi components. Although we won't actually write a functional component, we'll give you all the background information you need before you get your hands dirty.

When Do I Create a Custom Component?

As the producer of components, it's up to you to decide what types of components to create. How do you know if you need a new component or if an existing component will work? If all three of the statements below are true, then you have a candidate for a Delphi custom component:

- You need to simplify a difficult or complicated task.
- You need something that is easy to re-use.
- There isn't an existing component that can do the same task.

To reiterate, if the task is already simple, there's no reason to make it a component. Making a component for a simple job can just make it more difficult to implement and understand. If your task is something you are only



going to do once or twice during development, then you're better off writing the code and saving it in a library of functions, procedures, or class definitions. There is no need to clutter the Component Palette with components you'll never use again.

If a component that fits the bill already exists, then most of your work may be done. Completing the job may be a simple matter of attaching code to event handlers of a pre-existing component.

Any of the following reasons can be just as important:

- You want to add some custom functionality to your application (e.g. a control to give it a unique look and feel).
- You want to encapsulate the functions of a DLL (which are procedural by their very nature) to make them easier to use.
- You want to play (the most important reason of all!). Make a few components so you understand them and the VCL better.

The Family Tree

To create a custom component, you need to know which VCL component to use as a basis for your new component. Most components will be derived from a small group of ancestor class definitions:

- *TComponent* is as far down the VCL family tree as most component designers will ever need to go. It's the base class for most non-visual components, and introduces the capability of components to read and write themselves to streams. Also, it can own other components. It's used to create components such as *TTable*, *TDatabase*, and *TTimer*.
- *TWinControl* is the base class for any component that requires input focus from Windows. This can be used for encapsulating existing Windows controls like *TScrollBar*.
- *TCustomControl* is a direct descendant from *TWinControl* and can get input focus from Windows. It's the first component to have *Canvas* and *Paint* methods that allow you to customize its appearance. This can be used for most components that require input focus like *TNoteBook*, *TMediaPlayer*, or *THintWindow*.
- *TGraphicControl* does not have a Canvas or a Paint method, and instead uses its parent's canvas. It's primarily used for creating "window dressing" like *TLabel*, *TShape*, or *TBevel*.

Components can also be derived from one of the top-most components (such as *TEdit*, *TLabel*, *TForm*, etc.) if you are only modifying or expanding existing functionality. For example, you would modify *TButton* to create a specialized button.

When you need a custom component that behaves like one of the VCL components, pay particular attention to the VCL's custom classes (shown in Figure 1). Custom classes contain all the properties and methods of their namesakes (*TCheckBox*, *TComboBox*, *TDBGrid*, etc.), but the properties are not published. This is done so you can surface only the behavior you want to make available to the component user.

Component Guts

The skeleton of a Delphi custom component is simple, yet powerful in its design. The basic structure of a component derived from a *TComponent* is shown in Figure 2 (as generated by the Component Expert).

It's a standard class definition with only a few exceptions:

- The **private** section is used for the internal workings (implementation details) of the component. Anything declared within this section can only be seen from within the class unit. For method declarations, this is where you would hide anything you do not want the component user to access.
- Anything declared within the **protected** section can be seen from within the unit of the class and any new class derived from it. It's common to place methods that you want to override in a descendant class in this section.
- The **public** section is used for the run-time interface. Anything declared within this section can be seen by anything else in the application. This is where you would place properties or methods that you want component users to access only at run-time.
- The **published** section is similar to the **public** declaration. However, properties declared here will be visible in the Object Inspector in design mode.

It's important that you fully understand the differences between the declaration sections because they will have a direct impact on how your component works and how the developer works with the component. For instance, if you declare a method in the **private** section of your component and at some point someone wants to override that method — it just won't happen. By declaring the method as **private** you have locked all users out of that method.

On the other hand, if you declared a potentially sensitive variable or method in the **public** section, component users can misuse it and render the component useless. Therefore, be very careful when planning the interface of your components. They can come back to haunt you.

Something else to notice about the component skeleton is its *Register* procedure.

```
procedure Register;
begin
  RegisterComponents('Test', [MyComponent]);
end;
```

This procedure registers the component with the Delphi component library and places it on the Component Palette. Notice the 'Test' part of the call. This tells Delphi which page of the Component Palette to place your component on. If the name

<i>TCustomCheckBox</i>
<i>TCustomComboBox</i>
<i>TCustomDBGrid</i>
<i>TCustomEdit</i>
<i>TCustomGrid</i>
<i>TCustomGroupBox</i>
<i>TCustomLabel</i>
<i>TCustomListbox</i>
<i>TCustomMaskEdit</i>
<i>TCustomMemo</i>
<i>TCustomOutline</i>
<i>TCustomPanel</i>
<i>TCustomRadioGroup</i>

Figure 1: The Visual Component Library's custom classes.

```

unit Unit1;

interface

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

type
  MyComponent = class(TComponent)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Test', [MyComponent]);
end;

end.

```

Figure 2: A component created by the Delphi Component Expert.

you provide is not already on the Palette, Delphi will create a new page with that name.

Variables, Methods, and Properties

The variables, methods, and properties added to your component skeleton give it behavior. *Instance variables*, sometimes called *fields*, are variable instances that belong to the class. These variables can be of any valid type (e.g. integer, double, record). Even another class is a valid instance variable type.

Let's add some instance variables to our *MyComponent* class:

```

type
  MyComponent = class(TComponent)
  private
    Int: integer;
    D: double;
  protected
    { Protected declarations }
  public
    C: SomeOtherClass;
  published
    { Published declarations }
  end;

```

Remember, since we made the *Int* and *D* instance variables private, they won't be visible to classes declared outside this unit.

Methods are simply procedures or functions that work as members of a class. You declare them as you would any other procedure or function inside the declaration for *MyComponent*:

```

type
  MyComponent = class(TComponent)
  private
    Int: integer;
    D: double;

```

```

protected
  procedure SomethinCool; virtual;
public
  C: SomeOtherClass;
  function HowBigIsD: Double;
published
  { Published declarations }
end;

```

Notice we use the **virtual** directive as part of the *SomethinCool* procedure declaration. This allows us to override *SomethinCool* in descendant classes. There are four such directives that you would primarily use with methods:

- The **virtual** directive is used to create virtual methods. *Virtual methods* are methods that can be overridden in descendant classes to modify a class' behavior. The result of this is a small performance penalty in calling virtual methods as opposed to normal (static) methods such as *HowBigIsD*. Having virtual methods in a class also occupies more memory because each class instance keeps all its virtual methods in a table named the Virtual Method Table (or VMT).
- The **dynamic** directive is used to create dynamic methods. *Dynamic methods* function almost identically to virtual methods, but are implemented slightly differently resulting in a larger performance penalty. However, the Dynamic Method Table (DMT) requires less space than the VMT, so they are more memory efficient.
- Use the **override** directive whenever you override a method in a descendant class. Never re-use the **virtual** or **dynamic** directives when overriding a method.
- The **abstract** directive must always be paired with **virtual** or **dynamic**. Using this directive creates a procedure without functionality and is designed to be overridden by descendant classes. In other words, **abstract** helps you define an interface to a class, but the implementation is up to the descendants of the class.

Methods are defined in the **implementation** part of your unit. You must use the class name and the dot scoping operator when defining methods. For example:

```

implementation

procedure MyComponent.SomethinCool;
begin
  { Call the Windows API function to beep }
  MessageBeep(0);
end;

function MyComponent.HowBigIsD: Double;
begin
  Result := D;
end;

```

Properties allow you to publish a safe and intuitive interface for your custom components — exposing enough functionality for users get a lot out of your component, but not so much that they shoot themselves in the foot.

Defining a property is easy, just use the **property** keyword followed by the property name, type, and some directives telling the property how to read or write itself. As an illustration, we'll add a property *IProp* to the *MyComponent* to access instance variable *Int*:

```

type
  MyComponent = class(TComponent)
  private
    Int: integer;
    D: double;
  protected
    procedure SomethinCool; virtual;
  public
    function HowBigIsD: Double;
  published
    property IProp: integer read Int write Int;
  end;

```

The **read** and **write** directives mean that whenever the user reads *IProp*, its value is taken from *Int*. Whenever a user writes to *IProp*, the value is written to *Int*.

You can also read from and write to methods of *MyComponent*. For example, you can make the *IProp* property change the value of *Int* by writing to a method. Just add a *SetInt* procedure to the *MyComponent* function and change the **write** directive for the *IProp* property as shown below:

```

type
  MyComponent = class(TComponent)
  private
    Int: integer;
    D: double;
    procedure SetInt(Value: integer);
  protected
    procedure SomethinCool; virtual;
  public
    function HowBigIsD: Double;
  published
    property IProp: integer read Int write SetInt;
  end;

```

The *SetInt* method is defined as:

```

procedure MyComponent.SetInt(Value: integer);
begin
  if Int <> Value then
    Int := Value;
end;

```

In effect, this says “change the value of *Int* to match *Value* only if they are not already equal”. This little check for equality may seem extraneous, but it actually helps avoid adding overhead to our program’s execution as we’ll show in a moment.

In addition to **read** and **write**, you can also use the **default** directive to assign a default property value. When doing so, you must also assign that value in your component’s constructor, or Delphi will have problems loading your component from a .DFM file. You can make a property read-only by omitting the **write** directive.

Events

An *event* is a mechanism for linking an occurrence (a mouse-click, for example) to specific Object Pascal code. Properly, an event is a pointer to a specific method in a specific object instance.

Creating a published event is a three-step process:

- Create the event instance variable. The variable should be of *TxxxEvent* type. You can use one of the pre-defined types,

such as *TNotifyEvent*, or create one.

- Create the event property. The property will allow users to assign code to the event in the Object Inspector.
- Call the event somewhere in your component’s code.

The first two steps are straightforward based on earlier examples of instance variables and properties. Let’s add them to *MyComponent*:

```

type
  MyComponent = class(TComponent)
  private
    Int: integer;
    D: double;
    procedure SetInt(Value: integer);
    FOnIntChanged: TNotifyEvent;
  protected
    procedure SomethinCool; virtual;
  public
    function HowBigIsD: Double;
  published
    property IProp: integer read Int write SetInt;
    property OnIntChanged: TNotifyEvent
      read FOnIntChanged write FOnIntChanged;
  end;

```

Now you have to simply call the event at some point in your code. As its name implies, we’ll call the event when the value of *IProp* is changed. To do this requires a couple of modifications to the *SetInt* method:


```

procedure MyComponent.SetInt(Value: integer);
begin
  if Int <> Value then
  begin
    Int := Value;
    if assigned(FOnIntChanged) then
      FOnIntChanged(Self);
  end;
end;

```

The new line of code checks to see if the user has assigned a value to the *OnIntChanged* event, and, if so, calls that method — passing the class instance (*Self*) as a parameter. Notice our check to make sure *Int* is not equal to *Value* now pays off. The *OnIntChanged* event will now only fire when the value is actually changed (as opposed to being assigned the same value).

Conclusion

By now you know most of the basics for creating custom components in Delphi. The most important advice we’ve given is to put some thought into the design of your custom component before you begin. Next month, we’ll get our hands dirty and create some functional components. Until then, happy hacking! 

Jim Allen is an engineer in Borland’s Technical Support Department. Jim supports Delphi, object-oriented programming, and applications development in various languages. He is also responsible for internal training for database development and related subjects.

Steve Teixeira is a Senior Engineer at Borland International Technical Support. He writes for several industry periodicals, and is the co-author of *Delphi Developer’s Guide* from Sam’s Publishing. You can reach him on CompuServe at 74431,263 or via the Internet at steixeira@wpo.borland.com.



AT YOUR FINGERTIPS

B Y D A V I D R I P P Y
DELPHI / OBJECT PASCAL | ALL LEVELS



Caution has its place, no doubt, but we cannot refuse our support to a serious venture which challenges the whole of the personality. If we oppose it, we are trying to suppress what is best in man — his daring and his aspirations.

— Carl Jung, Swiss psychiatrist

Welcome to “At Your Fingertips”, the bi-monthly tips column that will help guide you through the vast Delphi universe. It will be a challenging journey, no doubt, as all great adventures are, but there will be great rewards for the persistent, the patient, and the creative. So let’s throw caution to the wind, dig in, and learn something!

What’s the easiest way to add a status bar to my form?

Creating a status bar for a form — like the one shown in **Figure 1** — is a great way to keep the user informed while your application is running. For example, you might want to display a message in the status bar indicating the completion status of a lengthy process.

Delphi makes this easy by providing us with the standard Panel component. First, place a Panel component (from the Component Palette’s Standard page) on the form. Don’t worry about its size or alignment — setting the *Align* property to *alBottom* will handle this for us. Use the Object Inspector to set the following properties for *Panel1*: set *Align* to *alBottom*, *Alignment* to *taLeftJustify*, and *BevelInner* to *bvLowered*. Leave the *Caption* property alone for now.

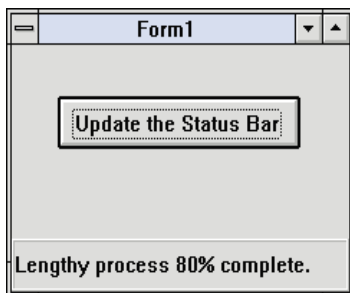


Figure 1: Delphi makes it easy to add a status bar to a form.

Your completed status bar should now look similar to the one shown in **Figure 1**. Note that by setting the *Align*

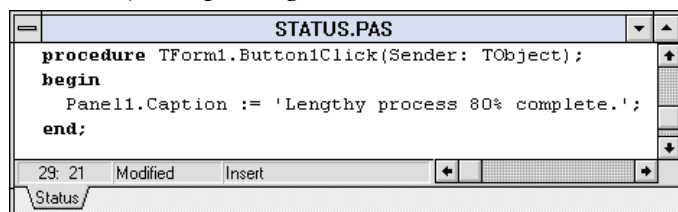


Figure 2: This *OnClick* event handler updates the status bar.

property to *alBottom* rather than sizing the Panel manually, the user can resize the form, and the Panel will automatically adjust itself to fit the width of the form. This is a convenient and elegant feature.

Using the Object Inspector, simply set the Panel’s *Caption* property to the message you want to appear in the status bar. The Object Pascal code shown in **Figure 2** demonstrates how this is accomplished. — *D.R.*

How can I allow a user to search for a value in a table?

A common feature in a database application is the ability to search a column for a particular value. **Figure 3** shows a form containing a DBGrid component with two columns: **Manufacturer** and **Title**. In this example, the user enters the name of a manufacturer to search for in the *Edit1* field and presses the **Locate** button.

If the value is found in the **Manufacturer** column, the DBGrid’s record marker moves to the record containing the value. If the

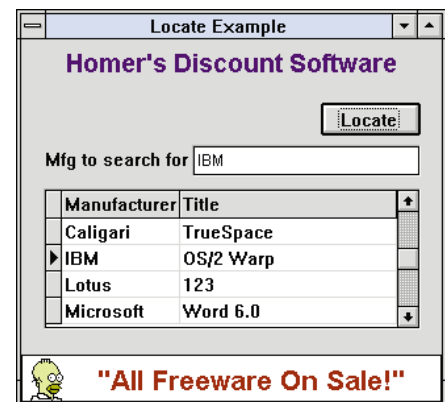
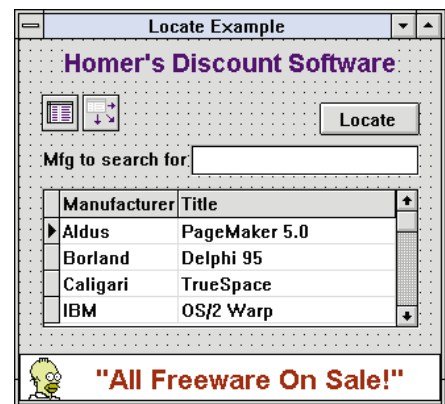


Figure 3 (Top): The Locate Example form in design mode. **Figure 4 (Bottom):** At runtime, the record marker advances to IBM after a successful locate.

value in the *Edit1* field is not found, a message box is displayed to notify the user. Figure 4 shows the form at runtime with “IBM” entered into the *Edit1* field. As you can see, the locate was successful because the record marker moved to the row containing “IBM” in its **Manufacturer** column.

Searching for a value on an indexed field requires very little coding in Delphi. All the necessary logic is found in the *OnClick* event handler of *Button1* (the Locate button), as shown in Figure 5. This code uses the *SetKey* and *GoToKey* methods to search the **Manufacturer** column for the value entered in the *Edit1* field.

```

SEARCH.PAS
procedure TLocate_Example.Button1Click(Sender: TObject);
begin
  Table1.SetKey; {Put Table1 into search mode}
  Table1.Fields[0].AsString := Edit1.Text;
  if not Table1.GoToKey then {Attempt search}
    MessageDlg(Edit1.Text+' could not be found.',
      mtError, [mbOK], 0);
end;
    
```

Figure 5: The *OnClick* event handler of *Button1* performs the locate.

Note that in this example, the locate operation is case-sensitive. Therefore, entering “ibm” would not have located the value “IBM” in the **Manufacturer** column. — D.R.

How can I allow a user to filter records in a DBGrid?

Like the locate example above, the ability to filter the data presented to a user is a fundamental element of any data-intensive application. Fortunately, Delphi makes filtering on indexed fields a painless process. The example form in Figure 6 shows a DBGrid (named *DBGrid1*) containing two fields — **ID** and **Name**. Also shown are two edit fields — *Edit1* and *Edit2* — and buttons for setting and canceling a range.

To set a filter, the user enters a starting value in the field labeled **Start** and an ending value in the field labeled **End**. When the **Set Range** button is pressed, *DBGrid1* will only display records that fall within the given ID range. Pressing the **Cancel Range** button will cause all the data to be displayed again.

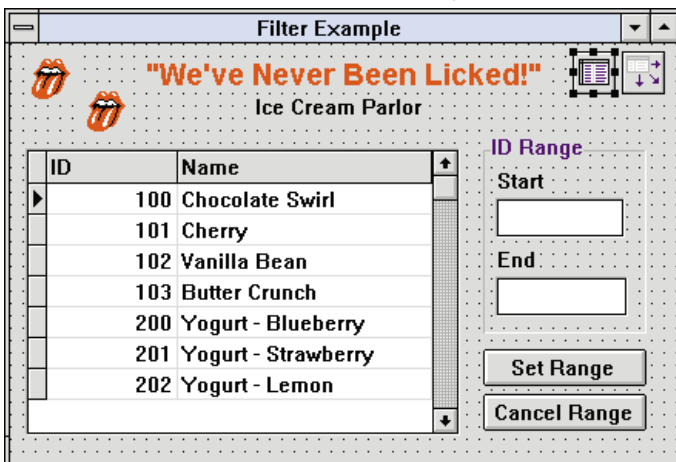


Figure 6: The Filter Example form in design mode.



```

RANGE.PAS
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.SetRangeStart;
  Table1.Fields[0].AsString := Edit1.Text;
  Table1.SetRangeEnd;
  Table1.Fields[0].AsString := Edit2.Text;
  Table1.ApplyRange;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Table1.CancelRange;
end;
    
```

Figure 7 (Top): Only the records within the range are displayed. Figure 8 (Bottom): Sample Object Pascal code for applying and canceling a range.

In Figure 7, a range has been set to display only the records with ID numbers falling between 200 and 299. When the **Set Range** button is pressed, only “Yogurt” entries will be shown in the DBGrid. To display every record, the user can simply press the **Cancel Range** button.

The Object Pascal code for applying a filter is located in the *OnClick* event handler of *Button1*, as shown in Figure 8. The primary methods for filtering data are *SetRangeStart*, *SetRangeEnd*, and *SetRange*. Together, these methods allow you to set the starting and ending values of the range of data you want to view.

To cancel a range programatically, simply call the *CancelRange* method as shown in the *OnClick* event handler of *Button2* (see Figure 8). — D.R. Δ

All files referenced in this article (projects, forms, tables, etc.) are available on the 1995 Delphi Informant Works CD located in `INFORM95\APRIL\DR9504`.

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





By *Alistair Ramsey*

Calling DLLs Dynamically

Developing Modular Applications Using the Windows API

When developing modular applications, module names can be stored in a file or database table. If any of these modules are dynamic link libraries (DLLs) however, what do you do when you can't hard-code the library's name?

This article discusses a technique to call procedures from DLLs using the Windows API within Delphi when the name of the DLL and procedure are not known at compile-time.

Why Use this Technique?

In Delphi, it's possible to compile a program as a DLL by specifying externally available procedures or functions using an Object Pascal **exports** clause. To use one of these procedures in an application, the calling program must make a **forward** declaration of the procedure's name with an **export** and **far** directive along with the name of the DLL.

This method is fine when the DLL and procedure names are known before compiling. However, the application may have a list of sub-module names that are compiled as DLLs, but are read into a menu when Delphi begins. The names of these DLLs may be stored in an .INI file or a database table along with the name of a procedure. Their names may change over time, and therefore aren't known before compile-time. In this example, we need a technique that reads DLL names into a string and calls the unnamed DLLs using the Windows API.

Some DLL Basics

A DLL is essentially a module of executable code. However, a DLL cannot be *run* in the sense that an .EXE file can. A DLL contains code that can be called by an .EXE at run-time.

Before Windows and DLLs became common tools for programmers, any code that was going to be re-used would typically be pre-compiled and stored in some sort of library. In Turbo Pascal these common pieces of code would be compiled into *unit files* (a .TPU or .TPW file). In a C environment these would be compiled as object code (.OBJ) files and stored in a library (.LIB) file. Whichever language was used, these could then be compiled into many different application (.EXE) files, and would only require recompiling if the unit or object file source code was changed.

The fact that the unit or object file must exist and be accessible to the compiler at compile-time is important. With a DLL, the code in the library is not linked into the .EXE at compile-time (as with the static linking described earlier), but is dynamically linked at run-time.

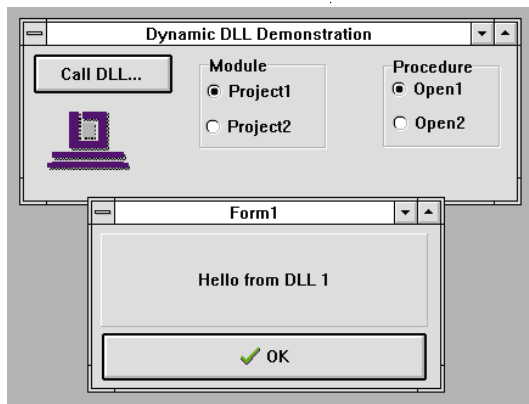


Figure 1: The demonstration program at run-time.

DLLs have several advantages. They allow you to update applications by updating the individual DLLs rather than recompiling the executable (.EXE) file. Since the executable doesn't have the code linked into it, it's smaller. Also, multiple executables can use the same DLL, which increases the reusability of the code.

In Delphi it's possible to define procedures in a DLL and then call those procedures at run-time. When the main program (the executable) is compiled, the DLL doesn't even need to exist. When a DLL is registered by an application, an internal user count is incremented by one. When the DLL is unloaded this count is decreased.

The Building Blocks

The code in this article uses the following Object Pascal language elements: the *TFarProc* type, the *GetMem* procedure, the *StrPCopy* function, the *FileExists* function, and the *MessageDlg* function.

It also uses the Windows API functions **LoadLibrary**, **GetProcAddress**, and **FreeLibrary**. With the exception of *TFarProc* — which is a pointer to a procedure — the language elements should be fairly familiar.

Here's a brief description of each Windows API function: The **LoadLibrary** function takes a single argument, the name of the DLL, as a *PChar*. If the DLL loads correctly, it returns a handle to the DLL. (In Delphi this is defined as a *THandle* and is just a two-byte word type.) Otherwise an error code is returned. If the returned value is less than 32, an error has occurred. Some error codes correspond to DOS file errors (e.g. 2 indicates that the DLL file was not found, 3 indicates an invalid path, etc.). Other return codes are specific to the Windows environment (e.g. 10 indicates that the Windows version was incorrect).

The **GetProcAddress** function returns a run-time address for the required procedure from the specified DLL. It takes two arguments, a DLL handle and procedure name, as a *PChar*.

The **FreeLibrary** function takes a valid *THandle* and unloads the library from Windows. This function decreases the user count of the DLL. When the count reaches zero, the DLL is finally unloaded.

The Technique

First you need a handle to the DLL. Delphi defines a type, *THandle*, that you can use for this purpose. Make a call to the Windows API **LoadLibrary** function which returns a *THandle*. You should check the value of the returned *THandle* — a value less than 32 indicates failure.

As with all Windows API functions, the name of the DLL must be passed as a null-terminated string (*PChar*) type — not the Pascal pre-defined **string** type. This Object Pascal statement shows the call:

```
DLL_Handle := LoadLibrary(DLLPchar);
```

Next, you need to get the address of the procedure to call by using the **GetProcAddress** function. This takes the *THandle* returned from **LoadLibrary** and a procedure name — again, a null-terminated string (*PChar*) — as arguments. If it fails, it

returns **Nil**. Otherwise, an address is returned.

Assign the returned address to a variable of procedural type (*Proc* in this example) and call the procedure:

```
TFarProc(@Proc) := GetProcAddress(DLL_Handle, Proc_Name);
{ The next line calls the procedure }
Proc;
```

The type casting using *TFarProc* sets the procedural type variable to the specified address. After the call has completed, the DLL should be unloaded using a call to **FreeLibrary**:

```
FreeLibrary(DLL_Handle);
```

This method is used by our company in an application that is implemented as an .EXE "shell". The main functionality of the system is implemented as a series of DLL modules that can be supplemented over time. The names of available modules are held in a table. At run-time this table is read and a drop-down menu containing the module names is constructed.

A generic procedure is assigned to the *OnClick* event handler for each menu option. The event handler then calls a generic procedure, *GenericModuleCaller*, to call the required procedure.

A Working Example

In the demonstration program accompanying this article, the form has a button and two radio button groups. Selecting a module and procedure and then clicking the button will call the DLL using the method described above. The valid combinations are Project1/Open1 and Project2/Open2. Other combinations will fail.

When clicking the button with a valid combination, the DLL module 1 is loaded (as shown in **Figure 1**) and displays a form. Note that the DLL procedure doesn't need to have a visible form, it could be a portion of code that executes a process.

The project's source code follows this article. **Listings Five** and **Six** contain the project and unit files (respectively) for the main application. (Note that the code expects the Project1.DLL and Project2.DLL files to be in the \Windows directory on drive C:.) **Listings Seven** and **Eight** contain the project and unit files (respectively) for the first DLL form. The source and unit files for the second DLL form are identical, except that all references to "1" instead refer to "2".

Conclusion

So there you have it. As we've seen, Delphi provides the flexibility to enable us to call DLLs dynamically at run-time. **▲**

The project referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM95\APRIL\AR9504.

Alistair Ramsey is a programmer/analyst at Dunstan Thomas Ltd. in the UK. He has been programming in Pascal for 10 years and is now a senior member of the Dunstan Thomas Delphi Team. Alistair will be representing Borland UK at the upcoming *International Developer's Challenge* in the US. You can contact him at aramsey@dtport-mhs.compuserve.com, or call 44 (1) 705-822254.

Begin Listing Five: DLLDEMO.DPR

```

program Dlldemo;

uses
  Forms,
  Dyn_dll in 'DYN_DLL.PAS' {Form1};

{$R *.RES}

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

```

End Listing Five**Begin Listing Six: DYN_DLL.PAS**

```

unit Dyn_dll;

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    GroupBox1: TGroupBox;
    RadioButton1: TRadioButton;
    RadioButton2: TRadioButton;
    GroupBox2: TGroupBox;
    RadioButton3: TRadioButton;
    RadioButton4: TRadioButton;
    Image1: TImage;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
    procedure GenericModuleCaller(ModuleName,
                                  ProcName : string);
  public
    { Public declarations }
  end;
var
  Form1: TForm1;

implementation

{$R *.DFM}

var
  ModuleName,
  ProcName : string;

procedure TForm1.GenericModuleCaller(
  ModuleName,ProcName : string);

```

```

var
  { Library and procedure names as
  null-terminated strings }
  DLLPchar, Proc_Name : PChar;
  { Procedural type declaration }
  Proc : procedure;
  { DLL handle }
  DLL_Handle : THandle;
begin
  DLL_Handle := 0;
  GetMem(DLLPchar, Length(ModuleName) + 1);
  StrPCopy(DLLPchar, ModuleName);
  if not FileExists(ModuleName) then
    MessageDlg('Fatal Error: No DLL',mtWarning,[mbOk],0)
  else
    begin
      GetMem(Proc_Name,Length(ProcName)+1);
      StrPCopy(Proc_Name, ProcName);
      if DLL_Handle = 0 then
        DLL_Handle := LoadLibrary(DLLPchar);
      if DLL_Handle >= 32 then
        begin
          TFarProc(@Proc) :=
            GetProcAddress(DLL_Handle, Proc_Name);
          if TFarProc(@Proc) = Nil then
            MessageDlg('Fatal Error'+#13+
              'in GetProcAddress()',
              mtWarning, [mbOk], 0)
          else
            Proc;
        end
      else
        MessageDlg('Fatal Error with DLL',
          mtWarning, [mbOk], 0);
    end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if RadioButton1.Checked then
    ModuleName :=
      'C:\WINDOWS\' + RadioButton1.Caption + '.DLL'
  else
    ModuleName :=
      'C:\WINDOWS\' + RadioButton2.Caption + '.DLL';
  if RadioButton3.Checked then
    ProcName := RadioButton3.Caption
  else
    ProcName := RadioButton4.Caption;

  GenericModuleCaller(ModuleName, ProcName);
end;
end.

```

End Listing Six

Begin Listing Seven: DLL1.DPR

Library Dll1;

usesForms,
Dllform1 in 'DLLFORM1.PAS' {Form1};

{\$R *.RES}

exports

Open1;

begin

end.

End Listing Seven**Begin Listing Eight: DLLFORM1.PAS**

unit Dllform1;

interface**uses**SysUtils, WinTypes, WinProcs, Messages,
Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, Buttons, ExtCtrls;**type**TForm1 = **class**(TForm)

Panel1: TPanel;

BitBtn1: TBitBtn;

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form1: TForm1;

procedure Open1; **export;****implementation**

{\$R *.DFM}

procedure Open1;**begin****try**

Application.CreateForm(TForm1, Form1);

Form1.ShowModal;

finally

Form1.Free;

end;**end;****end.****End Listing Eight**