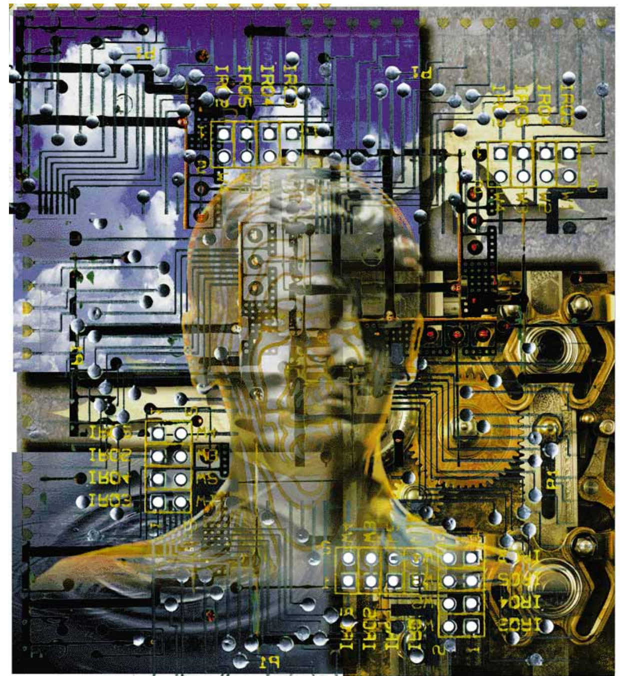
September 1995 - Volume 1, Number 5

Creating Components

Assembling Delphi's Building Blocks



Cover Art By: *Victor Kongkadee*

ON THE COVER

8 **The TSlideBar Component** — Robert Vivrette
Mr Vivrette gets our component issue off to a rollicking start with his fully-implemented SlideBar. Not only will you come away with a useful and attractive Windows control, you'll learn how it was built. Many techniques are presented along the way including: handling focus, handling keyboard and mouse input, painting a control's canvas, storing images in a resource file, and more.

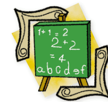
16 **A Dynamic Toolbar** — Gary Entsminger
Here's a floating Toolbar that your application's users can modify on-the-fly — adding and deleting buttons as the need arises. Mr Entsminger tackles several topics while discussing the component, including: unit variables, the *FormCreate* event, exception handling, *TNotifyEvent*, the *Sender* parameter, hints, glyphs, the API **ShellExecute** function, and more. Download it and put it to use!

41 **A Stopwatch Component** — Richard Holmes
More awesome components! And this one is not for the faint of heart. Mr Holmes uses embedded assembly language and in-line code to access the Windows Virtual Timer Device to create an extraordinarily precise software stopwatch. Even if you're not up to writing assembler, the Stopwatch components are useful as they come — one is even designed specifically for profiling programs.

FEATURES

23 **Informant Spotlight** — Thomas Miller
This month's "Spotlight" features a blow-by-blow comparison of Delphi and PowerBuilder from a recent convert. A long-time PB developer, Mr Miller puts both products through their paces before concluding "It's time to leave PowerBuilder behind."

31 **DBNavigator** — Cary Jensen, Ph.D.
In this month's DBNavigator, Mr Jensen concludes a two-part series on data validation. This time the focus is on validating input for data-aware *TField* components. Topics discussed include: creating input masks, the *Required* property, and the *OnValidate* and *BeforePost* events.



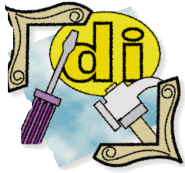
36 **OP Basics** — Charles Calvert
It's the second of a three-part series on Object Pascal strings. This month Mr Calvert shares some string-handling techniques (including how to strip trailing blanks), and introduces the Object Pascal *Length*, *GetDate*, *Delete*, *FillChar*, *Copy*, and *Move* functions.

DEPARTMENTS

2 **Delphi Tools**
5 **Newsline**

Delphi
T O O L S

New Products
and Solutions



Delphi Accounting Package Released

ColumbuSoft of Columbus, OH has released ***Accounting for Delphi***, a series of general accounting modules written entirely in Object Pascal (Delphi's native language) and sold with the source code. Purchasers receive a license that allows them to resell their compiled applications without additional royalties or fees.

Using the same fundamental approach as previous ColumbuSoft products, Accounting for Delphi is a batch-oriented, double-entry system. The system provides the means for Delphi developers to provide general accounting functionality that integrates with their custom or vertical market applications.

Accounting for Delphi includes general ledger, accounts payable, accounts receivable, order entry, inventory/purchasing, fixed assets, payroll, and job costing modules. Each can be used alone or with other modules without changes to the code.

The source code includes utility functions that can be used in other applications, including a reusable report printer with integrated on-screen preview. ReportSmith or other report generators are not required.

A free demonstration version is available. It has several source code files, and a developer's help file with technical documentation and file structures.

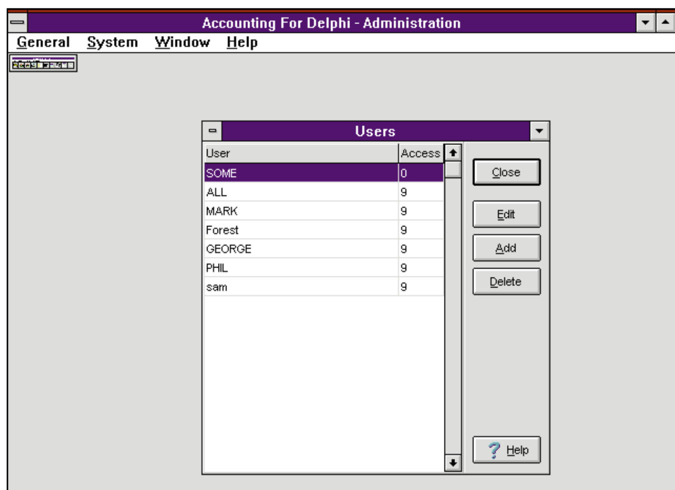
Price: US$500 for the first module, US$250 for any additional modules.

Contact: ColumbuSoft, 1525 Norma Road, Columbus, OH 43229

Phone: (800) 692-2150 or (614) 885-7789

Fax: (614) 885-2077

E-mail: 76702.556@compuserve.com



New Delphi Books

Developing Windows Applications Using Delphi
Paul Penrod
John Wiley & Sons
ISBN: 0-471-11017-5

Developing introduces object-oriented programming techniques, and then develops a Windows application step-by-step. It covers topics such as Object Pascal, GUI, debugging and testing applications, event handling, exception handling, error handling, using the run-time libraries, and compiling executables.
Price: US$29.95 (353 pages)
Phone: (212) 850-6630

Delphi: A Developer's Guide
Vince Kellen & Bill Todd
M&T Books
ISBN: 1-55851-455-4

Delphi guides readers through every aspect of Delphi, including: Delphi's database controls, VCL components, the Borland Database Engine, InterBase, SQL Server, and object-oriented programming and techniques.
Price: US$44.95
(820 pages, CD-ROM)
Phone: (800) 488-5233

New HeadConv 1.0: C DLL Header Converter Expert for Delphi

HeadConv 1.0, a C DLL Header Converter Expert for Delphi, is now available. Targeted at the serious Delphi developer, the HeadConv Expert assists the conversion of C DLL header files to Delphi import units, giving Delphi users access to third-party C DLLs.

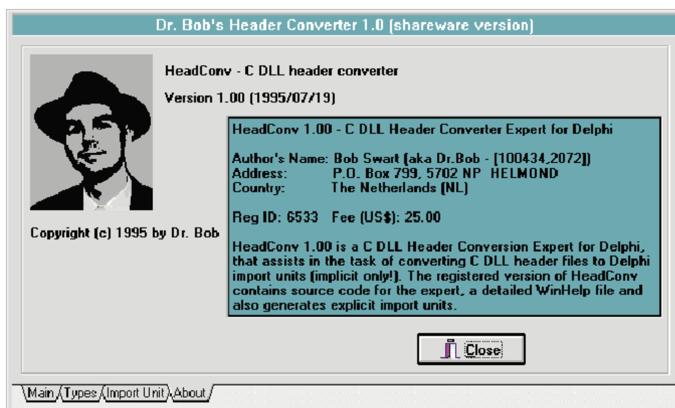
HeadConv has full support for functions and procedures, argument and return types (128 custom type conversions), and generates implicit Delphi import units. The expert is integrated in the Delphi IDE, simplifying the conversion process by opening automatically within the IDE. There is limited (non-complex) support for typedefs, structs, unions, enums, and conditional compilations. HeadConv cannot perform 100 percent of the conversion, but offers major assistance in converting C DLL header files.

The shareware version of HeadConv 1.0 is available on CompuServe from the Informant Forum (GO ICG-FORUM: Library 14), and the Delphi Forum (GO DELPHI: Library 22). The file name is HDCNV1.ZIP. All registered users will receive source code for the expert and stand-alone EXEs (source code for the parser is not provided), a detailed WinHelp file, and the capability of generating explicit import units. Registered users may also submit feature requests, and will receive regular updates by CompuServe mail when available.

Price: US$25, registrations can be made in the SWREG Forum (id #6533).

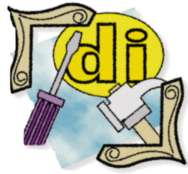
Contact: Bob Swart, P.O. Box 799, 5702 NP, Helmond, the Netherlands

E-mail: drbob@pi.net or 100434,2072



Delphi TOOLS

New Products and Solutions



ProtoView Releases Delphi Tools

ProtoView Development Corporation of Cranbury, NJ has released ***ProtoView Interface Component Set*** (PICS) version 1.5 for Delphi, a DLL/VBX controls toolkit. PICS is a collection of Windows user-interface objects for creating Delphi forms. The PICS Interface Component Set includes a hierarchical list box with multiple sort levels, volume dial, font selection, date, multi-directional button, numeric edit, percent bar, time, and icon/bitmap button controls.

PICS includes design-time visual setting through the Delphi Object Inspector, node searching functions, movable sub-trees, expandable hierarchical levels, and bitmap graphics for individual nodes. In addition, PICS has a data/calendar control that allows graphical calendar selection and formatting of various data values. Users can press a date button and display a calendar for the year specified.

PICS features a time control that allows graphical 24-hour time selection and formatting with various styles, a numeric edit control that allows picture masking and scientific notation formats for numeric input, and display styles including LED readout, counter, and normal.

Additional PICS for Delphi controls include a multi-directional arrow control, a font selection control, and a versatile Push Button control.

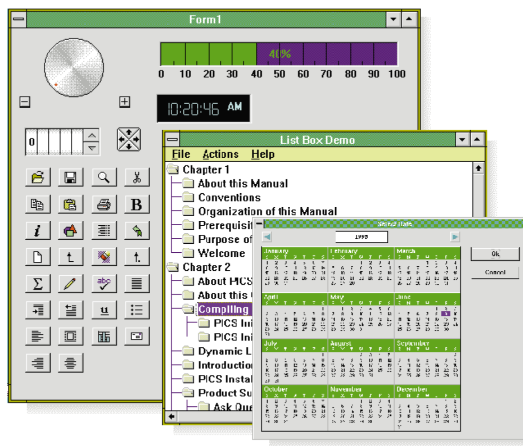
Applications created using the ProtoView Interface Component Set may be distributed royalty-free.

Price: PICS 1.5 for Delphi is US$149; source code, US$495.

Contact: ProtoView, 2540 Route 130, Cranbury, NJ 08512

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

Fax: (609) 655-5353



New Delphi Books

Mastering Delphi
Marco Cantu
Sybex
ISBN: 0-7821-1739-2

This book introduces programmers to all of Delphi's features and techniques, including the secrets of the integrated development environment, programming language, the custom components, and Windows programming in general. Topics covered include OLE, DDE, DLL, and graphics. The companion disk contains the source code to the examples from the book and an assortment of utilities.

Price: US$39.99 (1,500 pages, CD)
Phone: (510) 523-8233

New AccuSoft Image Format Library 5.0

AccuSoft Corporation of Westborough, MA has announced the release of ***AccuSoft Image Format Library 5.0***, a raster imaging DLL/VBX toolkit. It features 36 formats including TIFF, JPEG, Photo CD, G3, G4, PCX, GIF, DIB, WPG, BMP, TGA, PICT, EPS, and WMF.

This library also features advanced color reduction, 30 percent faster JPEG, automatic thumbnails, new file information function, file I/O replacement, status bar through call backs for functions, new scanning features, and group three and four raw data can be output directly.

The AccuSoft VBX custom control provides developers with a complete imaging toolkit, according to the company. There are no function calls since everything is implemented as properties and events. This version also has over 130 properties.

The VBX 32 has all features of the VBX 16 but is two to three times faster. Extra features include anti-aliased display, advanced scanner control, and sub-degree rotation.

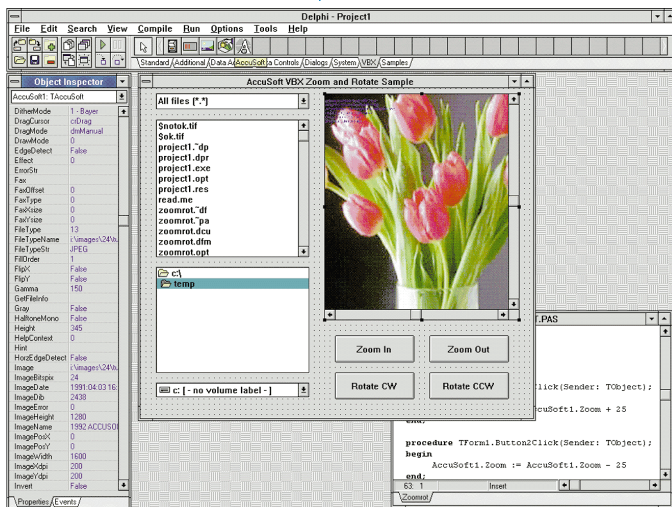
The VBX 32 Pro Gold has all the features of the standard 32-bit version but is two to four times faster. It also supports ImageAccel, large image, and black and white, Group IV and JPEG images. No special drivers are required.

Prices: Starts at US$495.

Contact: AccuSoft Corporation, Two Westborough Business Park, Westborough, MA 01581

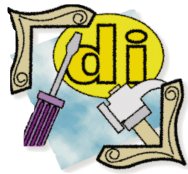
Phone: (508) 898-2770

Fax: (508) 898-9662



Delphi TOOLS

New Products and Solutions



Delphi Training

The 4GL Consulting Group Ltd., of San Mateo, CA is offering beginning and advanced courses in Delphi. Taught by a developer, these classes cover Object Pascal, object-oriented programming, client/server database applications and Delphi components.

4GL will be offering classes throughout this month. For more information and registration materials, contact Ian Hart at (415) 348-4848 or fax (415) 349-4683.

The course instructor may be contacted via CompuServe at 72143,467.

SilverWare Windows Communications Tool Kit Ships

SilverWare Inc., of Dallas, TX has announced the release of ***The SilverWare Windows Communications Tool Kit*** version 5.01, a multi-language communication library for Microsoft Windows. The library is a Windows dynamic link library (DLL) accessible from any Windows language or program that can make FAR PASCAL function calls and pass 16- and 32-bit parameters by reference and value. It offers complete support for the following languages: Object Pascal, ObjectPAL, C/C++, CA-Visual Objects, Clarion for Windows, Clip4Win, Visual dBASE, FiveWin, Power Builder, Turbo Pascal for Windows, and Visual Basic.

The SilverWare Windows Communications Tool Kit has built-in, high-level dialog box functions to decrease development time. These dialog box functions enable developers to prompt users for COM port selection, UART settings, auto dialer, modem defaults, file transfer options, and more, with a single function call.

It also features: low/high level support, beyond COM1/COM2, multi COM I/O boards, hardware/software flow control, file transfers, Smartmodem support, auto dialer, high-level remote input, immediate, transmit, asynchronous timer functions, comprehensive return code system, documentation, and examples for every language.

Price: US$299 (no royalties, includes free technical support and a 30-day money back guarantee).

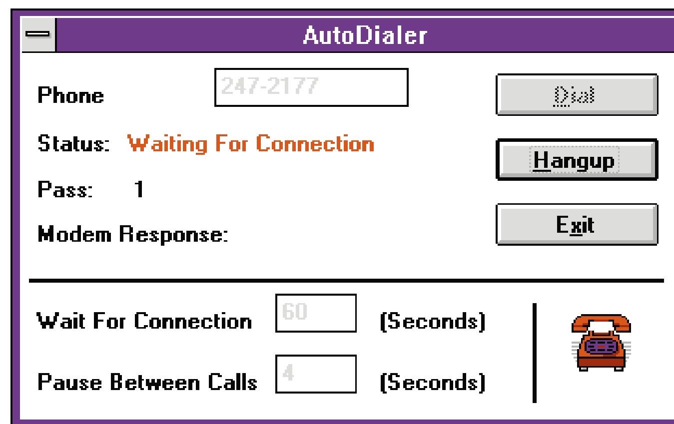
Contact: SilverWare Inc., 3010 LBJ Freeway, #740, Dallas, TX 75234

Phone: (214) 247-0131

Fax: (214) 406-9999

BBS: (214) 247-2177

WWW Internet Home Page URL: http://rampages.onramp.net/~silver

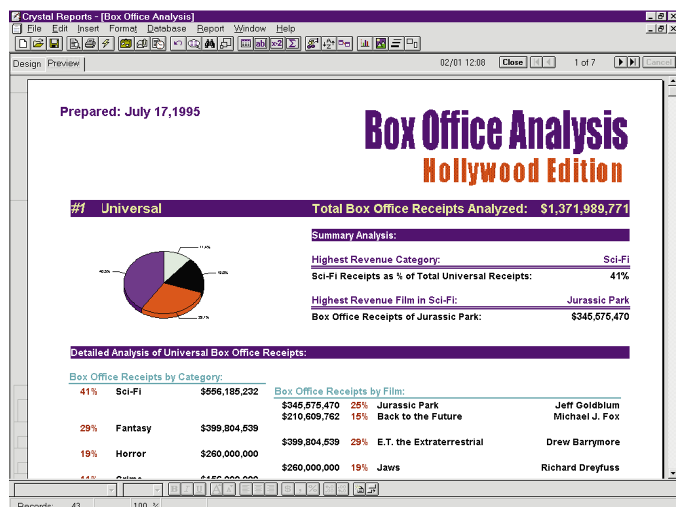


Crystal Announces 32-bit Windows Versions

Crystal of Vancouver, BC has announced a new 32-bit version of ***Crystal Reports*** for Windows 95 and Windows NT. It will be available in both 16-bit and 32-bit versions, and will run on all Windows platforms including Windows 3.1.

Crystal Reports' new features include enhanced graphing with customizable graph types, import and export capabilities for Lotus Notes, support for Access 2.0 OLE picture fields, and support for the Microsoft Access Engine 2.5. It can export to Excel 5.0, save report options with a report (simplifying report distribution), drill down on graphs, and supports the new Borland Database Engine (IDAPI) and Paradox 5.0 for Windows.

According to the company, developers can deliver reporting solutions for their 32-bit environments using the 32-bit Report Engine and OCX. In addition, this version offers the speed and stability of a 32-bit operating system, a tabbed interface, and long-file-name support.



Price: Crystal Reports Standard 4.5, US$195; Crystal Reports Professional 4.5, US$395; Upgrades to Crystal Reports Professional 4.5 from previous versions of Crystal Reports Standard or Professional plus OEM versions of Crystal Reports, US$199.

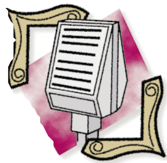
Contact: Crystal, a Seagate Software Company, 1095 West Pender Street, 4th Floor, Vancouver, BC, Canada V6E 2M6

Phone: (604) 681-3435, or (800) 877-2340

Fax: (604) 681-2934

News LINE

September 1995



Virtual User Group Meets

The Informant CompuServe Forum will host a Delphi Virtual User Group meeting Wednesday, Sept. 13, 1995 at 6 p.m. (PST). During this hour long session, we'll discuss the Borland Conference and the latest trends in the Delphi Community. To access the Informant CompuServe Forum type "GO ICGFORUM" at any CompuServe "GO" prompt.

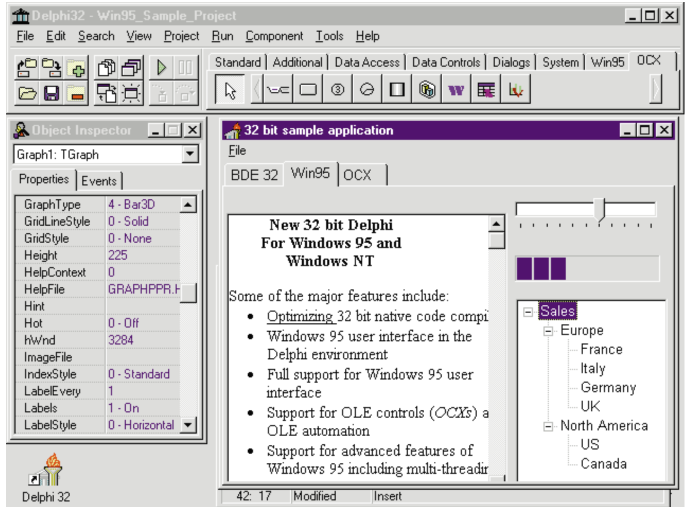
Delphi 32: Supports OCXes, OLE Automation, and More

Scotts Valley, CA — Borland International is scheduled to unveil product details at their San Diego conference regarding their new 32-bit versions of Delphi and Delphi Client/Server. Delphi 32's enhancements include an optimized 32-bit native code compiler, a Windows 95 user-interface in the Delphi IDE, a high performance 32-bit Borland Database Engine and SQL links, additional example programs and demonstrations, and improved documentation (including a printed language reference manual).

Delphi 32 will support OLE controls (OCXes), OLE automation, as well as provide full support for Windows NT, and Windows 95.

With OLE automation, developers will be able to create or control scriptable applications with a variety of other applications including Paradox for Windows, Visual dBASE, Microsoft Office, WordPerfect Office, and others.

The current versions of Delphi and Delphi Client/Server are compatible with Windows 3.1, Windows for Workgroups 3.11, Windows 95 beta releases, Windows NT 3.5 and 3.51, and OS/2.



Developers currently using Delphi can create applications that look similar to a Windows 95 application by using components such as notebook tabs, outline controls, spin controls, and tool help. Then, when 32-bit versions of Delphi are available, developers can recompile their existing applications for true 32-bit performance on Windows 95 or Windows NT without rewriting code. Developers using low-level code that is dependent on Windows 16-bit segmented architecture, or Windows 3.1 features not supported by Windows 95, will need to alter their code as necessary.

With Delphi 32, developers will not be able to create 16-bit applications. However, applications created in Delphi 32 that do not use 32-bit specific features can be recompiled with the 16-bit version of Delphi and then run on Windows 3.1.

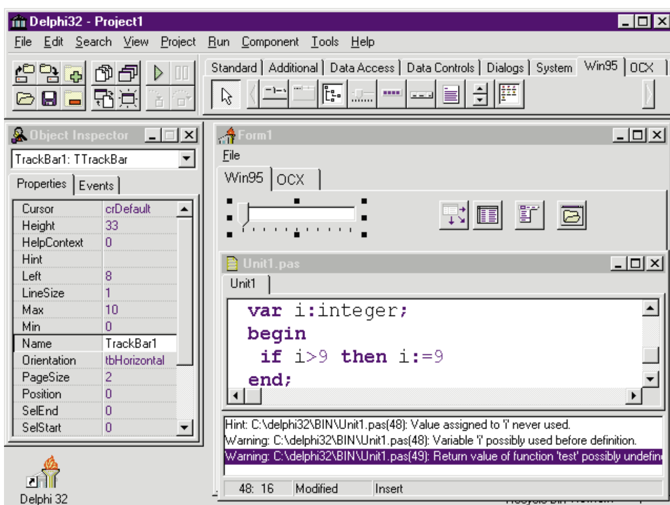
Delphi 32 is expected to meet all the Windows 95 logo requirements. It will include additional components to support new Windows 95 specific features such as rich text editing, Windows 95 style notebook tabs, progress bars, already said, etc. These components can then be added to the Component Palette.

In addition, Delphi 32 supports long file names, new dialog boxes, styles, and immediate access to Windows 95 API including facilities such as multi-threading, plug and play, MAPI, and more. In addition, Delphi 32 will make it easy to create applications that meet Windows 95 logo requirements.

Currently, the 32-bit versions of Delphi and Delphi Client/Server are in beta testing, and are expected to be available about 90 days after the commercial release of Windows 95.

Borland is planning to offer special upgrade pricing for registered Delphi users. Customers who purchased Delphi Client/Server with maintenance will receive the new 32-bit version free of charge.

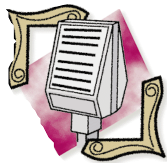
Borland expects the 16- and 32-bit operating systems to co-exist for the next 18 to 24 months and will continue to sell and support the 16-bit version of Delphi after Delphi 32 is released.



News

LINE

September 1995



Borland Makes Profit in First Quarter

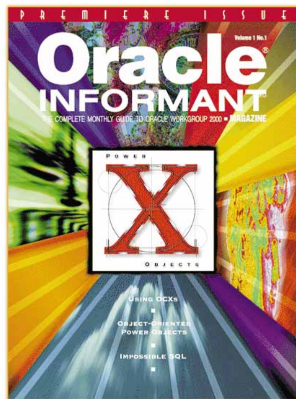
Scotts Valley, CA — Borland International Inc. has announced revenues for its first quarter ending June 30, 1995 of US$53.8 million. Revenues from the first fiscal quarter of the prior year were US$69.1 million, however it included US$24.5 million from the sale of Paradox licenses to Novell. Excluding the non-recurring revenue from the sale of the Paradox licenses, first quarter fiscal 1996 revenues increased 20 percent from the same quarter of the prior year.

The net income for the June 30, 1995 quarter was US$2.8 million or US$.10 per share, compared with net income of US$61.4 million or US$1.88 per share in the first quarter a year ago. Included in the prior year's results is a US$99.9 million non-operating gain on the sale of Borland's Quattro Pro spreadsheet product line to Novell, Inc., the Paradox license revenue of US$24.5 million, and a one-time charge for purchased technology of US$16.2 million related to the company's acquisition of ReportSmith. Excluding these non-recurring transactions, Borland would have reported an operating loss of US$35 million, on revenues of US$44.6 million, in the quarter ending June 30, 1994.

Total operating expenses for the quarter were US$43 million, a 37 percent decrease from US$68.5 million for the same quarter of the previous year, exclusive of the write-off of purchased technology of US$16.2 million. The lower expenses in the quarter ended June 30, 1995 reflect the restructuring efforts started this year.

According to Gary Wetsel, president of Borland, the quarter's results reflect Borland's efforts to reduce costs and the continued success of Delphi.

ICG to Publish *Oracle Informant*



Elk Grove, CA — Informant Communications Group, Inc. (ICG) has announced it will be publishing its third product-specific technical magazine, *Oracle® Informant*. *Oracle Informant* will contain in-depth technical articles on client/server development with Oracle® Workgroup/2000 tools. The premiere issue of *Oracle Informant* is scheduled for Winter 1995.

Oracle Informant will place heavy emphasis on database application development using Oracle Workgroup/2000 products. Featuring regularly-appearing articles covering technical issues regarding Oracle7 Workgroup Server, Personal Oracle7, Oracle Power Objects, Oracle Objects for OLE, and Oracle Mobile Agents, Oracle developers will have an independent, comprehensive source of in-depth technical information.

Oracle Informant will also feature news from the Oracle community, third-party product information, product and book reviews, and Oracle user group information each month.

Magazine-only subscriptions to *Oracle Informant* are available to US subscribers for US$49.95 a year (12 issues). An optional subscription to the *Oracle Informant* Companion Disk is also available. The *Oracle Informant* Companion Disk contains all the source code and support files for each article appearing in *Oracle Informant*. One-year magazine and Companion Disk subscriptions are available for US$119.95 a year (12 issues and disks). *Oracle Informant* will also be available at major newsstand outlets.

Authors and developers interested in contributing articles can obtain a writer's style guide and editorial calendar by contacting ICG Associate Editor Carol Boosembark at (916) 686-6610, ext. 16 or via e-mail at 75702.1274@compuserve.com.

Vendors interested in advertising can obtain an *Oracle Informant* Media Kit by contacting ICG Advertising Director Lynn Beaudoin at (916) 686-6610, ext. 17 or via e-mail at 74764.1205@compuserve.com.

Borland Developers Conference, London 1996 Announced

London, England — Borland International, Desktop Associates Limited, and Dunstan Thomas Limited announced the third Borland Developers Conference, London 1996 will be held in London, England on April 28 through 30 1996 at the Royal Lancaster Hotel, Lancaster Gate, London.

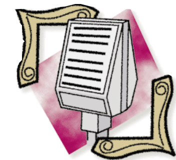
The event will feature over 40 sessions covering all Borland mainstream products: Delphi, Paradox, and dBASE, C++, client/server solutions, InterBase, and many other topics including case studies. The sessions will cover programming, solutions, tools and techniques, and methodologies.

Pricing for the Borland Developers Conference, London 1996 is £495 (US$742). Those attending only one day pay £275 (US$412). For a complete brochure call the conference office at +44 0181 788 0057.

News

L I N E

September 1995



Software World and Client/Server Developers to Meet in San Jose

Andover, MA — DCI's Software World and Client/Server Developers Conference and Exposition is heading to the San Jose Convention Center in San Jose, CA on Oct. 10-12, 1995. The event will feature over 250 exhibitors, several management and technical tracks, keynote presentations, and special events.

The management tracks include Managing Complex Software Projects, The Future of Database Management, Deploying Complex Applications, Professional Rapid Application Development, and Practical Business Process Re-Engineering Strategies. For technical growth, the show will feature tracks on Distributed Objects, Next Generation of Component-based Development, Windows 95 and OS/2, Visual Programming — Tips and Techniques, Cross Platform Solutions, and Leveraging Lotus Notes.

Attendees will hear from several keynote speakers including Ed Yourdon, consultant and methodologist; Phillip White, President, CEO and Chairman of the Board, Informix Software; George Schussel, Chairman and CEO, DCI; Rodney Knowles III, Director of Special Projects, Atlanta Committee for the Olympic Games; and Steve Mills, General Manager of Software Solutions Division, IBM.

For more information call: (508) 470-3880, fax: (508) 470-0526, or e-mail: DCIconf1@aol.com.

Borland Ships Visual dBASE 5.5 and Compiler

Scotts Valley, CA — Borland International Inc. has released its Visual dBASE 5.5 database and Visual dBASE Compiler for the Microsoft Windows 3.1 and Windows 95 operating systems. The company also announced a new product: Visual dBASE Client/Server.

Visual dBASE 5.5 is the only second-generation, object-oriented Xbase database. It features new productivity tools for users and developers, performance enhancements, and client/server capabilities. The separate Visual dBASE Compiler allows developers to create and deploy stand-alone .EXE applications royalty-free to users.

Using the new client/server version of Visual dBASE, developers can create front-ends to existing Oracle, Sybase, Microsoft SQL Server, Borland InterBase, and Informix database servers. The product includes Visual dBASE, the Visual dBASE Compiler, native Borland SQL Links, a single-user Local InterBase Server and Borland's new Data Pump Expert.

The estimated street prices for Visual dBASE 5.5 and the Visual dBASE Compiler are US$349.95 each. Special upgrade prices are available for current dBASE users and competitive products. The estimated street price for Visual dBASE Client/Server is US$695. For more information or to place orders, call Borland at (800) 233-2444.

10th Annual PC Expo in Chicago

Fort Lee, NJ — The Blenheim Group has announced that the Tenth Annual PC Expo will be held October 3 through 5 at the McCormick Place East in Chicago, IL. The event will host over 200 exhibitors, featuring applications supporting Windows 95, OS/2, and Macintosh operating systems. Blenheim expects to attract over 30,000 attendees, including corporate volume buyers from the business and government sector. Volume resellers in the audience will include software developers, dealers, VARs, and consultants.

Keynote speakers for PC Expo feature: Pallab Chatterjee, president of Personal Productivity Products business, Texas Instruments, Inc.; James P. McNeil, executive vice president, corporate development, Cheyenne Software; and Robert E. Lawton, vice president of marketing, The Wollongong Group, Inc.

PC Expo will also have an "Internet Theater" enabling attendees to learn about the resources available on the Internet, as well as gain hands-on experience. In addition, there will be several areas for specific products and services. These include an Internet Pavilion, Networking Pavilion, Multimedia Pavilion, and Technology Recruitment Center.

There will be a variety of seminars, half-day workshops, and tutorials consisting of in-depth user case studies available throughout the event, such as networking, emerging technologies, technology management, Windows, client/server, Internet, and Groupware.

For more information call The Blenheim Group at (800) 829-3976 or visit their home page at: http://WWW.shownet.com.

Software Development '95 East
Over 200 venders are expected to attend this year's Software Development East in Washington, D.C., Oct. 2-6. The event features more than 150 lectures, workshops, and tutorials, covering topics such as C++, Windows 95/NT development, object-oriented programming, database programming and design, and more. For more information call Miller Freeman Inc. at (800) 441-8826, fax (415) 905-2222, e-mail sd95east@mfi.com, or visit their home page at: http://www.mfi.com/sdconfs/

By Robert Vivrette

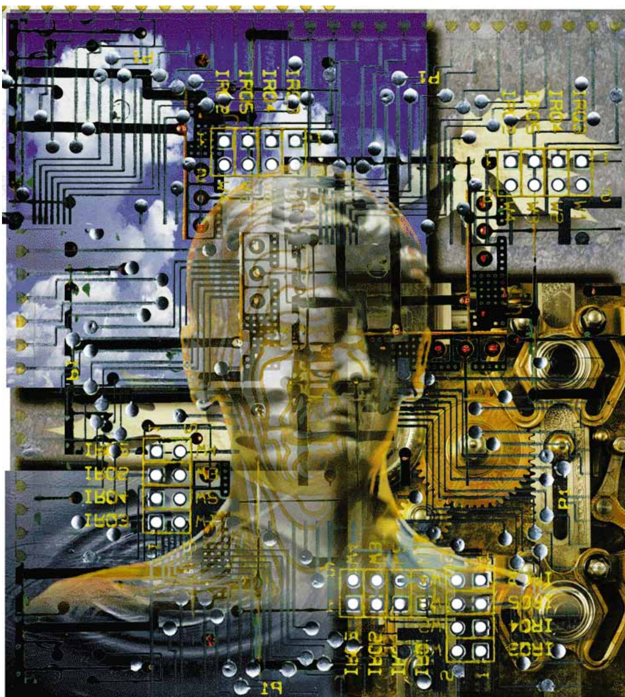
The TSlideBar Component

Component Design Techniques for the Initiated

Since its release in February, Delphi has created quite a stir in the computer industry. One of the more prominent changes I have seen is in the arena of component design. I'm sure you've noticed the proliferation of component design articles in various programming journals.

The component design phenomenon is one of the exciting features of Delphi. I don't think you can find a better platform to write incredibly powerful and robust components. Each component that you write in Delphi can be just as capable and efficient as those that are pre-installed. And once you've developed a new component, it is seamlessly integrated into the environment. (How many custom control articles did you see for Visual Basic when it first came out? Not many I bet.)

This article will discuss some of the more advanced elements of component design in Delphi. The featured example is a slide bar component, but the focus will be on how you can implement these more advanced capabilities into your custom components.



The *TSlideBar* component (see Figure 1) is functionally similar to a standard scroll bar, but it is more attractive and consumes less screen real estate. While developing this component, we'll cover these topics:

- Handling the focus
- Receiving input from the keyboard
- Mouse capture and mouse events
- Painting on a control's canvas
- Storing images in a resource file
- Transparent areas and masking
- Working with a *TStringList*

The *TSlideBar* Component

Before going into the details, let's look at what we want from this component. At its most elementary level, a slide bar component is really just a way of enabling the user to select a number. So, why not just use an edit box and have the user enter a number?

There are several reasons for not using an edit box. First, that approach requires the user to use the keyboard, and it would be convenient if the user could also select a number with the mouse. Second, the number may be irrelevant. For example, the number may

be an index for a list of strings. Suppose you are using the control to indicate a movie rating. Typical responses may be G, PG, PG-13, and R. The slide bar could be used to display these four choices without requiring the user to enter anything. In addition, the user does not have to be concerned that the program has assigned 0 to G, 1 to PG, 2 to PG-13, and 3 to R.

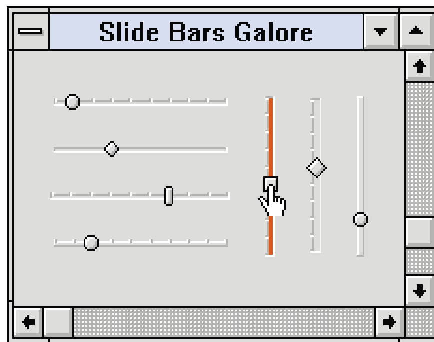


Figure 1: A demonstration of the SlideBar component. Note that some have different thumbs, width trenches, and orientations (vertical vs. horizontal). In addition, some are raised while others are lowered, some have tick marks, and one of the controls currently has the focus (as indicated by the red highlight). Finally, the control features an optional "pointing hand" custom cursor.

So we want the basic ability to obtain a number from the component (i.e. its "position"). But let's take it a bit further. The thumb that we are sliding back and forth may not fit the design scheme of the application as a whole. So, we'll allow the developer to have different thumbs. We can also allow the slot that the thumb moves along to be raised or lowered, and of variable width.

Naturally, we want to have the mouse move the thumb. Therefore the component will have to be "mouse aware", responding to mouse clicks and mouse movement. In addition, the user may want to use the keyboard, so support should be implemented for the component to respond to keyboard events. Since the slide bar can be selected with the keyboard, it should indicate when it has focus. We'll also allow the slide bar to be oriented vertically or horizontally.

There are a few other things I added to the component to spice it up. The *TSlideBar* can display small tick marks along the slot to "tell" the user there are fixed positions that the thumb will jump to. A custom cursor was also added, so that when the mouse passes over the control, it presents a pointing hand — a more appropriate mouse pointer for a slide bar control.

Finally, I decided it would be handy if the control could hold a list of strings for its various positions. That way, the control itself could report which string has been selected, rather than getting the control's position and running through a big **case** statement to determine the string.

Focus! Focus!

In a Windows application, at any time, the user *must* be able to clearly see which control on a form currently has focus. Most controls indicate this by altering their appearance. A button for example, will have its text surrounded by a dotted line. An edit box will show a blinking insertion point, or highlight some of the text inside. Scroll bars (when they are allowed to receive focus) often have a blinking thumb.

The *TSlideBar* component should be no different. Fortunately, managing which object has focus in an application is primarily the operating system's job. All that needs to be done from the component's perspective is to recognize when the component has gained or lost focus, and determine how to indicate this to the user.

Detecting a focus change is just a matter of including handlers that will trap the appropriate Windows messages. In this case, we are looking for the `WM_SETFOCUS` and `WM_KILLFOCUS` messages. In the **private** section of your component's class declaration, you would include these lines of code:

```
private
  procedure WMSetFocus(var Message: TWMSetFocus);
    message WM_SETFOCUS;
  procedure WMKillFocus(var Message: TWMKillFocus);
    message WM_KILLFOCUS;
```

When the user tabs through the controls on the form, or clicks on a particular control, Windows first sends a `WM_KILLFOCUS` message to the control that had focus. Then, a `WM_SETFOCUS` message is sent to the control that is gaining focus. From the component's side, we simply write handlers for these messages. In the case of *TSlideBar*, I just want the component to repaint itself when it gains or loses focus, as follows:

```
procedure TSlideBar.WMSetFocus(var Message: TWMSetFocus);
begin
  Refresh;
end;
```

The *WMKillFocus* procedure would also be the same. Now, when the component is being repainted, it's a simple matter of looking at the *Focused* property. If the component currently has focus, this will return *True*. Otherwise, it will return *False*.

Based on the result, the center portion of the slot is then colored appropriately. To add a bit more pizzazz to the control, the *FocusColor* property was added to define the color that will be used when the component has focus. Inside the routine for drawing the slot is the code for managing the focus:

```
{ Now color a filled rectangle in the center
  if the control has focus }
if Focused then
  Brush.Color := FocusColor
else
  Brush.Color := clSilver;

Pen.Style := psClear;
{ Draw the focus highlight }
Rectangle(X1+1, Y1+1, X2+1, Y2+1);
```

There are different ways of implementing focus features. For example, in many cases the *WMSetFocus* procedure could draw the focus highlight. However, I chose not to do this with *TSlideBar*, because the thumb sits over the focus highlight and I would have to redraw it as well. (Once I start doing that, then I might as well redraw the whole thing.)

A Key Feature

There is nothing more frustrating than being forced to use a Windows control with only the mouse or keyboard. A well-designed control must work with either device and should not require the user to constantly switch between the two input devices.

Again, adding this feature doesn't require expending too many brain cells. First, we must override the *KeyDown* event that we are inheriting from up the object tree. In the **private** section of the class definition, you should add a line such as:

```
private
  procedure KeyDown(var Key: Word;
                    Shift: TShiftState); override;
```

At a minimum, we want to enable the user to move the thumb using the arrow keys. I also added support for Home and End to move the control to its minimum and maximum values respectively. In addition, I decided to add support for Page Up and PageDown. Since the arrow keys only move the slide bar's position by one, you would be in trouble if the range of the slide bar was 1000. Therefore, Page Up and PageDown were enabled to move the slide bar's position by 10 percent of its total length. To enable these functions, the *KeyDown* procedure was used (see Figure 2).

The *Position* property is (of course) the slide bar's current position. The *Max* and *Min* properties are the maximum and minimum values that the slide bar can reach.

```
procedure TSlideBar.KeyDown(var Key: Word;
                            Shift: TShiftState);
var
  b : Integer;
begin
  b := MaxInt(1,(Max-Min) div 10);
  case Key of
    VK_PRIOR : if (Position-b) > Min then
                 Position := Position - b
               else
                 Position := Min;
    VK_NEXT  : if (Position+b) < Max then
                 Position := Position + b
               else
                 Position := Max;
    VK_END   : if IsVert then
                 Position := Min
               else
                 Position := Max;
    VK_HOME  : if IsVert then
                 Position := Max
               else
                 Position := Min;
    VK_LEFT  : if Position  > Min then
                 Position := Position - 1;
    VK_UP    : if Position  < Max then
                 Position := Position + 1;
    VK_RIGHT : if Position  < Max then
                 Position := Position + 1;
    VK_DOWN  : if Position  > Min then
                 Position := Position - 1;
  end;
end;
```

Figure 2: The *KeyDown* procedure.

Note: When constructing **case** statements, try to list the items in ascending order. Here for example, VK_PRIOR is a constant with the value of 21, VK_NEXT has the value 22, and so on to VK_DOWN that has a value of 28. When a **case** statement is sorted in ascending order, the compiler can optimize the code to execute faster. If they are not in sequential ascending order, the compiler must use a less efficient method to generate the needed code. All it takes is one line out of sequence to foil optimization, so make sure you pay attention.

When you add the code shown in Figure 2 and run the program — it doesn't work! Rather than moving the thumb's position, the arrow keys are moving the focus between controls. To get your control to pay attention to the arrow keys, you must trap a Windows message called WM_GETDLGCODE and tell it that you will handle the arrow keys. Like our WM_SETFOCUS and WM_KILLFOCUS handlers above, we must add the following line to the **private** section:

```
private
  procedure WMGetDlgCode(var Message: TWMGetDlgCode);
    message WM_GETDLGCODE;
```

The procedure's code is as simple as it gets:

```
procedure TSlideBar.WMGetDlgCode(var Message:
                                 TWMGetDlgCode);
begin
 Message.Result := DLGC_WANTARROWS;
end;
```

With the *TSlideBar* component, we are only interested in trapping the arrow keys. However, there may be instances in other components where you will want to catch other keys.

The *WMGetDlgCode* handler can also use any combination of other constants including, but not limited to:

- DLGC_WANTALLKEYS
- DLGC_WANTCHARS
- DLGC_WANTTAB
- DLGC_WANTMESSAGE

You can read more about this topic by searching on WM_GETDLGCODE in the Windows API on-line help file (WINAPI.HLP).

Build a Better Mouse Trap

Normally, a Windows control or form only receives mouse messages when the mouse cursor is over its client area (i.e. the area bounding the dimensions of the control or the inside area of a form). Sometimes however, it becomes necessary for a control to receive mouse messages even when the mouse is outside this area. This is known as "capturing" the mouse.

Fortunately with the Delphi component designer, mouse capture is (for the most part) handled for you. For example, with the *TSlideBar* component, we don't want the user to have to keep the mouse within the slide bar's boundaries. If mouse capture was not on, the mouse messages would immediately stop after the mouse leaves the control.

By default, however, mouse capture has been turned on for components that descend from the *TControl* object. As a result, when you select the slide bar's thumb (by holding down the left mouse button), you can drag the mouse all over the screen and the thumb will move appropriately. As long as you keep the mouse button down, that control has "captured" the mouse. When the mouse button is released, it immediately releases the capture.

You can modify whether the control will capture the mouse by setting and/or clearing the *csCaptureMouse* flag from the component's *ControlStyle* property. If you are going to change a component's *ControlStyle* you should do it in the *Create* method. For example, to ensure that a control does not capture the mouse, the code would look similar to this:

```
ControlStyle := ControlStyle - [csCaptureMouse];
```

Also keep in mind that when a component has captured the mouse and you move outside its client area, the MouseMove messages being generated may be reporting negative X and/or Y coordinates (since they are relative to the component's coordinate system). Make sure your program knows how to handle this.

Since the *TControl* object has the *csMouseCapture* flag on by default, we don't need to include any code in our component to manage the mouse capture. It will just work. However, this is an important concept to understand for developing well-behaved components.

Besides providing for the mouse capture, the *TSlideBar* must also respond to various mouse events (to enable the mouse to move the thumb). Here are the basic activities we want to trap:

- If the user clicks on either side of the thumb, the thumb should shift a single position in the appropriate direction.
- If the user selects the thumb, he or she should be able to slide it back and forth. The thumb should slide smoothly while being dragged, and not hop between positions. (This would ruin the visual effect that you are holding onto the thumb.) When the left mouse button is released, the thumb should "click" to the closest position (indicated by the tick marks).

To respond to these activities, we must detect the *MouseDown*, *MouseUp*, and *MouseMove* events. The *TSlideBar* component overrides these procedures as follows:

```
protected
  procedure MouseUp(Button: TMouseButton;
    Shift: TShiftState; X,Y: Integer); override;
  procedure MouseDown(Button: TMouseButton;
    Shift: TShiftState; X,Y: Integer); override;
  procedure MouseMove(Shift: TShiftState;
    X,Y: Integer); override;
```

The *MouseUp* and *MouseMove* methods have some pretty boring calculations in them and would not add much to this discussion. To summarize their behavior, however, *MouseUp* determines if the thumb was being dragged, and if so, finds the closest position to "click" the thumb to. If it was not being dragged, it determines if the *MouseUp* event occurred to the left or right of the thumb. If the click occurred on the left, the thumb is moved one position to the left. If the click occurred on the right, the thumb is moved one position to the right.

MouseMove only moves the thumb if the mouse button is down, and it is dragging the thumb. The *MouseDown* event, however, is short and interesting:

```
procedure TSlideBar.MouseDown(Button: TMouseButton;
  Shift: TShiftState; X,Y: Integer);
begin
  SetFocus;
  Dragging := PtInRect(ThumbRect,Point(X,Y));
  if IsVert then
    DragVal := Y
  else
    DragVal := X;
end;
```

First, the *TSlideBar* component tells Windows that it now has the focus. This makes sense since the user has just clicked the mouse button on the control. Next, a Boolean value called *Dragging* is set to *True* or *False* provided the *MouseDown* event occurred within the rectangle that bounds the thumb's current position. If so, the *MouseMove* events will enable the thumb to move with the mouse.

The third line simply saves the X or Y position of the mouse click depending on whether the component is oriented in a horizontal or vertical position. If the slide bar is a vertical one, we are interested in where the mouse is on the Y scale so the thumb can be moved appropriately. If the slide bar is horizontal, we would then be interested in the X coordinate.

Painting the Town

One of the basic features of a custom component is its appearance. Sometimes you may inherit a component's appearance from an ancestor, but often you must provide your own "look" to custom components. The control's appearance is *not* an aspect of component design that should be left for last.

Fortunately for us, the object we are likely to descend from (*TCustomControl*) has most of what we need for drawing our own component. The key element *TCustomControl* provides is the *Canvas* property. With a canvas, we can easily draw until we are content with the component's appearance.

First, you must override the inherited *Paint* method. This *Paint* method only provides a dashed rectangle at design time, and often that is not something we want. In the **protected** section of your component class declaration you would add a line:

```
protected
  procedure Paint; override;
```

This tells Delphi that you are going to create a custom *Paint* method, and that you are not interested in the *Paint* method inherited from higher up the object tree.

For the *TSlideBar* component, I divided the painting responsibilities into a number of procedures: *DrawTrench*, *DrawThumbBar*, *RemoveThumbBar*, *SaveBackground*, and *WhereIsBar*. Therefore, the *Paint* procedure is quite simple:

```
procedure TSlideBar.Paint;
begin
  DrawTrench;
  WhereIsBar;
  SaveBackground;
  DrawThumbBar;
end;
```

The *Paint* method will be called every time Windows determines that the component must be redrawn, and the listed procedures are all that are needed to completely draw the *TSlideBar*. However, don't be fooled by the illusion that you can control when the *Paint* method is called.

There are many times that the procedure will be called and those calls will be out of your control. For example, your component may be covered by a dialog box or window from another application. In this case, once the obstruction is cleared, Windows marks all items that were covered as requiring repainting.

It's interesting to note however, that many of the drawing behaviors of the *TSlideBar* component do not go through the *Paint* method. In many cases, I don't want the entire control to be repainted, only a portion of it. For example, when the thumb is moving, forcing the entire control to repaint is not a good decision. If a user is doing this fast enough, or is dragging the thumb, the control will flicker erratically. This occurs because each move redraws *all* the component's elements.

So, let's apply a little common sense to the issue. If the thumb is moving, then that should be the only thing that repaints. Right? For the most part yes, but there is one problem. When the thumb moves, we must be able to repaint the area of the slot that it had just covered. The slot may have focus, and it may have also covered over one or more tick marks as well. At this point you may be thinking that flicker didn't look so bad after all.

With the *TSlideBar*, a background bitmap is maintained, in addition to the ThumbBar bitmap. This background bitmap is the same size as the ThumbBar bitmap and will always hold the image behind the bitmap. Then, whenever I want to move the thumb, I follow these procedures:

- Place the background image over the current location of the thumb.
- Get the background image of the new location of the thumb.
- Place the thumb in its new location.

If I stick to this sequence in all situations when the thumb is moving, the control will be doing the minimum amount of painting necessary.

Now, instead of refreshing the control after every move of the thumb, I can do only the procedures that are absolutely necessary to maintain the component's correct appearance. Here is the code that is called when the thumb moves:

```
procedure TSlideBar.SetPosition(A: Integer);
begin
  RemoveThumbBar;
  FPosition := A;
  WhereIsBar;
  SaveBackground;
  DrawThumbBar;
  if Assigned(FOnChange) then
    FOnChange(Self);
end;
```

The *RemoveThumbBar* procedure places the saved background image over the thumb's current location. Then, the new position is set (*FPosition* is the private variable that holds the current thumb's position). Next, *WhereIsBar* is called to set the region that the thumb will soon occupy (its new position). The background image is saved in that region. Finally, the thumb is drawn in its new position and calls any defined *OnChange* event that the programmer might have assigned.

By keeping the drawing to a minimum, the *TSlideBar* will be able to quickly update itself in response to mouse and keyboard events.

Being Resourceful

Resources are your friend. Resource files are a way of organizing data and storing it in with a program or component. They are generally used to hold the user-interface elements of a program such as bitmaps, icons, cursors, strings, version information, dialog boxes, etc. Developers can also define their own resource types if necessary (e.g. a proprietary graphics format).

Resource files can be generated with a number of programs such as Resource Workshop and (to a more limited extent), the Image Editor that comes with Delphi. The Image Editor is limited to three basic graphic types: bitmaps, cursors, and icons. Once a resource file has been created, it can be compiled with a program or unit so that it is readily available. In a Delphi component, the resource is bound with the unit (.DCU) when it is compiled. In this way, a component can have access to this data even at design time (e.g. the *TSlideBar* component).

To more efficiently manage system memory, Windows has a certain amount of flexibility when it deals with resource data. Even though resource data is bound with a program or component, Windows will often choose to leave the data on the disk until it is needed. In addition, by default resources are marked as "discardable" so that if Windows must, it can release the resource and memory it is using. If the program needs the resource again later, Windows will reload it. All this is completely transparent to the programmer and user.

However, there are some minor performance penalties that are associated with allowing Windows to have this kind of flexibility. Clearly, if a resource is being frequently swapped on and off the hard disk, the user may be able to see the side effects. The program may hesitate very briefly while Windows retrieves the resource.

If necessary, the programmer can override these default behaviors by marking a resource as "fixed", rather than "discardable" or "movable". This prevents Windows from playing with it. In addition, you can mark a resource as "pre-load" or "load-on-call" to control when Windows will load the resource. (Pre-load loads the resource when the program starts. Load-on-call loads the resource only the first time the resource is referenced.) However, unless it is required for performance reasons, it's better programming practice to let Windows manage the resources on its own by keeping the default options.

The resources used by *TSlideBar* consist of a collection of bitmaps for the different thumbs, and a cursor. (We'll discuss the cursor later.) Each bitmap is given an identifying label as shown in Figure 3.

Each bitmap included in the resource file also has a *mask* associated with it. A mask is used to make sections of a bitmap transparent so you can see areas that the bitmap is sitting on. The thumbs need masks because not all of them are rectangular. If we try to move a circular thumb around on the form, it would have four small corners that would be painted along with it. To solve this problem, a mask is created (see Figure 4).

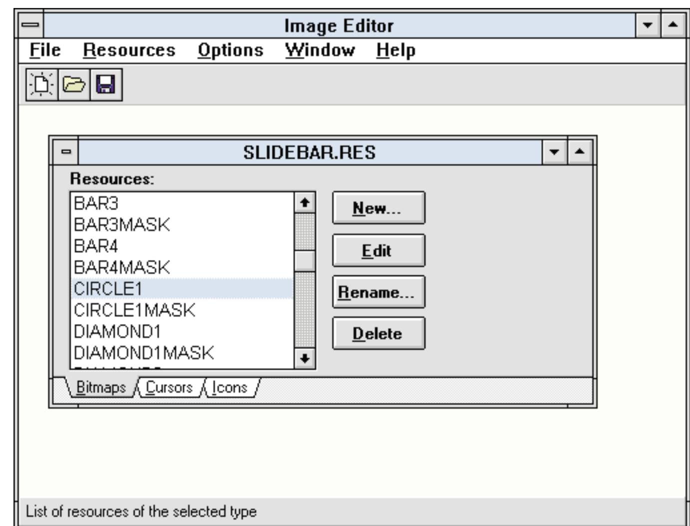
Figure 4 shows how *Circle1Mask* is used to determine the portions of *Circle1* that we want. (Think of it as a kind of "cookie cutter".) If you stacked the two bitmaps, the only portions of *Circle1* that would be painted on the control are those areas that would show through the black portions of *Circle1Mask*. The white portions of the mask would be replaced with whatever background pixels happened to be under the thumb.

Loading the bitmaps into the *TSlideBar* component is fairly straightforward. First, we must set up some *TBitmap* objects to hold the three we will be dealing with: ThumbBar, ThumbBar Mask, and a storage bitmap for the Background pixels. In the *Create* method of the component, we would set these up with this code:

```
constructor TSlideBar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ThumbBmp  := TBitmap.Create;
  MaskBmp   := TBitmap.Create;
  BkgdBmp   := TBitmap.Create;
  { The rest of the Create method goes here ... }
end;
```

Whenever a new thumb style is chosen, the *SetThumbStyle* method is called. (It is the "write" access method for the *FThumbStyle* private variable.) The code in Figure 5 shows a portion of what this method would resemble. Then, when it is time to paint the thumb on the screen, the *real* work begins (see Figure 6). Although this code looks a little convoluted, it's effective. Windows performs these actions when it deals with minimized icons of running applications. Each icon has a transparent area around it so that the desktop color (or image) will show through.

The first step is to create a temporary bitmap to hold an intermediate step in the process. The temporary bitmap is made to



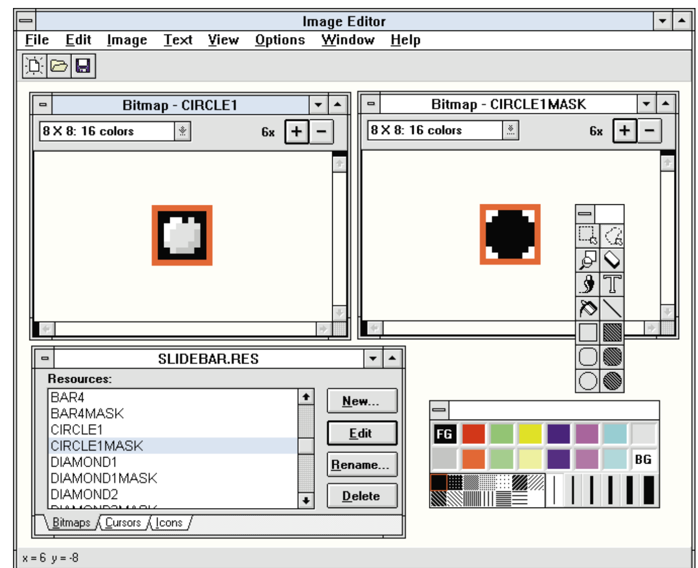


Figure 3 (Top): The Delphi Image Editor. Each bitmap is assigned an identifying label. **Figure 4 (Bottom):** Creating the mask for the circular thumb in the Image Editor.

the same dimensions as the currently selected thumb. Then, the background area (that would normally reside under the thumb) is copied onto this temporary bitmap. Next, the mask bitmap is painted onto the temporary bitmap using the *SrcAnd* copy mode. This causes Windows to perform some math on the combination of the pixels in such a way that the black areas of the mask are erased from the destination image.

Now, we paint the bitmap of the thumb over the resulting temporary image. This time however, we use the *SrcPaint* copy mode that causes the image we want to be merged with the existing background. Then we copy the resulting image to the canvas with the *CopyRect* command.

Notice also that this whole procedure is within a **try...finally** structure. This ensures that the temporary bitmap is released even if an error occurs during the drawing process. This prevents resource leaks and is a feature you should always keep in mind if your component uses memory or other resources.

```
procedure TSlideBar.SetThumbStyle(A: TThumbStyle);
begin
  if ThumbStyle <> A then
    begin
      FThumbStyle := A;
      case ThumbStyle of
        tsCircle1 : ThumbBmp.Handle :=
          adBitmap(HInstance,'Circle1');
        tsSquare1 : ThumbBmp.Handle :=
          LoadBitmap(HInstance,'Square1');
        { Same for the rest of the bitmaps }
      end;
      case ThumbStyle of
        tsCircle1 : MaskBmp.Handle :=
          LoadBitmap(HInstance,'Circle1Mask');
        tsSquare1 : MaskBmp.Handle :=
          LoadBitmap(HInstance,'Square1Mask');
        { Same for the rest of the masks }
      end;

      Refresh;

    end;

end;
```

```
procedure TSlideBar.DrawThumbBar;
var
  TmpBmp : TBitMap;
  Rect1  : TRect;
begin
  try
    { Define a rectangle to mark the dimensions
      of the thumb }
    Rect1 := Rect(0,0,ThumbBmp.Width,ThumbBmp.Height);
    { Create a working bitmap }
    TmpBmp         := TBitmap.Create;
    TmpBmp.Height  := ThumbBmp.Height;
    TmpBmp.Width   := ThumbBmp.Width;
    { Copy the background area onto the working bitmap }
    TmpBmp.Canvas.CopyMode := cmSrcCopy;
    TmpBmp.Canvas.CopyRect(Rect1,BkgdBmp.Canvas,Rect1);
    { Copy the mask onto the working bitmap with SRCAND }
    TmpBmp.Canvas.CopyMode := cmSrcAnd;
    TmpBmp.Canvas.CopyRect(Rect1,MaskBmp.Canvas,Rect1);
    { Copy the thumb onto the working bitmap with
      SRCPAINT }
    TmpBmp.Canvas.CopyMode := cmSrcPaint;
    TmpBmp.Canvas.CopyRect(Rect1,ThumbBmp.Canvas,Rect1);
    { Now draw the thumb }
    Canvas.CopyRect(ThumbRect,TmpBmp.Canvas,Rect1);
  finally
    TmpBmp.Free;
  end;
end;
```

Figure 5 (Top): The *SetThumbStyle* method. **Figure 6 (Bottom):** The *DrawThumbBar* method.

Finally, a reference to the resource file in the source code must be included to bind the resource data to the rest of the component. To do this you use the `$R` compiler directive as follows:

```
{ $R SLIDEBAR.RES }
```

Let's Give 'em a Hand!

One of the nice features of the *TSlideBar* component is that it has an embedded cursor that can be used at run-time. It's a little more visually correct to have a hand or finger (versus an arrow) move a sliding thumb. You do have the ability to change any of the predefined cursors by altering the component's *Cursor* property, but you're limited to the cursors that are provided with Delphi. What if you want your own?

In this case, we do want our own because none of the stock cursors would make any more sense than the arrow does. Therefore, a custom cursor was included with *TSlideBar*. The cursor is created using Image Editor or Resource Workshop, and then saved into an .RES file. This .RES file is then bound with the component when it is compiled.

Once we have the cursor linked with the component, all that remains to do is to obtain a handle to an *HCursor*, and then display it at the appropriate time. Since I did not want to force everyone to use this new cursor, I created a Boolean property called *HandCursor*. If *HandCursor* is set to *True*, the component will switch to the custom cursor whenever the mouse passes over it. If *HandCursor* is *False*, the component will use whichever cursor was defined in the inherited *Cursor* property.

To prepare for managing the cursor, we need to declare two *HCursor* variables — one to hold a pointer to the custom cursor, and the other to hold a pointer to the original cursor (so it can be restored). In the component's **private** section, declare two variables:

```
private
  HandPointer    : HCursor;
  OriginalCursor : HCursor;
```

Then we must remove the cursor from the resource file. Since a variable was declared to hold a pointer to the new cursor, we can load it from the resource in the *Create* method by using this line of code:

```
HandPointer := LoadCursor(HInstance,'HandPointer');
```

Next, we must find a safe place to grab the original cursor. It happens to work nicely in the *WMGetDlgCode* procedure, so the following code is added there:

```
OriginalCursor := GetClassWord(Handle,GCW_HCURSOR);
```

It's important that the original cursor is saved at a point in the component's life where it has a cursor defined. Otherwise you will be saving garbage. It cannot be done in the *Create* method (where I originally tried it) because the cursor is not defined at that point (i.e. before the component is completely created).

The logical place to make the switch between the original and new cursor would be in the *MouseMove* event handler. After all, if the control was receiving *MouseMove* events, that indi-

cates that the mouse is over the component, right? Here's code to make the switch:

```
procedure TSlideBar.MouseMove(Shift: TShiftState;
                              X, Y: Integer);
begin
  if HandCursor then
    SetClassWord(Handle, GCW_HCURSOR, HandPointer)
  else
    SetClassWord(Handle, GCW_HCURSOR, OriginalCursor);
    { Continue with the rest of MouseMove... }
end;
```

You're Just Stringing Me Along

While completing the *TSlideBar* component, I had a brainstorm. Much of the time, a slide bar component is used to pick different textual values from a list. A common approach to this may be to capture the slide bar's position when it changes, and run that value through a big **case** statement to obtain a string value that can then be reported in a *TLabel* on the form. Why not allow the component to hold its own strings? To accomplish this, I simply added a *TStringList* object to the component in its *Create* method:

```
FLabels := TStringList.Create;
```

Then, I added a write access method to the string list to enable the strings to be edited at design time. Since *TStringList* is a complete object, it features a property editor for its strings. By double-clicking on the *Labels* property in the Object Inspector, the component will display the *TStringList* property editor, allowing you to enter the strings that the component will report depending on its current position:

```
procedure TSlideBar.SetLabels(A: TStringList);
begin
  FLabels.Assign(A);
end;
```

Finally, I added a **public** procedure called *CurrentLabel* that obtains the current string value from the *TSlideBar* component:

```
function TSlideBar.CurrentLabel: string;
begin
  if ((Position-Min+1) <= Labels.Count) and
     (Position >= Min) then
    CurrentLabel := Labels[Position-Min]
  else
    CurrentLabel := '<Un-Defined>';
end;
```

In your program, you can add a procedure to the *TSlideBar* component's *OnChange* event that fetches the current label and then feeds it into a *TLabel* on the form. From working with a huge **case** statement, we have boiled it down to a single line of code:

```
procedure TForm1.SlideBar1Change(Sender: TObject);
begin
  Label1.Caption := SlideBar1.CurrentLabel;
end;
```

Conclusion

Although the complete Object Pascal listing for the *TSlideBar* component was too lengthy to present in this article, there is enough information here to give you a good sense of how to implement many of these features in your components. [The entire component and source is available on diskette and for download. See below.]

Delphi provides developers with an extremely powerful and versatile tool in its component design capabilities. With these, a programmer can easily develop powerful and feature-rich components and controls that rival any of the controls that are provided in Windows itself. ∆

The TSlideBar component (including its .PAS and .RES files) is available on the 1995 Delphi Informant Works CD located in INFORM\95\SEP\RV9509.

Robert Vivrette is a contract programmer for a major utility company and Technical Editor for *Delphi Informant*. He has worked as a game designer and computer consultant, and has experience in a number of programming languages. He can be reached on CompuServe at 76416,1373.

ON THE COVER

DELPHI / OBJECT PASCAL

By Gary Entsminger

A Dynamic Toolbar

Using Delphi's Built-In Events to Build a User-Configurable Toolbar

Question: Why is it necessary to drag down from the Olympian fields of Plato the fundamental ideas of thought in natural science, and to attempt to reveal their earthly lineage?
Answer: In order to free these ideas from the taboo attached to them, and thus to achieve greater freedom in the formation of ideas and concepts.
— Albert Einstein, *Relativity, the Special and General Theory*

A toolbar is a panel of controls, usually located just below the menu bar at the top of a form. Typically, a toolbar behaves like a menu. For example, a user clicks on a button or a menu item, and the application responds to the *Click* event. In Delphi, most code executes in response to events. Thus, most components contain an *OnClick* event procedure that you can modify to suit specific applications. The *OnClick* event procedure, you'll soon discover, can be the star of an application.

How to Create a Toolbar

It's easy to add a toolbar to any Delphi application:

- Add a Panel component to a form.
- Set the Panel's *Align* property to *alTop* to force the toolbar to align itself to the top of the form even when the form is resized.
- Add controls (usually SpeedButtons) to the panel.
- Assign control properties. For example, a SpeedButton component needs a glyph, a bitmap image indicating what the SpeedButton does when clicked.
- Modify the *OnClick* event procedure for each control to indicate what the application should do when the control is clicked.

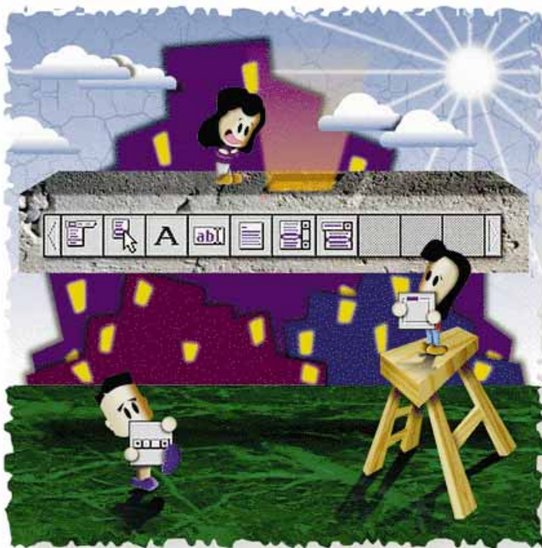


Figure 1 shows a SpeedButton toolbar at design time.

Figure 1: The sample application at design time.

But what if you want to let users create, destroy, or change the properties of the controls on a toolbar at run-time? You can, but it's trickier because it means an application must create new objects (say, SpeedButtons) at run-time and then assign their *OnClick* event procedures to actions that are unknown at design time.

In this article, we'll discuss this assignment problem and others that arise when creating a dynamic toolbar. We'll build a toolbar that acts as a generic "program launcher".

The sample project (Speedbar.DPR) uses a menu to allow users to customize the toolbar. Between application sessions, the toolbar's state is maintained in a table that is loaded each time the application opens. Since one way to use the toolbar is as an application launcher, we'll also allow it to optionally be a floating window (i.e. on top of all other windows).

The User Interface: A Form and Components

In Delphi, building the user interface is easy. From the default project, add the following components from the Component Palette to a form:

- Menu from the Standard page
- Panel from the Standard page
- OpenDialog from the Dialogs page
- Table from the Data Access page

We'll use the Menu component to allow users to issue commands. The Panel will contain the SpeedButtons users add at run-time. The OpenDialog component makes it easy for users to select files to execute, and bitmaps for the SpeedButtons. The table will preserve the toolbar's state between application sessions.

Edit the MenuItems Property

Double-click the Menu component to open the Menu Designer. This dialog box enables you to edit the Menu component's *MenuItems* property. First, add two items to the menu bar with captions `&Buttons` and `&Form`. (The ampersand indicates that the letter following the ampersand will be the underlined shortcut key. For example, Alt B will select the **Buttons** menu item.)

Under the **Buttons** item add the **Add button** (`&Add button`) and **Remove button** (`&Remove button`). Under **Form** add **Normal** (`&Normal`) and **Always on top** (`&Always on top`). Figure 2 shows the form under development with the Menu Designer open. When you're done, close the Menu Designer.

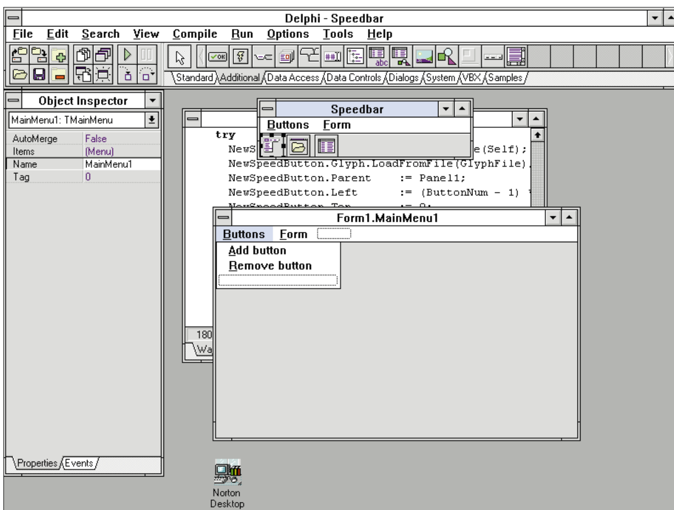


Figure 2: Building the menu with the Menu Deigner.

Making the Toolbar Dynamic

Now things get more interesting (call this Brainstorming 103). We already decided to add and remove SpeedButtons at run-time and save the toolbar between application sessions. Let's also opt for automatically loading and saving the toolbar.

The program uses a table to store the data for the toolbar between sessions. How about storing the SpeedButtons themselves during each session?

Each time we create a SpeedButton, Delphi allocates memory for it. If a user removes a SpeedButton from the toolbar, we want to recover that memory as soon as possible. To recover that memory we use the SpeedButton's built-in *Free* method. The only catch is that we must know which SpeedButtons to free — that is, which SpeedButtons were created at run-time. We can keep track of SpeedButtons by maintaining them in a list, using Delphi's nifty, ready-made solution — the *TList* object.

At the beginning of each session, we'll create a list of SpeedButtons. Then, each time a user adds or removes a SpeedButton, we'll update the list. If we need to remove all SpeedButtons, we'll iterate through the list, removing them one by one. We'll use a list instead of an array because a list is dynamic like the toolbar. We don't know how many SpeedButtons a user might add to a toolbar, and we don't have to specify a list size at design time.

If you think maintaining a list of SpeedButtons might get expensive (in terms of memory and other resources), relax. Every Object Pascal object (and therefore every component, such as a SpeedButton) is really a pointer, a variable containing the address of a memory location. Therefore, a list of objects is really a list of *pointers* to objects, not a list of the objects themselves.

When should loading and saving occur? A good time to load is when the *Form.OnCreate* event procedure executes. Saving could occur when the application terminates, or each time a user adds a new SpeedButton to the toolbar. Let's opt to save the toolbar each time it is modified. This way, the table will always be current.

There are a few more details we need to worry about, including: how to execute a file, how to create an event for each new SpeedButton, and how to protect the application from I/O exceptions. Before we discuss these details, however, let's finish designing more of the application.

Create the Toolbar Table

First, use the Database Desktop to create a table called Speedo.DB. There are at least two important pieces of information to save between sessions to reload a toolbar — the SpeedButton's *Hint* and *Glyph* properties. Both of these properties are strings. The *Glyph* property specifies a file containing a bitmap to appear on the SpeedButton. The *Hint* property is the text (or balloon help) that

appears when the user moves the mouse pointer over a SpeedButton. In addition, in the toolbar application, *Hint* serves a second purpose — it contains the full path of a file or application to execute.

Next, add two string fields to Speedo.DB: Hint and Glyphfile. These correspond to the *Hint* and *Glyph* properties. Make the Hint field a key field. We'll use this key later to remove SpeedButtons from the toolbar. Figure 3 shows the structure of this table during a Database Desktop session. After you create the table, set the table component's *TableName* property to `Speedo.DB` (see Figure 4).

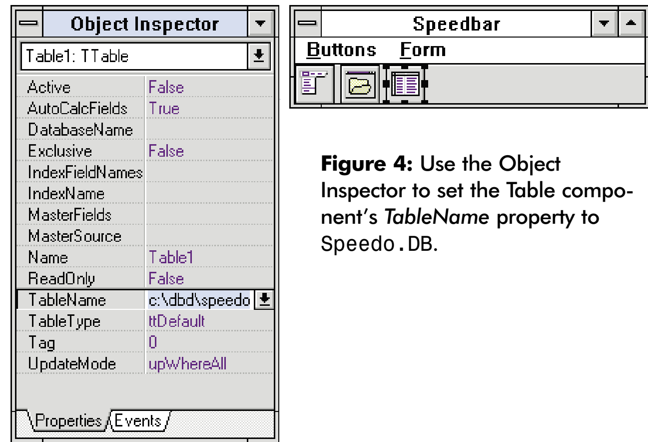


Figure 4: Use the Object Inspector to set the Table component's *TableName* property to `Speedo.DB`.

Unit Variables

When you create a new form or use the form in the default project, Delphi creates a form variable in the **var** section of the form's unit:

```
var
  Form1: Tform1;
```

Add the following variables to that **var** section:

```
  NewSpeedButton: TSpeedButton;
  ButtonNum : Integer;
  ButtonList : TList;
  RemoveButton : Integer; { test flag }
```

These variables will be visible throughout the unit:

- *NewSpeedButton* is a *TSpeedButton* component.
- *ButtonNum* is a SpeedButton counter.
- *ButtonList* maintains the list of SpeedButtons on the toolbar.
- *RemoveButton* is a flag that enables a SpeedButton click event to perform differently depending on how the flag is set. (We'll discuss this flag in detail later.)

The complete listing of the sample application is shown in Listing One beginning on page 21. Please refer to it as we step through the code and discuss its more interesting aspects.

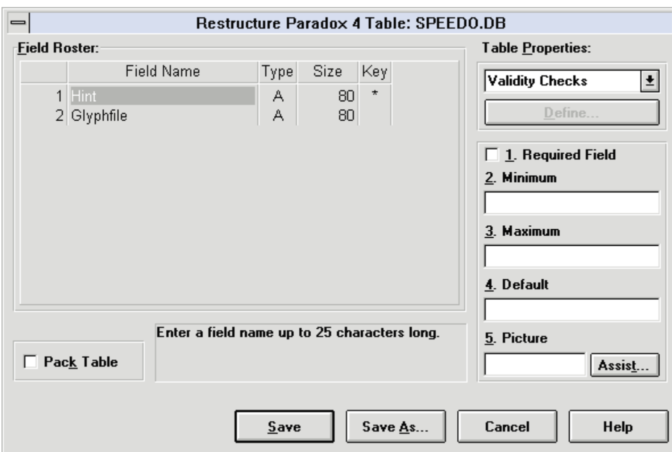


Figure 3: A Database Desktop session. This is the structure of the Speedo table (Speedo.DB).

The FormCreate Event Procedure

The *FormCreate* event procedure is one of the first events that occurs when you run an application. Thus, it's a good event for initializing variables and performing other initialization tasks. In the sample application, when the form opens we'll create a list to hold the toolbar's SpeedButtons, reset *ButtonNum* to `0`, and get the previous session's buttons from the table and load them onto the toolbar.

GetButtonsFromTable

The *GetButtonsFromTable* procedure opens the Speedo.DB table, moves to the first record in the table, reads the `Hint` and `Glyphfile` fields, and then creates new SpeedButtons based on those fields until the end of file is reached.

Note that the procedure uses a **try...finally** block. It's used because at least three things can go wrong in this procedure: the table open can fail, the table read can fail, and the *CreateNewSpeedButton* procedure can fail. If any of these failures occur, we'd like to ensure that the table is closed. Note that if we try to close a table that isn't open, it's no problem (no error occurs).

Why use a **try...finally** block as opposed to a **try...except** block? In a **try...finally** block, the **finally** part of the block is always executed. No matter what happens in *GetButtonsFromTable*, we want to close the table.

Creating a New SpeedButton

The *CreateNewSpeedButton* procedure is the heart of the *GetButtonsFromTable* procedure. *CreateNewSpeedButton* relies on a very interesting generic Object Pascal type, *TNotifyEvent*.

TNotifyEvent is a type of event that notifies a component when a specific event occurs. *OnClick* for example, is of type *TNotifyEvent*, and it notifies a control that a click event occurred on the control.

When a user presses a key or clicks the mouse, an event occurs. Windows sends this event to the window or object that was the focus of the key press or mouse click. For example, if the click occurs on a SpeedButton, *you* determine the

response your application makes by attaching code to the *OnClick* event procedure for the SpeedButton.

But how do you specify behavior for an *OnClick* event procedure that you're adding at run-time? You can assign an *OnClick* event procedure to any *TNotifyEvent* type that has the following form:

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
```

Sender indicates the object that generated the event, and **procedure** can be any procedure. You create specific *TNotifyEvent* types for any descendent of *TObject* (e.g. menu items, SpeedButtons, etc.) by specifying the *Sender* type:

```
type
  TSpeedButtonNotifyEvent =
    procedure(Sender: TSpeedButton) of object;
```

After you've defined a *TSpeedButtonNotifyEvent* type, you can declare a variable of *TSpeedButtonNotifyEvent* type, assign a procedure to the variable, create a new SpeedButton, and assign its *OnClick* event procedure to a *TNotifyEvent* type (in this case, a *TSpeedButtonNotifyEvent* type):

```
var
  EventName: TSpeedButtonNotifyEvent;
begin
  EventName              := GenericSpeedButtonClick;
  NewSpeedButton         := TSpeedButton.Create(Self);
  NewSpeedButton.OnClick := TNotifyEvent(EventName);
end;
```

This sequence is the crux of the *CreateNewSpeedButton* procedure. It begins with a local **var** declaration of a *TSpeedButtonNotifyEvent*. It then uses a **try...except** block to attempt the creation of a new SpeedButton. Why use a **try...except** block? The statement:

```
NewSpeedButton.Glyph.LoadFromFile(GlyphFile);
```

attempts to load a bitmap from a file. It's possible that this file doesn't exist or contains an error. The **try...except** block tests for an `EInOutError` in the **except** part of the block. [For more information regarding **try...except** blocks and exception handling, see Gary Entsminger's article "Exceptional Handling" in the June 1995 *Delphi Informant*.]

The *CreateNewSpeedButton* procedure creates a new SpeedButton, sets the properties of several SpeedButtons, including the location (the *Left* property) of each SpeedButton based on the current button number and the *Hint* property, and adds the newly created SpeedButton to the button list. It then specifies an event name (*GenericSpeedButtonClick*), and sets the SpeedButton *OnClick* event to the *GenericSpeedButtonClick*. The **try...except** block ensures that if there's a problem with any of these details that the application can recover.

(Note that this code assumes the bitmaps used are all 26 pixels in width. You must make the appropriate adjustment to the assignment statement for `NewSpeedButton.Left` if you are using bitmaps of other widths. This code also assumes that you're using the 2-image bitmaps supplied in the \Delphi\Images\Buttons directory as the glyphs.)

The *AddButtonsClick* procedure handles the chores of displaying a file browser dialog box to allow the user to associate a file with a button (see Figure 5). It then displays a browser again so the user can select a glyph for the button.

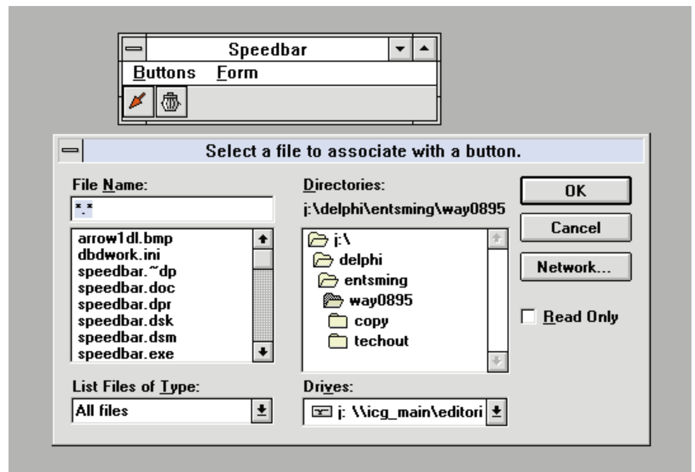


Figure 5: A file browser dialog box is displayed when **Buttons | Add button** is selected. In this manner, the user can associate a file with the button.

GenericSpeedButtonClick

At the heart of the *CreateNewSpeedButton* procedure is the *GenericSpeedButtonClick* procedure. It uses the *Sender* parameter to determine which object (i.e. which SpeedButton) generated the event. In the sample application, the *Sender* is everything. If we know the *Sender*, we know which SpeedButton's corresponding *Hint* property to read. Also, as mentioned earlier, *Hint* contains the file to execute.

GenericSpeedButtonClick is a bit complex because of the *RemoveButton* flag (variable) mentioned (and created) earlier. While designing this application, I encountered a problem. How could I provide a way for users to remove a specific SpeedButton? One possibility was to show the user a list of available SpeedButton hints. For example, after the user selects a hint from the list, that hint could be matched to its corresponding SpeedButton and deleted. This works, but required more work than I preferred.

Therefore, I devised a simple, quicker alternative. The user selects a button to remove by clicking on it. However, we also want the user to be able to click on a SpeedButton and execute a file. That's where the *RemoveButton* variable comes in. If the user selects **Buttons | Remove button**, the *RemoveButton* flag is turned on. Otherwise, it's off and the toolbar executes the file.

If the flag is on, the toolbar asks the user if the button is to be removed. It does this by displaying an Information box with **Yes** and **No** buttons (see Figure 6). If the user selects **Yes**, the button is removed from the table and list (freeing memory). The toolbar application then immediately returns to run mode by turning the flag off (see Figure 7). Thus, the *OnClick* event is assigned at run-time, and has two functions depending on the setting of the *RemoveButton* flag.

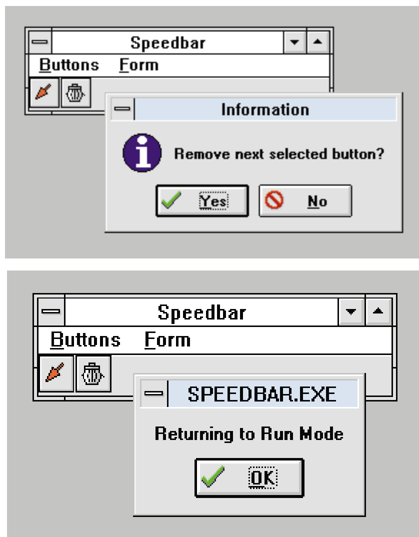


Figure 6 (Top): Selecting **Buttons | Remove button** displays this information dialog box. **Figure 7 (Bottom):** This message is displayed when you click the **No** button on the Information dialog box shown in Figure 6.

Let's quickly look at the remaining custom methods. *DeleteItemFromTable* uses the existing key of a table to find an item to delete. *FreeButtons* traverses the button list, releasing the memory allocated to each SpeedButton and removing the SpeedButton from the button list.

ShellExecute

The last statement in *GenericSpeedButtonClick* calls the *ExecuteFile* method. It converts the Pascal strings for *Command*, *Params*, and *WorkDir* to the null-terminated strings required by the Windows API function **ShellExecute**. It then invokes **ShellExecute** to open the file.

If **ShellExecute** is asked to open a file, it runs the file if it's an executable. Otherwise, it opens the file. Optionally, you can ask **ShellExecute** to *print* a file.

Staying On Top

Finally, let's wrap up the application by adding *Normal* and *AlwaysOnTop* menu click event procedures. These allow the application to either "float" above all other windows on the desktop or to behave normally.

In Visual Basic, you must call the Windows API to create a floating form. However, in Delphi a floating form is a snap. Simply change the form's *FormStyle* property to either *fsStayOnTop* or *fsNormal*. Then, to keep the user informed of the form's status, place a check next to the active menu item.

The FormClose Event Procedure

The *FormClose* event procedure occurs when you close a form (or a single-form application). Thus, it's a good event for cleanup tasks. In this application, when the form closes we'll release the memory allocated to the button list.

Conclusion

There's another way you can go with this of course. Instead of using a Paradox table to hold values between sessions, you could use an .INI file. Both approaches are fine, although the Paradox table does require a lot of overhead (the Borland Database Engine, or BDE) if you want to distribute the toolbar as part of an application and don't need the BDE for anything else. [For a detailed description of how to implement an .INI file using Delphi, see Douglas Horn's article "Initialization Rites" in the August 1995 *Delphi Informant*.]

And that does it. Have fun with this one, and if you customize the application, tell me about it. ∆

This article is adapted from material for Gary Entsminger's forthcoming book *The Way of Delphi* (Prentice-Hall).

The demonstration program referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\SEP\GE9509.

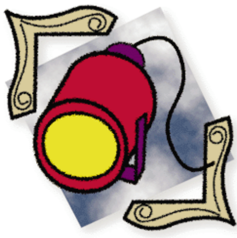
Gary Entsminger is the author of *The Tao of Objects*, an *Introduction to Object-oriented Programming, 2nd ed.* (M&T 1995) and *Secrets of the Visual Basic Masters, 2nd ed.* (Sams, 1994). He is currently working on *The Way of Delphi*, an advanced Delphi book for Prentice Hall, and is Technical Editor for *Delphi Informant*. He can be reached on CompuServe @71141,3006.

Begin Listing One: Toolbar.PAS

```
unit Toolbar;

interface

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

type
  { Define a SpeedButton Notify Event type }
  TSpeedButtonNotifyEvent =
    procedure(Sender: TSpeedButton) of object;
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    Style1: TMenuItem;
    Normal1: TMenuItem;
    AlwaysOnTop1: TMenuItem;
    Buttons1: TMenuItem;
    OpenDialog1: TOpenDialog;
    Panel1: TPanel;
    Table1: TTable;
    AddButtons: TMenuItem;
    Removebutton1: TMenuItem;
    { events }
    procedure Normal1Click(Sender: TObject);
    procedure AlwaysOnTop1Click(Sender: TObject);
    procedure AddButtonsClick(Sender: TObject);
    procedure Removebutton1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
private
  { Private declarations }
  procedure CreateNewSpeedButton(ButtonNum:Integer;
                NewApp: string; GlyphFile: TFileName);
  procedure SaveSpeedButton(ButtonNum: Integer;
                     NewApp: string; GlyphFile: TFileName);
  procedure GetButtonsFromTable;
  procedure DeleteItemFromTable(Command: string);
  procedure FreeButtons;
  procedure ExecuteFile(Command,Params,WorkDir: string);
  { Private generic event handler }
  procedure GenericSpeedButtonClick(Sender: TSpeedButton);
public
  { Public declarations }
  end;

var
  Form1: TForm1;
  NewSpeedButton: TSpeedButton;
  ButtonNum : Integer;
  RemoveButton : Integer;   { Test flag }
  ButtonList : TList;

implementation

{$R *.DFM}

{ Form open and close events }
{ When the form opens, create a list to hold the
  SpeedButtons, reset the button count to 0, and
  get previous session's buttons from a table. }
procedure TForm1.FormCreate(Sender: TObject);
begin
  ButtonList := TList.Create; { Create SpeedButton list }
  ButtonNum  := 0;
  GetButtonsFromTable;
end;

{ When the form closes:
  * release the memory allocated to the button list. }
procedure TForm1.FormClose(Sender: TObject;
                           var Action: TCloseAction);
begin
  { Free the SpeedButton list }
  ButtonList.Free;
end;

procedure TForm1.GetButtonsFromTable;
var
  NewApp: string;
  Glyphfile: TFileName;
begin
  try
    Table1.Open;
    Table1.First;
    while not Table1.EOF do
      begin
        ButtonNum  := ButtonNum + 1;
        NewApp     := Table1.FieldByName('Hint').AsString;
        GlyphFile  :=
          Table1.FieldByName('GlyphFile').AsString;
        CreateNewSpeedButton (ButtonNum, NewApp, GlyphFile);
        Table1.Next;
      end;
  finally
    Table1.Close;
  end;
end;

{ Generic execute file routine }
procedure TForm1.ExecuteFile(Command, Params, WorkDir:
                             string);
begin
  { Convert Pascal string to Null-termintated strings }
  Command := Command + #0;
  Params := Params + #0;
  WorkDir := WorkDir + #0;
  { Run/open application/file }
  if ShellExecute(Application.MainForm.Handle, 'Open',
                 @Command[1], @Params[1], @WorkDir[1],
                 SW_SHOWNORMAL) < 32 then
    MessageDlg('Could not execute ' + Command, mtError,
               [mbOK],0);
end;

procedure TForm1.FreeButtons;
var
   I : Integer;
begin
  { Go through the button list until the end is reached;
    * release the memory allocated to each SpeedButton
    * remove the button from the list. }
  for I := 0 to ButtonList.Count do
    begin
      NewSpeedButton := ButtonList.Items[0];
      NewSpeedButton.Free;
      ButtonList.Remove(NewSpeedButton);
    end;
      ButtonNum := 0;
end;

{ Use the existing key to find the item to delete;
  or alternatively define a key here. }
procedure TForm1.DeleteItemFromTable(Command: string);
begin
  Table1.Open;                 { Open table }
  Table1.FindKey([Command]);   { Use existing key }
  Table1.Edit;                 { Hint field is the key }
  { Delete this item, button.hint, from table }
  Table1.Delete;
  Table1.Close;                { Close table }
end;
{ Generic SpeedButton click can either set up a new
  SpeedButton toolbar or execute a file }
procedure TForm1.GenericSpeedButtonClick(
            Sender:TSpeedButton);
```

```
var
  Command, Params, WorkDir: string;
begin
  { Get the file name and path from hint }
  Command := Sender.Hint;
  { Edit table - remove button }
  if RemoveButton = 1 then
    begin
      DeleteItemFromTable(Command);       { Remove button }
      FreeButtons;       { Remove buttons, release memory }
      GetButtonsFromTable; { Reload new table of buttons }
      RemoveButton := 0;    { Reset button count }
      ShowMessage('Returning to Run Mode');
    end
  else
    { Execute file }
    begin
      Params  := '';
      WorkDir := '';
      ExecuteFile(Command, Params, WorkDir);
    end;
end;

procedure TForm1.CreateNewSpeedButton(ButtonNum: Integer;
            NewApp: string; GlyphFile: TFileName);
var
  EventName: TSpeedButtonNotifyEvent;
begin
  try
    NewSpeedButton := TSpeedButton.Create(Self);
    NewSpeedButton.Glyph.LoadFromFile(GlyphFile);
    NewSpeedButton.Parent    := Panel1;
    NewSpeedButton.Left      := (ButtonNum - 1) * 26;
    NewSpeedButton.Top       := 0;
    NewSpeedButton.Hint      := NewApp;
    NewSpeedButton.ShowHint  := True;
    NewSpeedButton.NumGlyphs := 2; { Assume 2-image bitmap }
    ButtonList.Add(NewSpeedButton); { Add instance to list }
    { Assign new event, created at runtime, to the
      SpeedButton OnClick event, GenericSpeedButtonClick }
    EventName := GenericSpeedButtonClick;
    NewSpeedButton.OnClick := TNotifyEvent(EventName);
    except
      on E: EInOutError do
        begin
          MessageDlg('Unable to create SpeedButton. ' +
          E.Message, mtInformation, [mbOK],0);
          { Free button resources }
          NewSpeedButton.Free;
          ButtonNum := ButtonNum - 1;
        end;
      end;
end;

{ Use the Open Dialog to get a file name and a
  corresponding bitmap for the new SpeedButton to be
  associated with the file. }
procedure TForm1.AddButtonsClick(Sender: TObject);
var
  NewApp: string;
begin
  OpenDialog1.Title :=
    'Select a file to associate with a button.';
  OpenDialog1.Filter   := 'All files|*.*';
  OpenDialog1.FileName := '';
  if OpenDialog1.Execute then
    begin
      NewApp:= OpenDialog1.FileName;
      if FileExists(NewApp) then
        begin
          OpenDialog1.InitialDir := '';
          OpenDialog1.Title :=
            'Select a glyph for the SpeedButton.';
          OpenDialog1.Filter := 'Bitmap files|*.bmp';
          OpenDialog1.FileName := '';
          if OpenDialog1.Execute then
            begin
              ButtonNum := ButtonNum + 1;
              CreateNewSpeedButton(ButtonNum, NewApp,
                                   OpenDialog1.FileName);
              SaveSpeedButton(ButtonNum, NewApp,
                              OpenDialog1.FileName);
              { Set Remove Button flag to false }
              RemoveButton := 0;
              ShowMessage('In Run Mode');
            end;
        end
      else
        ShowMessage('File does not exist.');
    end;
end;

{ Remove button click event simply sets a remove flag,
  which the generic click event checks before deciding
  how to carry out the click event }
procedure TForm1.Removebutton1Click(Sender: TObject);
begin
if MessageDlg('Remove next selected button?',
     mtInformation, [mbYes, mbNo], 0) = mrYes then
  RemoveButton := 1
else
  begin
    RemoveButton := 0;
    ShowMessage('Returning to Run Mode');
  end;
end;

{ Add each new SpeedButton to end of SpeedButton table }
procedure TForm1.SaveSpeedButton(ButtonNum: Integer;
                    NewApp: string; GlyphFile: TFileName);
begin
  with Table1 do   { Add to Speedo table }
    try
      Open;
      Last;         { Move to the end of the table. }
      Insert;
      FieldByName('Hint').AsString       := NewApp;
      FieldByName('GlyphFile').AsString  := GlyphFile;
      Post;
    finally;
      Close;
    end;
end;

{ Form style click event procedures: an AlwaysOnTop click
  makes the form 'float' above all other forms and a
  Normal click allows the form to behave normally }
procedure TForm1.Normal1Click(Sender: TObject);
begin
  AlwaysOnTop1.Checked  := False;
  Form1.FormStyle       := fsNormal;
  Normal1.Checked       := True;
end;

procedure TForm1.AlwaysOnTop1Click(Sender: TObject);
begin
  Normal1.Checked       := False;
  Form1.FormStyle       := fsStayOnTop;
  AlwaysOnTop1.Checked  := True;
end;

end.
```

End Listing One



Informant Spotlight

Delphi / Power Builder

By Thomas Miller

Leaving PB Behind

Delphi vs. PowerBuilder: A Blow-by-Blow Comparison

The perfect enterprise development tool would allow us to program through telepathy, debug the program, and automatically write the documentation and on-line help. It would compile and optimize the program in Assembler, be operating-system-level compliant with Windows, Windows NT, Macintosh, OS/2, and 10 flavors of UNIX. It would also support all databases through native APIs, and integrate seamlessly with workgroup products.

Back to reality. We buy programming tools to avoid tedious programming in C/C++ and assembler and facilitate timely delivery of software. Currently however, there are no full fledged enterprise development tools on the market.

We may begin to see these all-encompassing tools that support multiple platforms, multiple databases, group productivity, and rock solid compiled code in two or three years. For now, however, PowerBuilder 4.0 (PB) from PowerSoft, Inc. is "king of the hill". Borland's Delphi is the new kid in town and a more-than-able challenger. Yet, some developers are apprehensive about using Delphi for several reasons: 1) Delphi is a new product, 2) Borland has had its much-publicized problems, and 3) PB is well-accepted.

It's my contention that Delphi is the best Windows development tool on the market. It's fast, easy-to-use, and has the world-renowned Pascal language in its pedigree. For those who aren't convinced of Delphi's capabilities, this article is for you. It's time for you to leave PB behind.

Painters and Palettes

PB's integrated development environment (IDE) is divided into "Object Painters" (see Figure 1). These are the building blocks used to create an application. On the other hand, Delphi uses forms and an object palette metaphor (the Component Palette) for development (see Figure 2). As any good programmer will tell you: "I can get the system to do anything you want." For the purposes of this discussion, however, only those functions that are specifically designed into each development system are being compared.

Let's compare PB to Delphi using PB's painters as the "jumping off point".

The **PB Application Painter** creates an application shell, opening script, library paths, and allows you to set the application icon. **Delphi** creates

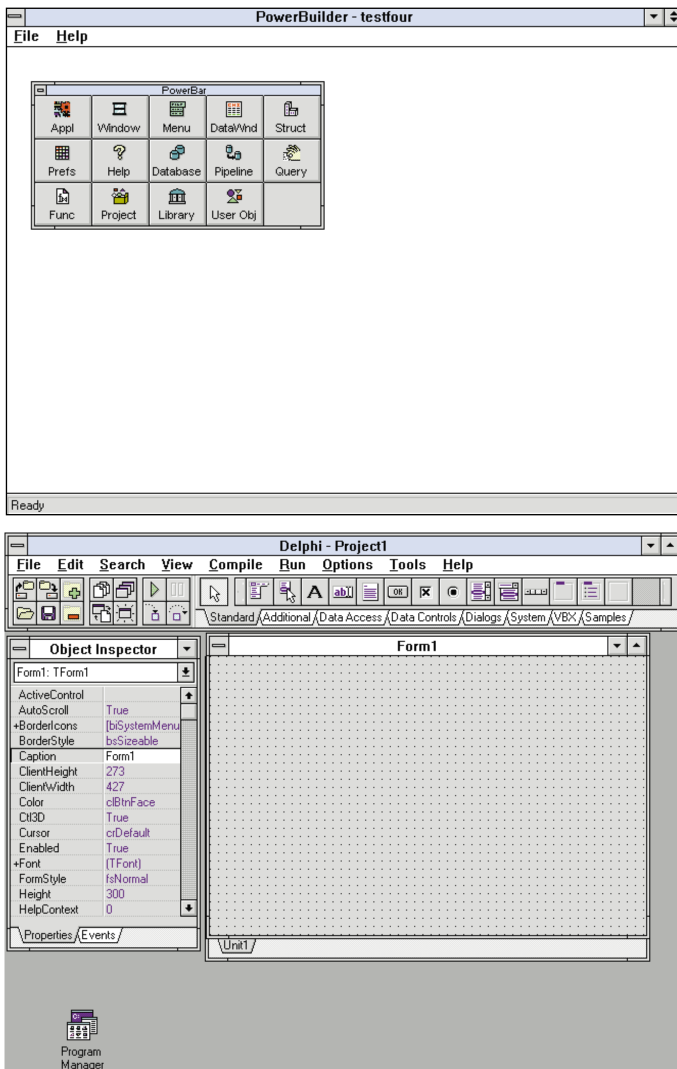


Figure 1 (Top): The PowerBuilder 4.0 opening screen. The floating palette contains icons that open the major painters used by PowerBuilder. **Figure 2 (Bottom):** Delphi's opening screen. Delphi uses a single-document interface (SDI) that is compliant with Windows 95. The system window, located at the top of the screen, contains the menu, Speedbar, and a Component Palette. The Object Inspector is on the left. It allows you to set an object's properties and events. The form is on the right. It allows you to design a window of an application.

a project shell, opening code, allows you to set the application icon, and default help file for context-sensitive help.

Delphi doesn't support libraries in the same sense as PB. In PB, the libraries are used to segregate the code into smaller manageable pieces of deliverable code. In Delphi, they are extensions to the development environment and are referred to as the Visual Component Library (VCL). Not all objects in the VCL are visual. To break up large projects, Delphi supports DLLs with restrictions which are discussed later in this article. PB doesn't directly support context-sensitive help.

The **PB Project Painter** allows for configuring a macro for compiling an application. **Delphi** enables you to configure compiler options, linker options, and directories for additional resources. This is a new painter for PB (version 4) and was sorely needed for compiling large projects. Compiling code has five separate steps. And, compiling 15MB of source code can take three hours to complete. If you change one object in the program, PB highly recommends you recompile the entire program. In addition, PB recommends compiling only on your local hard drive, and turning off Windows SMARTDrive. When you compile in PB, you are really pre-compiling for the p-code interpreter. New releases of the interpreter fix many bugs, but inevitably introduce new ones. In addition, sub-routines that worked fine for months are suddenly non-functional.

In contrast, Delphi is fast, easy, and specifically supports incremental re-compiles. A 15MB source program will compile in about 10 to 15 minutes. The compiler options allow you to set extra debugging features for in-house testing and then later turn them off for distribution of your program. This allows for tighter, faster deliverable programs that are true EXEs and DLLs.

The **PB Window Painter** assembles various other objects or controls (including data window controls) into a functioning window. The **Delphi form** is an object where you can assemble other objects to create a functioning window. The functionality of these two objects is similar.

The **PB Menu Painter** enables you to design and program menus (main menus). **Delphi** has a **Menu Designer** for creating MainMenu and PopupMenu components. PB enables you to make menu items active, inactive, visible, or hidden at run-time. Delphi gives you complete control of the menu at run-time to include adding, deleting, changing and merging parent, and child menus.

The **PB Structure Painter** enables you to graphically declare a structure. **Delphi** refers to the structure data type as a *record*. There is no graphical wizard to aid in declaring this type of variable. Delphi does not facilitate any graphically-assisted coding. This would require an extra step to convert from the graphical representation to Object Pascal before the code is compiled. This is a trade off between ease-of-use and speed and reliability.

The **PB Preference Painter** enables you to set many of the preferences used by PB (see Figure 3). **Delphi Environment Options** allow you to set many preferences for the Delphi IDE (see Figure 4).

PB presents more options under its Preferences Painter. Unfortunately, most are codes and difficult to set. Delphi gives you an adequate number of environment settings that are easy to access, and you are always welcome to dig into the source code to change any or all defaults.

The **PB Database Painter** allows you to create and manipulate tables and columns, and set extended attributes for columns in a table. The Database Painter also allows you to browse and manipulate data. The **Delphi Database Desktop** supports data browsing and manipulation.

PB's Database Painter is one of the best tools available for creating a database from scratch. It supports indexes, keys, and extended column attributes. If you have a system that is set in stone, the extended attributes are wonderful.

If you're in a rapid application development situation (i.e. making little changes all the time), the modifications you make to existing extended attributes are not automatically updated to objects that already use the extended attributes. Suppose you create an extended attribute for an Employee Type drop-down list box. When you originally set the extended attribute there are three employee types. The extended attribute is then associated to three different windows. Later, the human resource department tells you that there are four Employee Types. You fix the extended attribute, recompile the program, and drop down the Employee Type drop-down list box to find only three types. You are required go to each control and manually refresh it before compiling.

The extended attributes are maintained in the active database type. If you start off with Sybase, this makes it almost impossible to use the data window for Oracle. PB stores information about databases in its own system catalogs. It creates catalogs in the actual database. For example, if you are using Sybase, it creates its system tables in the Sybase database. This is somewhat similar to setting up aliases for databases in the BDE, but is more messy and requires more set-up work

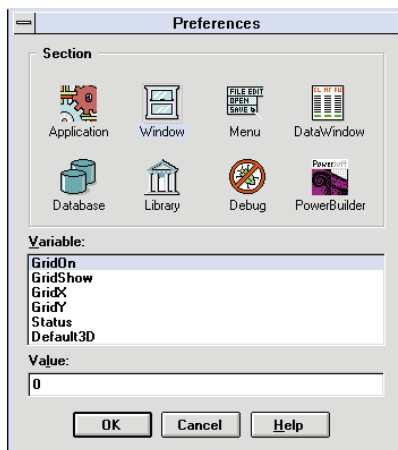
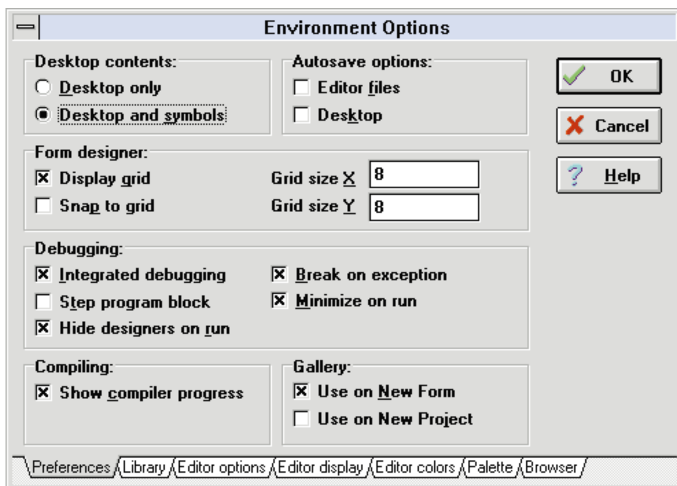


Figure 3 (Top Left): PB's Preferences Painter.
Figure 4 (Bottom): Delphi's Environment Options dialog box.



because it creates its own tables in each of the databases. This makes it difficult to "point" to new databases "on-the-fly". The advantage with extended attributes is that a lot of data validation rules can be done at database setup time.

This is a perfect example of *acting* object-oriented versus *being* object-oriented. Delphi lacks a database table maintenance tool. Both systems have capable browsers.

The **PB Database Profile Painter**: This allows you to set your database connection information. Most databases require a white paper to implement. The **Borland Database Configuration Utility** is simple and straightforward. PB's database connection is difficult to set up, and not all the ODBC drivers work consistently. PB does include distributable ODBC drives. With Delphi you will have to purchase them separately. There have been very few complaints about configuring the Borland Database Engine (BDE).

The **PB Query Painter** (4.0) features an updated interactive wizard to assist in creating a SQL statement (see Figure 5). The Query Painter is well-integrated with other data manipulation tools. **Delphi's Visual Query Builder** is an interactive utility that assists in creating a SQL statement (see Figure 6). PB has designed an intuitive and easy-to-use interface. Delphi's is somewhat complex to use, even with the manuals.

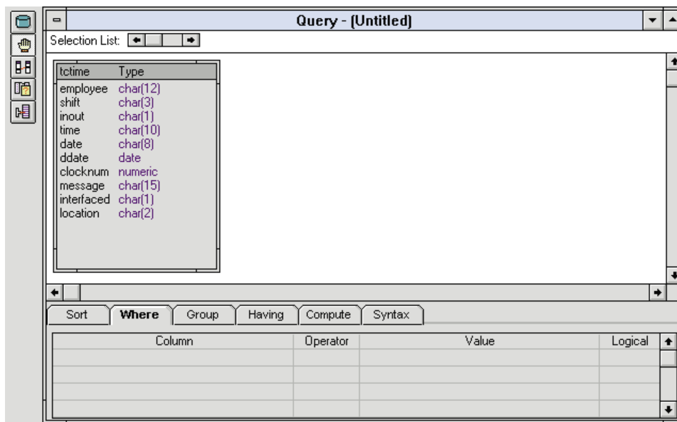
The **PB Pipeline Painter** allows translation from one vendor database to another. **Delphi's BatchMove Component** also allows translation from one vendor database to another. PB and Delphi are functionally 95 percent the same.

The **PB Function Painter** allows for graphically registering a function. **Delphi** has no graphical "wizard" to aid in registering functions. (As stated earlier, Delphi does not facilitate any graphically-assisted coding.)

The **PB Library Painter** enables hierarchical graphical display of objects grouped in libraries. **Delphi** graphically displays its programming units.

One of Delphi's weakest points is its inability to support distributable libraries. This is only a problem for very large projects (50 or more windows). In Pascal distributable overlays are referred to as "overlays", and are not currently supported in Delphi or Object Pascal. Unfortunately, PB's libraries don't serve one of their primary functions, which is compiled code segregation. As mentioned earlier, when you make a change to one library, PB recommends you re-compile and redistribute the entire system. PB's libraries are not truly object-oriented, and are not designed to be shared between multiple programs. This defeats another major benefit of libraries.

The PB User Object Painter graphically enables you to program extensions to the programming environment, and access the Windows API and non-visual objects. PB's user objects are a very crude equivalent of Delphi's components, but because PB is not object-oriented, you can't create real



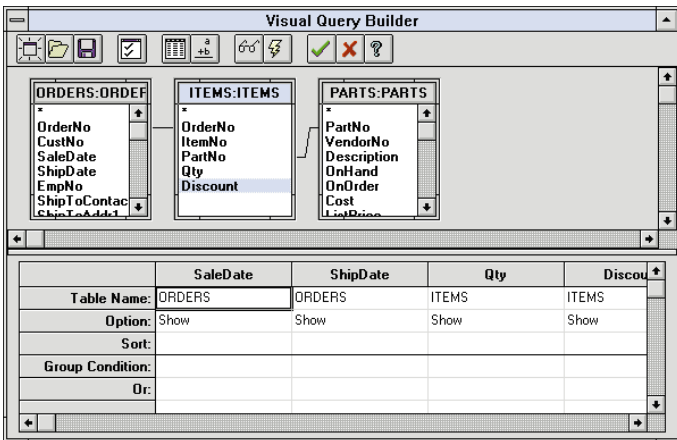


Figure 5 (Top): PB's Query Builder. **Figure 6 (Bottom):** Delphi's Visual Query Builder.

components. Nevertheless, user objects help with standardizing and re-use. Non-visual user objects are PB's equivalent of classes. **Delphi** supports these programming capabilities non-graphically. In addition, Delphi extends the programming environment through the VCL. There are also non-visual components.

PB has tried to make advanced programming graphical. It just hasn't worked. The user-objects are difficult and complex to implement. Delphi has taken a more practical, direct approach and requires you to write straight code and place it appropriately. Writing in assembler, accessing DLLs, and writing to the Windows API is complex enough without adding a graphical encapsulation layer.

This covers most of the major sub-systems in PB and Delphi that are easily compared on a direct basis. Now we'll compare data access, events, and scripting. This is more of an "apples and oranges" type comparison.

PB Data Access, Events, and Coding

If you have ever attended a PB seminar you have definitely heard that "The DataWindow is our crown jewel." Unfortunately, these days it's a chipped and cracked crown jewel.

The DataWindow is easy to work with and can create a data access window quickly. It is arguably the most flexible database layout tool on the market. It does all the coding for you in the background, prints reports, and allows easy sorting and filtering. For a very simple, direct data access window, it's tops. Now let's look at the downside.

The DataWindow will not support many non-data-related objects. For example, you can't place a button or an edit box on the DataWindow. This can make it difficult to design an effective graphical user interface (GUI) for your end-user.

The DataWindow uses a data replication algorithm to access data. This is a major problem in several areas. First, it takes a long time to retrieve all the data to the workstation and problems with synchronization can arise. For example, if the replication gets out of sync with the server database, you have to retrieve the data from the server again. Large amounts of data replicated locally consume huge amounts of resources and can slow the workstation to a crawl. And, when two DataWindows are sharing the same data, you have to manage the synchronization manually. Furthermore, the SQL code is written for you, making it difficult to change it programmatically.

When distributing an off-the-shelf program, it's difficult to programmatically set the database owner. PB writes extended attributes about the database into the DataWindow. When you change a column attribute (i.e. size or not null) in the database, you have to manually fix each DataWindow accessing the table to recognize the table change. This is because the DataWindow is not truly object-oriented.

Also, the DataWindow is not always sensitive to programmatic changes to its buffers (e.g. window "A" assigns a value to column "1" in window "B"). In advanced programming, the DataWindow has three data buffers that must be managed. Suddenly, the DataWindow is not the "do-all automatic database interface" tool it claims.

In PB, the DataWindow represents most of the user interface. It becomes important for the DataWindow to be flexible in handling design issues in the development environment and at run-time. One word for the PB faithful: dwModify. Need I say more?

For the rest of you, *dwModify* is a C++/Assembler type syntax to modify attributes of the DataWindow programmatically. It is not for the faint of heart! PB has converted much of the old DataWindow command structure to a BASIC-type command structure in version 4. However, there is still enough functionality available through the old, arcane, syntax of dwModify to be annoying.

Windows is an event-driven operating system. Therefore, it's important that any Windows development tool properly trigger events and related code. PB is sorely lacking in this area. PB recommends using events sparingly. This is partly due to the large amount of overhead associated with events.

The main reason, however, is that many PB events do not interact well with each other. As an illustration, set up a simple DataWindow attached to an empty table. Next, place message boxes in each of the following DataWindow's events: EditChanged, ItemChanged, ItemFocusChanged, and RowFocusChanged. As you add the first row to the DataWindow, the events will fire off in one order. As you add the second row to the DataWindow, the events will fire in a second (and different) order, and when you delete the two rows to leave the DataWindow blank, the events fire in a third, different order.

GetFocus (pre-event) and LoseFocus (post-event) are two critical types of events in Windows programming. Remove these two event types from an event-driven system and you have a structured system (DOS). PB has a major problem triggering post events. The DataWindow is an object with related pre-events and post-events, and data fields are not objects and do not directly support any events. This is another example of acting object-oriented instead of being object-oriented.

As an example, set up a simple DataWindow with a field requiring verification against a second table. (An order form with part numbers.) In the ItemChanged, ItemFocusChanged, and LoseFocus events, enter the following script:

```
if dw_1.GetColumnName() = "Part_Number" then &
  MessageBox("EventName","Verify Part Number in Parts Table")
```

Now add a button to the window. We'll use this to change focus from the DataWindow to the Button. The message box represents a subroutine to verify that the entered part number exists and to retrieve associated data from the parts table (i.e. price, cost, description, etc.) Enter a part number in the field and click the button. What happens? Nothing. Let's see why.

PB does not run any code unless an object has focus. Using this premise, we'll analyze why each event failed to give us the desired result:

- ItemChanged Event doesn't run because GetColumnName doesn't return Part_Number because the button has focus.
- ItemFocusChanged doesn't run because GetColumnName doesn't return Part_Number because the button has focus.
- LoseFocus doesn't run because when the code runs, the button has focus and GetColumnName doesn't return Part_Number.

Some of you may ask: "Why aren't you using the EditChanged event?" This would work, but consider that for each keystroke, the system would verify the part number and display a dialog box with the message "Part number not on file". A 10-character part number would return nine error messages. In short, PB's inability to deal with post events is a major problem and totally unacceptable in an event-driven environment.

Script (code) in PB is easy, direct, and conforms to industry standards. If you are familiar with BASIC, Visual Basic, C, C++, FORTRAN, or dBASE, you can quickly settle into the syntax provided by PB. PB provides over 600 functions, allows you to add additional functions through registering external DLLs, and allows access to the Windows API. FUNCky Library, available from PowerSoft, is an add-on function pack that provides more than 500 functions.

Client/Server Data Access

For quite a while, PB has been renowned for its database access (the DataWindow facilitates this access). The basic database access of the DataWindow has not changed since Version 2 was released more than three years ago.

PB replicates data from the server database to the workstation. This is the first problem. Client/server database engines were specifically designed to return blocks of data as needed. Let's say you send a query to the database that has a 5,000 record result set. In PB, it will return all 5,000 records to the local workstation before any operations can be performed on the data. RetrieveAsNeeded (a function that only returns enough data to fill the screen) is simply a ruse to make the screen *look* active. Then, in a separate database call, PB requests the rest of the data. Either way, you often have to wait as long as 15 minutes until you can access the data set on the screen.

Delphi doesn't directly support sorting or filtering. This is a mixed blessing. Ideally, you want to sort and filter at the server to save network overhead and not burden the workstation. On the other hand, you cannot dynamically change the sorting or filtering without re-querying the database. This will depend on the functionality required by your application.

Most client/server systems are designed to return a specific amount of data chunks, thereby optimizing data access. Suppose you have a Novell Network on an Ethernet architecture running Oracle. If you set the Ethernet Packet size to 16K, the Novell disk block size to 16K and the Oracle workstation requester data retrieve parameter to 128K, you have just optimized the transport of data from the server to the workstation. Assume our 5,000-record query represents 2000K of data. When the workstation reaches the last record in the first "chunk" of data returned (128K), it then requests the next chunk of data (128K). This increases overall speed, reliability, and decreases the possibility of the workstation data set getting out of sync with the server database.

New technology and other products have easily eclipsed PB's capabilities. ODBC is now a mature API and supported by all popular development environments and database systems. Many companies have developed general database access engines that are flexible and can be attached to multiple development environments.

Delphi Data Access, Events, and Coding

Delphi is a new breed of tool featuring a general database access engine that works under the traditional client/server paradigm. In addition, Delphi sports an IDE that is truly object-oriented.

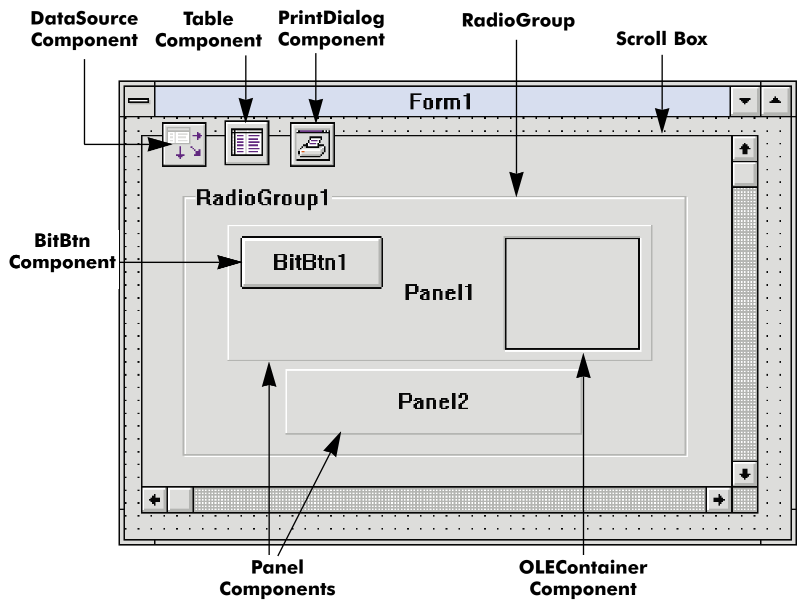
When you first start Delphi, the most noticeable difference is the lack of individual painters. Delphi is a "top-down" development tool. First you open a form (window), and then place objects on the form: panels, buttons, edit boxes, and/or other non-visual objects (see Figure 7). The objects you add to a form can then contain their own associated objects. Thus the parent/child/grandchild etc. relationship is created (see Figure 8).

If this sounds like more work — it is! To lay out a simple window minus the code may take 15 minutes to do in PB, and as much as 45 minutes in Delphi. However, don't get discouraged — there are numerous gains in other areas of Delphi that more than compensate for the difference in time and capability.

Delphi's Component Palette is the centerpiece of its development environment. Some 75 objects, visual and non-visual, are available from this palette. The palette is broken into the following general categories: Non-Data Aware Components, Data Access Components, Data Aware Components, Windows Non-Visual Dialog Components, and System Components.

Individual components worth special mention are:

- **Scrolling Panel:** This allows for controlling the scroll bars when the panel shrinks vertically or horizontally smaller than a user-defined threshold. (Not available in PB.)
- **TabPages:** This is two components working together allowing multi-page panels with tabs at the bottom of the page, similar to a spreadsheet. (Not available in PB.)
- **TabbedNotebook:** This component has multiple pages with large folder-style tabs at the top of the page (see Figure 9). (Not available in PB.)
- **DataSource:** This is an object that, when linked with supporting objects, creates the data connection to the database. It automatically manages commits, rollbacks, adds, saves, deletes, undos and refreshes the data set. (PB requires extra coding.)
- **DBNavigator:** A graphical object that controls scrolling, add, edit, delete, undo, and refresh functions for data aware controls. (PB requires extra coding.)
- **DBLookupList / DBLookupCombo:** These two controls allow you to point to another table for population and, on selection, return the selected item to the current table and column. (PB requires extra coding.)
- **Dialog Objects:** Include Windows File, Save, Font, Print, Printer Setup, Color Palette, Find, and Find and Replace dialog boxes. (PB requires extra coding.)



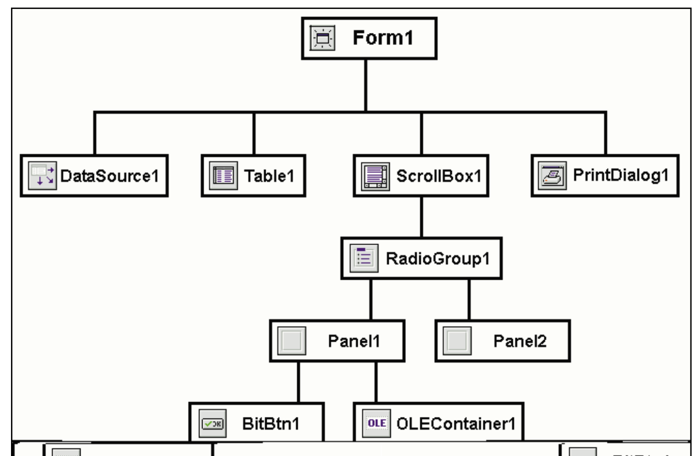


Figure 7 (Top): A Delphi form with the following components: DataSource, Table, PrintDialog, ScrollBox, RadioGroup, two Panels, BitBtn, and an OLEContainer. **Figure 8 (Bottom):** The object hierarchy for the form shown in Figure 7.

- **System Components:** These include Timer, PaintBox, FileListBox, DirectoryListBox, DriveComboBox, FilterComboBox, MediaPlayer, OLEContainer, DDEClientConversation, DDEClientItem, DDEServerConversation, and DDEServerItem. (The same functionality in PB requires extra coding, or is not available.)

By using just one of these objects to replace the coding necessary in PB, you have recovered your lost 30 minutes. The only object that is missing is a multi-line grid component. This is one of the most popular and powerful interface items in PB and the only large hole in the Delphi GUI design objects repertoire. Delphi's grid component is a traditional spreadsheet type interface that only supports an edit box interface. However, all other design GUI elements found in PB are directly supported, surpassed, or easily reproduced with minor programming in Delphi. Of

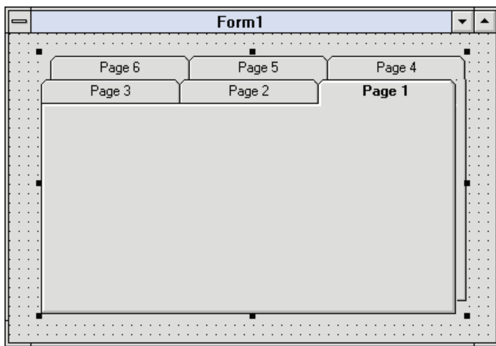


Figure 9: Delphi's TabbedNotebook component, an amalgamation of the TabSet and Notebook components.

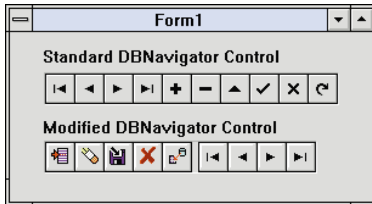


Figure 10: This Delphi form shows a DBNavigator control and an extended DBNavigator control at run-time.

course, if you feel there is something lacking, you may modify and extend any of the existing components, or acquire a third-party component. There are several third party grid components that give you some of the functionality of PB grids (e.g. those from Woll2Woll and Orpheus). More important, Delphi and all its components are written in Object Pascal making Delphi completely extensible. For example, Figure 10 demonstrates Delphi's extensibility with a standard DBNavigator component and its extended counterpart. Try that with PB!

In Delphi, you control objects through *properties* and *events*. Delphi provides access to properties and events via its Object Inspector. In Delphi, all properties are in one easy-to-read dialog box, while PB spreads them across several menu items.

In general, Delphi supports 20 to 25 percent more properties per object than PB. Setting most properties in Delphi is usually accomplished through a drop-down list box and can programmatically be set using the same keywords as found in the Object Inspector. For instance, the *TabStop* property can be either *True* or *False* in the Object Inspector. To programmatically change *TabStop* to *False*, the statement is:

```
TabStop := False
```

No *dwModify* and no guessing. If the attribute is supported in the Object Inspector, it can be controlled programmatically.

PB's event handling is average at best. Not all items have events and many events don't run properly. Delphi's true object-oriented paradigm has no problems with pre-events and post-events, or with multiple events interacting with each other. You are free to program 10 or more events related to one object. Overhead isn't an issue with Delphi because the code is compiled into a true .EXE that will run up to 20 times faster than PB's p-code.

Delphi's data access is faster, and more reliable than PB's. It requires substantially less coding to set up database access. Delphi directly supports parent/child table relationships in the database objects, auto-synchronizes two views of the same table (the equivalent of PB share data), and even supports stored procedures.

Conversely, PB only supports stored procedures with certain databases. Delphi has two data control components: Table and Query. Table gives you direct access to the entire table, is fast, and easy-to-implement. The Query component allows you to access the table by limiting the result set with a SQL statement.

There is, however, room for improving Delphi's Query component. In PB, SQL statements are executed extremely fast. Delphi's Query component seems to "hiccup" once before it sends the SQL statement to the database server. Depending on the size of the result set, Delphi's Query component is still — hands-down — faster than PB. The Query component returns result sets to an array — not variables — and doesn't support cursors at the database server. These are all minor inconveniences at best. The Query component is effective for simple, basic SQL statements. And the StoredProcedure component handles very complex SQL sub-routines with ease. You will have to be creative with some SQL sub-routines to avoid putting them into stored procedures.

Delphi is based on Object Pascal. Object Pascal rivals the power and flexibility of C/C++, but is strongly typed (restrictive variable declaration) to keep you out of trouble. The syntax is different from other mainstream programming tools and takes some time to get comfortable with. Blocks of code start with **begin** and finish with **end** (see Figure 11). This includes sub-blocks of code found within **if** statements. Each statement ends with a semicolon. This saves you from using the ampersand character to continue the line as you must do with PB.

In Delphi, the only place this isn't true is in the **if** statements. Placing a semicolon in the wrong place is similar to placing an `endif` (in PB) in the wrong place. In Delphi, this type of error is much more difficult to find.

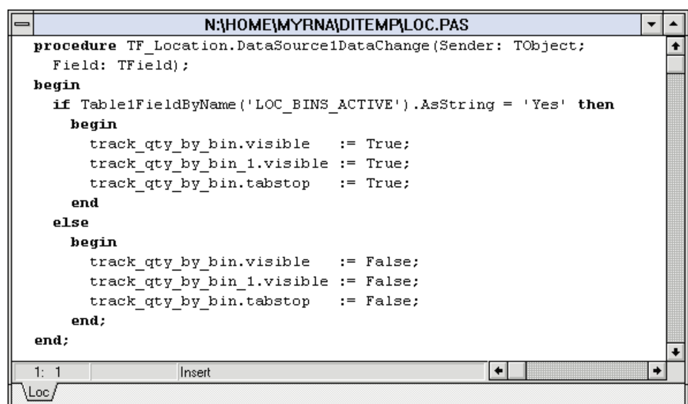


Figure 11: An example of an Object Pascal **if** statement. Notice the first **end** keyword is not followed by a semicolon.

Large project support is not currently available in Delphi. You can put modal windows in DLLs which limits you to search windows, message boxes, and other inconsequential windows. Your more important main interface window (MDI Child) cannot reside in a DLL. This is best described by an excerpt from one of the demo programs shipped with Delphi: "Note that the TDllForm1 form is always used modally. Borland does not recommend using a modeless form from a DLL because the *OnActivate* and *OnDeactivate* mechanisms do not work when control is transferred between a DLL-owned form and another DLL or EXE owned form." In English, this translates to: "The form called from a DLL does not recognize the parent form in the EXE as its (the DLL form's) parent form." Theoretically, it is possible for two modeless windows to have focus at the same time. This DLL problem needs to be fixed, or Delphi needs to support an old-but-reliable way to split up an EXE file — libraries.

Object Pascal takes some getting used to. For example, the Object Pascal **case** statement only supports ordinal variable types. It's disappointing that Object Pascal **case** statements don't support string variables. Also, as you program events in the source code file, the events are added at the end of the file. When you get 2000 lines of code, you are jumping all over the file trying to find related code (the code compiles fine). The colon/equal sign combination (:=) is the assignment operator, while the equal sign (=) is strictly a comparison operator. On the positive side, once you become comfortable with the different syntax, Object Pascal is very robust. In addition, there is a lot of available shareware and third-party support to extend Object Pascal's capabilities.

Conclusion

On paper, PB *should* easily outclass Delphi. It's a well-rounded development environment with easy-to-use, intuitive tools (i.e. its Painters). Once you start peeling back the pretty face, however, PB doesn't look as good. Slow database access, partial support for some databases, a slow p-code interpreter, compiler problems, and events that don't run properly (or at all), are just some of the system-wide problems. In short, PB doesn't need more functionality to capture the hearts of programmers. What's already there just needs to be fixed!

Delphi is an incomplete development environment and is missing a database structure maintenance tool. The included report writer is average at best and the SQL-related tools are complex and difficult to use. However, Delphi's core programming elements (Windows layout, programming language, and database access) are excellent. I would pit Delphi's capabilities in these areas against any other tool on the market. Compared to PB's system-wide problems, Delphi only has five weaknesses in its core area that need to be addressed:

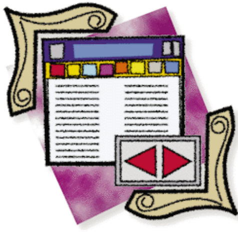
- *TQuery* objects should initiate faster.
- The Query component must be more flexible and allow for procedural SQL sub-routines and database cursors.
- Large projects must be supported better by allowing large EXEs to be divided into libraries, or DLLs should be allowed to recognize the forms in EXEs as the parent. This will make systems easier to manage, and increase code reuse.
- A more well-rounded grid component.
- Better OLE support.

Also, PB's reporting capability is good and Delphi's is average at best. (Crystal Reports by Crystal is an excellent reporting tool and can be used with either product.) I expect some of the other holes to be filled when the next version of Delphi is released. Meanwhile you can use PB's Database Painter until Delphi has a similar utility.

If your SQL is only average and you rely on wizards to help with SQL statements, get a book and learn how to program in SQL. No matter how good the wizard is, you will eventually have to write a complex SQL statement by yourself.

There isn't a complete enterprise development tool available, but for core database programming prowess, Delphi is the new "king of the hill". It's fast, provides easy database access, a completely object-oriented programming environment, and features reliable code generation and complete extensibility. ∆

Thomas Miller is President of Business Software Systems, Inc., a consulting firm specializing in implementing accounting, distribution, manufacturing, and business management systems. They are currently working on their own distribution system written in Delphi and supporting Oracle, Sybase, and Btrieve back-ends. You can reach Thomas at 76652,2065.



DBNavigator

Delphi / Object Pascal

By Cary Jensen, Ph.D.

Data Validation: Part II

Delphi's Data-Aware Components

Last month's article introduced the basics of data validation, and explored how to apply validation to edit controls not associated with database tables. This month's "DBNavigator" continues this discussion by considering the issues that apply to data-aware components, that is, those that permit you to edit tables.

Table-Level Validation

Validation can be produced in a number of ways when data-aware controls are involved. The first line of defense against invalid data lies in the data tables themselves. At a minimum, the data type of the individual fields in a table prohibit data of an incompatible type from being entered.

For example, a table will not allow letters to be entered into a field when the field type is numeric. Likewise, fields that are of the type Date or Time require that the entered data conform to particular characteristics. By comparison, an Edit component accepts any type of alphanumeric data.

The validation provided by table field type is provided by descendants of the *TField* component. These types can easily be seen when you instantiate (create) fields using the Fields Editor. Figure 1 shows the VendNo field selected in the Fields Editor. This field also appears in the Object Inspector, where you will notice that it is a *TFloatField* component. By default, this component accepts only numeric values.

Even when you do not explicitly instantiate your fields, Delphi represents the fields using *TField* components. Like those you instantiate, the purpose of these components is to provide the first line of defense against obviously invalid data.

However, the data type of the fields in a table provide only limited protection from unacceptable data. For example, while a field may have a data type of Date and a name of DateOfBirth, unless further steps are taken, any valid date can be entered. For instance, if the table is designed to hold the names of a company's current employees, it's unlikely that the date 12/31/1065 would be considered acceptable.

Most table formats provide for further restrictions to be placed on entered data. Paradox tables, for example, have a feature called *validity*

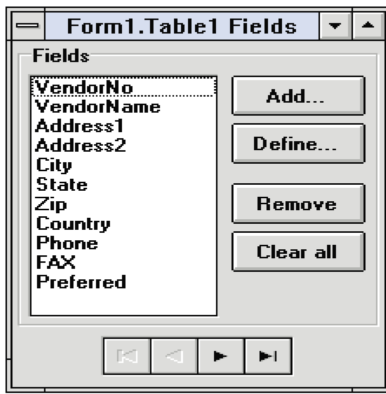


Figure 1: When a field is instantiated, a *TField* descendant is created to represent the field. These *TField* components provide default validation based on field type.

checks. Using validity checks you can specify that dates earlier than January 1, 1910 (for example) cannot be entered into the DateOfBirth field.

SQL tables provide you with even greater control over acceptable values for fields in the form of *triggers*. A trigger is a mechanism on the database server that executes a custom code routine in response to certain events. For example, you can create a trigger that will execute a validation routine when any SQL client (such as Delphi) attempts to post the contents of a changed record (i.e. save the record to the table). Furthermore, if the code executed by the trigger determines that the contents of the record being posted are invalid, the record is rejected and no changes are made to the database.

The advantages of table-based validation are many. First, the validation is stored with the table to which it applies. Consequently, every application that has access to the data is affected. Second, table-based validation often means that it is unnecessary to add additional validation to controls on forms. This is particularly advantageous when the same tables appear on many different forms in an application.

Table-based validation impacts Delphi applications through *exceptions*. When Delphi attempts to post a new or modified record, the record will only be posted if the record passes all table-based validation. If the record cannot be posted, an exception is raised. Figure 2 displays an exception created when a record violates a minimum value validity check for a Paradox table.

The exception shown in Figure 2 deserves additional comment. Paradox for Windows developers have come to expect that field-level validation will occur when a field is departed (except for Required validity checks that are applied when the record is posted). In Delphi applications these validity checks are not tested until the record is being posted, even when Paradox tables are being used.

Field-Level Validation

There are two basic approaches to field-level validation in Delphi applications. One is to set the properties of the field so invalid data cannot be entered, and the second is to use code to check the data after it has been entered. These approaches are rarely exclusive. You will often find yourself applying both techniques, sometimes even to the same field.

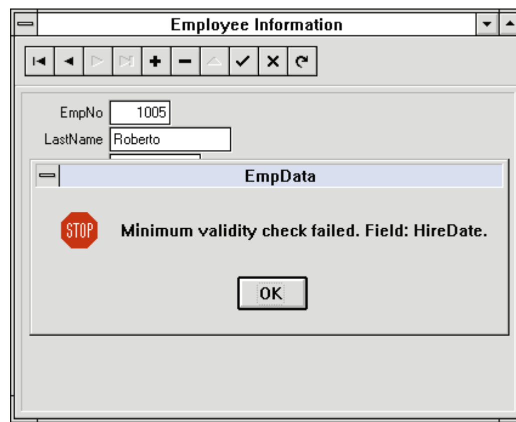


Figure 2: An exception raised by Delphi when an attempt is made to post a record containing invalid data to a table.

One characteristic that both these techniques share is their need for you to instantiate (create) the field object. As discussed in a previous article [Cary Jensen's "DBNavigator" column in the June 1995 *Delphi Informant*], this is done using the Fields Editor.

The following example demonstrates both types of field-level validation. It makes use of the \DBDEMOS files installed with Delphi. If you want to follow along with this demonstration but do not have the files on your computer, you should reinstall Delphi using a Custom installation to install only the sample files.

Begin by creating a new project. On the form, place the following components: a DataSource, Table, DBGrid, and DBNavigator. The form you create may resemble the one in Figure 3.

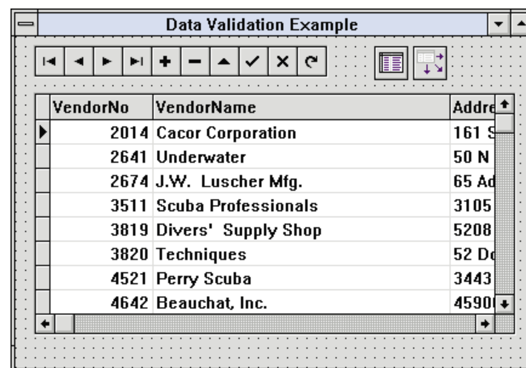


Figure 3: A new form for validation demonstration.

Select the DataSource component and set its *DataSet* property to *Table1*. Select the Table component and set its *DatabaseName* property to `DBDEMOS` (the alias defined by Delphi when you install the sample files). Set the *TableName* property to `VENDORS.DB`, and the *Active* property to *True*.

Now select the DBGrid and set its *DataSource* property to *DataSource1*. The contents of the Vendors table should appear in the DBGrid. Finally, select the DBNavigator and also set its *DataSource* property to *DataSource1*.

It is now time to instantiate the fields of the Table component. Double-click the Table component to display the Fields Editor. (Alternatively, you can select the Table compo-

nent, right-click it, and select **Fields editor** from the pop-up menu.) Select the **Add** button from the Fields Editor dialog box. When the Add Fields dialog box is displayed, all fields should be highlighted. Select **OK** to instantiate a field object for each of the fields in the Vendor table.

Validating Fields Using Properties

The primary properties that you use to control field-level validation are *EditMask* and *Required*. You can use the *EditMask* property to provide an input template when the field requires data in a particular format. This usually applies only to fields that hold highly structured data such as dates, times, zip codes, phone numbers, and other similar data. On the other hand, the *Required* property can be used by any field that accepts user input.

To demonstrate the use of both the *EditMask* and *Required* properties, use the Object Inspector to select the object named `Table1Phone`. Set the *EditMask* property to the following value:

```
!000-000-0000;1;_
```

In this example you entered the *EditMask* property directly. Instead, you could have opened the property editor for the *EditMask* property by clicking the ellipsis that appears when the *EditMask* property is selected. The *EditMask* property editor is the Input Mask Editor dialog box shown in Figure 4.

Using this property editor you can select from a set of predefined edit masks. By clicking the **Masks** button on this dialog box you can load a country-specific mask file (.DEM) for international applications.

Whether you create a mask, or use one from the Input Mask Editor, the *EditMask* has three parts, each separated by semicolons. In the first part of this edit mask, the exclamation point, specifies that leading blanks will not be saved. The 0 character specifies that a number is required in that position, while the dash character (-) specifies that a dash will appear in those exact positions in the field.

The second part of this *EditMask* can contain only the number 0 or 1. If the number 0 appears there, literal characters within the mask appear in the field, but are not saved to the underlying table. When a 1 appears in this position (as it does in the example mask) the literal characters are saved to the table.

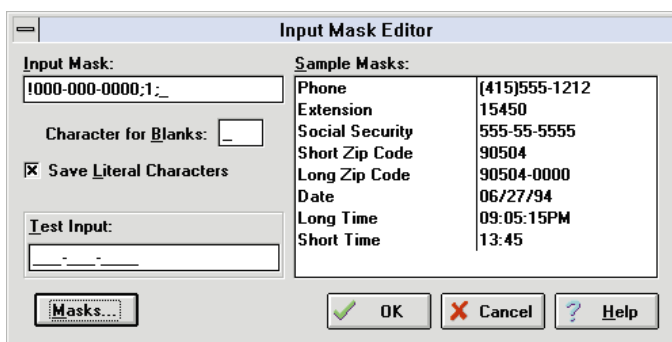


Figure 4: The Input Mask Editor dialog box.

The final position includes a single character. This character is used in the mask to refer to blank spaces. The underscore character (_) is identified as the blank character in the mask used in this example. (Note that the third part of the mask is required even though this particular mask does not contain blank characters.)

Select the `Table1VendorName` object from the Object Inspector. Set the *Required* property to *True*. When the *Required* property is set to *True*, the field cannot be left blank. If you do leave the field blank and then attempt to post the record, Delphi will generate an exception.

Beware that using the *Required* property for an instantiated field does not alone ensure that valid data will be entered. Let's say a field includes an EditMask component that contains literal characters (such as the dash characters in the mask we're using in this example). If the user changes that field, but then erases the entered characters, Delphi does not consider the field to be blank since the literal characters appear there. Consequently, although a valid value has not been entered, and the field looks as if no value was entered, the *Required* property will not produce an exception.

You're done. Compile the form and run it. Press Insert to insert a new record. Enter the value 1 in the VendorNo field. Next, press ↓ to attempt to leave the record. An exception is generated by the *Required* property as shown in Figure 5. (Note: If you run this from the integrated development environment — the IDE — and **Break On Exception** is checked on the Preferences page of the Environment Options dialog box, press F9 to continue running after the exception.)

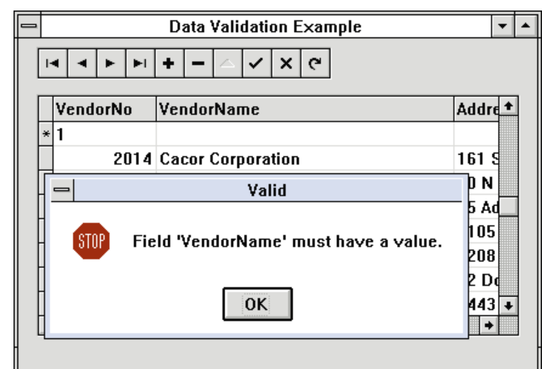


Figure 5: When you leave the VendorName field blank and then attempt to move to or insert another record, this exception is raised. This occurs because this field's *Required* property is set to *True*.

Following the exception, enter a name in the VendorNo field. Now press Tab until you arrive at the Phone field. Notice that a mask appears in this field. Enter the 10 characters of a phone number, beginning with an area code. Now press ↓ to leave the field. Since both the *Required* property on *Table1VendorName* field and the *EditMask* property in the *Table1Phone* field are satisfied, the new record is posted. Close the form to return to Delphi.

Validating Fields from Event Handlers

Validating fields using event handlers is more direct when you are using data-aware components instead of an Edit component. (An Edit component is a text box that is not associated with a field in a table.) An Edit component provides only a few events that are appropriate for attaching validation code, with the *OnExit* event being the best (although less than ideal) solution.

When it comes to field validation, data-aware field objects are easier to use. This is because instantiated fields possess an event handler that best represents the data validation needs of developers. This event handler is *OnValidate*.

The *OnValidate* event triggers whenever Delphi attempts to accept a value entered by the user. You can attach code to this event, and explicitly test the value that the user has entered into the field. If your code determines that the value is invalid, you can prevent the value from being accepted by raising an exception.

In most cases, you will raise an exception to prevent a value from being accepted in one of two ways. The first is to use the **raise** keyword to create a new exception, and the second is to create a silent exception using the *Abort* procedure. The **raise** keyword is used in the following demonstration.

When you use **raise** within an *OnValidate* event handler, you can create an exception that will display an error message and prevent the field from being accepted. While there are a number of ways of doing this (including techniques requiring that you define an exception object, as demonstrated in last month's "DBNavigator"), the easiest way is to use **raise** with the *Create* method of the exception class. This can be accomplished with the following line of code:

```
raise Exception.Create('Your error message goes here')
```

Begin by selecting `Table1VendorNo` from the Object Inspector. Go to the Events page of the Object Inspector and select *OnValidate*. Enter the following code into the *OnValidate* event handler:

```
if Table1VendorNo.Value < 1 then
  raise Exception.Create('Positive Vendor number
    required');
```

Run the program, insert a new record, and enter a value of less than one (e.g. -1) in the VendorNo field. When you press Tab to leave the field the exception is generated, displaying the message shown in Figure 6. (Again, if you're running this code from the Delphi IDE, the above note regarding **Break On Exception** applies.)

Record-level validation is applied before the entire record is posted to a table. As mentioned earlier, an exception is raised automatically if the record being posted violates one or more requirements of the table to which the record is being posted. You can also use custom code to evaluate the record, and raise an exception (silent or otherwise) if your code determines that the record is invalid.

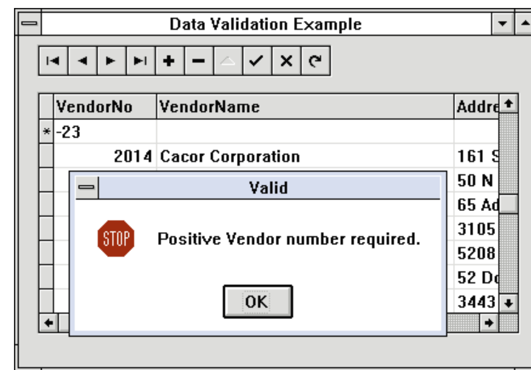


Figure 6: An exception raised by an *OnValidate* event handler.

Like the preceding demonstration of field-level validation, record-level validation involves using event handlers. For our example, we'll use the *BeforePost* event handler, which belongs to all descendants of the *TDataSet* class (this includes Table, Query, and StoredProc components). The code you add to the *BeforePost* event handler is used to evaluate the fields (or other related controls) on the form. If your code determines that the record is not valid, generating an exception prevents the record from being posted.

The following example demonstrates this technique. In this case, however, the use of a silent exception is demonstrated. Begin by selecting the Table component. Then, from the Events page, open the *BeforePost* event handler and add the following code:

```
if Table1Address1.Value = '' and
   Table1Address2.Value <> '' then
begin
  MessageDlg('Address1 cannot be blank',
       mtError,[mbOK],0);
  Abort;
end;
```

Compile the program and run it. Insert a new record, entering a positive vendor number and a vendor name. Then press Tab to move to the field Address2 and enter an address. Next, attempt to leave the record by pressing ↓ or Insert. This will trigger the *BeforePost* event handler. Your code will detect that Address1 is blank and that Address2 is not, and will display the error dialog box shown in Figure 7. After you accept this dialog box, the *Abort* procedure generates a silent exception.

Like a raised exception, a silent exception prevents the record from being posted. The main difference is that a silent exception does not display a dialog box. In this example, a custom dialog box was displayed using the *MessageDlg* function. The primary advantage of raising a silent exception is that you can use any dialog box, and even associate a custom dialog box with a help context from your application's help file. When you raise an exception, the standard Delphi exception dialog box is displayed, and you have no control over its appearance and help context.

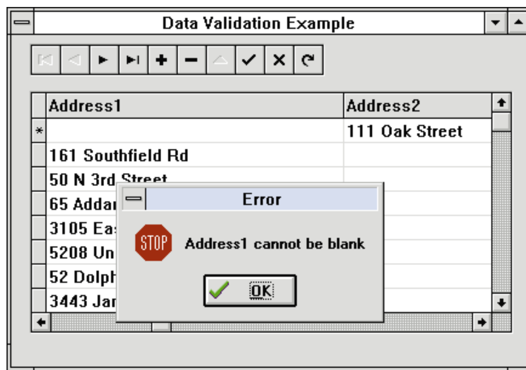


Figure 7: Code in the *BeforePost* event handler displays an error message.

Form-Level Validation

Form-level validation is performed on the entire form, not just a single field or record. Most commonly, this type of validation is employed when the user has finished entering data into the form. Like field-level and record-level validation, form-level validation is performed using event handlers. Specifically, this takes place in the form's *OnCloseQuery* event handler.

Before continuing, it's important to note that closing a form does not destroy it. It only makes the form invisible to the user. (You destroy a form using either the *Release* or *Free* methods.)

The objects on a closed form can still be accessed by other forms in your application. Typically, the code you place in the *OnCloseQuery* event handler ensures that this data is valid, and therefore usable by forms that may read the data from the closed form.

Unlike the *OnValidate* and *BeforePost* event handlers, you do not raise an exception if your validation determines that the form should not be closed. Instead, you assign the value *False* to the Boolean formal parameter *CanClose*, which is passed by reference to the *OnCloseQuery* event handler. The default value for this parameter is *True*.

The following demonstrates a simple use of the *OnCloseQuery* event handler. In this case, the only validation that occurs is when the user confirms his or her intention to close the form. Begin by selecting the form's *OnCloseQuery* event handler from the Object Inspector. Enter the following code:

```
if MessageDlg('Close this form',mtConfirmation,
              [mbYes,mbCancel],0) <> mrYes then
   CanClose := False;
```

Compile the form and run it. Next, double-click the Control menu to close the form. This will trigger the form's *OnCloseQuery* event handler, and the dialog box shown in Figure 8 will be displayed. If you select **Yes**, the form will close. Otherwise, the code in the event handler assigns the value *False* to *CanClose*, and the form remains open.

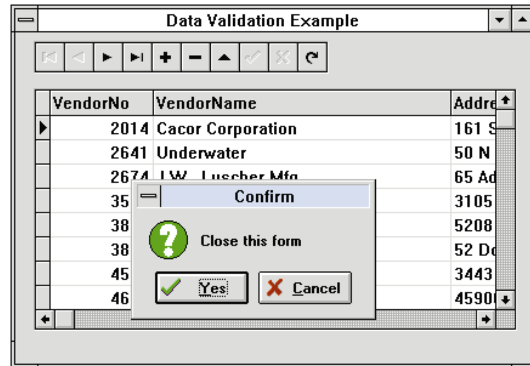


Figure 8: This Confirm dialog box asks the user to confirm a request to close a form.

Conclusion

Data validation is an important part of your Delphi applications. Whenever possible, you should use the validation offered by your data tables, and the properties of your form's components to ensure that entered data is valid.

When this does not provide the level of validation that you require, you can use the valuable event handlers that Delphi provides to add this essential capability to your applications. ∆

The demonstration project referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\SEP\CJ9509.

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

OP BASICS

DELPHI / OBJECT PASCAL

By Charles Calvert

Strings: Part II

Stripping and Manipulating Object Pascal Strings

Last month we introduced Object Pascal strings, and began to discuss the basics of string manipulation — searching for a substring within a string, and parsing lengthy strings. As promised, in this month's installment, we'll talk about stripping blanks from the end of a string, and functions for manipulating strings.

Stripping Blanks

A classic problem is the need to strip blanks off the end of a string. Consider the code fragment in Figure 1.

At first glance you might expect this code to print the number 2.03 to the screen. However, it will not because *Str2Real* cannot handle the extra spaces appended after the characters 2.03. It's quite likely that a problem similar to this could occur in a real-world program. For instance, a programmer might ask the user to enter a string, and the user may accidentally append a series of blanks to it (or perhaps the extra characters were added by other means). To ensure that your program will run correctly, you must strip those extra blank characters from the end of the string.

```
uses
  MathBox;

procedure TForm1.Button1Click(Sender: TObject);
var
  S: string;
  R: Real;
begin
  S := '2.03       ';
  R := Str2Real(S);
  WriteLn(R:2:2);
end;
```

```
function StripBlanks(S: string): string;
var
  i: Integer;
begin
  i := Length(S);
  while S[i] = ' ' do begin
    Delete(S,i,1);
    Dec(i);
  end;
  StripBlanks := S;
end;
```

Figure 1 (Top): In this code example, the first line in the **begin** section contains characters that need to be stripped. **Figure 2 (Bottom):** The *StripBlanks* function.

The *StripBlanks* function (see Figure 2) can be used to remove space characters from the end of a string. *StripBlanks* will not change the string that you pass into it, but creates a second string that it passes back to you as the function result. This means you must use this function in the following manner:

```
S2 := StripBlanks(S1);
```

where *S1* and *S2* are both strings. You can also write code that looks similar to this:

```
S1 := StripBlanks(S1);
```

StripBlanks has one local variable, `i`, which is an integer:

```
var
  i: Integer;
```

This variable is set to the length of the string passed to the function:

```
i := Length(S);
```

The *Length* function is one of the more fast and simple routines in the Object Pascal language. In effect, it does nothing more than this:

```
function Length(S: string): Integer;
begin
  Length := Ord(S[0]);
end;
```

In short, it returns the value of the length byte that is the first character of a string. [See last month's article for a discussion of the length byte.] The next line in the *StripBlank* function checks the value of the last character in the string under investigation:

```
while S[i] = ' ' do begin
```

More explicitly, it checks to see whether it is a blank. If it is a blank, the following code is executed:

```
Delete(S,i,1);
Dec(i);
```

The built-in *Delete* function takes three parameters. The first is a string, the second is an offset into the string, and the third is the number of characters you want to delete from the first parameter. In this case, if you passed in the string `'Sam   '`, which is the name "Sam" followed by three spaces, the last space would be lopped off so that the string would become `'Sam  '`, where "Sam" is followed by two spaces.

The function then decrements the value of `i` and returns to the top of the loop to see if the next character is a space:

```
while S[i] = ' ' do begin
```

If the next character is a space, it is also deleted from the end of the string. The entire process is repeated until the last character in the string is no longer a space. At that point, the function ends and a string is returned that is guaranteed not to have any spaces appended to the end of it.

Becoming familiar with functions such as *StripBlanks* is essential for all serious programmers. It isn't really that this one particular function is so important (although I do use this function fairly often). What *is* important is that *StripBlanks* is the kind of function that solves a common programming problem, and that it does so by manipulating a chunk of data — byte by byte.

Date-Based File Names

Now let's turn our attention to another Delphi function that manipulates strings. This function is not likely to be used daily by most programmers, but when you do need it, it's very handy. Programmers often end up making reports or gathering data on a daily basis. For instance, I sign onto an on-line service nearly every day, and frequently need to store the information I glean from cyberspace in a file containing the current date.

In other words, if I sign onto CompuServe and download the current messages from the Delphi forum, I don't want to store that information in a file called DELPHI.CIS. I want a file name that includes the current date, so I can easily tell what files were downloaded on a particular day. In short, I want to automatically generate file names that look like this: DE022595.TXT, PA022695.TXT, DE022795.TXT, and so on, where 022795 is a date of the type *MMDDYY*.

The *GetTodayName* function (see Figure 3) fits the bill. This function takes two parameters: a two-letter prefix, and a three-letter extension, and creates a file name of the type we've just discussed. The function begins by calling the built-in Pascal function *GetDate*, which returns the current year, month, day, and day of week as Word values. If the date were Tuesday, March 25, 1994, the function would return the following:

```
Year        := 1994
Month       := 3
Day         := 25
Day-of-Week  := 2      { 0 = Sunday }
```

Assuming that the user of this function passed in `DE` in the *PRE* parameter, and `TXT` in the *EXT* parameter, it would be fairly easy to use the *IntToStr* function to create something like this:

DE3251994.TXT

```
{---------------------------------------------------
       Name: GetTodayName function
Declaration: GetTodayName(Pre, Ext: string): string;
       Unit: StrBox
       Code: S
       Date: 03/01/94
Description: Return a filename of type PRE0101.EXT,
             where PRE and EXT are user supplied strings,
             and 0101 is today's date. PRE must not be
             longer than 2 letters.
---------------------------------------------------}
function GetTodayName(Pre, Ext: string): string;
var
  y, m, d, dow : Word;
begin
  GetDate(y,m,d,dow);
  Year := Int2StrPad0(y, 4);
  Delete(Year, 1, 2);
  GetTodayName := Pre + Int2StrPad0(m, 2) +
                  Int2StrPad0(d, 2) +
                  Year + '.' + Ext;
end;
```

Figure 3: The *GetTodayName* function.

There are several problems with this result, the biggest being that it is 12 characters long — too long for a legal DOS file name. To resolve the problems, we need to change the month to a number such as 03, to keep the day as 25, and to strip the 19 from the year:

DE032594.TXT

To achieve this end, *GetTodayName* needs a special function that will not only convert a number to a string, but also pad it with an appropriate quantity of zeros. The *Int2StrPad* function is what we need (see Figure 4).

```
{---------------------------------------------------
       Name: Int2StrPad0 function
Declaration: Int2StrPad0(N: LongInt; Len: Integer): string;
       Unit: MathBox
       Code: N
       Date: 03/01/94
Description: Converts a number into a string and pads
             the string with zeros if it is less than
             Len characters long.
---------------------------------------------------}
function Int2StrPad0(N: LongInt; Len: Integer): string;
var
  S : string;
begin
  Str(N:0,S);
  while Length(S) < Len do
    S := '0' + S;
  Int2StrPad0 := S;
end;
```

Figure 4: The *Int2StrPad* function.

This very useful function first uses the built-in Pascal routine called *Str* to convert a LongInt into a string. If the string that results is longer than *Len* bytes in length, the function simply exits and returns the string. However, if the string is less than *Len* bytes, the function appends zeros in front of it until it is *Len* characters long. Here is the transformation caused by each successive iteration of the **while** loop if *N* equals 2 and *Len* equals 4:

```
2     { First iteration }
02    { Second iteration }
002   { Third iteration }
0002  { Fourth iteration }
```

The function checks to see if the string is four characters long. If it isn't, the function adds a zero to the beginning of the string:

```
S := '0' + S;
```

In the case of the *GetTodayName* function, the value passed in the *Len* parameter is 2, because we want to translate a number such as 3 or 7 into a number such as 03 or 07.

The final trick in the *GetTodayName* function is to convert a year such as 1994 into a two-digit number such as 94. Clearly, this can be easily achieved by merely subtracting 1900 from the date. However, that sound of hoofbeats in the distance is the rapid approach of the year 2000.

Subtracting 1900 from 2001 would not achieve the desired result. The code therefore first converts the year into a string, then simply lops off the first two characters with the *Delete* function:

```
Year := Int2StrPad0(y,4);
Delete(Year,1,2);
```

In this case, a 4 is passed to *Int2StrPad0*, because the year was originally a four-digit number.

As mentioned earlier, the *Delete* function is built into the Delphi language. It deletes characters from a string, starting at the offset specified in the second parameter. The number of characters to be deleted is specified in the third parameter. (You can search on "Delete" in Delphi's on-line help for more details.)

If you step back now and view the *GetTodayName* function as a whole, you can see that it allows you to pass in the first two letters and the extension for a file name. In return, it supplies you with a string containing the prefix and extension, plus the current date. It's great to have functions like this available to you when you need them.

Using the Move and FillChar Functions

The next two built-in Delphi methods we'll examine are fast and powerful. Speed of this sort is a luxury, but it comes replete with some dangers that you must be sure to sidestep. In particular, neither the *FillChar* nor the *Move* function has much in the way of built-in error checking. *FillChar* is usually used to initialize an array, record, or string. It will, however, fill a structure not only with zeros but with whatever character you specify.

FillChar takes three parameters. The first is the variable you want to copy bytes into, the second is the number of bytes you want to fill, and the third is the character you want placed in those bytes:

```
procedure FillChar(var X; Count: Word; value);
```

Consider the following array:

```
var
  MyArray: array[0..10] of Char;
```

Given this array, the following command will set all the members of this array to #0:

```
FillChar(MyArray,SizeOf(MyArray),#0);
```

If you want to fill the array with spaces, you could use the following statement:

```
FillChar(MyArray,SizeOf(MyArray),#32);
```

This code would fill the array with the letter “A”:

```
FillChar(MyArray,SizeOf(MyArray),'A');
```

The key point to remember when you're using *FillChar* is that the *SizeOf* function can help you ensure that you are writing the correct number of bytes to the array. The big mistake you can make is writing too many bytes to the array — this is much worse than writing too few. If you think of the memory theater example we covered in last month's article, you can imagine ten members of the audience sitting together, all considering themselves part of *MyArray*. Right next to them are two people who make up an integer. They are busy remembering the number 25. Now you issue the following command:

```
FillChar(MyArray, 12, #0);
```

All the people who are part of *MyArray* will start remembering #0, which is fine. However, the command will keep right on going past the members of *MyArray* and tell the two folks remembering the number 25 that they should both now remember #0. In other words, the Integer value will also be “zeroed out” as well, and a bug has been introduced into your program. You should understand that the result described here is a best-case scenario. The worst case scenario is that the extra two bytes belong to another program. This means that your program will generate a General Protection Fault (or GPF). The moral is that you should always use the *FillChar* procedure with care.

Copy or Move It

A function similar to *FillChar* is called *Move*. Its purpose is to move a block of data from one place to another. A typical use of this function might be to move one portion of a string to a second string, or to move part of an array into a string. The *Copy* function can also be used for similar purposes. The advantage of the *Copy* function is that it is relatively safe. The disadvantages are that it is less flexible, and can be slower under some circumstances.

Move takes three parameters. The first is the variable you want to copy data from, the second is the variable you want to move data to, and the third is the number of bytes you want to move:

```
procedure Move(var  Source, Dest; Count: Word);
```

The following code is an example of a typical way to use *Move*. If you enjoy puzzles, you might want to take a moment to see if you can figure out what it does:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S1,S2: string;
begin
  S1 := 'Heebee Gee Bees';
  Move(S1[12], S2[1], 4);
  S2[0] := #4;
  Edit1.Text := S2;
end;
```

This code first sets `S1` to a string value. It then indexes 12 bytes into that string and moves the next four bytes into a second string. Don't forget to count the spaces when you are adding the characters in a string. (And don't forget that the first byte is the length byte.) Finally, it sets the length byte of the second string to `#4`, which is the number of bytes that were moved into it. After executing this code, the final statement assigns the word “Bees” to *Edit1.Text*. This statement accomplishes the same task using the *Copy* function:

```
S1 := 'Heebee Gee Bees';
S2 := Copy(S1, 12, 4);
WriteLn(S2);
```

The first parameter to *Copy* is the string you want to get data from, the second is an offset into that string, and the third is the number of bytes you want to use. The function returns a substring taken from the string in the first parameter.

The *Copy* function is easier to use and safer than the *Move* function, but it is not as powerful. If at all possible, you should use the *Copy* function. However, there are times when you can't use the *Copy* function, particularly if you need to move data in or out of at least one variable that is not a string. Also, it is worth remembering that *Move* is very fast. If you have to perform an action repeatedly in a loop, you should consider using *Move* instead of *Copy*.

As easy as it is to write data to the wrong place using the *FillChar* statement, you will find that the *Move* statement can lead you even further astray in considerably less time. It will, however, rescue you from difficult situations, provided you know how to use it.

Moving On

The following function puts the *Move* procedure to practical use. As its name implies, the *StripFirstWord* function (see Figure 5) is used to remove the first word from a string. For instance, it would change the following string: `'One Two Three'`, into `'Two Three'`.

The first line in this function introduces you to the built-in *Pos* (position) function, which locates a substring in a longer string. In this case for instance, the *Pos* function is used to find the first instance of the space character in the string passed to the *StripFirstWord* function. The function returns the offset of the character it is looking for.

More specifically, the *Pos* function takes two parameters. The first is the string to search for, and the second is the string you want to search. Therefore the statement `Pos(#32,S)` looks for the space character inside a string called `S`.

If you passed in this line of poetry — “The pure products of America go crazy” — the *Pos* function would return the number 4, which is the offset of the first space character in the sentence. However, if you passed in a simpler string such as

```
{---------------------------------------------------
       Name:  StripFirstWord function
Declaration:  StripFirstWord(S : string) : string;
       Unit:  StrBox
       Code:  S
       Date:  03/02/94
Description:  Strip the first word from a sentence,
              return the shortened sentence. Return original
              string if there is no first word.
---------------------------------------------------}
function StripFirstWord(S : string) : string;
var
  i, Size: Integer;
  S1: string;
begin
  i := Pos(#32, S);
  if i = 0 then begin
    StripFirstWord := S;
    Exit;
  end;

  Size := (Length(S) - i);
  Move(S[i + 1], S[1], Size);
  S[0] := Chr(Size);
  StripFirstWord := S;
end;
```

Figure 5: The *StripFirstWord* function.

"Williams", *Pos* would return 0 because there is no space character in the string. If the function does not find a space character in the string, it returns an empty string:

```
if i = 0 then begin
  StripFirstWord := '';
  Exit;
end;
```

The built-in *Exit* procedure simply exits the function without executing another line of code. This is the *StripFirstWord* function's sole, and rather limited, exercise in error checking.

If the offset of a space character is returned by the *Pos* function, the *Move* function transfers an "offset" number of characters from the string that is passed in to a local string named S1:

```
i := Pos(#32, S);
  ...
Move(S[1], S1[1], i);
S1[0] := Chr(i-1);
```

The second line of code sets the length byte for the newly created string that contains the first word in the sentence. The next three lines of code excise the first word from the original sentence:

```
Size := (Length(S) - i);
Move(S[i + 1], S[1], Size);
S[0] := Chr(Size);
```

The first step is to determine the number of characters in the sentence after the first word is removed. This is found by subtracting the number returned by *Pos* from the total length of the sentence. *StripFirstWord* then moves the remaining portion of the string from a position "offset" characters deep in the following string:

```
She was a child and I was a child, In a kingdom by the sea
```

to the very first spot in the string:

```
was a child and I was a child, In a kingdom by the sea sea
```

The extra characters — represented in this case by the second occurrence of "sea" — are then lopped off by setting the length byte to the appropriate number of characters:

```
was a child and I was a child, In a kingdom by the sea
```

The function then returns the first word of the sentence, and also the shortened sentence.

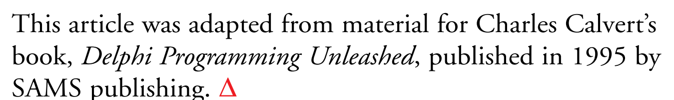
The *StripFirstWord* function is not perfect. For instance, some readers may have noticed that the function would not perform as advertised if the first characters of the string passed to it were spaces. Overall, however, it does the job required of it.

Of course, you could write a function that strips spaces from the beginning of a string. Then you could first pass a string to the new function you created, and then pass it on to the *StripFirstWord* function. (We'll do this in the next article of this series.)

Conclusion

Next month, we'll talk more about the *StripFirstWord* function, and explore parsing the contents of a text file and converting the data into fundamental Delphi types.

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



Delphi projects that demonstrate the principles discussed in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\SEP\CC9509.

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

By *Richard D. Holmes*

A Stopwatch Component

Embedding Assembly Language in Object Pascal to Call the Windows Virtual Timer Device

The hallmark of Delphi is flexibility combined with performance. One example of this flexibility is Delphi's ability to combine a very high-level language, Object Pascal, with embedded assembly language and even in-line machine code. Using these features, I have built two high-speed software stopwatches.

Class *TStopWatch* is a simple stopwatch designed for timing external events. It can measure intervals as short as 25 microseconds (on a 486/66). Class *TProfilingStopWatch* is a specialized descendant of *TStopWatch* that is designed for profiling programs. By compensating for overhead, *TProfilingStopWatch* can measure intervals as short as 2-3 microseconds. In comparison, the shortest interval that can be measured with the DOS/Windows system clock is 55,000 microseconds.

Classes *TStopWatch* and *TProfilingStopWatch* are non-visual components that can be installed on the Delphi Component Palette. Using them is as simple as placing a stopwatch object on the form and then calling the methods: *Start*, *Stop*, and *ElapsedTime*. Two example programs are presented: one that times external events and one that profiles code performance.

Unit VTIMERDV

The Windows Virtual Timer Device (VTD) provides access to the PC's hardware timer within the confines of Windows' protected memory management system. The 8253 Programmable Interval Timer chip ticks at a rate of 1.196 MHz, giving the VTD a resolution of 0.836 microseconds. Calling a Windows' virtual device driver, however, requires assembly language routines. The routines used here are based on those of Rick Grehan in his article "The Software Stopwatch" (*Byte*, April 1995). Additional information can be found in "Timers and Timing in Microsoft Windows" on the Microsoft Developer's Network CD-ROM.

The Delphi unit VTIMERDV (see Listing Two on page 45) encapsulates the assembly language routines that are needed to call the VTD and to process the results. The public interface to unit VTIMERDV exports just one function, *VTD_GetTime*, which returns a 64-bit real number that represents the time in seconds since Windows was started. The implementation of the unit uses the private procedure *VTD_GetEntryPoint*. Noteworthy features in the implementation of VTIMERDV include the use of embedded assembly language, the use of embedded machine code, and the presence of an **initialization** section.

The procedure *VTD_GetEntryPoint* calls the Window software interrupt $2F to get the VTD's entry point. This address is stored in the variables *wVTDSegment* and *wVTDOffset*. (I've used a modified "Hungarian" convention for naming variables and constants. The first letter or letters of a variable's name represent its data type: *w* for Word, *l* for Longint, *d* for Double, *btn* for *TButton*, etc.) In the listing for this procedure, notice how the ability to reference Object Pascal variables within an **asm** block makes it easy to integrate assembly language routines into a Delphi program.

The function *VTD_GetTime* calls the VTD to get the number of hardware timer ticks that have elapsed since Windows was started and converts the returned value from ticks to seconds. The implementation of this routine requires a combination of embedded assembly language and in-line machine code. This is amazingly simple to do in Delphi, since the Object Pascal statements within the body of a procedure or function can be freely intermixed with **asm** statements and **inline** statements.

As shown in Listing Two, the implementation of *VTD_GetTime* consists of an **asm** block, followed by an **inline** block, followed by another **asm** block. The final result is a high-performance function that goes well beyond the native capabilities of Delphi, yet is implemented entirely within the Delphi environment.

The first **asm** block calls the VTD, which returns the lower 32 bits of the tick count in the EAX register and the upper 32 bits in the EDX register. This poses a real challenge. How can we access these 32-bit registers? Delphi's built-in assembler is a 16-bit assembler. It can manipulate the 16-bit registers AX and DX, but not their 32-bit counterparts EAX and EDX. The solution is to use in-line machine code. By using the $66 opcode prefix to toggle the operand size, we can access the 32-bit registers even though the executable itself is just a 16-bit program. Cool!

By transferring the EAX and EDX registers to adjacent memory locations, we can also emulate a 64-bit integer variable, even though the largest integer type that Delphi supports is the 32-bit Longint. The last instruction within the **inline** block pushes this 64-bit integer variable onto the coprocessor's floating point stack, where it is implicitly converted to an 80-bit real. The second **asm** block multiplies the tick count by the conversion factor *dSecondsPerTick*, stores the calculated time as a 64-bit real in the Object Pascal return variable *@Result*, and pops the coprocessor stack to clean up.

Important note: All of this works only on an 80386 or higher CPU with a hardware coprocessor or floating point unit.

Initialization sections are another innovation in the Delphi programming model. The **initialization** section of a unit contains code that is executed when the program is loaded, prior to the execution of any routines in the body of the unit. In unit VTIMERDV, the procedure *VTD_GetEntryPoint* must be called before the function *VTD_GetTime* can be used. However, *VTD_GetEntryPoint* only needs to be called once. It is therefore an ideal candidate to be placed in the unit's **initialization** section. The other action needed within the **initialization** section is to set the value of the conversion factor, *dSecondsPerTick*.

A minor shortcoming of Delphi's **asm** and **inline** statements is that you cannot reference Object Pascal constants symbolically, although you can reference variables by name. It was therefore necessary to declare the conversion factor, *dSecondsPerTick*, as **var** rather than **const** and to initialize its value.

Class TStopWatch

The class *TStopWatch* is a Delphi component that represents an electronic stopwatch. *TStopWatch* uses unit VTIMERDV in its implementation, but itself contains no traces of the low-level details that were the center of attention in VTIMERDV (see Listing Three on page 46). These details have been successfully encapsulated within VTIMERDV.

The methods that control a *TStopWatch* object are the familiar ones used to operate a stopwatch:

- *Start* starts the watch. If the watch is already running, *Start* does nothing. Note that starting the watch does not reset the accumulated time. This allows multiple sequences of *Start* and *Stop* to be used to measure the cumulative time for a family of related events without requiring the programmer to create variables and write code to perform the accumulation.
- *Stop* stops the watch and adds the time that has elapsed since *Start* to the accumulator. If the watch is already stopped, *Stop* does nothing.
- *Reset* stops the watch and resets the accumulated time to zero. It is not necessary to call *Reset* before using a *TStopWatch* object, since the constructor initializes all fields to their reset values.
- *ElapsedTime* reads the watch and returns the elapsed time in seconds. If the watch is stopped, it returns the accumulated time. If the watch is running, it returns the current "split time".
- *IsRunning* returns a Boolean value that indicates whether the watch is running.

Example 1

Class *TStopWatch* is a non-visual component. It should be installed using the standard procedures for customizing the Delphi component library (see Chapter 2 of the *User's Guide*). By default it installs itself on the System page of the Component Palette (see Figure 1). To create a *TStopWatch* object, simply drag a stopwatch from the palette and place it on the form. Then add the code to operate the stopwatch at the appropriate points in your application. Like other non-visual components, the *TStopWatch* object will be visible at design time, but not at run-time.

Example 1 (see Listing Four on page 48) uses two stopwatches that are controlled by buttons on the form, as shown in Figure 2. It shows how *TStopWatch* can be used to time external events. In this case the external events are just button presses, but the possibilities for timing external events are limited only by your ability to make those external stimuli visible to your program.

Figure 1 (Top): The *TStopWatch* and *TProfilingStopWatch* components installed on the Component Palette. **Figure 2 (Bottom):** The user interface for Example 1.

This example also shows that an application can use two or more stopwatches. Each stopwatch can be started and stopped independently and there are no restrictions on how many stopwatches can be running concurrently. This is because only one timer is running on the 8253 Timer Chip. These components are simply retrieving and storing numbers from this timer and then performing arithmetic on those values. This is *not* like using the *TTimer* component in Windows that consumes a software timer from a limited pool of timers that Windows manages.

Class *TProfilingStopWatch*

Class *TProfilingStopWatch* is a specialized descendant of *TStopWatch* that is designed for profiling the execution of Delphi programs. It automatically removes the overhead of calling the Windows VTD from the elapsed time. Figure 3 shows why this overhead should be removed from a stopwatch that times blocks off code, but not from a stopwatch that times external events.

The overhead in calling the Windows VTD can be divided into two parts: preparing to read the hardware timer and returning afterwards. In Figure 3, these are shown as intervals a and b, respectively. A program that triggers the stopwatch at time t_1 will actually read the hardware timer at time u_1 and will not regain control of the processor until time v_1.

For a stopwatch that is used to measure the interval between two external events, t_1 and t_2, no correction is needed for overhead. The measured interval $x = u_2 - u_1$ is equal to $t_2 - t_1$, since the lag time a is identical for both events. The shortest interval, x, that can be measured between two external events is therefore $b + a$. On a 486/66 PC, this is about 22 microseconds.

On the other hand, for a stopwatch that profiles code, the desired result is the interval between v_1 and t_2 , which represents the amount of processor time actually used by the code. Processor time used by the stopwatch should be excluded. The essence of class *TProfilingStopWatch* is to separately measure the overhead, $z = b + a$, and use it to calculate the interval, $y = u_2 - u_1 - z$, which is of equal duration to $t_2 - v_1$. In principle, the shortest interval that can be measured with a profiling stopwatch is one tick of the hardware timer (0.840 microseconds). In practice, the limiting factor is the uncertainty in calibrating the overhead, which appears to be on the order of 2-3 microseconds.

By comparison, the DOS system clock, which ticks only 18 times per second, has a resolution of 55 milliseconds (55,000 microseconds). Although the Windows API function **GetTickCount** returns the number of milliseconds since Windows was started, the value is only updated every 55 milliseconds. Therefore, its actual resolution is identical to that of the DOS clock. The resolution of *TStopWatch* is therefore over 2,000 times better than the system clock used by DOS and Windows, and the resolution of *TProfilingStopWatch* is nearly 20,000 times better.

To reiterate, these components should be applied as follows: Use *TStopWatch* to measure the interval between external events, but use *TProfilingStopWatch* to measure the processor time consumed by a block of code.

In addition to automatic calibration and removal of overhead, class *TProfilingStopWatch* also provides five methods that can be used to manually control the calibration for overhead. These calibration methods will be of interest primarily to users attempting to measure intervals that are short compared to the observed overhead:

- *CalibrateOverhead* measures the overhead of making 100 calls to *Start* and *Stop*, and stores the average value in the class variable *dOverheadForStop*. It also measures the overhead of making 100 calls to *ElapsedTime* to read a split time and stores the average value in the class variable *dOverheadForSplit*. *CalibrateOverhead* is called in the **initialization** section for unit StpWatch. It can also be called later to recalibrate the overhead.
- *OverheadForStop* returns the current value of *dOverheadForStop*.
- *OverheadForSplit* returns the current value of *dOverheadForSplit*.

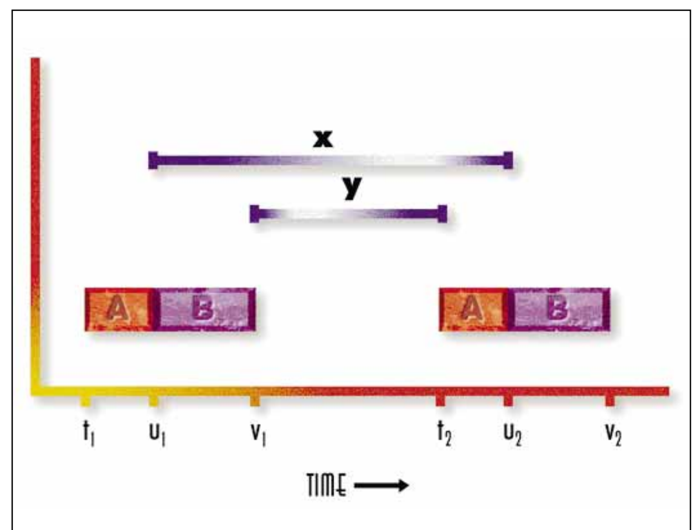


Figure 3: The difference between timing external events (x) and profiling code (y).

- *SetOverheadForStop* allows the value of *dOverheadForStop* to be set directly.
- *SetOverheadForSplit* allows the value of *dOverheadForSplit* to be set directly.

Note that these calibration methods are class methods and that the variables *dOverheadForStop* and *dOverheadForSplit* are class variables, not instance variables. At any point in time, all *TProfilingStopWatch* objects within a program use the same values of *dOverheadForStop* and *dOverheadForSplit*. This ensures that differences in the measured overhead will not bias one stopwatch relative to another. The overhead corrections will be consistent for all instances and the measured times can be freely compared against one another. Of course, Object Pascal doesn't directly support Smalltalk-style "class variables", but with the additional encapsulation provided by units, it's easy to fake it.

Example 2

Example 2 (see Listing Five on page 49) uses four profiling stopwatches to time the execution of three blocks of code, and to compare their execution time against the total time for the enclosing procedure. It illustrates the use of repeated calls to *Start* and *Stop* to accumulate the total time for a family of related events, the use of *ElapsedTime* to obtain split times, and the use of the *TProfilingStopWatch* calibration methods.

Figure 4 shows the user interface for this program. The **Run** button executes the test procedure, *btnRunClick*, that is being profiled. The three blocks to be timed within this procedure are the "Empty Loop", the "Split Loop", and the "Work Loop". The total time is measured by *ProfilingStopWatch1*, the Empty Loop is timed by *ProfilingStopWatch2*, the Split Loop is timed by *ProfilingStopWatch3*, and the Work Loop is timed by *ProfilingStopWatch4*.

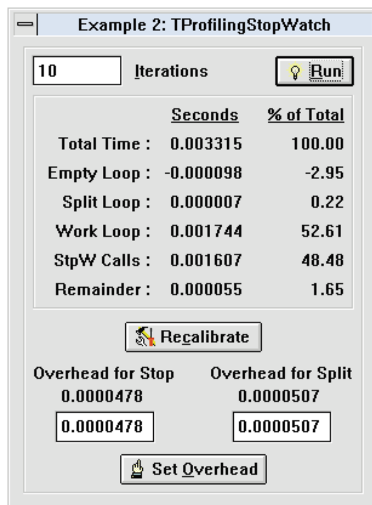


Figure 4: The user interface for Example 2.

The number of iterations for all three loops is controlled by the value entered in the **Iterations** edit box. The Work Loop, however, contains a nested inner loop, so that the amount of work that it performs is proportional to the *Iterations* parameter squared.

The Empty Loop does nothing but start *ProfilingStopWatch2* and then immediately stop it. Since the stopwatch is not reset between iterations, the reported time is cumulative. If the calibration for the overhead of stopping the watch was perfect, the measured time would be zero, regardless of the number of iterations. Dividing the accumulated time for the Empty Loop by the number of iterations gives an estimate of the accuracy in the calibration of *dOverheadForStop*. In Figure 3, for example, the actual time was 28 microseconds for 20 iterations. The calibration error was therefore 1.4 microseconds per *Stop*.

The Split Loop does nothing but call *ElapsedTime* to read the split time. If the calibration for the overhead of taking a split was perfect, the measured time would be zero, regardless of the number of iterations. Dividing the accumulated time for the Split Loop by the number of iterations gives an estimate of the accuracy in the calibration of *dOverheadForSplit*. In Figure 3, the actual time was 22 microseconds for 20 iterations. The calibration error was therefore 1.1 microseconds per *Split*.

In practice, the limiting factor in the resolution of *TProfilingStopWatch* is the uncertainty in calibrating the overhead, which can fluctuate by 10 percent or more.

Keep in mind, however, that calibration uncertainties are significant only if you are trying to measure intervals that are short compared to the measured overhead. For example, the time required to execute some Windows API functions can be as much as a thousand times greater than this.

The **Recalibrate** button executes the class method procedure *CalibrateOverhead* and displays the new values of *dOverheadForStop* and *dOverheadForSplit*. After recalibrating, press the **Run** button again to see the effect on the measured time for the Empty Loop and the Split Loop. You can also explore the effect of changes in *dOverheadForStop* and *dOverheadForSplit* by entering new values in the corresponding edit boxes and then pressing the **Set Overhead** button.

By default, each *TProfilingStopWatch* object automatically corrects for the overhead of calling the Windows VTD. However, it can do this only for the overhead incurred by its own calls to the VTD. If nested timers are used, as here, the elapsed time for the outer timer will include the overhead incurred by the inner timers.

The solution is to count the number of calls made by the inner timers and to subtract this from the elapsed time for the outer timer, after multiplying by the values returned by *OverheadForStop* and *OverheadForSplit*. This is the time reported for **StpW Calls**.

The value reported for **Remainder** consists of all the remaining parts of procedure *btnRunClick*. This is mostly just the control code for the Empty Loop, and its proportion of the total time is small, as expected.

Conclusion

Unit VTIMERDV shows the flexibility of Delphi in allowing the programmer to use assembly language or even machine code to perform operations that cannot be implemented directly in Object Pascal. Yet, at the same time, everything was

done within the Delphi environment and within the structural framework of Object Pascal with its excellent support for strong type checking and safe, structured programming.

Class *TStopWatch* builds upon the foundation provided by VTIMERDV to create an easy-to-use Delphi component that provides a high-speed timer with the ability to resolve external events that are separated by intervals as short as 25 microseconds. Class *TProfilingStopWatch* is a specialized descendant of *TStopWatch* that can profile code blocks as short as 2-3 microseconds. ∆

The stopwatch components and demonstration forms referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\SEP\RH9509.

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

Begin Listing Two: The VTimerDV unit

```
{ Access the Windows Virtual Timer Device }
unit VTimerDv;

interface

function VTD_GetTime: Double;

implementation

var
  wVTDSegment: word;
  wVTDOffset:  word;
  { Used as a 64-bit signed Integer }
  aVTDTicks: array [1..2] of LongInt;
  { Actually a constant }
  dSecondsPerTick: Double;

procedure VTD_GetEntryPoint;
begin
  { Call Windows to get address of VTD entry point. }
  asm
    { Code to return API entry point }
    mov ax,$1684
    { Code for Virtual Timer Device (VTD) }
    mov bx,$05
    { Clear ES:DI }
    xor di,di
    mov es,di
    { Call Windows }
    int $2F
    { Save results }
    mov wVTDSegment,es
    mov wVTDOffset,di
  end;
end;

function VTD_GetTime: Double;
begin
  { Call VTD to get current "micro-ticks". }
  asm
    { Load the entry point }
    mov dx,wVTDSegment
    mov ax,wVTDOffset
    { Push our call-back address }
    push cs
    mov bx, offset @RetSpot
    push bx
    { Push the entry point }
    push dx
    push ax
    { Load ID for function VTD_Get_Real_Time }
    mov ax,$100
   { Call the VTD }
    retf

  @RetSpot:
    { Just a place to come home to }
    nop
  end;

  { The 64-bit tick count is returned in the 32-bit
    registers EAX and EDX. Since Delphi's built-in
    assembler does not support 32-bit registers, use
    machine code to transfer the registers into two
    adjacent 32-bit LongInts, which emulate a 64-bit
    Integer. Then push the 64-bit Integer onto the
    coprocessor stack. }
  inline
    (
    { Toggle operand size }
    $66/
    { mov aVTDTicks[1],eax }
    $89/$06/>aVTDTicks/
    { Toggle operand size }
    $66/
    { mov aVTDTicks[2],edx }
    $89/$16/>aVTDTicks+4/
    { fild[64] aVTDTicks }
    $DF/$2E/>aVTDTicks
    );

  { Convert from ticks to seconds. }
  asm
    { Multiply by the conversion factor }
    fmul dSecondsPerTick
    { Store in @result and pop the stack }
    fstp @result
  end;
end;

initialization
  VTD_GetEntryPoint;

  { The built-in assembler cannot reference Object
    Pascal constants. So must declare this as a
    variable and explicitly initialize it. }

  { Timer rate is 1.196 MHz }
  dSecondsPerTick := 0.836E-6;
end.
```

End Listing Two

Begin Listing Three: The StpWatch Unit

```
{ Exports classes TStopWatch and TProfilingStopWatch. }
unit StpWatch;

interface

uses Classes, VTimerDv;

type
  TStopWatch = class(TComponent)
  { A simple stopwatch, appropriate for timing external
    events. }
  private
    TimerIsRunning: Boolean;
    dStartTime:     Double;
    dCurrTime:      Double;
    dAccumTime:     Double;
  public
    { Uses default constructor. All data fields are
      initialized to zero. }
    function  IsRunning: Boolean; virtual;
    procedure Start; virtual;
    procedure Stop;  virtual;
    procedure Reset; virtual;
    function  ElapsedTime: Double; virtual;
  end;

  TProfilingStopWatch = class(TStopWatch)
  { A specialized stopwatch for profiling Delphi
    programs. Automatically corrects for the overhead
    of calling the Windows VTD. Calibration is done
    via class methods and class variables, thereby
    ensuring that overhead corrections are identical
    for all instances. }
  private
    lNumSplits: LongInt;
  public
    { Uses default constructor. All data fields are
     initialized to zero. }
    procedure Stop;  override;
    procedure Reset; override;
    function  ElapsedTime: Double; override;
    class procedure CalibrateOverhead;
    class function  OverheadForStop: Double;
    class function  OverheadForSplit: Double;
    class procedure SetOverheadForStop
      (dNewOverheadForStop: Double);
    class procedure SetOverheadForSplit
      (dNewOverheadForSplit:Double);
  end;

procedure Register;

implementation

var
  { Class variables shared by all instances of
    TProfilingStopWatch. }
  dOverheadForStop:  Double;
  dOverheadForSplit: Double;

function TStopWatch.IsRunning: Boolean;
begin
  result := TimerIsRunning;
end;

procedure TStopWatch.Start;
begin
  if TimerIsRunning then
    { do nothing }
  else
    begin
    dStartTime := VTD_GetTime;
    TimerIsRunning := True;
    end;
end;

procedure TStopWatch.Stop;
begin
  { Stop the timer and update the accumulator. }
  if TimerIsRunning then
    begin
      dCurrTime  := VTD_GetTime;
      dAccumTime := dAccumTime + (dCurrTime - dStartTime);
      TimerIsRunning := False;
    end
  else
    ;  { do nothing }
end;

procedure TStopWatch.Reset;
begin
  { Stop the timer and reset the accumulator. }
  TimerIsRunning := False;
  dAccumTime := 0.0;
end;

function TStopWatch.ElapsedTime: Double;

begin

  { Return the elapsed time in seconds. }
  if TimerIsRunning then
    begin
      { If the timer is running, return the current
        "split" time. }
      dCurrTime := VTD_GetTime;
      result := dAccumTime + (dCurrTime - dStartTime);
    end
  else
    result := dAccumTime;
end;


procedure TProfilingStopWatch.Stop;
begin
  { Stop the timer and update the accumulator. }
  if TimerIsRunning then
    begin
      dCurrTime  := VTD_GetTime;
      dAccumTime := dAccumTime + (dCurrTime - dStartTime)
                    - dOverheadForStop -
                    lNumSplits*dOverheadForSplit;
      TimerIsRunning := False;
    end
  else
    ;  { do nothing }
end;

procedure TProfilingStopWatch.Reset;
begin
  { Stop the timer and reset the accumulator. }
```

```
  dAccumTime := 0.0;
  lNumSplits := 0;
  TimerIsRunning := False;
end;

function TProfilingStopWatch.ElapsedTime: Double;
begin
  { Return the elapsed time in seconds. }
  if TimerIsRunning then
    begin
      { If the timer is running, return the current
        "split" time. }
      dCurrTime := VTD_GetTime;
      inc(lNumSplits);
      result := dAccumTime + (dCurrTime - dStartTime)
                - lNumSplits*dOverheadForSplit;
    end
  else
    result := dAccumTime;
end;

class function TProfilingStopWatch.OverheadForStop: Double;
begin
  result := dOverheadForStop;
end;

class function TProfilingStopWatch.OverheadForSplit:
Double;
begin
  result := dOverheadForSplit;
end;

class procedure TProfilingStopWatch.SetOverheadForStop
                  (dNewOverheadForStop: Double);
begin
  dOverheadForStop := dNewOverheadForStop;
end;

class procedure TProfilingStopWatch.SetOverheadForSplit
                  (dNewOverheadForSplit: Double);
begin
  dOverheadForSplit := dNewOverheadForSplit;
end;

class procedure TProfilingStopWatch.CalibrateOverhead;
  { Measure the overhead of starting and stopping the
    stopwatch. }
const
  NumCalls = 100;
var
  CalibrationWatch: TProfilingStopWatch;
  dSplitTime: Double;
  i: Integer;
begin
  CalibrationWatch  := TProfilingStopWatch.Create(nil);
  dOverheadForStop  := 0.0;
  dOverheadForSplit := 0.0;

  { Measure the overhead of stopping the watch. }
  CalibrationWatch.Reset;
  for i := 1 to NumCalls do
    begin
      CalibrationWatch.Start;
      CalibrationWatch.Stop;
    end;
  dOverheadForStop :=
    CalibrationWatch.ElapsedTime / NumCalls;

  { Measure the overhead of reading a split time. }
  CalibrationWatch.Reset;
  CalibrationWatch.Start;
  for i := 1 to NumCalls do
    dSplitTime := CalibrationWatch.ElapsedTime;
  CalibrationWatch.Stop;
  dOverheadForSplit := (CalibrationWatch.ElapsedTime -
                        dOverheadForStop) / NumCalls;
  CalibrationWatch.Free;
end;

procedure Register;
begin
  RegisterComponents('System',[TStopWatch]);
  RegisterComponents('System',[TProfilingStopWatch]);
end;

initialization
  TProfilingStopWatch.CalibrateOverhead;
end.
```

End Listing Three

Begin Listing Four: The Main Unit

```
{ Example 1 -- How to use class TStopWatch. }
unit Main;

interface

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

type
  TMainForm = class(TForm)
    StopWatch1: TStopWatch;
    StopWatch2: TStopWatch;
    btnStart1:  TButton;
    btnStop1:   TButton;
    btnReset1:  TButton;
    btnSplit1:  TButton;
    btnStart2:  TButton;
    btnStop2:   TButton;
    btnReset2:  TButton;
    btnSplit2:  TButton;
    lblTime1:   TLabel;
    lblTime2:   TLabel;
    cbRunning1: TCheckBox;
    cbRunning2: TCheckBox;

    Label1:     TLabel;
    Label2:     TLabel;
    procedure FormActivate(Sender: TObject);
    procedure btnStart1Click(Sender: TObject);
    procedure btnStop1Click(Sender: TObject);
    procedure btnReset1Click(Sender: TObject);
    procedure btnSplit1Click(Sender: TObject);
    procedure btnStart2Click(Sender: TObject);
    procedure btnStop2Click(Sender: TObject);
    procedure btnReset2Click(Sender: TObject);
    procedure btnSplit2Click(Sender: TObject);
    procedure DisplayResults;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{ $R *.DFM }

var
  dTime1: Double;
  dTime2: Double;

procedure TMainForm.FormActivate(Sender: TObject);
begin
  dTime1 := 0.0;
  dTime2 := 0.0;
  DisplayResults;
end;

procedure TMainForm.btnStart1Click(Sender: TObject);
begin
  StopWatch1.Start;
  DisplayResults;
end;

procedure TMainForm.btnStart2Click(Sender: TObject);
begin
  StopWatch2.Start;
  DisplayResults;
end;

procedure TMainForm.btnStop1Click(Sender: TObject);
begin
  StopWatch1.Stop;
  dTime1 := StopWatch1.ElapsedTime;
  DisplayResults;
end;

procedure TMainForm.btnStop2Click(Sender: TObject);
begin
  StopWatch2.Stop;
  dTime2 := StopWatch2.ElapsedTime;
  DisplayResults;
end;

procedure TMainForm.btnReset1Click(Sender: TObject);
begin
  StopWatch1.Reset;
  dTime1 := StopWatch1.ElapsedTime;
  DisplayResults;
end;

procedure TMainForm.btnReset2Click(Sender: TObject);
begin
  StopWatch2.Reset;
  dTime2 := StopWatch2.ElapsedTime;
  DisplayResults;
end;

procedure TMainForm.btnSplit1Click(Sender: TObject);
begin
  dTime1 := StopWatch1.ElapsedTime;
  DisplayResults;
end;

procedure TMainForm.btnSplit2Click(Sender: TObject);
begin
  dTime2 := StopWatch2.ElapsedTime;
  DisplayResults;
end;

procedure TMainForm.DisplayResults;
begin
  lblTime1.Caption := FormatFloat('0.000000',dTime1);
  lblTime2.Caption := FormatFloat('0.000000',dTime2);
  if StopWatch1.IsRunning then
    cbRunning1.State := cbChecked
  else
    cbRunning1.State := cbUnchecked;
  if StopWatch2.IsRunning then
    cbRunning2.State := cbChecked
  else
    cbRunning2.State := cbUnchecked;
end;

end.
```

End Listing Four

Begin Listing Five: TProfilingStopWatch

```
{ Example 2 -- How to use class TProfilingStopWatch. }
unit Main;

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

type
  TfmMain = class(TForm)
    { Non-Visual components }
    ProfilingStopWatch1: TProfilingStopWatch;
    ProfilingStopWatch2: TProfilingStopWatch;
    ProfilingStopWatch3: TProfilingStopWatch;
    ProfilingStopWatch4: TProfilingStopWatch;
    { Visual components }
    btnRun:            TButton;
    btnSetOverhead:    TButton;
    btnRecalibrate:    TButton;
    ebIterations:      TEdit;
    ebStopOverhead:    TEdit;
    ebSplitOverhead:   TEdit;
    lblTotalTime:      TLabel;
    lblTotalPct:       TLabel;
    lblEmptyTime:      TLabel;
    lblEmptyPct:       TLabel;
    lblWorkTime:       TLabel;
    lblWorkPct:        TLabel;
    lblOverhead:       TLabel;
    lblOverheadPct:    TLabel;
    lblRemainder:      TLabel;
    lblRemainderPct:   TLabel;
    lblStopOverhead:   TLabel;
    lblSplitOverhead:  TLabel;
    lblSplitTime:      TLabel;
    lblSplitPct:       TLabel;
    Label1:   TLabel;
    Label2:   TLabel;
    Label3:   TLabel;
    Label4:   TLabel;
    Label5:   TLabel;
    Label6:   TLabel;
    Label7:   TLabel;
    Label8:   TLabel;
    Label9:   TLabel;
    Label10:  TLabel;
    Label11:  TLabel;
    procedure FormActivate(Sender: TObject);
    procedure btnRunClick(Sender: TObject);
    procedure btnSetOverheadClick(Sender: TObject);
    procedure btnRecalibrateClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  fmMain: TfmMain;

implementation

{ $R *.DFM }

procedure TfmMain.FormActivate(Sender: TObject);
begin
  ebIterations.Text   := '10';
  ebStopOverhead.Text :=
    FormatFloat('0.0000000',
                TProfilingStopWatch.OverheadForStop);
  ebSplitOverhead.Text :=
     FormatFloat('0.0000000',
                 TProfilingStopWatch.OverheadForSplit);
  btnRun.SetFocus;
  btnRunClick(self);
end;

procedure TfmMain.btnRunClick(Sender: TObject);
var
  I, j, lIterations, lNumInnerStops,
    lNumInnerSplits: LongInt;
  X, dTotalTime, EmptyTime, WorkTime, Overhead,
    dRemainder, dSplitTime: Double;
begin
  lIterations := StrToInt(ebIterations.Text);

  ProfilingStopWatch1.Reset;
  ProfilingStopWatch2.Reset;
  ProfilingStopWatch3.Reset;
  ProfilingStopWatch4.Reset;
  lNumInnerStops  := 0;
  lNumInnerSplits := 0;

  { Start the overall timer. }
  ProfilingStopWatch1.Start;

  { Run ProfilingStopWatch2 in an empty loop. }
  for i := 1 to lIterations do
    begin
      ProfilingStopWatch2.Start;
      ProfilingStopWatch2.Stop;
      lNumInnerStops  := lNumInnerStops  + 2;
    end;

  { Run ProfilingStopWatch3 in a loop with splits. }
  ProfilingStopWatch3.Start;
  for i := 1 to lIterations do
    { Get split time }
    dSplitTime := ProfilingStopWatch3.ElapsedTime;
  ProfilingStopWatch3.Stop;
  inc(lNumInnerStops);
  lNumInnerSplits := lNumInnerSplits + lIterations;
  { Recalculate x by repeated multiplication. }
  ProfilingStopWatch4.Start;
  inc(lNumInnerStops);
  X := 1.0;
  for i := 1 to lIterations do
    for j := 1 to i do
      begin
        X := sqrt(X);
        X := ln(X);
        X := exp(X);
        X := X * X;
      end;

  ProfilingStopWatch4.Stop;
  inc(lNumInnerStops);

  { Stop the overall timer. }
  ProfilingStopWatch1.Stop;
```

```
  { Format and display the results. }
  dTotalTime := ProfilingStopWatch1.ElapsedTime;
  dEmptyTime := ProfilingStopWatch2.ElapsedTime;
  dSplitTime := ProfilingStopWatch3.ElapsedTime;
  dWorkTime  := ProfilingStopWatch4.ElapsedTime;
  dOverhead  :=
    lNumInnerStops*TProfilingStopWatch.OverheadForStop +
    lNumInnerSplits*TProfilingStopWatch.OverheadForSplit;
  dRemainder := dTotalTime - dEmptyTime - dSplitTime -
    dWorkTime - dOverhead;

  lblTotalTime.Caption :=
    FormatFloat('0.000000',dTotalTime);
  lblEmptyTime.Caption :=
    FormatFloat('0.000000',dEmptyTime);
  lblSplitTime.Caption :=
    FormatFloat('0.000000',dSplitTime);
  lblWorkTime.Caption  :=
    FormatFloat('0.000000',dWorkTime);
  lblOverhead.Caption  :=
    FormatFloat('0.000000',dOverhead);
  lblRemainder.Caption :=
    FormatFloat('0.000000',dRemainder);
  lblTotalPct.Caption  := '100.00';
  lblEmptyPct.Caption  :=
    FormatFloat('0.00',100.0 * dEmptyTime/dTotalTime);
  lblSplitPct.Caption  :=
    FormatFloat('0.00',100.0 * dSplitTime/dTotalTime);
  lblWorkPct.Caption   :=
    FormatFloat('0.00',100.0 * dWorkTime/dTotalTime);
  lblOverheadPct.Caption :=
    FormatFloat('0.00',100.0 * dOverhead/dTotalTime);
  lblRemainderPct.Caption :=
    FormatFloat('0.00',100.0 * dRemainder/dTotalTime);
  lblStopOverhead.Caption := FormatFloat('0.0000000',
    TProfilingStopWatch.OverheadForStop);
  lblSplitOverhead.Caption := FormatFloat('0.0000000',
    TProfilingStopWatch.OverheadForSplit);
end;

procedure TfmMain.btnSetOverheadClick(Sender: TObject);
begin
  TProfilingStopWatch.SetOverheadForStop
    (StrToFloat(ebStopOverhead.Text));
  TProfilingStopWatch.SetOverheadForSplit
    (StrToFloat(ebSplitOverhead.Text));
  lblStopOverhead.Caption := FormatFloat('0.0000000',
    TProfilingStopWatch.OverheadForStop);
  lblSplitOverhead.Caption := FormatFloat('0.0000000',
    TProfilingStopWatch.OverheadForSplit);
end;

procedure TfmMain.btnRecalibrateClick(Sender: TObject);
begin
  TProfilingStopWatch.CalibrateOverhead;
  lblStopOverhead.Caption := FormatFloat('0.0000000',
    TProfilingStopWatch.OverheadForStop);
  lblSplitOverhead.Caption := FormatFloat('0.0000000',
    TProfilingStopWatch.OverheadForSplit);
end;

end.
```

End Listing Five