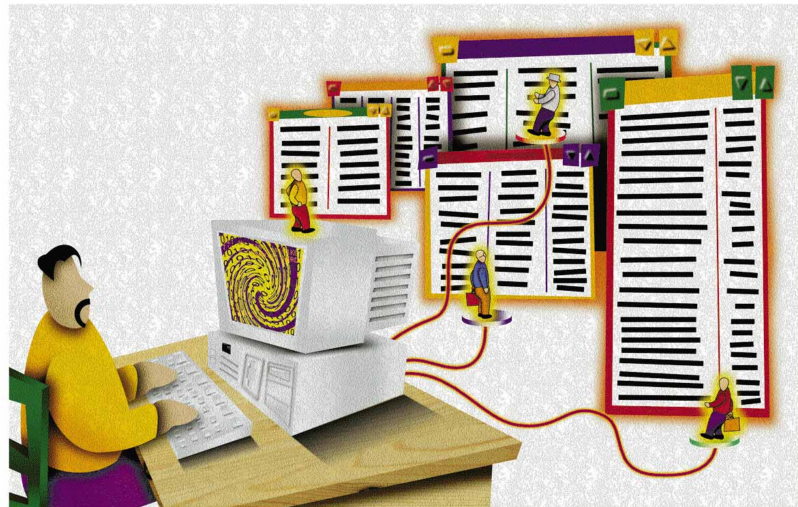
July 1995 - Volume 1, Number 3

Working with Data

Delphi as a Database Front-End



Cover Art By: *Victor Kongkadee*

ON THE COVER

6 The DBTracker Utility — John O'Connell
"Did the record post?" It's a question of data integrity, and any database veteran can tell you the importance of knowing when critical database events occur. Mr O'Connell provides us with an invaluable tool for tracking these elusive events.

16 Moving to Local InterBase — Marco Cantù
Although "real" client/server topology calls for separate client and server processors, Mr Cantù points out that you can obtain many of the benefits of a SQL server by moving your data to the Local InterBase product that comes with Delphi.

FEATURES

21 DBNavigator — Cary Jensen, Ph.D.
Dr Jensen continues his exploration of the *TField* object. This month's topics include calculated fields, using the *OnCalcFields* event to display lookup data, and how to manipulate the creation order of objects so they are there when you need them.

26 Informant Spotlight — Robert Vivrette
One of the (extraordinarily) few complaints you'll hear about Delphi is "Why are the executables it creates so big?" The short answer is that Delphi makes you pay once, up front for a multitude of Windows niceties. And as Mr Vivrette explains — it's a bargain.

31 OP Tech — Bill Todd
The Memo component is cool, no doubt, but it does lack some important cursor controlling functionality. Fortunately, thanks to Mr Todd and a couple of Windows API messages, it's just not a problem.

34 Visual Programming — Sedge Simons
It ain't Object Pascal, but it *is* a vital portion of every Delphi form. Mr Simons introduces the .DFM file, explains its function, and describes how developers might exploit it to perform a number of programming tasks.

REVIEWS

37 Developers Visual Suite Deal
Product review by Douglas Horn

40 *Instant Delphi Programming*
Book review by Jerry Coffey

40 *Delphi by Example*
Book review by Jerry Coffey

41 *teach yourself...Delphi*
Book review by Jerry Coffey

DEPARTMENTS

2 Delphi Tools

4 Newsline

Delphi

TOOLS

New Products and Solutions

Pinnacle's Graphic Server Ships

Pinnacle Publishing of Kent, WA has upgraded its Graphics Server. Its improved tool kit, ***Graphics Server 4.0***, allows developers to add graphing and charting features to their Windows applications.

Graphics Server 4.0 includes 16 and 32-bit versions of its DLLs, so it allows graphing functionality to be ported from Windows 3.1 to Windows 95 and Windows NT 3.5 with no change in code. In addition to standard 16-bit VBXes, developers will be able to use 16 and 32-bit OLE custom controls (OCX).

It also features an editing interface that allows users to change graph type, style, perspective, and other parameters with simple button clicks. Graphs can be edited through either the toolbar or dialog box invoked by right-clicking with the mouse.

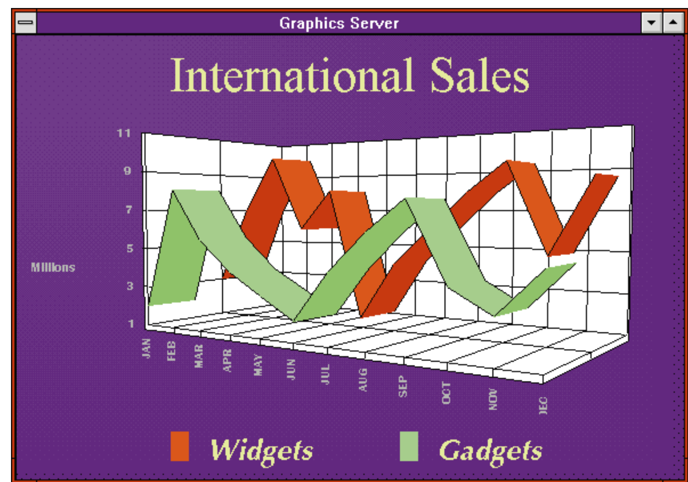
Graphic Server's graphing capabilities were also upgraded for this new release. These include true 3D graphs that users can rotate on their X or Y axis. The point of view can be interactively moved above or below the horizon for viewing multiple data sets. Four new graph types have been added including box-whisker, candlestick, surface, and time-series, plus three new variations on existing types: lin/log, log/log, and open-high-low-close. Statistical functions have also been augmented with the addition of spline and moving average curve fits, error bars, and a new Fast Fourier Transform function that enables users to transform data into the frequency domain.

Price: US$299

Contact: Pinnacle Publishing, 18000 72nd Avenue South, Suite 217, Kent, WA 98032

Phone: (800) 788-1900, or (206) 251-1900

Fax: (206) 251-5057



Delphi Training

HTR, Inc. is an authorized Delphi Training center offering two Delphi classes: *Client/Server Application Development Using Delphi* and *Advanced Client/Server Application Development*. The classes run through July at various cities nationwide. The three-day *Client/Server Application Development Using Delphi* class costs US$1050, and *Advanced Client/Server Application Development* is a two-day event priced at US$700. For more information, or to enroll, call (301) 881-1010.

Sax Software Announces Sax Comm Objects

Sax Software of Eugene, OR has begun shipping ***Sax Comm Objects v 2.0 Professional Edition***, a package that combines Sax Comm Objects and the Sax Basic Engine scripting language.

Using the Professional Edition, developers can write serial communications programs that include customized common dialog boxes. The Professional Edition includes the Standard Edition of Sax Comm Objects. Because the Standard Edition is a custom control, it provides all the VBX and OCX power needed to get connected. It supports the most popular file-transfer protocols, including ZModem, YModem, XModem, Kermit, and CompuServe B+.

The built-in terminal emulation lets developers create Windows front-ends for character-based programs such as those used by UNIX systems or by many on-line services.

Sax Basic Engine is a custom control that offers developers a Basic-syntax scripting language. The comm extensions are written specifically by Sax Software to make joining the Comm Objects and the Basic Engine with a single function call. The Professional Edition includes complete source code for Sax Comm Objects and for the comm extensions to the Basic Engine.

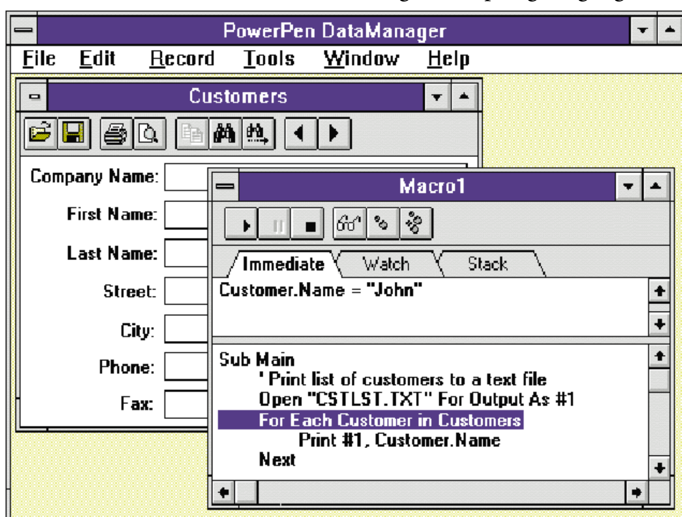
Sax Software charges no royalties or run-time fees, and all Sax Software products include an unconditional 30-day money-back guarantee.

Price: US$495.

Contact: Sax Software, 950 Patterson Street, Eugene, OR 97401

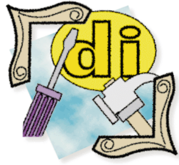
Phone: (503) 344-2235

Fax: (503) 344-2459



Delphi TOOLS

New Products and Solutions



Free Trial-Run Components for Delphi from TurboPower

TurboPower Software, of Colorado Springs, CO has announced a free, Trial-Run version of *Orpheus*, a collection of native VCL data entry components for Delphi. The Trial-Run components offer the full functionality of Orpheus, but run only when the Delphi development environment is running. The free tools include abbreviated on-disk documentation.

Orpheus includes validated data-aware fields for string, numeric, currency, and date/time variables. It has a form controller for managing validation error events and configurable command/key mappings. It features a text editor with 16MB capacity, undo/redo, word wrap, and more. Orpheus includes a list box with unlimited capacity, multiple selection and colors, along with a notebook page with Windows 95-style top or side tabs.

It has a flexible table that holds edit fields, combo boxes, bitmaps, checkboxes, and more. It also includes two, four, and five-way spinners. The components in Orpheus are native VCL components.

The free tools are available on the *Delphi Informant* Companion Disk and for download from the Informant Forum on CompuServe (ICGFORUM). Library 12: Delphi 1.x. File name: ORPHEUS.ZIP. The file is also available from the Informant Bulletin Board at (916) 686-4740. Library name: DIWINLIB. In addition, it's available on Internet (ftp from rmii.com:/pub2/turbopower) and from the TurboPower BBS.

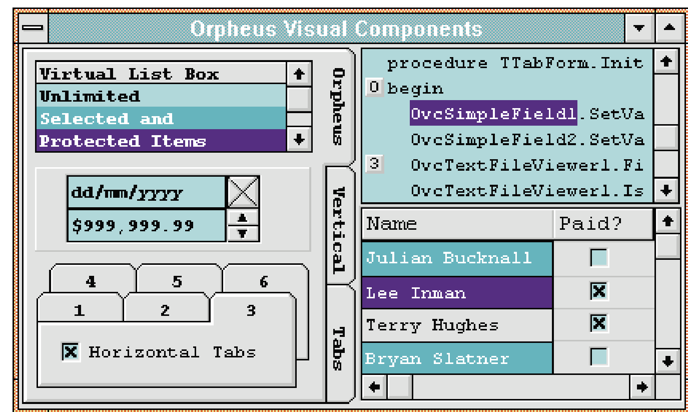
Price: US$199, includes full source, comprehensive printed documentation, free support from Delphi experts by phone, e-mail, and fax, and a 60-day money-back guarantee.

Contact: TurboPower, P.O. Box 49009, Colorado Springs, CO, 80949-9009

Phone: (719) 260-6641

Fax: (719) 260-7151

TurboPower BBS: (719) 260-9726



Access Mainframe Data from PC with TransPortal PRO

The Frustum Group, Inc. of New York, NY has released ***TransPortal PRO, version 4.1***. This data exchange toolkit integrates PC applications with on-line host screen based applications.

Developers can create PC front-ends for complicated or unfriendly host systems. PC applications using TransPortal PRO can retrieve host data in real time. TransPortal PRO requires no additional mainframe software and follows existing terminal security procedures. TransPortal PRO works with more than 30 PC programming languages in DOS, Microsoft Windows, and OS/2 using the same syntax and one function call.

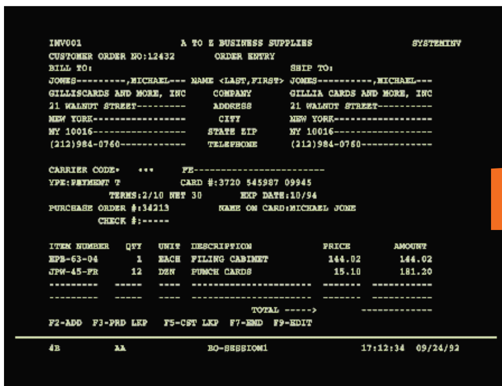
TransPortal PRO documentation includes sample code for all supported languages, on-line help in Windows and DOS, enhancements to Gupta SQL Windows, a new installation program, and improvements to TurboMap. Using TurboMap, PC programmers can automate complicated mainframe terminal operator procedures.

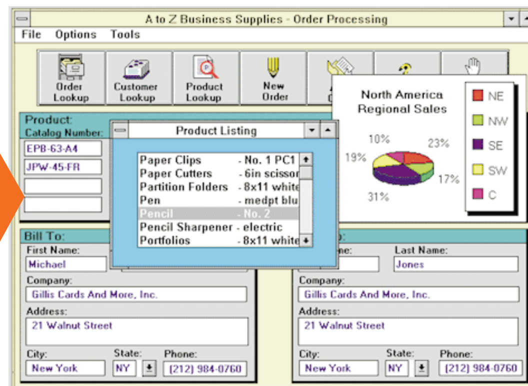
Price: US$2450

Contact: The Frustum Group, Inc., 525 North Broadway, White Plains, NY 10603

Phone: (914) 428-7200 or (800) 548-5660

Fax: (914) 428-0795





New Delphi Books

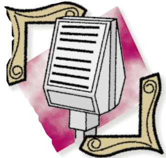
Delphi Programming EXplorer
By Jeff Duntemann, Jim Mischel, & Don Taylor
Coriolis Group Books
ISBN: 1-883577-25-X
Delphi Programming EXplorer alternatively covers the theory and practice of Delphi programming. It also addresses Object Pascal basics, object-oriented programming, and includes VB to Delphi conversion tips.
Price: US$39.99 (627 pages), disk
Phone: (800) 410-0192

Using Delphi, Special Edition
By John Matcho & David Faulkner
QUE Corporation
ISBN: 1-56529-823-3
Using Delphi, Special Edition targets intermediate to expert programmers, and stresses application development and delivery. Coverage includes database programming, ReportSmith, DDE, OLE, DLLs, and the Windows API.
Price: US$29.99 (573 pages)
Phone: (317) 581-3500

News

L I N E

July 1995



Softbite International, Borland International, and **Informant Communications Group** have announced two additional international dates for the ***1995 Delphi World Tour***. The event is scheduled to stop in London on October 12-13, and Amsterdam on October 16-17. Other international dates are planned, but no details were available at press time.

In the US, the two-day seminar is scheduled to stop in Columbus, Boston, Los Angeles, Philadelphia, Chicago, Seattle, Dallas, New York, Atlanta, San Francisco, Washington DC, Denver, Minneapolis, Orlando, Phoenix, Detroit, Houston, and Raleigh/Charlotte.

For more information contact Softbite International by calling (708) 833-0006, or sending an e-mail to register@softbite.mhs.compuserve.com.

Informant Communications Group has secured **Synectics Software** as their ***African distributor of Delphi Informant*** and ***Paradox Informant Magazines***. Synectics Software territories include: Republic of South Africa, Lesotho, Swaziland, Namibia, Botswana, Zimbabwe, Mozambique, Mauritius, Angola, Zambia, and Malawi.

For information, call 27 11 789-4316, fax 27 11 886-5697, or e-mail 100075,1636.

Borland Developers Conference Heads to San Diego

Scotts Valley, CA — Borland International has announced their 6th Annual Borland Developers Conference is slated for August 6-9, 1995, in San Diego, CA. Housed in the San Diego Conference Center, the event will spotlight four product-specific tracks: Delphi, Paradox, dBASE, and Borland C++.

Focusing on Paradox application development in Windows and Windows 95, the Paradox sessions will cover: using interface design, event-driven programming, and real-world Paradox programming solutions. Specific topics include: Leveraging Windows 95 in Paradox Application Development, Multi-Form Application Development, Upsizing: Programming ObjectPAL and SQL, Security in Paradox Applications, Programming Paradox for Windows with OLE 2.0, and Delivering and Deploying Paradox Applications.

The Delphi tracks will present a thorough introduction to Delphi, including the Delphi object model, visual programming environment, client/server development, and much more. Specific topics include: Delphi and Windows 95, Rapid Application Development with Delphi, Visual Programming with Delphi, Creating Delphi Components, Using Borland C++ DLLs with Delphi, Object-Oriented Application Design, The Delphi Object Model, Accessing Data with Delphi, Client/Server Issues for Delphi, Delphi Exception Handling, Delphi SQL programming, and OLE 2.0 Programming with Delphi.

For dBASE, Borland focuses on programming dBASE applications in Windows and Windows 95. It addresses issues such as solutions, programming, tools and techniques, and more. The Borland C++ sessions teach C++ Windows and Windows 95 programming techniques including templates, Object Windows Programming, exception handling, and DLL programming.

Borland is also offering elective tracks on emerging industry trends, business solutions, hardware, and operating system issues, InterBase, ReportSmith, and many others. Those attending a track will receive the appropriate technical paper, source code, and examples.

The conference registration prices are: single, US$1195, and 3 or more from the same company, US$1145 each. Each attendee is eligible to receive one free Borland product: Delphi for Windows (CD-ROM), Paradox for Windows (CD-ROM or 3 1/2 disks), Borland C++ 4.5 (CD-ROM), or dBASE for Windows (CD-ROM or 3 1/2 disks).

Attendees may also attend pre-conference tutorial available on Sunday August 6 and receive a free lunch.

To register call Borland at (800) 350-4244.

Delphi Updates Available

Scotts Valley, CA — Borland International has released Delphi update files. The updates are contained in three files: DELCSPAT.ZIP for Delphi Client/Server Edition (including VCL patches); DELPATCH.ZIP for Delphi Desktop Edition; and VCLPATCH.ZIP, for updating Desktop Edition users' VCL code.

According to the readme file provided with the update files, the following problems have been addressed: improved compatibility with Windows 95 M8 beta for MDI (e.g. new child) and OLE2 (e.g. insert object); IDE debugger compatibility fix for Windows NT; fixed MDI design mode problem when minimizing MDI Child window; significant updates to OLE2 API unit (see \DELPHI\DOC\OLE2.INT); fixed unit version problem in DLIB.EXE; fixed problem in Browser when you double-click a reference to a .PAS file that is not already open in the editor; fixed **Options | Rebuild Library** problem when the current project has an active Dataset; fixed Alt Tab problem in Grid control; fixed DBGrid to allow cancel of *SetKey* mode; *TForm.DefineProperty* now calls its inherited method; support for owner draw in *TOutline*; *DBImage.CutToClipboard* now correctly updates the Clipboard; in *TDataSource.OnDataChange*, fixed invalid pointer in the Field Parameter; fixed various demo problems.

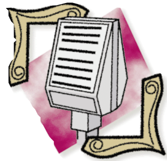
In addition, Borland has released the Delphi VCL Reference Manual and Object Pascal Language Reference in Adobe Acrobat format. These manuals are also available in hard copy for an additional fee. For information call Borland at (408) 431-1000. Note: Both manuals are already available as Windows on-line Help with the CD-ROM release of Delphi and Delphi Client/Server.

The update files and manuals are available for download from the Delphi Forum on CompuServe, Library 2.

News

L I N E

July 1995



Borland International Inc. recently posted a US$51 million loss for the fourth quarter of fiscal 1995 (ending March 31st), but it's less than the US$76 million loss from the fourth quarter of last year. Also, revenues increased 9 percent to US$55.5 million from US$51 million in 1994. The company also reported a loss of US$12.2 million, or 43 cents a share for fiscal 1995. Compared to a loss of US$69.9 million, or US$2.62 a share for fiscal 1994, Borland is looking stronger. Borland's president Gary Wetsel said the company was pleased with its progress. According to Wetsel, Borland had three primary objectives during the last quarter: to reduce the cost structure, focus on the software developer market, and launch Delphi.

Coming to Chicago July 25 - 28, **Windows World** and **Enterprise Computing Solutions** will be held simultaneously at the McCormick Place. ECS '95 covers the four areas of enterprise computing: hardware platforms, connectivity, strategic applications, and operating environments. Dedicated to Windows-based enterprise solutions, Windows World features information on client/server products, enterprise systems and servers, imaging systems, e-mail, messaging, Internet links, and new strategic applications. To receive a registration form via fax, call (617) 449-5554 and enter code 72. Have your fax number ready and a form will be faxed to you within 24 hours.

DB/Expo: A Look at Borland's RAD Pack for Delphi and New dBASE

San Francisco, CA — With close to 100 new product announcements and presentations, DB/Expo attracted 31,108 attendees to the 7th annual conference and exposition held May 1-5 at the Moscone Convention Center in San Francisco, CA. This year's database, client/server, and networking event featured 900 exhibits from more than 200 IT companies worldwide.

Borland International announced the release of the RAD Pack for Delphi, a new companion tool set that combines many of Borland's products into one package. It includes the Visual Component Library source code and the Resource Workshop for extracting and modifying standard Windows resources, such as icons, cursors, bitmaps, and dialog boxes.

It also features a Resource Expert that converts standard resource scripts into Delphi forms, a Delphi Language Reference Guide, and the Turbo Debugger. The RAD Pack will be available through major resellers or directly from Borland for US$249.95.

Borland also showed its dBASE for Windows upgrade, code-named Voyager. Recently introduced at Spring COMDEX, this 16-bit product is based on visual tools and a business programming language, and will run under both Windows 3.1 and Windows '95. It also has client/server features and performance enhancements. Borland said a dBASE compiler for Windows will be available with Voyager allowing developers to create and deploy stand-alone (.EXE) applications royalty-free. The Voyager demonstration is part of a spring tour of trade shows, user group meetings, training seminars, and other special review events designed to inform developers about the new software.

In addition, Microsoft and Gupta announced they will market a single product that combines Gupta's software development tools with Microsoft's database products. Other strategic alliances include BMC Software Inc.'s plans to distribute DataTools, Inc.'s backup and recovery products.

DB/Expo New York 1995 is the event's next stop. It's scheduled for December 4-8, 1995 at Jacob K. Javits Center. For more information contact Blenheim at (415) 966-8440.

Borland's New Companion Products Group includes Delphi/Link and Poet

Scotts Valley, CA — Borland International Inc. is currently developing its Companion Products Group, a compilation of utilities and third-party tools for developers using Borland and other products.

Its first release, RAD Pack for Delphi, was launched at DB/Expo. It's a new companion tool set that combines many of Borland's products into one package. The RAD Pack is now available from major resellers or directly from Borland at a suggested retail price of US$249.95.

Other products in the group include Delphi/Link for Lotus Notes from Brainstorm Technologies, Poet Software from Poet Software, and Object Master from ACI. According to Borland, several new products are scheduled to launch just after Windows '95 is released, including products supporting Borland languages and databases, Microsoft's Visual Basic, and Powersoft's Powerbuilder. A majority of the tools within Borland's Companion Products Group are priced between US$100 and US$200.

For more information about the Companion Products Group or to order the RAD pack for Delphi, call Borland at (800) 233-2444 ext.1310.

Windows Component Resource Goes On-Line

Layton, UT — Imagicom is now providing a free comprehensive component resource service to Windows developers. The service has information on OCX, VBX, DLL, and Delphi custom controls organized by component type, file format, and vendor.

Imagicom's service is located on a World Wide Web (WWW) home page and can be accessed by any Internet browser. The home page is similar to an interactive catalog, where the user can point-and-click until the correct information is found. Once a developer has selected the right grouping, a specific component can be located from text descriptions, product images, and downloadable demonstration copies.

Vendors can put basic information about their components on the service for free. Imagicom charges a fee to maintain higher levels of service such as downloadable demonstration copies, independent Web pages, and images of custom controls.

Imagicom's home page is http://www.xmission.com/~imagicom/. For more information send e-mail to imagicom@xmission.com.

ON THE COVER

DELPHI / OBJECT PASCAL / IDAPI

By John O'Connell

The DBTracker Utility

Inside Delphi's Database Control Events

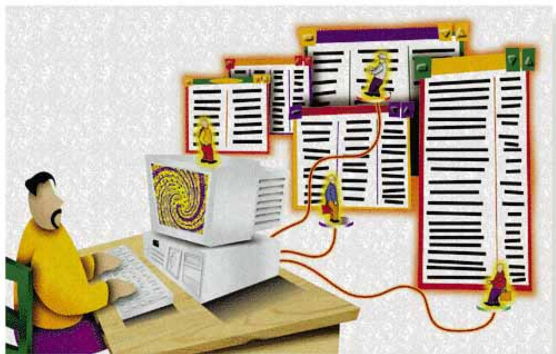
Delphi's database connectivity is based on a number of components that work together to allow your applications access to database tables. At a basic level, database access is provided by a component descended from *TDataSet*, such as *TTable*, *TQuery*, and *TStoredProc*.

A *TTable* component provides access to a table's records via IDAPI — the Borland Database Engine (BDE). A *TQuery* retrieves selected records from a table based on selection criteria, and a *TStoredProc* enables procedures and functions to be executed on a remote database server. These components are known collectively as *datasets*.

The *TDataSource* component is the link between data-aware controls such as *TDBEdit* or *TDBGrid* and a dataset. *TDataSource* is the middle layer in Delphi's database connectivity scheme. The *TTable* component is the lower layer, and data-aware controls are the upper layer.

Another component of Delphi's database connectivity is the *TField* component. It allows programmatic access to a field in the record. The *TField* object also performs default formatting and validating of field values. A *TField* object is created when a dataset is opened at run-time, or at design time by using the Fields editor for a *TDataSet* descendant component. In the latter case, properties and event handlers can be defined for the *TField* at design time. The list of *TFields* in a dataset can be accessed using the *Fields* array property at run-time. [For an in-depth introduction to *TFields*, see Cary Jensen's article "The TField Class" in the June 1995 *Delphi Informant*. He continues the discussion in this month's article "The TField Class: Part II", beginning on page 21.]

The *TDataSource* can be seen as a record buffer between data-aware controls and datasets. The *TTable*'s current record is copied to the *TDataSource* buffer and read by the data-aware controls using that datasource.



Datasource and dataset components generate notification events at various stages during manipulation of the database tables associated with them. This article will discuss the events generated by the *TTable* dataset and *TDataSource* using Paradox tables.

A *TTable* generates notification events for these reasons:
- The dataset is being opened or closed.
- The dataset is about to enter or exit edit mode.
- The dataset is about to insert or delete a record.
- The dataset is about to post or cancel changes to a record.

- A new record has just been created in the dataset.
- A dataset's calculated field needs recalculation.

A *TDataSource* generates notification events for the following reasons:
- The datasource's view of the record needs to be refreshed.
- The state of the dataset pointed to by the datasource has changed.
- The data in the current unposted record has changed.

The tables in Figures 1, 2, and 3 list these events and the possible states for a *TTable* or *TDataSource*.

But when do these events occur in an operating database application? It would be very useful to track these events as they happen to understand the order in which events occur in response to table manipulation by user interaction and Object Pascal code.

What DBTracker Does

The DBTracker utility (see Figure 4) does just this for one or two tables (which can be linked if desired). In this example I've used the CUSTOMER and ORDERS tables (linked one-to-many) in the DBDEMOS alias created by Delphi. This utility can be used with several tables without any changes to the source code (as we'll discuss later).

DBTracker tracks all *TDataSource* and *TTable* events and writes information about each event to a *TListBox* component. You can use the DBNavigator buttons to manipulate the tables, or use the buttons located in the **Methods** group box. Each button calls the named *TTable* method for the selected datasource's dataset table. For instance, this is useful when you want to track the exact stream of database events resulting from calling *TTable.Post*.

The table the method buttons call can be chosen from a combo box labeled **Call Method for**. It contains the names of all *TDataSource* components found in the form at run-time. When DBTracker is first started, the first *TDataSource's* dataset table is affected by the **Methods** buttons. The contents of the ListBox with the tracked database events can be saved to an ASCII text file and the list can also be emptied.

The master *TTable* in the form is called *tblMaster*, the detail *TTable* is called *tblDetail* — these are the datasets for the respective *TDatasource's dsMaster* and *dsDetail*. The tables are linked one-to-many by the detail table's secondary index.

The code behind DBTracker demonstrates a few techniques that could be used to achieve what may usually be required in a data-entry application. For example, it enables or disables buttons depending on the state of a table (or tables) as we'll see later.

Figure 4 shows the database events triggered just after the dialog box is displayed, and the master and detail tables are opened by code in the form's *FormCreate* event. Let's examine this list more closely:
- The master table is opened which triggers *BeforeOpen* for *tblMaster*.

TDataSet Event	When Triggered
BeforeOpen, AfterOpen	Before/after a dataset is opened
BeforeClose, AfterClose	Before/after a dataset is closed
BeforeInsert, AfterInsert	Before/after a dataset enters Insert mode
BeforeEdit, AfterEdit	Before/after a dataset enters Edit mode
BeforePost, AfterPost	Before/after a dataset posts changes to the current record
BeforeCancel, AfterCancel	Before/after a dataset cancels the current state
BeforeDelete, AfterDelete	Before/after a record is deleted
OnNewRecord	When a new record has been created
OnCalcFields	When calculated fields need to be recalculated

TDataSource Event	When Triggered
OnDataChange	When the view of the current record needs to be refreshed, usually as a result of moving to another record or the table being refreshed
OnUpdateData	When a field(s) value in the current record is about to be updated prior to posting the record
OnStateChange	When the mode or state of a dataset changes

State	When Triggered
dsInactive	When the dataset is closed
dsBrowse	When the dataset is in Browse state
dsEdit	When the dataset is in Edit state
dsInsert	When the dataset is in Insert state
dsSetKey	When the dataset is in SetKey state (*TTable.SetKey* has been called)
dsCalcFields	When the *OnCalcFields* event is called

Figure 1 (Top): The *TDataSet* events. **Figure 2 (Middle):** The *TDataSource* events. **Figure 3 (Bottom):** The State constants.

- Because *tblMaster* has a calculated field defined, *OnCalcFields* is triggered for *tblMaster*.
- The state for *dsMaster* and *tblMaster* changes from *dsInactive* to *dsBrowse* after *tblMaster* has opened.
- The *OnDataChange* event occurs for *dsMaster* when the displayed record changes from undefined to the first record in the master table.
- Finally the *AfterOpen* event for the master table is triggered.

A similar stream of events occurs for the detail table, *tblDetail*, except for the *OnCalcFields* event because there isn't a calculated field defined).

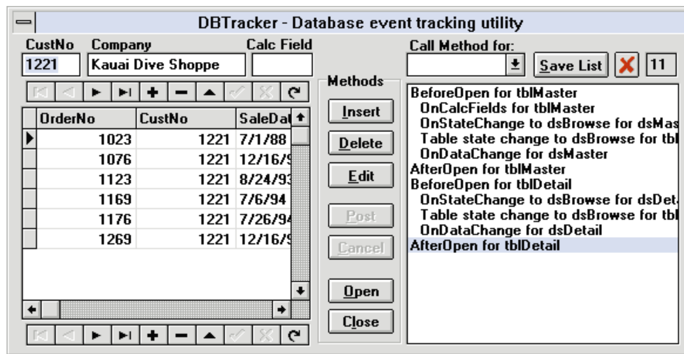


Figure 4: The DBTracker utility in action.

Using DBTracker

The buttons in the **Methods** group box explicitly call various *TTable* methods (*Insert*, *Delete*, *Edit*, *Post*, *Cancel*, *Open*, and *Close*, respectively). Pressing the equivalent buttons on the navigator toolbar has the same effect. However, using the **Methods** buttons allows you to clearly see what occurs between the moment the relevant *TTable* method is called, and the moment the method has completed. To specify which datasource the buttons will act upon, specify the desired datasource name from the combo box.

The **Open** and **Close** buttons allow you to open or close the *TTable* associated with the selected datasource. If the linked detail table *dsDetail* is closed, then the links between *tblDetail* and *tblMaster* will be ignored. If *dsMaster* is closed then *tblDetail* will effectively be an unlinked table with all records in view instead of the restricted view that's in force when *tblMaster* is open.

Let's look at the sequence of events for some common database usage scenarios.

Database Events for a Single Table

Viewing a Single Record

In this scenario *dsMaster* is open and *dsDetail* is closed. Moving through records or pressing the **Refresh** button on the Navigator toolbar causes *OnCalcFields* followed by *OnDataChange* to be triggered. Each time another record is reached, the dataset *tblMaster* signals that its calculated field requires recalculation and the record that the datasource is pointing to has changed. The *OnDataChange* event is triggered because the datasource needs to refresh its view of the record that in turn supplies the data-aware controls with the new field values for the record.

Pressing the **Insert** button causes the following chain of events:

```
BeforeInsert for tblMaster
  OnStateChange to dsInsert for dsMaster
  Table state change to dsInsert for tblMaster
  OnNewRecord for tblMaster
  OnDataChange for dsMaster
AfterInsert for tblMaster
```

Here, the states for *dsMaster* and *tblMaster* have changed from their previous states (which in this case were *dsBrowse*, the state at form startup). Because the datasource needs to refresh its view of the new empty record, the *OnNewRecord* event is triggered for *tblMaster* followed by the *OnDataChange* event.

Entering a value in the **CustNo** field in the new empty record and moving to the next field (by pressing Tab for example) triggers the following events:

```
OnCalcFields for tblMaster
OnDataChange for dsMaster.CustNo
```

Similar events occur for the **Company** field:

```
OnCalcFields for tblMaster
OnDataChange for dsMaster.Company
```

As a result of moving from a modified field, the value of the current record has changed in the underlying table. This causes the dataset to signal that its calculated field needs recalculating, and that the datasource's controls need refreshing with the changed field values of the record.

When the text in a data-aware control has changed and that control has lost focus, the text is sent to the underlying *TField* object. It then converts the text to the field's data type, validates the field value, and applies formatting.

Because the record has changed, the dataset notifies the associated datasource (*dsMaster*) that its view of the record needs to be updated, thus triggering the *OnDataChange* event. This ensures the data-aware controls are seeing exactly what's in the underlying record after the *TField* has done its bit. Note that the record itself isn't actually committed or posted to the table until *TTable.Post* is called or the entire record is refreshed.

Pressing the **Post** button triggers the following events:

```
OnUpdateData for dsMaster
BeforePost for tblMaster
  OnStateChange to dsBrowse for dsMaster
  Table state change to dsBrowse for tblMaster
  OnCalcFields for tblMaster
  OnDataChange for dsMaster
AfterPost for tblMaster
```

The *OnUpdateData* event is triggered by *tblMaster*'s *Post* method. It notifies all controls associated with *dsMaster* that the record in the table is about to change, so they can update their field values. During the post, the states for *dsMaster* and *tblMaster* switch from *dsInsert* to *dsBrowse*. The *OnCalcFields* event is triggered for the usual reason, and the *OnDataChange* event is triggered because the record has been refreshed after being successfully posted (i.e. written to the table).

Suppose we'd pressed the **Cancel** button rather than the **Post** button:

```
BeforeCancel for tblMaster
```

```
  OnStateChange to dsBrowse for dsMaster
  Table state change to dsBrowse for tblMaster
  OnCalcFields for tblMaster
  OnDataChange for dsMaster
AfterCancel for tblMaster
```

The chain of events triggered in this case is much the same, except the record is canceled and no changes are written to the table.

Pressing the **Edit** button generates the following chain of events:

```
BeforeEdit for tblMaster
  OnCalcFields for tblMaster
  OnStateChange to dsEdit for dsMaster
  Table state change to dsEdit for tblMaster
  OnDataChange for dsMaster
AfterEdit for tblMaster
```

In this case, the datasource's view of the record is refreshed before the table is ready for editing. This ensures the latest version of the record is being edited.

This is especially important where several applications are accessing the same table concurrently. It's possible that between the time a certain record is moved and edited, another application has edited that same record and posted the changes. This means you're viewing and editing the old version of the record! Thus moving from record to record (or getting ready to edit and insert a record) will always refresh the record.

If the datasource and table states are *dsInsert* or *dsEdit*, refreshing a table will cause a modified record to be posted and an unchanged record to be canceled. Try it yourself by editing and changing a master record in *tblMaster* and pressing the **Refresh** button on *dsMaster*'s navigator control.

However, there's a lot more going on behind the scenes than DBTracker is showing us when *TTable.Edit* is called. In fact, the single most important action to occur for a dataset before a record can be edited, *doesn't* generate a dataset or datasource event. This leads us to discuss the implications of using shared tables. (Paradox programmers will be familiar with these concepts.)

Record Locking and Shared Tables

Let's suppose the record you wish to edit is being edited by another user. When a record in a Paradox table is being edited, it is write-locked until the changes are posted (and the record unlocked) or canceled. This ensures that only one person at a time can make changes to a record. Therefore, when you start editing, an attempt is made to place a lock on the current record. If this fails (because another user has already locked the record) then the edit fails, an exception is raised, and the table state reverts to *dsBrowse*.

Calling *TTable.Edit* will attempt to lock the current record, as will *TTable.Delete* just before it tries to delete the record. When a record is locked by another user, you can't place a lock on it until the record is unlocked. This means you can't do anything to change the record. (Note that a Paradox record lock is a write-lock, so only you can write to it. There is no such thing as a record read-lock for Paradox tables.) If DBTracker tries to edit (or delete) a locked record, the only event that occurs is *BeforeEdit* (or *BeforeDelete*) before an exception is raised.

Another issue with shared tables is that of accuracy. For example, how do you know that the records being viewed in a DBGrid are current? Furthermore, how do you know that the record you're viewing even exists and hasn't been deleted by another user?

You don't — at least not until the moment after *TTable.Refresh* is called, a *TDBNavigator* object's **Refresh** button is pressed, or the set of records being viewed has just been scrolled into view.

At this stage, Paradox users will point out that Paradox can refresh tables automatically at preset intervals, so this isn't a problem with Paradox applications. Fortunately, the same effect can be achieved with Delphi by calling *TTable.Refresh* in response to a *TTimer's OnTimer* event. However, remember what refreshing does to a table whose state is *dsEdit* or *dsInsert*. If the datasource and table states are *dsInsert* or *dsEdit*, refreshing a table will cause a modified record to be posted and an unchanged record to be canceled. Therefore, only call *TTable.Refresh* when the dataset's state is *dsBrowse*. (Remember, if the datasource/table state is *dsInsert* or *dsEdit*, refreshing a table will cause a modified record to be posted and an unchanged record to be canceled.)

Database Events for 1-M Linked Tables

In this situation both *dsMaster* and *dsDetail* are open and linked. The events triggered when using multiple linked tables are a little different from the previous single table situation. These events will be triggered when moving from record to record in *tblMaster*:

```
OnCalcFields for tblMaster
OnDataChange for dsDetail
OnDataChange for dsMaster
```

The datasource *dsDetail* is told to refresh its view of the set of records displayed in the grid before *dsMaster* is told to refresh its view of the single record. (The events probably occur in that order to avoid displaying master and detail records that don't match, because the detail table is potentially slower to update — especially where the master-detail link is one-to-many.)

The events that occur when calling the various *TTable* methods for a detail table are similar to a single unlinked table. The important differences between these triggered events can be seen when calling the *Edit*, *Insert*, or *Delete* methods for *tblMaster*.

Calling *tblMaster.Edit* triggers the usual stream of events, as does calling *tblDetail.Edit*. However, watch what happens when calling *tblMaster.Post* when *tblDetail.State* is *dsEdit*:

```
OnUpdateData for dsMaster
OnUpdateData for dsDetail
```

```
BeforeCancel for tblDetail
  OnStateChange to dsBrowse for dsDetail
  Table state change to dsBrowse for tblDetail
  OnDataChange for dsDetail
AfterCancel for tblDetail
BeforePost for tblMaster
  OnStateChange to dsBrowse for dsMaster
  Table state change to dsBrowse for tblMaster
  OnCalcFields for tblMaster
  OnDataChange for dsMaster
AfterPost for tblMaster
```

According to Borland's documentation, the *OnUpdateData* event is triggered by *TTable.Post* or *TTable.UpdateRecord*. So why is *OnUpdateData* called for *dsDetail* when *tblDetail*'s current unmodified record is being canceled?

In fact, *tblDetail*'s record *was* modified when *tblMaster* entered edit mode. Remember the field in *tblDetail*'s current record that's used for the link with *tblMaster* will have been modified by *tblMaster*. So the record is being updated by a call to *TTable.UpdateRecord* that simply updates *tblDetail*'s record without posting it. The detail record is then canceled because, in effect, the record is unmodified (as there was no *OnDataChange* event triggered for any single field in *dsDetail*).

If *tblMaster.Cancel* is called instead of *tblMaster.Post* when *tblDetail.State* is set to *dsEdit*, the same sequence of events occurs except that *OnUpdateData* isn't called for *dsMaster*, only for *dsDetail*.

Another interesting sequence of events occurs when *tblMaster.Edit* is called when *tblDetail.State* is *dsEdit*:

```
OnUpdateData for dsDetail
BeforeCancel for tblDetail
  OnStateChange to dsBrowse for dsDetail
  Table state change to dsBrowse for tblDetail
  OnDataChange for dsDetail
AfterCancel for tblDetail
BeforeEdit for tblMaster
  OnCalcFields for tblMaster
  OnStateChange to dsEdit for dsMaster
  Table state change to dsEdit for tblMaster
  OnDataChange for dsMaster
AfterEdit for tblMaster
```

Placing *tblMaster* into edit mode actually cancels *tblDetail*'s unmodified record! What if *tblDetail*'s record had been modified and unposted? Then *tblMaster* would post the changes to *tblDetail*'s record before entering edit mode.

While it's all very well to discuss data control events, there's no substitute for using DBTracker yourself with various database scenarios to learn what happens when a certain button is pressed or method is called.

The Source Code

DBTracker was written with easy extensibility in mind (see Listing One beginning on page 12). Although just two tables are included in the form, more tables could be included and linked without any need for source code modifications. By simply copying and pasting *TDataSource* and *TTable* components, adding a few DBEdit components or a DBGrid component will provide all the functionality for tracking *TTable* and *TDataSource* events. Of course you'll need to set the relevant properties for your new components to make them work together.

Event Handlers

Tracking *TDataSource* and *TTable* events is pretty straightforward — just add the appropriate event handlers for all *TDataSource* and *TTable* objects. This doesn't have to be done for all DataSource and Table objects, just one of each. We can then copy and paste each object as needed at design time — each copy will use the same event handlers.

The event handlers log event information to the list box using the custom *LogEvent* method. Each event handler can identify the dataset or datasource that triggered the event by typecasting the *Sender* parameter to the appropriate type and then accessing the *Name* and/or *DataSet* properties.

The *OnStateChange* event handler, *TrackStateChange*, logs the new state and name of the datasource and checks to see if there is a *DataSet* property defined for *Sender* (the only parameter for a *TNotifyEvent*). If not, a warning message is logged. Otherwise the dataset's new state is logged. The datasource and dataset states will usually be the same.

We can identify *Sender* as a *TDataSource* by typecasting it as a *TDataSource* (using the **as** operator) and assigning it to `MySender`. We can do that because *TDataSource* is really a pointer to a dynamically allocated DataSource component.

With Delphi there isn't a "static" instance of a component or object because all components are created dynamically at runtime, and hence are referenced using pointers. So in effect, typecasting objects means "treat this pointer as a pointer to the type being cast as".

Whenever you reference an object's property or method, the pointer is automatically de-referenced so it doesn't seem as if you're dealing with a pointer. The following statement from *TrackStateChange* is pure pointer assignment — `MySender` points to the same dynamic object instance that `Sender` points to. Because *TObject* is the class all VCL objects are descended from, we can typecast it to the *TDataSource* descendant class that we know is the event sender:

```
MySender := Sender as TDataSource;
```

The following statement references the pointer so that the object's *Name* property can be accessed:

```
NameString := MySender.Name;
```

The *ToggleButtons* method called in *TrackStateChange* simply disables the relevant **Methods** buttons according to the state of the datasource.

The *OnDataChange* event is triggered for a number of reasons that were discussed earlier. This *TDataChange* event is handled by the *TrackDataChange* method. A *TDataChange* event has an additional parameter, called *Field*, that can be used to determine why *OnDataChange* was triggered. If *Field* is a nil pointer, then the event occurred as a result of moving from record to record in browse mode because more than one field contributed to the record changing. If *Field* is not nil and the table state is *dsEdit* or *dsInsert*, then the event occurred because a field's value changed and the unposted record was updated.

The *TTable* event handlers are very straightforward — they simply log each event with the sender's name. Most of the *TTable* events are triggered before and after a particular *TTable* action. When a record is posted, a *BeforePost* and *AfterPost* event is triggered. When a "before" event is logged all events subsequently logged are indented by one level. When an "after" event is logged the indenting is reduced by one level. This indentation displays which events occur between, before, and after events.

The `cboTables` *TComboBox* object lists the datasources that can be chosen for the **Methods** buttons to act on. The items in the *TStrings* object property of the ComboBox contain both the names of each datasource and a pointer to the named datasource instance.

A *TStrings* list is able to store both a string and pointer instead of just a string. Adding a string to a *TStrings* list is done using the *Add* or *Insert* methods. To add both the string and pointer to the list, use either the *AddObject* or *InsertObject* method (see the *FormCreate* method in Listing One). Strings and pointers to objects in the list can be accessed using the *Items* and *Objects* array properties respectively.

The *TStrings* object can be very useful for creating "bag" type arrays (containers) used to store items of different types. If you're familiar with Paradox's DynArray type, the same storage functionality can be achieved in Delphi by using a *TStrings* object.

As previously mentioned, DBTracker can easily be changed at design-time to include more *TTable*s and *TDataSource* components (whose properties can be set interactively) and then data - aware controls set up to work with them. All events will still be tracked without you having to modify any source code. The only caveat is that you must have a *TDataSource* called *dsMaster* and a *TTable* called *tblMaster* in the form. You may need to define a few field objects for those tables that are linked. You'll also need to define a calculated field called *MyCalcField* for *tblMaster*.

Extending DBTracker

I mentioned earlier that DBTracker was written with easy extensibility in mind. So how is this achieved? Let's start with the *FormCreate* method.

FormCreate checks the type of each component listed in the `Components` array property of the form to find all *TDataSource* components in the form at run-time. Each one found is added to the *TStrings* list of the `cboTables` *TComboBox* drop-down list labeled **Call Method for**. As mentioned, the *TStrings* list contains a list of names of *TDataSource* components in the form with a pointer to each *TDataSource* instance. This drop-down list allows the user to select the table to be affected when pressing one of the **Call Method** buttons.

The *TDataSource* and *TTable* variables `CurrentSource` and `CurrentTable` are used to track the DataSource and Table pair to be acted on by the **Methods** buttons. When the form is created `CurrentSource` is set to default to the first datasource found in the form. `CurrentSource's` *DataSet* property is used to initialize `CurrentTable`. If there isn't a *DataSet* property defined for `CurrentSource`, then `CurrentTable` cannot be set and the program will halt.

Assuming that `CurrentTable` is defined, all tables associated with the datasources are then opened. All *TTable*s must be not be open (i.e. their *Active* properties must be set to `False`) at design time. If a *TDataSource* has no *DataSet* property, a warning message is displayed stating the *TDataSource* has no *DataSet* and no associated table can be opened.

The two *TDBEdit* components are then set to have their *DataField* properties set to point to the first two fields in *tblMaster*. (Note that *tblMaster* doesn't necessarily have to be the master table in a multi-table form. I've used this as a convention to identify the table used by the two *TDBEdit* components for their data fields.) Finally, the form's buttons have their *Enabled* properties set.

The *TComboBox OnChange* event is handled by the custom *cboTablesChange* procedure. Here the `CurrentSource` and `CurrentTable` are set to that of the chosen datasource, and its *DataSet* property (if this is undefined then a warning error message is displayed). The **Methods** buttons have their *Enabled* properties set by the call to the custom *ToggleButtons* procedure.

Notice how the object pointer assignments were typecast using value typecasting, instead of the type-safe casting using the **as** operator. The difference between these ways of typecasting is this: if the typecast fails, then using **as** will raise an exception whereas the value typecast may cause undefined behavior later on (possibly causing a crash when the pointer is dereferenced). However, in this case we know that this particular typecast is safe.

The *SaveEventList* method that handles the *OnClick* event for the **Save List** button writes each item in the *TStrings* list to an ASCII file whose name is chosen using a SaveDialog component.

Conclusion

I have DBTracker set up as a form template in the gallery for those times when I need to create a quick form that can track the flow of database events in a complex multi-linked tables setup. I can quickly add those tables and datasources required by copying and pasting from the existing *TTable*s and *TDataSources*. Then all that's required is to set up the links between tables (and set their properties), place *TDBGrid* or *TDBEdit* components on the form (and set their properties), and run the form.

DBTracker was developed as a result of my need for a way of debugging code in a Data Access component's event handler. It also serves as a tool for learning and understanding how and when database events are triggered.

If desired, you could dig deeper into the data controls event model by logging the *TField* events, but I'll leave that exercise to you. ∆

The DBTracker utility is available on the 1995 Delphi Informant Works CD located in INFORM\95\JUL\JO9507.

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

Begin Listing One: FormCreate method

```
unit Track;

interface

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

const
    DbStates :
      array[dsInactive..dsCalcFields] of string[12] =
        ('dsInactive', 'dsBrowse', 'dsEdit',
         'dsInsert', 'dsSetKey', 'dsCalcFields');
type
  TFrmDBTrack = class(TForm)
    dsMaster: TDataSource;
    tblMaster: TTable;
    dsDetail: TDataSource;
    DBGrid1: TDBGrid;
    DBEdit1: TDBEdit;
    DBEdit2: TDBEdit;
    DBNavigator2: TDBNavigator;
    lstEvent: TListBox;
    btnClear: TBitBtn;
    tblDetail: TTable;
    DBEdit4: TDBEdit;
    Label1: TLabel;
    DBNavigator1: TDBNavigator;
    edCount: TEdit;
    cboTables: TComboBox;
    Label2: TLabel;
    GroupBox1: TGroupBox;
    btnInsert: TBitBtn;
    btnDelete: TBitBtn;
    btnEdit: TBitBtn;
    btnPost: TBitBtn;
    btnCancel: TBitBtn;
    btnOpen: TBitBtn;
    btnClose: TBitBtn;
    dlgSaveAs: TSaveDialog;
    btnSaveList: TBitBtn;
    tblMasterCustNo: TFloatField;
    tblMasterCompany: TStringField;
    tblMasterMyCalcField: TIntegerField;
    Label3: TLabel;
    Label4: TLabel;

    procedure btnInsertClick(Sender: TObject);
    procedure btnDeleteClick(Sender: TObject);
    procedure btnEditClick(Sender: TObject);
    procedure btnPostClick(Sender: TObject);
    procedure btnCancelClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
    procedure TrackDataChange(Sender: TObject;
                              Field: TField);
    procedure TrackStateChange(Sender: TObject);
    procedure TrackUpdateData(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure TrackAfterCancel(DataSet: TDataset);
    procedure TrackAfterClose(DataSet: TDataset);
    procedure TrackAfterDelete(DataSet: TDataset);
    procedure TrackAfterEdit(DataSet: TDataset);
    procedure TrackAfterInsert(DataSet: TDataset);
    procedure TrackAfterOpen(DataSet: TDataset);
    procedure TrackAfterPost(DataSet: TDataset);
    procedure TrackBeforeCancel(DataSet: TDataset);
    procedure TrackBeforeClose(DataSet: TDataset);
    procedure TrackBeforeDelete(DataSet: TDataset);
    procedure TrackBeforeEdit(DataSet: TDataset);
    procedure TrackBeforeInsert(DataSet: TDataset);
    procedure TrackBeforeOpen(DataSet: TDataset);
    procedure TrackBeforePost(DataSet: TDataset);
    procedure TrackOnCalc(DataSet: TDataset);
    procedure TrackNewRec(DataSet: TDataset);
    procedure btnOpenClick(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure cboTablesChange(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject;
                             var CanClose: Boolean);
    procedure SaveEventList(Sender: TObject);

  private
    ListLen: Word;
    IndentLevel: Byte;
    CurrentTable: TTable;
    CurrentSource: TDataSource;
    procedure LogEvent(EventStr: string);
    procedure ToggleButtons(StateParam: TDataSetState);
  public
  end;

var
  FrmDBTrack: TFrmDBTrack;

implementation

{$R *.DFM}

procedure TFrmDBTrack.ToggleButtons(StateParam:
                                    TDataSetState);
var
  ButtonEnabled: Boolean;
begin
  { If the table is in edit or insert mode then disable
    the Insert Delete and Edit buttons else disable the
    Post and Cancel buttons. If the table is closed/
    inactive then disable all pushbuttons. }
  ButtonEnabled :=
    not(StateParam in [dsEdit, dsInsert, dsInactive]);
  btnInsert.Enabled := ButtonEnabled;
  btnDelete.Enabled := ButtonEnabled;
  btnEdit.Enabled   := ButtonEnabled;

  if StateParam = dsInactive then
    begin
      btnPost.Enabled   := False;
      btnCancel.Enabled := False;
    end
```

```
  else
    begin
      btnPost.Enabled   := not ButtonEnabled;
      btnCancel.Enabled := not ButtonEnabled;
    end;
  ButtonEnabled    := (CurrentTable <> nil);
  btnOpen.Enabled  := ButtonEnabled;
  btnClose.Enabled := ButtonEnabled;
end;
procedure TFrmDBTrack.LogEvent(EventStr: string);
var
  IndentStr : string[32];
  i         : byte;
begin
  with lstEvent do
    begin
      IndentStr := '';
      if IndentLevel > 0 then
        begin
          for i := 1 to IndentLevel do
            IndentStr := IndentStr + '  ';
            Items.Add(IndentStr + EventStr);
        end
      else
        Items.Add(EventStr);
        Inc(ListLen);
        edCount.Text := IntToStr(ListLen);
        ItemIndex := Items.Count - 1;
    end;
    btnSaveList.Enabled := (ListLen > 0);
end;

procedure TFrmDBTrack.TrackDataChange(Sender: TObject;
                                      Field: TField);
var
  ds: TDataSource;
begin
  ds := Sender as TDataSource;
  if (Field = nil) then
    { Event was triggered by more than one field so
      event resulted from moving to another record }
    LogEvent('OnDataChange for ' + ds.Name)
  else
    if ds.State in [dsEdit, dsInsert] then
      { Event was triggered by a single field therefore
        this event resulted from a single field value
        being changed during editing }
      LogEvent('OnDataChange for ' + ds.Name + '.' +
               Field.FieldName)
    else
      { Event occurred for single field while
        not in Edit/Insert }
      LogEvent('OnDataChange for ' + ds.Name)
end;

procedure TFrmDBTrack.TrackStateChange(Sender: TObject);
var
  MySender: TDataSource;
begin
  MySender := Sender as TDataSource;
  LogEvent('OnStateChange to '+DbStates[MySender.State]+
           ' for ' + MySender.Name);
  { Don't try getting name of undefined dataset }
  if MySender.DataSet <> nil then
    LogEvent('Table state change to ' +
             DbStates[TTable(MySender.DataSet).State] +
             ' for ' + TTable(MySender.DataSet).Name)
  else
    LogEvent('WARNING -- No dataset defined for ' +
             MySender.Name);
    ToggleButtons(MySender.State)
end;

procedure TFrmDBTrack.TrackUpdateData(Sender: TObject);
begin
  with (Sender as TDataSource) do
    LogEvent('OnUpdateData for ' + Name);
end;

procedure TFrmDBTrack.btnInsertClick(Sender: TObject);
begin
  LogEvent('Begin TTable.Insert');
  Inc(IndentLevel);
  CurrentTable.Insert;
  Dec(IndentLevel);
  LogEvent('End TTable.Insert');
end;

procedure TFrmDBTrack.btnDeleteClick(Sender: TObject);
begin
  LogEvent('Begin TTable.Delete');
  Inc(IndentLevel);
  CurrentTable.Delete;
  Dec(IndentLevel);
  LogEvent('End TTable.Delete');
end;

procedure TFrmDBTrack.btnEditClick(Sender: TObject);
begin
  LogEvent('Begin TTable.Edit');
  Inc(IndentLevel);
  CurrentTable.Edit;
  Dec(IndentLevel);
  LogEvent('End TTable.Edit');
end;

procedure TFrmDBTrack.btnPostClick(Sender: TObject);
begin
  LogEvent('Begin TTable.Post');
  Inc(IndentLevel);
  CurrentTable.Post;
  Dec(IndentLevel);
  LogEvent('End TTable.Post');
end;

procedure TFrmDBTrack.btnCancelClick(Sender: TObject);
begin
  LogEvent('Begin TTable.Cancel');
  Inc(IndentLevel);
  CurrentTable.Cancel;
  Dec(IndentLevel);
  LogEvent('End TTable.Cancel');
end;

procedure TFrmDBTrack.FormCreate(Sender: TObject);
var
  i: word;
begin
  { Find all TDataSource objects on form and add them
    the TStrings list associated with the TCombobox }
  with cboTables do
    begin
      for i := 0 to Self.ComponentCount - 1 do
        if (Self.Components[i] is TDataSource) then
          { Add object name & pointer to the list }
          Items.AddObject(TDataSource(
            Self.Components[i]).Name,
            TDataSource(Self.Components[i]));
          { Form's main buttons will call TTable methods
            for CurrentTable which defaults to dataset
            of first TDataSource object found in form's
            component list. CurrentSource must have a
            dataset defined }
          CurrentSource :=TDataSource(Items.Objects[0]);
          if CurrentSource.Dataset <> nil then
            CurrentTable := TTable(CurrentSource.DataSet)
          else begin
            MessageDlg('The default datasource ' +
```

```
              CurrentSource.Name +
              #10' has no DataSet property defined.'#10+
              #10'This application will terminate',
              mtError, [mbOk], 0);
            Halt;
          end;
          IndentLevel := 0;
          { Open all tables in form, but check that each
            TDataSource in list has a dataset defined }
          for i := 0 to Items.Count - 1 do
            if (TDataSource(
                  Items.Objects[i]).DataSet = nil) then
              MessageDlg('The datasource ' +
                TDataSource(Items.Objects[i]).Name +
                ' has no DataSet property defined',
                mtWarning, [mbOK], 0)
            else
              TDataSource(Items.Objects[i]).DataSet.Open;
      end;
      DBEdit1.DataField := tblMaster.Fields[0].FieldName;
      DBEdit2.DataField := tblMaster.Fields[1].FieldName;
      ToggleButtons(CurrentSource.State);
      btnSaveList.Enabled := (ListLen > 0);
end;

procedure TFrmDBTrack.btnClearClick(Sender: TObject);
begin
  lstEvent.Clear;
  ListLen := 0;
  IndentLevel := 0;
  edCount.Text := IntToStr(ListLen);
  btnSaveList.Enabled := (ListLen > 0);
end;

procedure TFrmDBTrack.TrackAfterCancel(
                       DataSet: TDataset);
begin
  Dec(IndentLevel);
  with (DataSet as TTable) do
    LogEvent('AfterCancel for ' + Name);
end;

procedure TFrmDBTrack.TrackAfterClose(
                       DataSet: TDataset);
begin
  Dec(IndentLevel);
  with (DataSet as TTable) do
    LogEvent('AfterClose for ' + Name);
end;

procedure TFrmDBTrack.TrackAfterDelete(
                       DataSet: TDataset);
begin
  Dec(IndentLevel);
  with (DataSet as TTable) do
    LogEvent('AfterDelete for ' + Name);
end;

procedure TFrmDBTrack.TrackAfterEdit(DataSet: TDataset);
begin
  Dec(IndentLevel);
  with (DataSet as TTable) do
    LogEvent('AfterEdit for ' + Name);
end;

procedure TFrmDBTrack.TrackAfterInsert(
                         DataSet: TDataset);
begin
  Dec(IndentLevel);
  with (DataSet as TTable) do
    LogEvent('AfterInsert for ' + Name);
end;

procedure TFrmDBTrack.TrackAfterOpen(DataSet: TDataset);
begin
  Dec(IndentLevel);
  with (DataSet as TTable) do
    LogEvent('AfterOpen for ' + Name);
end;

procedure TFrmDBTrack.TrackAfterPost(DataSet: TDataset);
begin
  Dec(IndentLevel);
  with (DataSet as TTable) do
    LogEvent('AfterPost for ' + Name);
end;

procedure TFrmDBTrack.TrackBeforeCancel(
                       DataSet: TDataset);
begin
  with (DataSet as TTable) do
    LogEvent('BeforeCancel for ' + Name);
  Inc(IndentLevel);
end;

procedure TFrmDBTrack.TrackBeforeClose(
                       DataSet: TDataset);
begin
  with (DataSet as TTable) do
    LogEvent('BeforeClose for ' + Name);
  Inc(IndentLevel);
end;

procedure TFrmDBTrack.TrackBeforeDelete(
                       DataSet: TDataset);
begin
  with (DataSet as TTable) do
    LogEvent('BeforeDelete for ' + Name);
  Inc(IndentLevel);
end;

procedure TFrmDBTrack.TrackBeforeEdit(DataSet: TDataset);
begin
  with (DataSet as TTable) do
    LogEvent('BeforeEdit for ' + Name);
  Inc(IndentLevel);
end;

procedure TFrmDBTrack.TrackBeforeInsert(
                       DataSet: TDataset);
begin
  with (DataSet as TTable) do
    LogEvent('BeforeInsert for ' + Name);
  Inc(IndentLevel);
end;

procedure TFrmDBTrack.TrackBeforeOpen(DataSet: TDataset);
begin
  with (DataSet as TTable) do
    LogEvent('BeforeOpen for ' + Name);
  Inc(IndentLevel);
end;

procedure TFrmDBTrack.TrackBeforePost(DataSet: TDataset);
begin
  with (DataSet as TTable) do
    LogEvent('BeforePost for ' + Name);
  Inc(IndentLevel);
end;

procedure TFrmDBTrack.TrackOnCalc(DataSet: TDataset);
begin
  with (DataSet as TTable) do
    LogEvent('OnCalcFields for ' + Name);
end;

procedure TFrmDBTrack.TrackNewRec(DataSet: TDataset);
begin
```

```
  with (DataSet as TTable) do
    LogEvent('OnNewRecord for ' + Name);
end;

procedure TFrmDBTrack.btnOpenClick(Sender: TObject);
begin
  LogEvent('Begin TTable.Open');
  Inc(IndentLevel);
  CurrentTable.Open;
  Dec(IndentLevel);
  LogEvent('End TTable.Open');
end;

procedure TFrmDBTrack.btnCloseClick(Sender: TObject);
begin
  LogEvent('Begin TTable.Close');
  Inc(IndentLevel);
  CurrentTable.Close;
  Dec(IndentLevel);
  LogEvent('End TTable.Close');
end;

procedure TFrmDBTrack.cboTablesChange(Sender: TObject);
begin
  { Set CurrentSource and CurrentTable if user changes
    datasource to be acted on by Call Method buttons }
  with cboTables do
    CurrentSource :=
      TDataSource(Items.Objects[Items.IndexOf(Text)]);
  CurrentTable := TTable(CurrentSource.DataSet);

  if CurrentTable = nil then
    MessageDlg('The datasource ' + CurrentSource.Name +
               'has no DataSet property defined',
               mtWarning, [mbOK], 0);
  ToggleButtons(CurrentSource.State);
end;

procedure TFrmDBTrack.FormCloseQuery(Sender: TObject;
                                  var CanClose: Boolean);
begin
  if MessageDlg('Are you sure you wish to exit?',
                mtConfirmation,[mbYes,mbNo],0) <> mrYes
    then CanClose := False;
end;

procedure TFrmDBTrack.SaveEventList(Sender: TObject);
var
  LogFileName: string;
  LogFile: TextFile;
  i, j : Word;
begin
  { Write items in lstEvent to specified ASCII file }
  if dlgSaveAs.Execute then
    begin
      LogFileName := dlgSaveAs.Filename;
      AssignFile(LogFile, LogFileName);
      Rewrite(LogFile);
      j := lstEvent.Items.Count - 1;

      for i := 0 to j do
        writeln(LogFile, lstEvent.Items[i]);
      CloseFile(LogFile);
      MessageDlg('Events list written to ' + LogFileName,
                 mtInformation, [mbOK], 0);
    end;
end;

end.
```

End Listing One

By *Marco Cantù*

Moving to Local Interbase

The First Step Towards Upsizing a Delphi Application to Client/Server

U*psizing* is the process of taking an application that works on a PC and making it more robust by moving it to client/server. This is the opposite of *downsizing*, or taking a mainframe or mini-computer application and moving it to the client/server model.

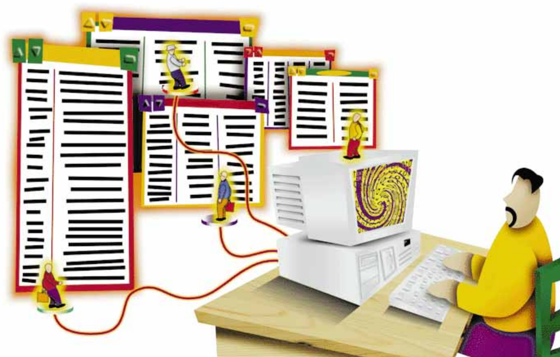
As we've heard, Delphi can be used to build client/server applications. Further, developing a new Delphi database application using a SQL server is similar to developing an application based on local files.

Having said this, however, what's involved in upsizing an application? To make it function, very little. And just a little more work is required to take advantage of SQL server features such as transaction processing.

Having What It Takes

In this article, we'll discuss what's involved in upsizing an existing Delphi database application (based on a local Paradox table), to a client/server application (based on a Local InterBase database). This can be seen as a first step in upsizing the application to use a remote SQL server. The next step — moving from a local SQL database to a remote SQL server — is generally quite simple, particularly if you use two versions of the same database.

To begin, I've built a simple database application that accesses a Paradox table (the sample Country.DB table that ships with Delphi), with a calculated field. The final version of the example will be similar, but connected to the Local InterBase server. In this article, we'll:



- Write an initial example program using a Paradox table.
- Write a program to copy the data from a Paradox table file to a Local InterBase database.
- Update the first example program to use it with the Local InterBase table.

Generating Code with the Database Form Expert

The first draft of the program can be built using the Database Form Expert. To do this, create a project and start the Expert by selecting **Help | Database Form Expert**. In the Expert, select the following:

Expert Grid

Name	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777815	32300003
Bolivia	La Paz	South America	1098575	7300000
Brazil	Brasilia	South America	8511180	150400000
Canada	Ottawa	North America	9976147	26500000
Chile	Santiago	South America	756943	13200000
Colombia	Bagota	South America	1138907	33000000
Cuba	Havana	North America	114524	10600000
Ecuador	Quito	South America	455502	10600000
El Salvador	San Salvador	North America	20865	5300000
Guyana	Georgetown	South America	214969	800000
Jamaica	Kingston	North America	11424	2500000
Mexico	Mexico City	North America	1967180	88600000
Nicaragua	Managua	North America	139000	3900000
Paraguay	Asuncion	South America	406576	4660000
Peru	Lima	South America	1285215	21600000
United States of America	Washington	North America	9363130	249200000

Figure 1: The expert-generated program, at run-time.

- a simple form based on a *TTable* object,
- the Country table (country.db) in the \delphi\demos\data directory,
- all the fields,
- and a grid for the output.

After generating the code, you can remove the previous blank form from the project (`Form1` by default), compile the program, give appropriate names to the files, and run it. The result (with some modifications) is shown in Figure 1.

Delphi's DBNavigator component is already transaction-oriented, so you can easily use it for transaction processing. In this example, the DBNavigator component's *VisibleButtons* property has been set so that *nbInsert*, *nbDelete*, *nbEdit*, *nbPost*, *nbCancel*, and *nbRefresh* are *False*. In addition, the *Caption* property of *Form1* was changed to `Expert Grid`.

The Fields of a Table

The grid of our example Expert Grid form includes all the fields in the source table. Of course, we could have removed some fields using the Database Form Expert, but how do we remove them once the code has been generated? And what about adding new fields?

The answer to these questions lies in the concept of the *field*. Field components (instances of the *TField* class) are non-visual components that are fundamental to each Delphi database application. Data-aware components are directly connected to these field objects, which correspond to database fields.

By default, *TField* components are automatically created by Delphi at run-time. This happens each time a DataSet component is active. These fields are stored in the *Fields* property of table and query components as an array of fields. We can access these values in a program with a statement like the following:

```
Table1.Fields[0].AsString
```

Alternatively, Field components can be created at design-time, using the Fields editor. In this case, you can set a number of properties for these fields. These properties affect the behavior of their data-aware components. When you define new fields at design-time, they are listed in the Object Inspector.

Creating a Calculated Field

To open the Fields editor, select the Table (or Query) component on the form, right-click to activate its SpeedMenu, and select **Fields editor**. Use the **Add** button to select which fields of the database table should be used by the Table or Query component (see Figure 2). [For an in-depth discussion of *TField* objects and the Fields editor, see Cary Jensen's article "The TField Object" in the June 1995 *Delphi Informant*.]

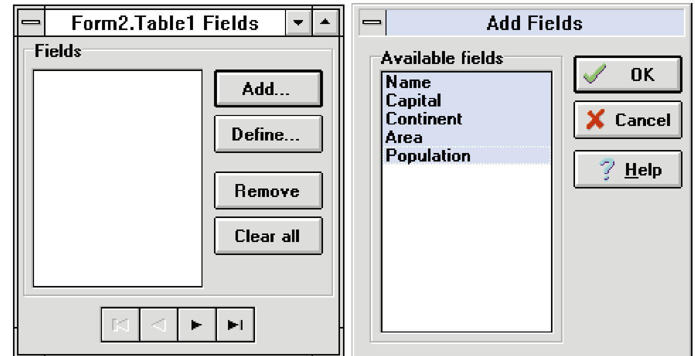


Figure 2: The Fields editor and Add Fields dialog box.

Click the **Define** button in the Fields editor to display the Define Field dialog box (see Figure 3). In this dialog box, you can define new calculated fields, and enter a descriptive field name that can include blank spaces. The dialog box generates an internal name — the name of the component — that you can modify.

Next, select a data type for the field, and check the **Calculated** box. For our calculated field, we'll use population and country area data from the Country table to compute the population density of each country.

To do this, we'll need to add another field to the table by clicking the **Add** button in the Fields editor. Then click on the **Define** button and enter a proper **Field name**, `Population Density` in this example, and **Field type** — `FloatField` — for the new calculated field. Note that as you type in the field name, the component name is generated by default.

Of course, we also need to provide a way to compute the new field. This is accomplished in the *OnCalcFields* event of the Table component with the following Object Pascal code:

```
procedure TForm1.Table1CalcFields(DataSet: TDataset);
begin
  Table1PopulationDensity.Value :=
    Table1Population.Value / Table1Area.Value;
end;
```

As you can see, this code accesses the other fields directly (e.g. `Table1Area.Value`). This is possible because the Fields editor has added the fields to the form. Here is the Object Pascal

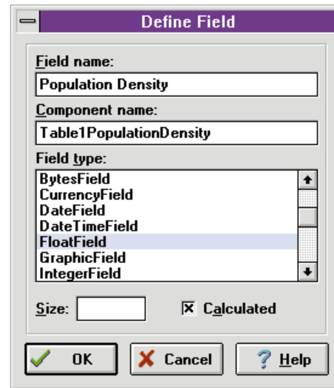


Figure 3: Defining a calculated field in the Define Field dialog box.

code the Fields editor added to the form definition:

```
Table1PopulationDensity: TFloatField;
Table1Area: TFloatField;
Table1Population: TFloatField;
Table1Name: TStringField;
Table1Capital: TStringField;
```

Also note that each time you add or remove fields in the Fields editor, the form's grid is automatically updated. Of course, you won't see the values of a calculated field because the method used to compute it must be compiled, and will be available only at run-time.

After defining components for the fields, we can use their properties to customize some of the grid's visual components. For example, as shown in Figure 4, we've changed the first column's name to **Country** (instead of **Name**). To do this, simply select Table1Name in the Object selector and change its *DisplayLabel* property to `Name`. Last, press Enter to accept the changes.

We also set a display format, adding a comma to separate thousands. For example, you can select Table1Area in the Object selector and enter `#,#` for its *DisplayFormat* property (likewise, select Table1Population and change its *DisplayFormat* property).

These changes have an immediate effect on the grid at design time. Also, the form's Caption property has been changed from Expert Grid to `Calculated Field`.

We can now compile and run the program. At run-time the grid will have proper values for the calculated field, **Population Density** (see Figure 5).

Copying a Table to the Server

Now that we have built the basic program, we can start the upsizing process by copying the table to a Local InterBase database.

There are basically two ways to copy a Paradox table to a SQL server. The first is to use an interactive tool, such as the Database Desktop. The other is to write a program in Delphi, based on the BatchMove component (located on the Data Access page of the Component Palette). We'll use the

Calculated Field

Country	Capital	Population	Area	Population Density
Argentina	Buenos Aires	32,300,003	2,777,815	
Bolivia	La Paz	7,300,000	1,098,575	
Brazil	Brasilia	150,400,000	8,511,180	
Canada	Ottawa	26,500,000	9,976,147	
Chile	Santiago	13,200,000	756,943	
Colombia	Bagota	33,000,000	1,138,907	
Cuba	Havana	10,600,000	114,524	
Ecuador	Quito	10,600,000	455,502	
El Salvador	San Salvador	5,300,000	20,865	
Guyana	Georgetown	800,000	214,969	
Jamaica	Kingston	2,500,000	11,424	
Mexico	Mexico City	88,600,000	1,967,180	
Nicaragua	Managua	3,900,000	139,000	
Paraguay	Asuncion	4,660,000	406,576	
Peru	Lima	21,600,000	1,285,215	
United States of America	Washington	249,200,000	9,363,130	
Uruguay	Montevideo	3,002,000	176,140	
Venezuela	Caracas	19,700,000	912,047	

Calculated Field

Country	Capital	Population	Area	Population Density
Argentina	Buenos Aires	32,300,003	2,777,815	11.63
Bolivia	La Paz	7,300,000	1,098,575	6.64
Brazil	Brasilia	150,400,000	8,511,180	17.67
Canada	Ottawa	26,500,000	9,976,147	2.66
Chile	Santiago	13,200,000	756,943	17.44
Colombia	Bagota	33,000,000	1,138,907	28.98
Cuba	Havana	10,600,000	114,524	92.56
Ecuador	Quito	10,600,000	455,502	23.27
El Salvador	San Salvador	5,300,000	20,865	254.01
Guyana	Georgetown	800,000	214,969	3.72
Jamaica	Kingston	2,500,000	11,424	218.84
Mexico	Mexico City	88,600,000	1,967,180	45.04

Figure 4 (Top): The grid at design time, after some changes to the properties of the field components. **Figure 5 (Bottom):** The output of the example, with the calculated **Population Density** field.

second approach to introduce the BatchMove component. Create a new project and place two Table components, a BatchMove component, and a Button component on the blank form. Change the *Caption* property of *Button1* to `Move`, and the form's *Caption* property to `Move Countries` (see Figure 6).

Connect the first table to the original Paradox table (Country.DB). We can immediately open it by setting its *Active* property to *True*. The second table should relate to the target database, indicating the name of the new table. (We cannot select Country as the target name, because there is already a table with this name in the IBLOCAL database.) Instead, we'll use America, since there are only American countries in the table. Once each table is properly set up, use them for the *Source* and *Destination* properties of the BatchMove component.

Last, set a proper value for the BatchMove component's *Mode* property. The default value is *batAppend* which appends records to an existing destination table. You can also set the following values:

- *batUpdate* updates matching records in an existing table

- *batAppendUpdate* updates matching records of, and appends new records to, an existing table
- *batCopy* creates a destination table with structure and content of source table
- *batDelete* deletes matching records in existing table

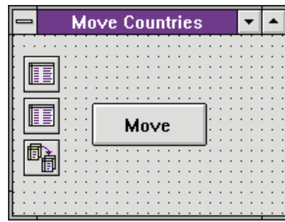


Figure 6: The example form, Move Countries (shown here at design time), is used to copy the tables.

The *batCopy* parameter is what we're looking for so select it.

Next, right-click on the form to activate its SpeedMenu, and select the **Execute** command. Delphi will create the new table (after asking for log-in information to the Local InterBase server).

We've done it! And without compiling the program! As an alternative, we can write one line of code for the **Move** button's *OnClick* event handler:

```
BatchMove1.Execute;
```

Now we're ready for the second step of the upsizing process.

Porting the Application

Now that the table has been moved to the Local InterBase server, we can turn our attention to modifying the application.

Open the old application (or a copy), select the Table component, and set its *Active* property to `False`. Then select the IBLOCAL database and the America table (that should appear in the list of available tables in the Object Inspector).

Activate the table again, and the live data is displayed. The difference is that now we are accessing a SQL server. When you run the program, the new calculated field will appear, exactly as it did in the previous version.

Using the Visual Query Builder

To build a more advanced version of our example, you can use a Query component instead of a Table component. Simply remove the Table component and add a Query component. Connect the data source to the Query component and write the appropriate SQL statement.

Since we are exploring the development of client/server applications, we'll use Delphi's Visual Query Builder to create the SQL statement. (Note that the Visual Query Builder is available only with the Client/Server version of Delphi.) To access the Visual Query Builder, right-click on the Query component to display its SpeedMenu. Then select **Query Builder** to display the Databases dialog box. Select the IBLOCAL database and enter the password. When the Add Table dialog box appears, select the America table. Then close the dialog box.

There are two ways to add columns to the answer set in the lower portion of the Visual Query Builder (see Figure 7). You can drag a column from the table window (**AMERICA:AMERICA** in our example) to the bottom half of the screen, or simply double-click on each column in the table window.

Next, we'll define a calculated field directly in the query. Click on the Expression button in the Visual Query Builder's toolbar (it has an "a + b" icon). In the Expression dialog box (see Figure 8), enter the **Expression Name** (**DENSITY** in this example), click on the **POPULATION** column, the division operator (**/**), and then on the **AREA** column. The expression will appear in the **Expression** box.

If it's correct, click on the **Close** button to return to the Visual Query Builder. A new column will be added to the table resulting from the query. Now you can click the Run button (a lightning bolt) to see the result. The calculated field will appear as if it were an original field of the table.

You can also look at the code of the SQL query by clicking on the SQL (eyeglasses) button:

```
SELECT AMERICA.NAME, AMERICA.CAPITAL,
       AMERICA.AREA, AMERICA.POPULATION,
       (AMERICA.POPULATION/AMERICA.AREA) as Density
FROM AMERICA AMERICA
```

Click on the OK (green check mark) button to copy the text of the query back to the Query component in the form. Now activate it, and you'll see the fields of the form in the grid, including the calculated **Density** field, at design-time. (If you do not own the Delphi Client/Server edition, simply write the SQL statement above in the *SQL* property of the Query component.)

There is still a minor problem, however. The division calculation we wrote in the SQL expression results in a floating-point number with several decimal places. The problem is that the default grid is not wide enough to accommodate all

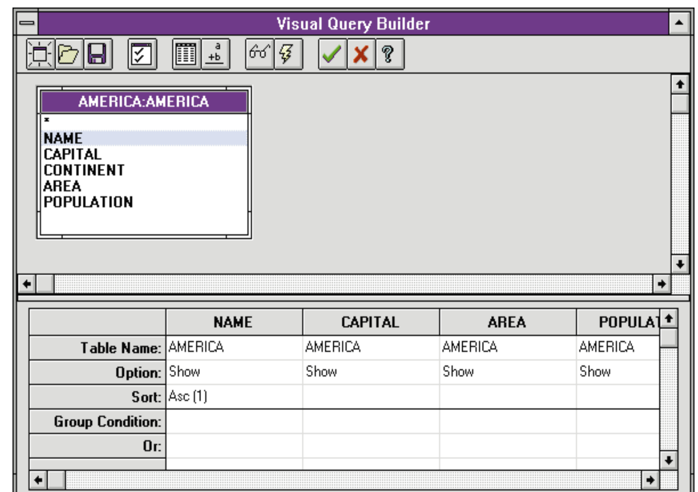
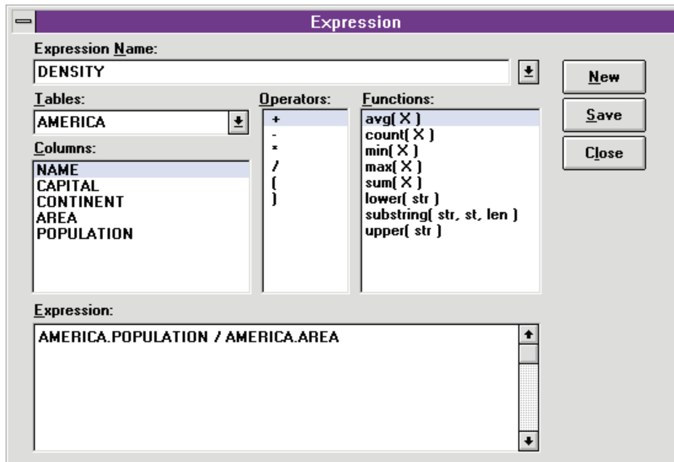


Figure 7: The Visual Query Builder.



Calculated Field

NAME	CAPITAL	AREA	POPULATION	DENSITY
Argentina	Buenos Aires	2777815	32300003	11.6278452668734
Bolivia	La Paz	1098575	7300000	6.64497189540996
Brazil	Brasilia	85111968	150400000	1.76708403687716
Canada	Ottawa	9976147	26500000	2.6563361586392
Chile	Santiago	756943	13200000	17.4385653873541
Colombia	Bagota	1138907	33000000	28.9751489805577
Cuba	Havana	114524	10600000	92.5570186161853
Ecuador	Quito	455502	10600000	23.2710284477346
El Salvador	San Salvador	20865	5300000	254.013898873712
Guyana	Georgetown	214969	800000	3.72146681614558
Jamaica	Kingston	33000	2500000	75.7575757575758
Mexico	Mexico City	1967180	88600000	45.0390914913734
Nicaragua	Managua	139000	3900000	28.0575539568345
Paraguay	Asuncion	406576	4660000	11.4615717602613
Peru	Lima	1285215	21600000	16.8065265344709
United States of America	Washington	9363130	249200000	26.6150315118983
Uruguay	Montevideo	176140	3002000	17.0432610423527
Venezuela	Caracas	912047	19700000	21.5997640472476

Figure 8 (Top): The Expression dialog box of the Visual Query Builder can be used to define calculated fields. **Figure 9 (Bottom):** The form with the SQL-calculated **Density** field at design time.

the numbers, so the values don't appear correctly. To solve the problem, we simply need to resize the grid's columns.

However, this is possible only if you define Field components for the query, using the Fields editor. Again, right-click on the form to access the Fields editor and add all the fields. Now, you can resize the grid's columns to obtain the desired effect (see Figure 9).

From Porting to Upsizing

In the last example, we obtained two interesting advantages. The first and most evident advantage is that now we can display a calculated field at run-time. This is possible because we do not need compiled Object Pascal code to make this computation. Instead, the Local InterBase server does it while processing a SQL statement.

This is a key point. We have moved a processing task from the client application to the SQL server. This means that we can move computations from the client computer to the server computer, which can usually handle the large amount of data involved in big queries more quickly and efficiently.

Although, admittedly, we are currently running both the client and server code on the same computer (since we're using Local InterBase), this doesn't modify the general perspective. A "true" client/server application distributes the processing workload of an application between a client and server computer. The client should be primarily involved in the user interface, and the server in data processing.

Conclusion

To obtain all the benefits of "true" client/server programming, we would need to go one step further and employ a SQL server that resides on another computer. The easiest migration would be to an InterBase database on another platform (e.g. Microsoft Windows NT, Novell NetWare, SCO UNIX, etc.).

Still, just by moving the data to Local InterBase, we receive many advantages over a Paradox database. Greatly enhanced data integrity via triggers, stored procedures, and transaction processing is just one example. ∆

This article was adapted from material for Marco Cantù's book, *Mastering Delphi*, published by Sybex.

The demonstration projects referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\JUL\MC9507.

Marco Cantù is a freelance writer and consultant based in Italy. Besides writing books and articles, he enjoys training Delphi and C++ programmers and speaking at conferences. You can reach Marco on CompuServe at 100273,2610.

By Cary Jensen Ph.D.

The TField Class: Part II

Using Calculated Fields

T*Field* components are a class of objects that permit you to directly manipulate the data displayed on a form. These objects are created automatically at run-time or manually at design time. By creating *TFields* at design time you can exert greater control over the default characteristics of the corresponding fields. For example, you can display only some of the fields from a table in a DBGrid component.

Last month's article examined how to use the Fields editor dialog box to instantiate fields associated with a Table object, and how to control their properties both at design time and run-time. This month we will explore *TFields* further. However, instead of instantiating *TFields* that belong to a Table, we will learn how to create calculated fields. [For a discussion of calculated fields in a client/server configuration, see Marco Cantù's article "Moving to Local InterBase" beginning on page 16.]

Reckoning with Data

Calculated fields differ from other *TFields* in one very important way—they are not associated with data in your table. Furthermore, the value displayed in a calculated field is always defined using Object Pascal code. In other words, as a developer you are responsible for assigning values to calculated fields.

At first thought this seems very inconvenient. Since most data control components (e.g. DBGrid, DBText, or DBEdit components) automatically display data from their associated *DataSource*, you may wonder why anyone would want to use a field requiring you to add code to display data.

However, when you use calculated fields to display data that is not stored in a *DataSource* the answer becomes more apparent. There are two instances where this is particularly valuable: displaying calculations and lookup data. Let's consider calculations first.

Displaying Calculated Values

Imagine you have a table of individual items being purchased by a customer for a particular order (a "line items" table). At a minimum this table holds the item Identification (ID) for each item being purchased, along with the corresponding price and quantity.

In most applications such a table doesn't include a total field to hold the product of price and quantity. Since the total can always be calculated from the data by multiplying the item price times the quantity ordered, a total field is redundant.

Form1

Truly Deep Diving Custom Order Form

Order Number 1004

Customer Code 2156 Davy Jones' Locker

Employee Code 145 Guckenheimer

Line Items:

ItemNo	PartNo	Description	Qty	Price	Total
1	1313	Regulator System	10	$250.00	$2,500.00
2	12310	Sonar System	10	$439.00	$4,390.00
3	3316	Stabilizing Vest	8	$430.00	$3,440.00
4	5324	Chisel Point Knife	5	$41.00	$205.00

Exit

Figure 1: As shown here, the total of price and quantity, as well as the custom company name and employee last name fields, are displayed using calculated fields.

Although storing a total value in the table isn't necessary, it's often desirable to *display* this value on a form. It is for this purpose that a calculated field is intended. Using Object Pascal code you can calculate the product of price and quantity, and assign this value to the calculated field. An example of a DBGrid that displays the product of the **Price** and **Quantity** fields in a calculated field (named **Total**) is shown in Figure 1.

The amount in the **Total** field is easily calculated. Simply attach the code that performs the assignment for calculated fields to the *OnCalcFields* event associated with the *TTable* component for which you have defined the calculated field. For example, suppose you have a Table component named `Table1` attached to the table named Items.DB and have created a calculated field named **Amount**. To assign a value to **Amount**, place the appropriate code in the *OnCalcFields* event associated with `Table1`.

Displaying Lookup Data

The second purpose of calculated fields is to display lookup data. *Lookup data* is information about one record that is stored in another table. For example, the table of items being sold to a customer must contain a part number (or some other value that identifies what the customer is purchasing). However, such a table rarely contains a description of the product. This description is normally located in another table — a parts table that holds one record for each part.

To display a part description from a parts table, without having to also enter and store this information in a line items table, create a calculated field that displays the description. Then, update the value of this field from the *OnCalcFields* event. Each time an *OnCalcFields* event triggers, you can locate the corresponding part record in the parts table and assign its description to the calculated field.

The *OnCalcFields* Event

The *OnCalcFields* event is triggered each time Delphi needs to update the data displayed in this field. This happens frequently. For example, if you edit a value displayed in a DBEdit component attached to a DataSource component, the *OnChange* event for that field will trigger an *OnCalcFields* event.

OnCalcFields events occur even when no change has been made to data. Each time it's necessary for a Table object to update the display of a record, an *OnCalcFields* event occurs. For example, *OnCalcFields* events occur for each displayed record when a form first opens, as well as when you advance to a new record.

If the DataSet (*TTable*, *TQuery*, or *TStoredProc*) is responsible for more than one record being displayed, the *OnCalcFields* event occurs once for each record. For example, if a DBGrid is associated with a detail table in a one-to-many form, there will be one *OnCalcFields* event generated for each record being displayed in the DBGrid when the form first initializes, as well as when the user advances to a new master record.

Adding Code to *OnCalcFields*

The *OnCalcFields* event doesn't occur unless you have instantiated at least one calculated field for the associated DataSet component. To do this use the Fields editor dialog box. Use the following steps to create a form on which you'll place a table that includes a calculated field.

Begin by creating a new project. Place the following components on the form: a DataSource, Table, DBNavigator, and DBEdit. Finally, place a Label component to the left of the DBEdit object (see Figure 2).

Display the Object Inspector and change the properties of your objects as follows: Set the *Caption* property of the Label to `Order Number:`. Set the DataSource's *DataSet* property to `Table1`. Set the DBNavigator's *DataSource* property to `DataSource1`.

Next, select the Table object and in the Object Inspector change its *DatabaseName* property to `DBDEMOS`. (This is the default alias created when you installed Delphi. If this alias doesn't exist, either create and use an alias for the directory where Delphi installed its sample database tables, or use the entire path in the *TableName* property.) Change the *TableName* property to `ORDERS.DB` (If you were able to set *DatabaseName* to `DBDEMOS`, you can select this table name from the *TableName* drop-down

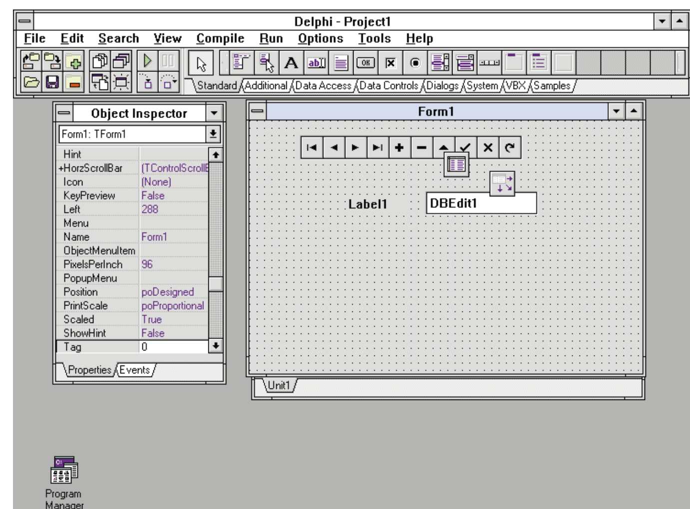


Figure 2: The *OnCalcFields* event demonstration form.

list.) You should now activate the connection to the table by setting *Active* to `True`.

Now set the DBEdit component properties by changing the *DataSource* property to `DataSource1` and setting its *DataField* property to `OrderNo`.

You are now ready to instantiate *TField* components. From the Form's window, double-click the Table object to display the Fields editor dialog box. First, add a *TField* component for the **OrderNo** field (from the ORDERS.DB table). Do this by clicking the **Add** button on the Fields editor dialog box, highlighting **OrderNo** from the **Available Fields** list, and then clicking **OK** to return to the Fields editor dialog box.

Next, add the calculated field. Click the **Define** button on the Fields editor dialog box to display the Define Field dialog box. In the **Field Name** text box enter `CalcDemo`. Delphi will automatically add `Table1CalcDemo` in the **Component Name** text box. You can optionally define an alternative component name, but this is usually not necessary.

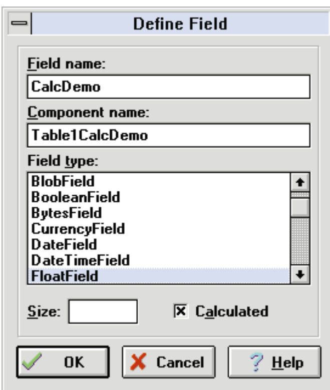


Figure 3: Use the Define Field dialog box to create calculated fields.

At this point, select the type of the calculated field. This field type must correspond to the data type you'll assign to this calculated field. In this example no data will be assigned to the calculated field, so the field type is irrelevant. For simplicity's sake, select **FloatField** (this field type does not require a size). The Define Field dialog box should look like Figure 3. Click the **OK** button.

The Fields editor dialog box should emulate the one displayed in Figure 4. The objects and properties are now in place. Because you instantiated the calculated field, an *OnCalcFields* event will be generated each time Delphi attempts to update the display of the DBEdit component. Close the Fields editor dialog box.

To see that this event is occurring, select the Table component and go to the Object Inspector's Events page. Double-click the *OnCalcFields* event to display the event handler. Enter the following code in the *Table1CalcFields* procedure:

```
Application.MessageBox('OnCalcFields', 'Here is one.',mb_OK)
```

Your screen should look like Figure 5.

You're done! Press F9 to compile and run your project.

Notice that even before the running form is visible, the message box is displayed as a result of an *OnCalcFields* event. This *OnCalcFields* event is produced when the form is initialized.

After you respond to this message box, the form will appear. Thereafter, the message box is displayed each time you move to a new record. It will also display if you edit the order number. (However, you shouldn't do this since the Orders table has detail records that rely on the value of the order number field.)

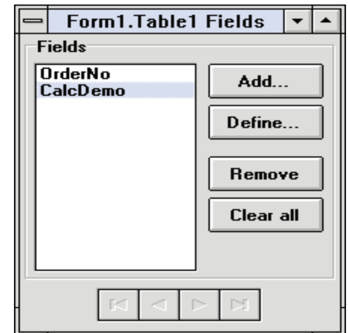


Figure 4: The Fields editor dialog box with one "regular" field and one calculated field.

More About *OnCalcFields* Events

There are a number of special characteristics of *OnCalcFields* events you need to be aware of to use them effectively. First, they occur frequently. As mentioned earlier, every time a user advances to a new record, or changes a value in a single field, an *OnCalcFields* event triggers for that record.

In some cases, advancing to another record can result in several *OnCalcFields* events. This occurs when you advance to another master table record and you have calculated fields in a detail table. Under these conditions, *OnCalcFields* events trigger for each record in the restricted view of the detail table. In short, it's usually desirable to keep the code you attach to the *OnCalcFields* events short and quick. It's important to remember that placing time-consuming operations in the *OnCalcFields* event can greatly affect your form's performance.

Another limitation of *OnCalcFields* events results from the special state in which Delphi places itself during the event. During the *OnCalcFields* event Delphi sets the *State* property of the associated Table, Query, or StoredProc object to *dsCalcFields*. When in this state, Delphi will not allow you to assign new values to *TField* objects that are not calculated fields. In other words, the only type of field you can assign a value to during the *OnCalcFields* event is a calculated field. (You can determine if a given field is a calculated field by inspecting its *Calculated* property.)

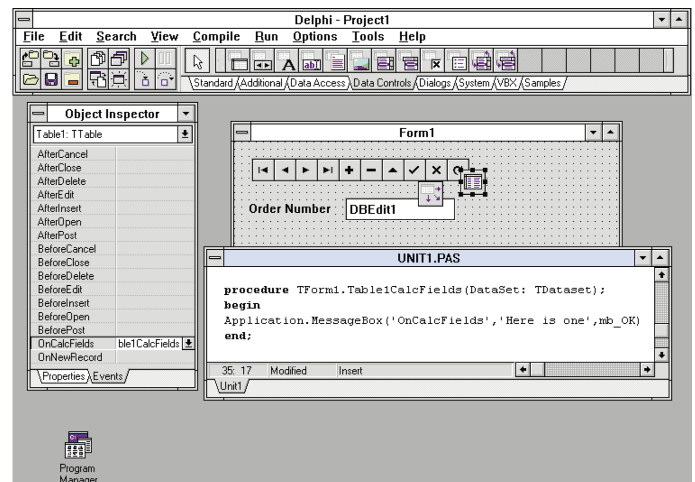


Figure 5: Sample code added to an *OnCalcFields* event handler.

Also, be mindful that *OnCalcFields* is triggered by many events that modify a DataSet. For example, navigating to a new record may generate one or more *OnCalcFields* events. You must avoid triggering these events with the code you add to the *OnCalcFields* event handler. Failure to do so may result in unwanted and uncontrollable recursion.

Under normal conditions, an *OnCalcFields* event triggers automatically each time you modify a field in a record. This occurs because the *AutoCalcFields* property of *TTable*, *TQuery*, and *TStoredProc* components is *True* by default. If you change the *AutoCalcFields* property to *False*, *OnCalcFields* events will only be triggered automatically when Delphi first loads and displays a record from a database.

Finally, remember that *OnCalcFields* events occur when objects are initialized on an opening form. Consequently, the creation order of objects can have an impact on the success of the code you place in an *OnCalcFields* event. For example, if you are using an *OnCalcFields* event to display lookup data, the lookup table must be created and opened first. You can easily control the creation order of objects on a form by selecting **View | Creation Order** from Delphi's menu.

Creating a Calculation and Lookup Fields Example

The following example builds on the project we created with the steps described earlier. (If you did not follow those steps, and want to create this example, simply refer to the previous example.)

In this example you'll add a DBGrid to the form. The modified form will display the line item details for a given order number. This DBGrid will also display information from three calculated fields. Two of these fields will display lookup information from another table, and the third field will display the results of a calculation.

First, add two additional Table and two additional DataSource objects to the form. Then add a DBGrid component. When you are done your form should look similar to Figure 6.

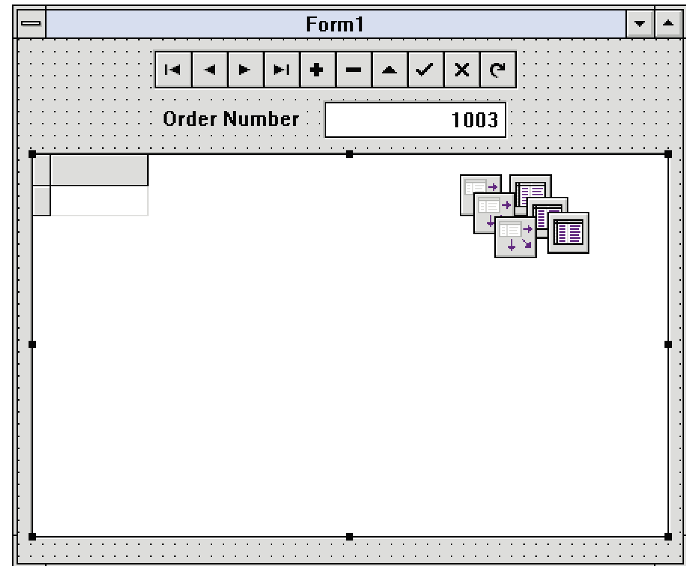


Figure 6: The modified form after adding five objects.

Select *DataSource2* and change its *DataSet* property to `Table2`. Select *DataSource3* and change its *DataSet* property to `Table3`. Select *Table2* in the Object selector and set its *DatabaseName* to `DBDEMOS`, its *TableName* property to `ITEMS.DB`, and its *Active* property to `True`. Likewise, select *Table3* and set its *DatabaseName* property to `DBDEMOS`, its *TableName* property to `PARTS.DB`, and its *Active* property to `True`.

Double-click on the *Table2* object to display its Fields editor dialog box. Select **Add** to display the Add Fields dialog box and select all fields. Click **OK** to return to the Fields editor. Now add the calculated fields. Click **Define** to display the Define Field dialog box. At **Field Name** enter `Description`, at **Field Type** select `StringField`, and at **Size** enter `25`. Click **OK**.

Click **Define** again to add the second calculated field. At **Field Name** enter `Price`, and at **Field Type** select `CurrencyField`. Click **OK** to return. Click **Define** again and at **Field Name** enter `Amount` and at **Field Type** select `CurrencyField`. Click **OK** to return. Your Fields editor dialog box should look like the one shown in Figure 7.

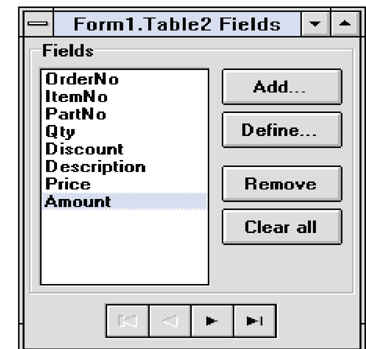


Figure 7: Fields instantiated for *Table2*.

Select the DBGrid and set its *DataSource* property to `DataSource2`. Delphi will now display live data in the DBGrid. The fields in the DBGrid will appear in the order that they appeared in the Fields editor dialog box.

Modifications

You'll want to make a few modifications to the DBGrid's display. First, remove the **OrderNo** field from display. This field is part of the master record, and is displayed in the DBEdit object. To do this, select the *Table2OrderNo* object and set its *Visible* property to `False`.

You'll also want to modify the order of the fields in the DBGrid. You can change a column's location by dragging the column heading for a field in the DBGrid to a new location. As you drag the mouse, the column will move. Drop the columns in the order you want them to appear. Probably the most reasonable order for these columns is **ItemNo**, **Price**, **Description**, **PartNo**, **Qty**, **Discount**, and **Amount**. (You can also change the order of the fields in a DBGrid by dragging the field names to new locations within the Fields editor window.)

You may want to make additional changes to the DBGrid to make it look better. For example, you can increase the size of the form, and then increase the size of the DBGrid. You can adjust the size of the individual fields in the DBGrid by drag-

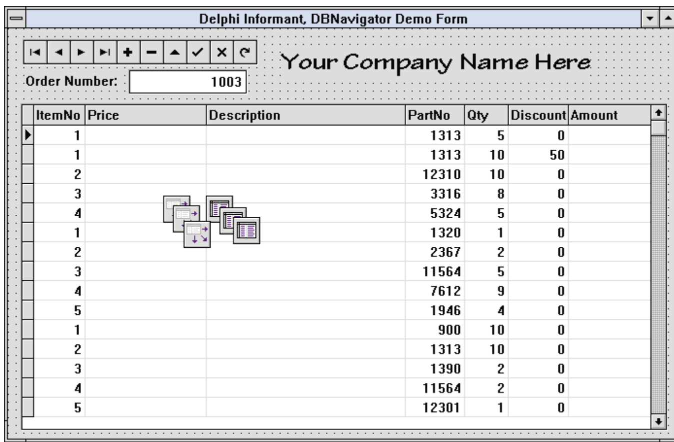


Figure 8: The example form has been spruced up with a Label component. The non-visual objects were placed over the DBGrid to remain unobtrusive during design.

ging the left column marker of the column header. Figure 8 shows an example of how the customized form might look.

It's now time to add the *OnCalcFields* event-handling code. Select *Table2* and display the Events page of the Object Inspector. Double-click on the *OnCalcFields* event to display its event handler. Enter the following code:

```
procedure TForm1.Table2CalcFields(DataSet: TDataSet);
var
  SubTotal : real;
begin
if Table3.FindKey([Table2PartNo.value]) then
  begin
    Table2Price.value := Table3ListPrice.value;
    Table2Description.value := Table3Description.value;
    SubTotal := Table2Price.AsFloat * Table2Qty.AsFloat;
    Table2Amount.value :=
      SubTotal - (SubTotal*(Table2Discount.AsFloat/100));
  end;
end;
```

Notice that this code first attempts to locate the current value displayed in the *Table2PartNo* field in *Table3*. It does this by using the *FindKey* method. (Use the on-line help for more information about *FindKey*.)

If *FindKey* returns *True* it means that a record in *Table3* has been located and corresponds to the part number in *Table2*. Under this condition the lookup fields can be assigned. The value in the *ListPrice* field of *Table3* is assigned to the calculated field *Table2Price*. And the value in the *Description* field of *Table3* is assigned to the calculated field *Table2Description*. This completes the lookup operation.

The last two steps in this code are used to produce the calculated field. The values displayed in *Table2Price* (a lookup field) and *Table2Qty* are multiplied, and then divided by the percentage represented by *Table2Discount*. The results of this calculation are assigned to the calculated field *Table2Amount*. (Note: This operation was performed in two steps for the sake of clarity. The entire calculation can be performed in a single statement.)

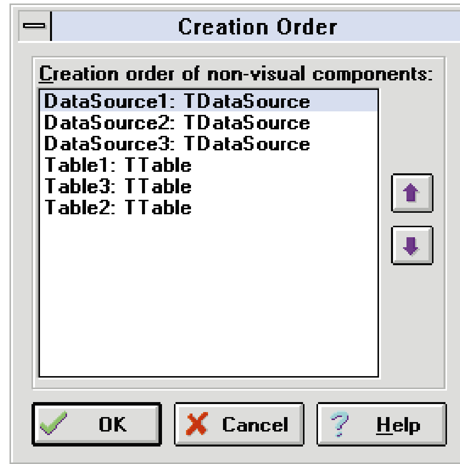
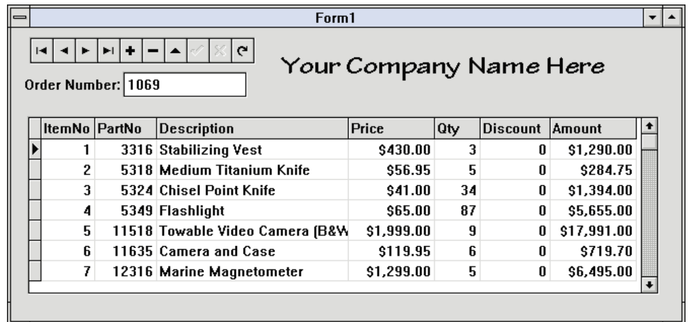


Figure 9 (Top): Adjusting the creation order to ensure that the *OnCalcFields* event does not try to reference *Table3* before it's created. **Figure 10 (Bottom):** A form making use of calculated fields.



There is one last required step. The code attached to *Table2*'s *OnCalcFields* event makes reference to *Table3*. Since this code will be executed when *Table2* is initialized, it's essential to ensure that *Table3* is created before *Table2*. You do this by selecting **Edit | Creation Order** from Delphi's menu. Drag *Table3* to a position above *Table2*, as shown in Figure 9. Complete this operation by closing the Creation Order dialog box.

This step is optional, although recommended. Remove the reference to the *Table1CalcFields* procedure from the *OnCalcFields* event for *Table1* because it is no longer needed.

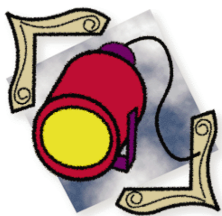
Press F9 to compile your project and run the form. Once the form is running, your screen will look like Figure 10. Recall that the **Description** and **Price** fields are produced by looking up the Parts table. Also, the **Amount** field is produced by a calculation.

Conclusion

Calculated fields are an important part of database programming. Using these fields you can display related data from other tables, and perform calculations to display data that doesn't otherwise need to be stored in the database. ∆

The demonstration projects referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\JUL\CJ9507.

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



INFORMANT SPOTLIGHT

DELPHI / OBJECT PASCAL

By Robert Vivrette

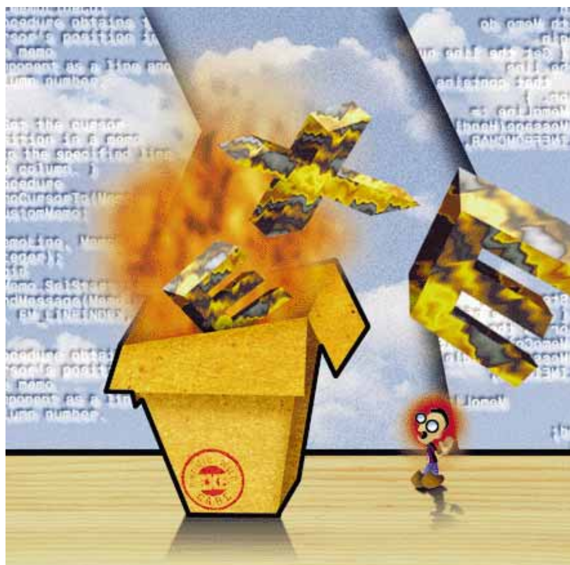
A Question of Size

Or, Why Are Delphi Executables So Big?

Everyone loves small, efficient programs. Perhaps that explains all the hubbub. Since the release of Delphi, there has been quite a bit of discussion about the size of executable files created with Delphi. In this article we'll discuss some of the issues relating to executable size, and present a detailed picture of what *exactly* goes into a Delphi executable.

First, let's see if we can understand the issue a little better. I created an example "do-nothing" Windows application with Delphi and with Borland Pascal 7. The application presents a normal, resizeable main window (see Figure 1). The main window (by default) has a control box in the upper-left corner, minimize and maximize buttons in the upper-right corner, and can be resized. There are no buttons, fields, or controls on the form. This type of application is what you get if you choose **File | New Project** in Delphi and then immediately compile the result. To see how big these resulting executable files are, take a look at Figure 2.

As you can see, Delphi applications are larger by more than a factor of 10. By using **Options | Project** and selecting Delphi's **Optimize for size and load time** option, the final executable is reduced by about 37KB. However, both options are nowhere near the size of a Borland Pascal 7 executable. Everyone else seemed to notice this as well. Borland's answer to this is accurate, if a bit vague. Here is the related excerpt from one of Borland's FAQ (frequently asked questions) sheets:

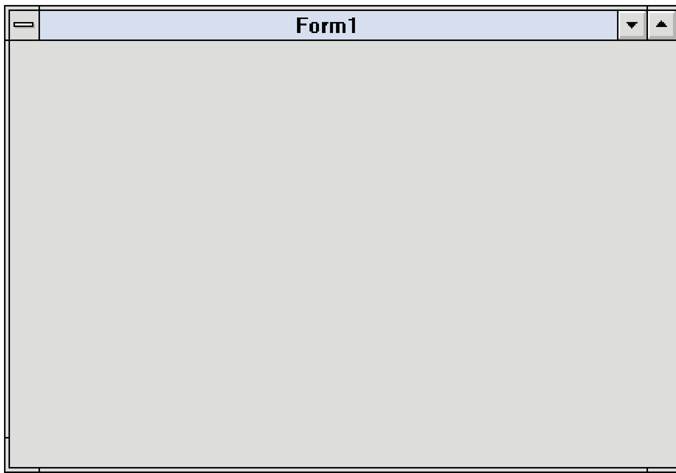


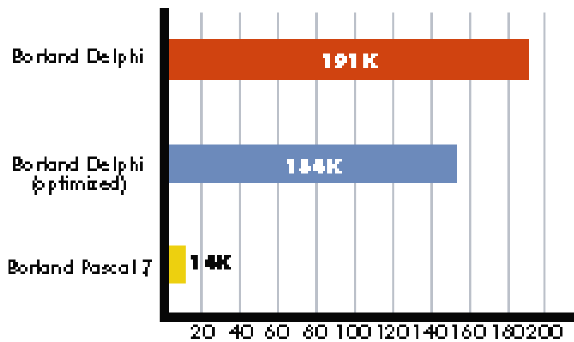
"*Delphi's VCL is based on RTTI and exceptions. This requires a 'footprint' of about 120Kb for an 'empty' application. The 200K you get has additional debug info or is not optimized by the compiler. Note that the size of your .EXE doesn't go to 400K for two buttons, but rather to 201K, i.e. after the footprint each additional control just adds the 'usual' amount of data/code size. Additionally, you can slim your* .EXE*'s down by checking the "Optimize For Size And Load Time" checkbox on the Linker page of the Options | Project dialog.*"

That's a satisfactory answer, but let's dig a little deeper. What is Delphi *doing* in that extra 140KB that has no apparent visual or functional effect on the program? Let's uncover the truth.

Getting Down to Basics

To answer this question, first we need to know which pieces of code are in the final executable. To do this select **Options | Project** and choose the Linker page. Then select the **Detailed** option in the **Map file** area. This





Start	Length	Name	Class
0001:0002	0067H	Project1	CODE
0001:0069	00BDH	Unit1	CODE
0001:0126	0ADAH	Printers	CODE
0001:0C00	0564H	TypInfo	CODE
0001:1164	0075H	WinProcs	CODE
0002:0002	0155H	Dialogs	CODE
0002:0157	0049H	Buttons	CODE
0002:01A0	22CAH	Menus	CODE
0003:0002	5E7FH	Graphics	CODE
0004:0002	6971H	Controls	CODE
0005:0002	76A1H	Forms	CODE
0006:0002	5075H	Classes	CODE
0007:0002	2242H	SysUtils	CODE
0007:2244	104EH	System	CODE
0008:0000	0E34H	DATA	DATA

Figure 1 (Top): What's going on? The executable file for a basic, "do-nothing" form can consume a whopping 191KB of disk space. **Figure 2 (Middle):** Here are the executable sizes for the form shown in Figure 1, created with Delphi, Delphi (optimized), and Borland Pascal 7. **Figure 3 (Bottom):** A portion of the map file created by Delphi by selecting the **Detailed** option in the **Map file** area. Among other things, this file is useful for discovering just what Delphi includes in the executable files it produces.

tells Delphi to create a text file that provides additional information about the content of an .EXE file. Named Project1.Map by default, a portion of this file is shown in Figure 3.

The information provided in this map file is similar to what Delphi includes in an .EXE file for use by an external debugger (if you have **Include TDW debug info** checked on the Linker page). This debug information identifies each source code file and maps the line numbers used with the corresponding address of that compiled code in the executable file. Using information like this, Turbo Debugger allows you to step through your program's lines of source code while it's running.

Indirectly, this same information tells which lines of code actually made it into the .EXE file after linking. Remember, the Delphi compiler performs "dead code elimination", so any code that's not referenced in a program won't be written to the executable, and hence will not be listed in the map file.

Equipped with this information, it becomes just a matter of reviewing this map file and finding the lines of source code in the .EXE file. This is not a pleasant job, I assure you. However, the resulting information is *very* useful, and helps us gain a much deeper appreciation for the efforts that went into creating Delphi.

Let's examine the results. Below is a list of the units included (at least most of them) in our sample program. Most of these units aren't represented in their entirety in the final executable, so I have listed only those features that are. Again, this is because Borland's smart linker removes code that isn't referenced. As a result, the information below provides a summary of all inherited behavior a Delphi developer receives with each and every application created — all in *only* 154KB!

The Forms Unit

As its name implies, the Forms unit is responsible for all the code necessary for creating and managing Forms. Since we do have a form, it should come as no surprise that this unit gets linked into our sample program. However, there's a lot more here than meets the eye.

CTL3D Support — The CTL3D and CTL3DV2 dynamic link libraries (DLLs) provide a 3-D look to dialog boxes and window controls. These DLLs essentially sub-class existing controls to provide them with an elegant 3-D look. You will really only see a difference in the form if you change its *BorderStyle* property to *bsDialog*. However, it also serves as the default behavior through the *ParentCTL3D* property of any controls placed on it.

Auto Scrolling Client Area — Though not immediately apparent, our simple main form example has the ability to automatically scroll its contents in case the window shrinks below a certain size. To test this, place a button near the lower-right corner of a form and run the program. When the form (and button) appears, resize the window so it partially obscures the button. Scroll bars will appear automatically to provide a way of scrolling the button back into view.

Default Windows Message Handling — This is an essential, low-level part of any Windows program. By providing default message handling, Delphi greatly minimizes development time. For C or C++ Windows programs, the programmer is generally responsible for writing and maintaining the code necessary to handle the flurry of message traffic for the application. For Delphi however, we only need to generate events and write handlers for those events. The maintenance of the Windows message queues is handled for us.

Form Recreation — A very useful capability, it allows us to change the basic class of a form (such as changing a normal resizeable window to a dialog box). Normally we would have to jump through a number of hoops to achieve this, but Delphi manages it by automatically recreating the form for us. To see this behavior, create an *OnClick* event for the form and insert the following code:

```
if BorderStyle = bsSizeable then
  BorderStyle := bsDialog
else
  BorderStyle := bsSizeable;
```

Now, by clicking on the form, it will toggle between a resizeable window style, and a fixed-size dialog box style.

Different Form Styles — Embedded in our sample application is support for all the types of forms Windows creates. These include: Standard resizeable forms, MDI Forms, MDI Children, and Dialogs. It also gives us the ability to force a window to stay on top of all other running applications.

Menus — Our form already has one (the System menu), but support for the application menu as well as any pop-up menus has also been provided. Try adding a pop-up menu to the form with four or five menu items on it. Now compile and check the executable size. Only about 300 bytes have been added. Not bad! We can also add or change the application's system menu.

Form Scaling — Using the *PixelsPerInch* and *Scaled* properties of a form, our sample also has the ability to dynamically size itself to handle different resolutions at run-time.

Color and Font Changes — Normally quite a chore, Delphi has provided an easy and elegant way to alter the entire appearance of the form. We can display text in any available font and can change the form's color based on a user's Windows color scheme.

User Events — Perhaps one of the more important capabilities, this support allows us to connect custom procedures to forms. We can attach code to a wide range of events such as Mouse Click/Movement, Drag/Drop, and Keyboard events.

Screen Object Management — The Screen object encapsulates all the basic information about the current display environment. We can easily access the screen resolution, pixels per inch, available fonts, and cursors. The Screen object can also provide a list of, and access to, all forms available in the current application.

Exception Handling — Our application also provides a rich exception management system. It smoothly handles error conditions relating to forms, and manages the use and restoration of memory from Windows memory pools. Much of the code also protects us against lost resources in case the application terminates abnormally.

Idle-State management — This allows the programmer to provide behavior for an idle application. For example, the programmer may want some special activity while waiting for user input. This is different from having timer events in that idle-state events only occur while the application is idle (i.e. no other events are occurring).

Hints — This provides support for the Help Hints (or "balloon help") that appear when the mouse pauses for a moment over a form or component. It also provides support for information in the Status bar. To see this, add a short sentence to the *Hint* property of the form and set *ShowHint* to *True*. After compiling, check the size of the executable. *No increase!*

The Controls Unit

This unit defines the basic *TControl* object with its descendants *TWinControl* and *TGraphicsControl*. This unit plays a key role in adding Delphi behavior to standard Windows controls (such as list boxes, buttons, etc.). The behaviors that appear in our sample application are listed below.

Drag-and-Drop — All the basic drag-and-drop behavior is here. This allows components or on-screen items to be moved to another location via drag-and-drop. A typical use for this behavior might be, for example, dragging items from one list box to another (as in the case of the Dual list box form template provided with Delphi).

Device Contexts Management — A lot of drudgery is eliminated for developers because they don't have to deal with Windows' *device contexts*. Many controls have new graphical behaviors (such as the speed buttons) that are automatically managed through Delphi's encapsulation of device contexts into canvases.

Parent and Children Lists — Our application automatically maintains internal lists so parent objects can keep track of their children. In the case of Delphi however, these lists enable the application to correctly destroy components that are contained within other parent controls. It also allows the parent font and hint overriding capabilities.

Control Scaling — Control scaling allows the control to change its scale in percentages or in relation to other objects. This behavior can be used, for example, when a form is resized, or when the application needs to adjust to displays with different resolutions.

Cursor Management — This allows the control to change the cursor when the mouse pointer moves over the control's boundaries. Examples of this might include the drag-and-drop cursors (visually allowing or restricting dropping of an object over a control) or changing the cursor to a shape more appropriate to the control (such as a hand for a spinner control).

Z-Order Management — In addition to X and Y screen coordinates, all Windows controls also have a *Z-order* (or stacking order) on the form. This determines which objects are stacked in front of other objects on the screen. The Delphi controls can change the stacking order so some controls can be moved in front or behind others.

Mouse Detection — This adds basic mouse detection for controls. Virtually any control can now respond to mouse clicks and mouse movement.

Popup Menus — Particularly with the new interface elements appearing in the latest versions of Windows and OS/2, users are demanding a higher level of ease-of-use. Pop-up menu support allows individual controls to provide context-sensitive menus at the click of a mouse button.

Tab-Order Management — This code allows us to modify the tab sequence of controls on a form. Typically the *TabOrder* and *TabStop* properties are only set at design time, but Delphi allows us to modify these at run-time as well.

Control Alignment — This allows controls to align in various ways relative to their parent controls or the form. By using this capability, we can have controls that dynamically adjust their position when a user resizes a form. You can see this behavior by placing a ListBox component on the form and setting its *Align* property to *alClient*. When the program runs, the list box will now dynamically adjust its size when the form containing it is resized.

User Events — Perhaps one of the more important capabilities, this support allows us to connect custom procedures to controls. We can attach code to a wide range of events such as Mouse Click/Movement events, Drag/Drop events, and Keyboard events.

Other Support — Our sample application also provides support for Short and Long Hints, Fonts, Colors, Palettes, and CTL3D.

The Printers Unit

You may not recall asking this program to print, but it can. To see this, add an *OnClick* event handler to a form and enter `Form1.Print` as the only line of code. When you click on the form, it will print itself. Granted you will only get a big gray rectangle on the page, but there is a significant amount of work behind that rectangle. The Printers unit adds the behaviors listed below.

Encapsulation of Printer Support — Printing in Windows is a lot of work. The creators of Delphi have encapsulated all the printing behavior into a neat little package called the Printer object. To begin printing use `Printer.BeginDoc` and to end printing use `Printer.EndDoc`. The Printer object has a canvas that we simply write on as we wish between these two commands. The Printer object saves the developer the unpleasant task of working with a printer device context.

Exception Handling — Exception handling for this unit allows the program to overcome potentially fatal problems, such as selecting an invalid printer, failures in the Windows print system, etc.

Management of Default Printers — Each computer has a list of available printers and Delphi grants us easy access to this list.

The Graphics Unit

The Graphics unit automatically manages a wide range of graphics chores for us. The foremost of these is the encapsulation of the Windows device contexts into the *Canvas* property. The Graphics unit also provides the support outlined below.
Loading Graphics from a Resource File — This allows the program to extract graphics data (icons, bitmaps, etc.) from an embedded resource file and incorporate this data into object properties (such as a Picture or Glyph).

Font Support — By encapsulating Windows font support into the Font object, the program can freely and easily display text anywhere on a form without requiring the programmer to issue low-level instructions to load or create the fonts needed.

Pens and Brushes — Need a green pen three pixels wide? How about a hatched brush with a clear border? Rather than create these, they are encapsulated in the Pen and Brush objects.

Color Constants — These make it easier to select colors for use in the application. Rather than request a specific RGB (Red/Green/Blue) color combination (i.e. the statement RGB(0,128,255) to refer to a bright, light blue), we can now refer to a color by means of color constants such as *clGreen* and *clWhite*. We also have access to colors that depend on the user's color scheme, such as *clButtonFace* and *clWindowText*. These allow the user to change the color preferences, and the program remains consistent with the user's desktop environment.

Canvases — Delphi captures all the behavior of Windows device contexts into the Canvas object. We simply draw, paint, write text, et cetera, on the canvas and it appears on the screen.

Graphics Exceptions and Memory Allocation — This code saves us from many graphics pitfalls. It handles graphics error conditions, and manages the use and restoration of memory from Windows memory pools.

Bitmap Manipulation — This encapsulates basic management of bitmap and icon images. It allows us to copy, move, and stretch images with very little effort. For example, if you place a *TImage* component that is 100 pixels square on a form, and then load a 50x50 pixel bitmap, it will display only in the top-left corner of the component's boundaries. However, simply setting the *TImage*'s *Stretch* property to *True* will extend the bitmap's size to cover the entire 100x100 area. Without this, we would have to write custom support for the Windows *StretchBlt* function.

Automatic Palette Support — Whenever we deal with graphics of different color "depths" (e.g. 16 vs. 256 colors), Windows needs to switch into the appropriate color palette so the colors on screen appear correctly. Delphi manages most of this for us automatically.

Metafiles — Windows Metafiles are a sort of "graphics macro language". They package all sorts of Windows GDI calls into a graphics file format that is easily exchangeable between Windows

programs. I recently had to add my own metafile support to a Borland Pascal 7 program and it wasn't a pleasant experience. The metafile format is an integral part of the Delphi graphics system; we can load, save, and display metafiles with no more effort than that required for simple bitmaps.

Clipboard Support — This allows Delphi to easily move graphics data (in many supported formats) to and from the Windows Clipboard.

Handle Caching — Delphi's graphics system can cache various graphic objects such as pens, brushes, fonts, etc. By caching these objects, the program can quickly flip between various objects without having to recreate them each time. This is entirely invisible to the programmer.

Management of Stock Objects — This adds some minor support for the various stock graphic objects that Windows has available (such as a black pen, or a white brush).

User Event Handling — These events are the "hooks" we can put in our code to extend an object's basic behavior. For example, events allow us to have a graphic respond to the mouse passing over it, or to have a bitmap know when it has changed.

The Dialogs and Buttons Units

The Dialogs unit adds some minor, low-level support for common dialog boxes. Additional segments of the Dialogs unit are not linked into the executable until we add one or more of the common dialogs from the Dialogs page of the Component Palette. The Buttons unit adds support for the *TSpeedBtn* and *TBitBtn* components. However, most of the unit has been linked. What remains is about four lines of code that clean memory usage of BitBtn glyphs.

The Classes Unit

The Classes unit contains the declarations for many of the base object classes used by Delphi. Much of the support provided is for low-level capabilities such as memory management, string management, exception handling, reading from and writing to streams, etc. Interestingly however, this unit also provides a Quicksort routine for sorting string lists, and an object reader/writer that has the functionality used in most object's *Read* and *Write* access methods.

The Menus Unit

Not surprisingly, this unit adds support for all menu types to our programs. The principal functionality is its ability to create, display, and allow modification of the menus and menu items. The two principal objects added to our program are the *TMainMenu* and *TPopupMenu* objects. The behaviors listed below are also included.

Pop-up Menus — Pop-up menu support is added giving the application the ability to show context-sensitive pop-up menus when the user right-clicks over a component or form. Because all base support is already there, adding a fully functional pop-up menu (with four or five menu items) to the main form only adds about 300 bytes to the program's size.

Menu Text and Shortcuts — This enables the user to easily define text and shortcut keys for menu items.

Checked and Disabled States — Some menu items allow a checked (or toggled) condition, while all menu items can be disabled if required. The code included allows the developer to access such menu items, read and modify their *Checked* state, and enable or disable them.

Moving and Changing Menu Items — Normally a fairly complex job, the developer can now change menu items and even merge menus from different forms.

Hints — Hints are automatically included as part of menu items. The Hints respond to an application's *OnHint* events and can, for example, provide status-line hints on an item currently selected in a menu.

Help Contexts — If a Help system is tied to the application, the menu items can be assigned context numbers allowing them to provide help information to the user at run-time. For example, a user might activate a pop-up menu by right-clicking on a form or component, and move the selection bar down to one of the menu items. Before clicking on that item, pressing F1 calls the assigned topic from the help system.

User Events — This code enables support for user-related events. For example, if we want something special to occur when a pop-up menu is activated (such as a sound effect), it could be done using the *OnPopup* event for the PopupMenu object.

Conclusion

As you can see, there is quite a bit of inherited behavior in Delphi's executables. Equipped with this knowledge, 154KB of inherited code doesn't seem like much does it? Somewhere down the line I'd like to see extensions to the compiler options that would allow programmers to tailor this inherited behavior to suit their needs — perhaps a series of check boxes disabling support for certain features, such as MetaFiles or PopupMenus, for example.

The developers of Delphi have gone to great lengths to provide a rich and powerful programming environment for all of us. By encapsulating complex Windows programming concepts into easy-to-use properties and methods, the Delphi environment saves programmers from having to deal with the drudgery of programming. Instead, it allows us to concentrate on the creative and innovative aspects of software development. ∆

Robert Vivrette is a contract programmer for a major utility company. 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.

OP Tech

Delphi / Object Pascal / Windows API

By Bill Todd

A Little Memo Help

Manipulating the Cursor in Delphi's Memo Components

Delphi includes two classes of memo control components, *TMemo* and *TDBMemo*, that enable the user to edit text. Since both of these controls encapsulate the Windows multi-line edit control they are limited to a maximum of 32KB of text.

To make editing the text in a memo control component easier, the *TMemo* and *TDBMemo* controls treat the text as an array of Pascal strings that you can access via the memo's *Lines* property. Each element in the string array corresponds to one line in the memo. This enables you to manipulate the text in a memo using all the Object Pascal run-time library procedures and functions for Pascal strings.

Access to the contents of the Windows multi-line edit control as an array of strings is provided by the *TStrings* class. *TStrings* includes the methods listed in Figure 1 that let you manipulate the text in a memo.

The *TStrings* class also includes a *Count* property that tells you the number of lines the memo contains. *Count* is zero-based, so the statement:

```
Panel1.Caption := Memo1.Lines[0];
```

displays the first line of the memo in `Panel1`, and:

```
Panel1.Caption := Memo1.Lines[Memo1.Lines.Count - 1];
```

displays the last line of the memo in the panel.

But Where's the Cursor?

While the *Lines* property provides an easy, intuitive interface between the Pascal String type and the text in a memo, there is one problem: There is no way to manipulate the cursor. *TMemo* does have the *SelStart* property that allows you to get or set the current cursor position. However, the position is given as the number of bytes from the beginning of the memo text including the carriage return and line feed characters at the end of each line. This is completely incompatible with the line and column position you need when working with the memo's *Lines* property.

Fortunately, *TMemo* and *TDBMemo* encapsulate the Windows multi-line edit component and there are two Windows API calls that save the day.

Method	Function
Add	Add a line to the memo.
Clear	Delete all lines from the memo.
Delete	Delete a line from the memo.
Exchange	Exchange the position of two lines.
IndexOf	Find line number of specified text.
Insert	Insert a line into the memo at any location.
Move	Move a line from one place to another.

Figure 1: Selected *TStrings* methods that enable you to handle text in a memo.

The first of these is the Windows `EM_LINEFROMCHAR` message. Sending this message to a Windows multi-line edit control along with a character offset returns the number of the line containing that character offset. Therefore, the statement:

```
MemoLine := SendMessage(Memo1.Handle, EM_LINEFROMCHAR,
                        Memo1.SelStart, 0);
```

returns the number of the line that contains the cursor.

The first parameter in the call to *SendMessage* is the memo component's window handle. This identifies the component the message will be sent to. The second is the constant that identifies the message to send, `EM_LINEFROMCHAR` in this case. The third parameter is the offset from the beginning of the memo of the character whose line number you want to find. In this case you want to find the line that contains the cursor. The memo's *SelStart* property is what you need, since it gives the cursor position as an offset from the beginning of the memo.

The second Windows message you need is `EM_LINEINDEX`. Sending the `EM_LINEINDEX` message to a memo returns the position of the first character of the line you specify as an offset from the beginning of the memo. For example:

```
LineStart := SendMessage(Handle, EM_LINEINDEX, 3, 0);
```

returns the offset of the first character of the fourth line of the memo. The first two parameters in the *SendMessage* call are the same as before. The third parameter is the number of the line whose offset you want to find.

Some Custom Procedures

Using these two messages you can write a generic procedure that returns the line number and column number of the cursor position. You can also write a second procedure that will move the cursor to any specified line and column number. As you have already seen, getting the number of the line that contains the cursor is a matter of sending the `EM_LINEFROMCHAR` message to the memo. To determine which column the cursor is in, subtract the offset of the first character of the line from the cursor position given by *SelStart* property. Figure 2 contains the complete code for the custom *GetMemoLineCol* procedure.

```
{ Get the line number and column number
  the cursor is positioned at in the memo. }
procedure GetMemoLineCol(Memo: TCustMemo;
                         var MemoLine, MemoCol: Integer);
begin
  with Memo do
  begin
    { Get the line number of the line
      that contains the cursor. }
    MemoLine := SendMessage(Handle, EM_LINEFROMCHAR,
                            SelStart, 0);
    { Get the offset of the cursor in the line. }
    MemoCol := SelStart - SendMessage(Handle, EM_LINEINDEX,
                                      MemoLine, 0) + 1;
  end;
end;
```

```
{ Set the cursor position in a memo
  to the specified line and column. }
procedure MemoCursorTo(Memo: TCustomMemo;
                       MemoLine, MemoCol : Integer);
begin
  Memo.SelStart := SendMessage(Memo.Handle, EM_LINEINDEX,
                               MemoLine, 0) + MemoCol -1;
end;

end.
```

Figure 2 (Top): The custom *GetMemoLineCol* procedure obtains the cursor's position in the memo component as a line and column number. **Figure 3 (Bottom):** Setting the cursor position of a Memo or DBMemo component to a specified line and column number.

This procedure takes three parameters. The first is the Memo or DBMemo component whose cursor position you wish to find. Note that the type of this parameter is *TCustomMemo*. *TCustomMemo* is used because it is an ancestor of both *TMemo* and *TDBMemo*. This allows the actual parameter to be of either type. The second two parameters are the line and column number that are returned. In this procedure the line is zero-based and the column is not.

Figure 3 shows the *MemoCursorTo* procedure that will set the cursor position of a Memo or DBMemo component to the line and column number you specify. (Again, the line is zero-based and the column is not.) This procedure uses the `EM_LINEINDEX` message to set *SelStart* to the offset of the first character of the specified line plus the column number minus 1.

To make it easy to use these procedures in any application, put them in a separate unit along with any other memo-related procedures you write. Then simply add the unit to the **uses** clause in any program that needs these procedures. Figure 4 shows the complete unit, MEMOS.PAS.

Figure 5 shows the form for a simple test program to demonstrate these procedures. The first button moves the cursor to line 3 column 5 in the text. The second displays the current cursor position in the panel at the bottom of the form.

```
unit Memos;

interface

uses WinProcs, SysUtils, StdCtrls, Dialogs, Messages;

{ Get the line number and column number the cursor
  is positioned at in the memo. }
procedure GetMemoLineCol(Memo: TCustomMemo;
                        var MemoLine, MemoCol : Integer);

{ Set the cursor position in a memo to the specified
  line and column. }
procedure MemoCursorTo(Memo: TCustomMemo;
                      MemoLine, MemoCol: Integer);
implementation

{ Get the line number and column number the cursor
 is positioned at in the memo. }
procedure GetMemoLineCol(Memo: TCustomMemo;
                        var MemoLine, MemoCol : Integer);
begin
  with Memo do
  begin
    { Get the line number of the line that
      contains the cursor. }
    MemoLine := SendMessage(Handle, EM_LINEFROMCHAR,
                           SelStart, 0);
    {Get the offset of the cursor in the line.}
    MemoCol := SelStart - SendMessage(Handle,
                                     EM_LINEINDEX,
                                     MemoLine, 0) + 1;
  end;
end;

{ Set the cursor position in a memo to the specified
   line and column. }
procedure MemoCursorTo(Memo: TCustomMemo;
                      MemoLine, MemoCol : Integer);
begin
  Memo.SelStart := SendMessage(Memo.Handle, EM_LINEINDEX,
                              MemoLine, 0) + MemoCol - 1;
end;

end.
```

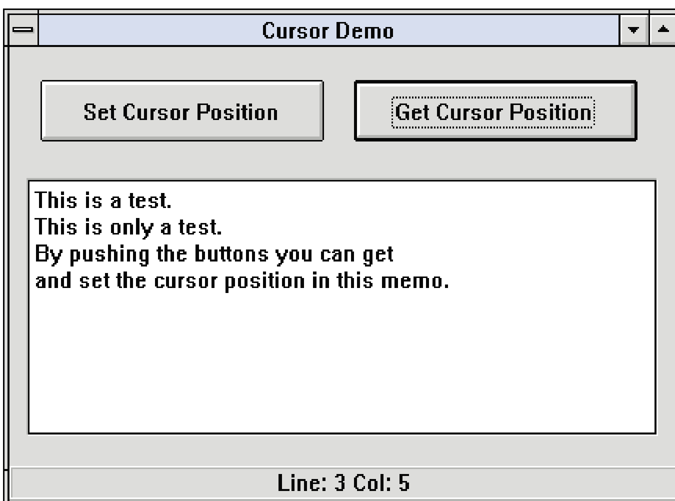


Figure 4 (Top): The complete code for the Memos unit. **Figure 5 (Bottom):** The Cursor Demo application in action.

```
procedure TForm1.SetCursorBtnClick(Sender: TObject);
begin
  { Move the cursor to line 3, column 5. }
  ActiveControl := Memo1;
  MemoCursorTo(Memo1, 2, 5);
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  MemoLine, MemoCol:  Integer;
begin
  { Display the memo's cursor position. }
  GetMemoLineCol(Memo1, MemoLine, MemoCol);
  Panel1.Caption := 'Line: ' + IntToStr(MemoLine) +
                   ' Col: ' + IntToStr(MemoCol);
end;
```

Figure 6: The *OnClick* event-handling procedures for the **Set Cursor Position** and **Get Cursor Position** buttons.

Figure 6 shows the code for the two buttons. Note the statement:

```
ActiveContol := Memo1;
```

in the *SetCursorBtnClick* procedure. When you change the cursor position in a Memo component, the memo must have focus or the cursor will seem to disappear. This statement moves focus from the button back to the memo before calling the *MemoCursorTo* procedure.

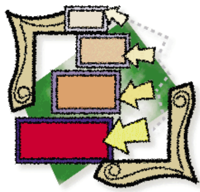
Conclusion

Delphi's memo components let you access the text in the memo as an array of strings. However, they provide no way to locate or move the cursor using the line and column position notation.

The Windows API messages EM_LINEINDEX and EM_LINETOCHAR let you convert the cursor position between line and column, and character offset so you can easily determine or change the cursor position. ∆

The demonstration unit and form referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\JUL\BT9507.

Bill Todd is President of The Database Group, Inc., a consulting firm based near Phoenix. He is also a member of Team Borland (supporting Paradox on CompuServe) and a speaker at all Borland database conferences. Bill can be reached at (602) 802-0178, or on CompuServe at 71333,2146.



VISUAL PROGRAMMING

DELPHI

By Sedge Simons, Ph.D.

Meet the .DFM File

Modifying Delphi's "File Behind the Form"

The graphical properties of a Delphi form and its components are stored in a .DFM file. In this article, we'll see how to open and edit this file, making short work of global changes or component creation, and giving you an additional approach for troubleshooting.

When you create a form, Delphi maintains both a unit file (with a .PAS extension), and a graphical form file (with a .DFM extension). A simple form and the corresponding unit are shown in Figures 1 and 2. Looking at the unit file in Figure 2, you'll notice that it does not include any specifications of the properties of the form or its components, only declarations of the objects themselves. Delphi stores the properties in a separate file, the graphical form file, which we'll refer to as the .DFM file.

The .DFM File

Believe it or not, Figure 1 is a .DFM file. You see, the Delphi Form Designer is really a graphical editor for the .DFM file. As you place, resize, and rearrange components with the Form Designer, Delphi updates the .DFM file accordingly, much like it updates declarations in the Object Pascal unit file.

The creators of Delphi included a second way to edit the .DFM file. From the Open File dialog box, select a file type of **Form file (*.DFM)** and select the .DFM file with the same name as the unit you're using. If you're currently editing the form in the Form Designer, Delphi will offer to save your changes and close the Form Designer. Naturally, Delphi does not allow you to edit the same .DFM file with two editors at once.

Figure 3 shows the .DFM file in the Code Editor. Remember that this is the same file shown in Figure 1, but it is now revealed through the Code Editor rather than the Form Designer.

We can make several observations looking at the listing in Figure 3:

- The .DFM file doesn't contain Object Pascal code.

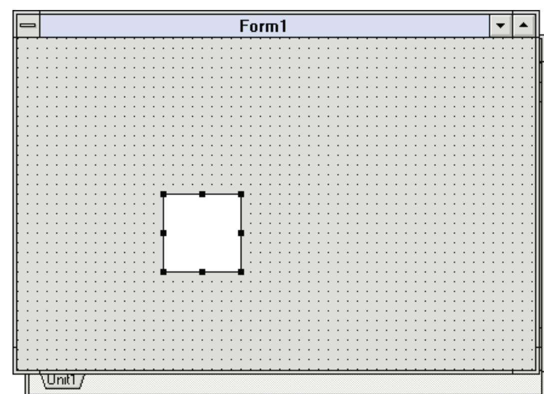


Figure 1: This sample form contains only one component, a *TShape* named *Shape1*.

```
DEMO1.PAS
unit Demo1;

interface

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

type
  TForm1 = class(TForm)
    Shape1: TShape;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

end.
20: 36    Insert
Demo1
```

Figure 2: This is the unit file, Demo1.PAS, that goes with the form in Figure 1. Notice that it doesn't specify any properties of the form or the component.

```
DEMO1.DFM
object Form1: TForm1
  Left = 200
  Top = 99
  Width = 435
  Height = 300
  Caption = 'Form1'
  Font.Color = clWindowText
  Font.Height = -13
  Font.Name = 'System'
  Font.Style = []
  PixelsPerInch = 96
  TextHeight = 16
  object Shape1: TShape
    Left = 120
    Top = 128
    Width = 65
    Height = 65
  end
end
12: 35    Insert
demo1.dfm
```

Figure 3: This is the .DFM file, Demo1.DFM, that defines the form in Figure 1. Graphical form files are automatically given the same name as the unit file, but with a .DFM extension.

- Many of the properties are familiar from the Object Inspector.
- A lot of familiar properties are not included in the .DFM file.

Let's take these observations one at a time.

It's not Object Pascal code. No, it's not Object Pascal, so don't use the Object Pascal assignment operator (:=) or statement terminator (;) by mistake. The .DFM file is stored more like a data file, and the Code Editor has converted it to a text format that you can edit. If you try to look at the file in Notepad or another editor, you will not see this converted view. You can, of course, use the Clipboard to copy and paste text between the Code Editor and Notepad, or other applications. You can also use **File | Save As** with a .TXT extension to save the file in ASCII format.

Many of the properties are familiar from the Object Inspector. Yes, these are the same properties. Some properties, like *Font*, are expanded using the familiar dot notation. You'll see *Font.Size*, *Font.Style*, and so on. You can change these properties here just as you can in the Object Inspector.

A lot of familiar properties are not included in the .DFM file. In fact, many are missing. How does Delphi know that *Shape1* is a rectangle, for example? The answer is that Delphi only stores those properties that differ from the default properties of the component. This is really an effect of inheritance. *Shape1* inherits the default properties of a *TShape* component, unless you set different properties. You can add any legitimate property settings to the .DFM file if you want to override the default properties.

This inheritance has a serious implication. If you change the default properties of components on your palette, then return to an old project, the default properties of components in that project will assume their new default values. This is how inheritance is supposed to operate, but if you're not expecting it, you might be shocked at how your project is changed. Imagine all your rectangles turning into ellipses because you defined *stEllipse* as the default shape for *TShape* components!

You'll probably never change the defaults of components that ship with Delphi, but the moral is to be thoughtful in choosing default properties in your own components. And be aware of how subsequent changes you make to your palette might affect old projects. If you feel you must change the default properties of a component, but are concerned about backward compatibility, consider creating a new component with new defaults instead.

Editing the .DFM File

Having access to the .DFM file in the Code Editor provides some powerful capabilities. Among the most useful is search-and-replace. Let's say you want to change the font used throughout a form from System to Arial. This process could be tedious in the form designer, and you might miss a few objects, particularly if they are on different notebook pages. (You couldn't use **Edit | Select All** in the Object Inspector since some components don't have a Font property.) Instead, just use the Code Editor to replace '`System`' with '`Arial`'. You could also replace colors (e.g. *clWindowText* with *clBlack*) so the colors on your form will not depend on the user's configuration.

You might also find that you can solve some alignment, centering, or sizing problems more easily by setting properties like *Left*, *Top*, *Width*, and *Height* through the Code Editor rather than by using the Object Inspector or Alignment Palette.

Occasionally, a debugging problem may send you to the .DFM file. If you've restructured a data table and changed some field names, references to the old names might linger in the .DFM file. If you're sure you've fixed all the references in the unit file and the problem persists, search the .DFM file for the offending field name.

Finally, consider the problem of creating many similar components. Copy-and-paste might be faster in the Code Editor than in the Form Designer, or you might prefer to use the Clipboard to paste text from another application. You could, for example, write a Word for Windows macro that generates the appropriate text for 100 rectangles in a 10-by-10 grid, then paste that text into the .DFM file. You might even create a Delphi project than writes a text file you can paste into a .DFM file!

If you do create or delete components in the .DFM file, don't forget to update the unit file. When you use the Code Editor to change the .DFM file, you don't have Delphi helping you keep everything synchronized.

Also remember that the .DFM file is strictly a design object. You will see your changes when you go back to the Form Designer, but the .EXE file is not changed until you recompile the project.

Conclusion

Delphi does so much of the work for us that most of the time we can forget that the .DFM file exists. But easy access to this file is one of the many features that makes Delphi so flexible. Hopefully these examples will have you thinking about even more ways of beefing up your RAD repertoire through the .DFM file. ∆

Dr. Simons is a senior systems analyst at Jensen Data Systems, Inc., a Texas-based provider of Database training, consulting, and application development. He writes applications and does consulting in Delphi and Paradox. You can reach him through CompuServe at 70771,75 or by calling Jensen Data Systems, Inc. at (713) 359-3311.

BY DOUG HORN

The Developers Visual Suite Deal

A Truly Sweet Deal from Visual Tools

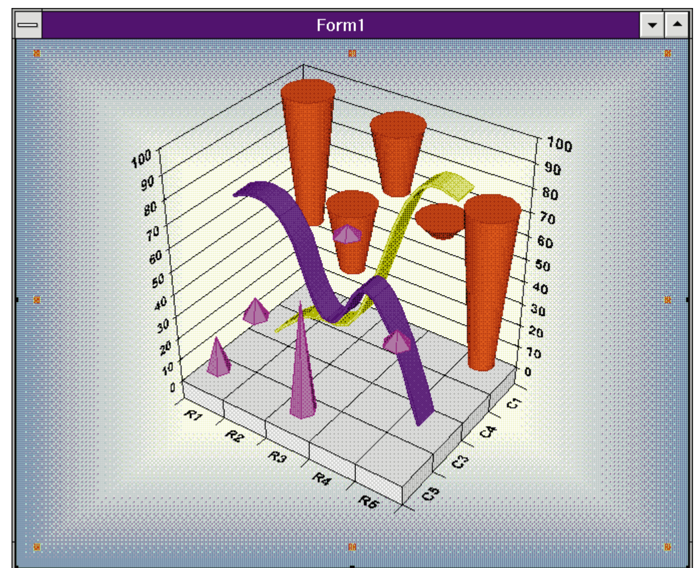
The Developers Visual Suite Deal from Visual Tools does not mince words. It promises to provide you with the VBX components necessary to build an Excel-compatible spreadsheet, word processor, spell checker, image viewer, and unparalleled charting package. On most of these promises, the Suite Deal delivers.

The Developers Visual Suite Deal is five component packages bundled together. These components can be used as dynamic linked libraries (DLLs), C class libraries, or Visual Basic controls (VBXes). This suite was clearly assembled with the serious developer in mind. It contains the tools necessary to build an integrated software application that could rival many on the market. Used individually, its tools are doubly valuable — they allow Delphi developers to use these popular functions in their applications with a minimum of programming.

First Impression

First Impression is arguably the Suite Deal's flagship product. It can create almost two dozen charts, including two and three-dimensional versions of pie, bar, and Gantt charts. What is more, First Impression's high degree of customization creates the appearance of hundreds of chart combinations.

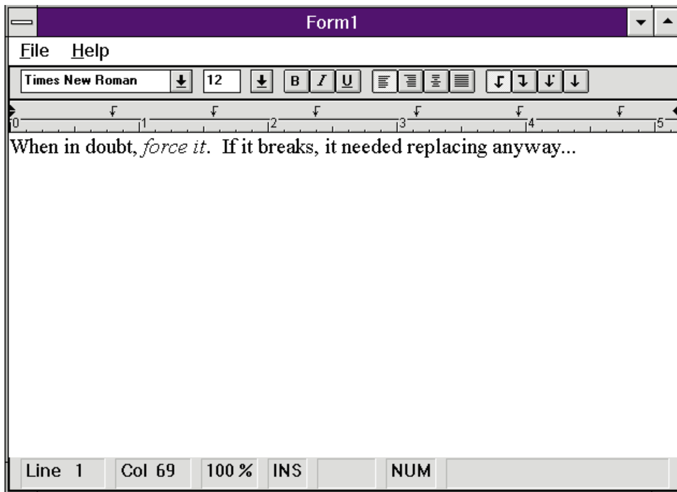
As a charting tool, First Impression compares favorably with the Chart F/X VBX included with Delphi (although a more advanced version of Chart F/X is available that more evenly approximates First Impression's capabilities). First Impression offers a more sophisticated array of charts and graphs including doughnut and combination charts. It also features a number of formatting options such as gradient fills, and screen or printer-optimized output.



The First Impression 3D combination chart shows impressive charting capabilities.

First Impression's fine-tuning does come at a price, however. It's not as highly integrated with Delphi as Chart F/X is. While First Impression's ability to allow end-users to customize charts at run-time is very nice, some features are only selectable at run-time, and not through Delphi code.

This drawback aside, First Impression adds a number of useful 'extras' that help create professional-looking charts. Backgrounds can include graphics, gradient or pattern fills, and a large assortment of other design elements. Three-dimensional charts allow the user to configure such parameters as view angles, lighting sources, line smoothing, and column shape. Furthermore, First Impression's three-dimensional shadowing and dithering are first rate, enhancing the appearance of these charts.



With VisualWriter, this simple word processor application was created in Delphi in about 10 minutes.

First Impression charts may be modified in one of two ways: directly, or through the use of a Formula One spreadsheet. While setting chart datapoints through Delphi code is possible, this direct approach requires much code and extensive use of the period key (.), and is certainly not the preferred method. The ability to link First Impression charts to Formula One spreadsheet files is better. These files can then be changed and updated to instantly reflect changes in the underlying data. This solution suffices, although it's not as useful as a direct DataSource link would be.

Formula One

Formula One is Visual Tools' Excel-compatible spreadsheet. As noted earlier, it forms the link between program data and First Impression charts. However, Formula One is capable of much more.

Formula One includes two VBX controls: a spreadsheet and an edit box. These two objects link easily via the spreadsheet control's EditBox property. Such thoughtful integration is found throughout the suite. Formula One spreadsheets can be modified by the end-user as any Excel-type worksheet file. However, the program also includes a more useful interface. Selecting the spreadsheet control's ApplicationDesigner property calls the Formula One application screen. From this screen, designers can create any number of pre-configured worksheets. Changes made to the worksheet here are reflected in the Delphi control. Once the sheet is complete, the developer can turn off whichever editing rights she chooses to protect the finished sheet.

While Formula One is called Excel-compatible, it contains a number of features — such as in-cell editing — that many users may find preferable to Excel's. On the other hand, although Formula One provides a great many mathematical and financial functions, it does not include Excel's more advanced add-in libraries or Solver capabilities. As a spreadsheet component, however, it has all the features most developers would probably use.

VisualWriter

Like Formula One, VisualWriter is a multi-control VBX. VisualWriter's four controls are text, button bar, status bar, and ruler. The text control — where end-users will type their documents — is the main component. It contains a property linking it to each of the others, much the same as Formula One spreadsheet controls link to edit boxes. With the properties linked, the VisualWriter controls almost immediately look and behave like a word processor. All that's left for the developer is to add a menu and a series of program functions, such as Open or File | Save.

VisualWriter allows developers to create a word processor as powerful as they choose it to be, since all functions are created by the developer. With the power of Delphi behind them, programmers should be able to conjure up everything from feature-rich stand-alone word processors and HTML generators, to simple letter writers integrated into larger applications.

VisualWriter can save files in its native format, or in rich text format (RTF) for exchange with other Windows word processor applications. It also offers two print methods — PrintForm that prints at screen resolution, or Print, the standard printer-resolution printing command.

The VisualWriter documentation is sparse compared to the thick manuals for First Impression and Formula One. It suggests a plan for creating a full-fledged word processor with VisualWriter components, but falls short of truly describing the process. Fortunately, most experienced developers will be able to figure out these details.

VisualSpeller

VisualSpeller is a 100,000-word spell checker VBX that functions seamlessly with VisualWriter components. VisualSpeller also works independently in any Delphi application.

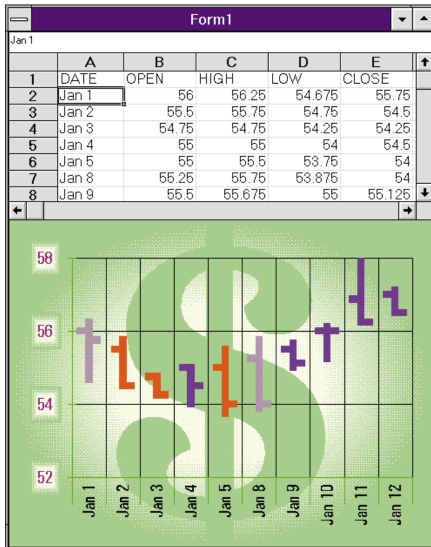
Visual Speller is excellent. It goes far beyond the capabilities of the spell checkers included with most Windows applications — not only does it allow multiple dictionaries and custom dictio-

Form1

=SUM(A2:D2)/COUNT(A2:D2)

	East	North	South	West	Average
Q1	4563	4521	4561	4556	4550.25
Q2	1235	388	855	907	846.25
Q3	2123	5696	4336	3667	3955.5
Q4	1699	2462	2008	1998	2041.75
Avg.	2405	3266.75	2940	2782	

Formula One spreadsheet and edit box components in a Delphi application.



A Delphi application linking a Formula One spreadsheet to a First Impression two-dimensional, open-hi-low-close chart, with an image and gradient-fill background.

naries, but also includes options available in few other applications. For example, one of the more useful options is "Allow Joined Words". As Madison Avenue and Silicon Valley continue to force words such as OpenDoc and VisualSpeller on us, it's nice to finally have a spell checker that recognizes them.

Another nice VisualSpeller feature is the ability to tell the component how to search for suggestions. There are eight techniques available, ranging from simple capitalization to exchanges, insertions, and deletions. In this vein, VisualSpeller also allows users to provide suggestions even for properly spelled words. This method, while tedious, is the best way to halt errors, such as *form* for *from* (which normal spell checkers never catch).

The Juicy Details

The fifth application in the Visual Tools suite is ImageStream, a graphics viewing program that handles dozens of graphic file formats, both vector and bitmapped images. The only catch is that the ImageStream VBX doesn't work with Delphi! Visual Tools acknowledges this known bug, but as of this writing, they have no plans for an update. This is unfortunate, as ImageStream may have been one of the more useful VBXes in the suite.

Visual Tools did, however, readily accept the fault for the bug rather than trying to lay the blame on Borland. In fact, in three test calls to Visual Tools' technical support line, I found the service to be excellent. Technical support is free, although it's a toll call. Most developers probably don't mind paying for the call as long as they don't spend an eternity in voice mail purgatory. (Visual Tools also provides Fax, BBS, and CompuServe technical support.)

The Visual Tools technical support crew answered all my calls within a minute, and provided correct answers to each question. But what impressed me the most was this: instead of seeming to read from a manual, the Visual Tools support representatives seemed to have a deep understanding for the issues we discussed — something that is becoming a rarity in the computer industry.

In general, the suite's documentation is also excellent. First Impression and Formula One are each extremely well-documented. Each has a user's guide and a separate function manual that describes programming with the components as VBXes or C class libraries and DLLs. The VisualSpeller documentation is also appropriate, although the VisualWriter manual lacks the finesse of the others.

Visual Tools allows Suite Deal buyers royalty-free licenses to distribute applications that include its components. Only ImageStream has a more restrictive license — limited to 100 run-time copies of a single commercial application per license.

INFORMANT FACT FILE

Developers Visual Suite Deal is a suite of five VBX components for Delphi developers, featuring high quality three-dimensional charting, Excel-compatible spreadsheet, word processor, 100,000+ word spell checker, graphics file viewer, and conversion components. Although the graphics VBX is incompatible with Delphi, the package is excellent and well worth the money.

Visual Tools
15721 College Blvd.
Lenexa, KS 66219

Price: Suite US$399;
First Impression only US$249;
Formula One only US$249;
VisualSpeller only US$149;
VisualWriter Pro version only US$249.

Phone: (913) 559-6500
Fax: (913) 599-6597

Each tool is easy to install. The setup program copies the necessary files to the hard drive and then offers a choice between copying the .VBX files to the \Windows\System directory or adding the VBX directory to the PATH statement. Once installed, the components can easily be added to the Delphi Component Palette.

Besides the component and help files, each control included at least one sample application. The more complex components like Formula One included several. Unfortunately, the sample applications are all in Visual Basic, Access, and C/C++. This is understandable at present, but hopefully Delphi will soon be added to the list of "required languages" for all componentware.

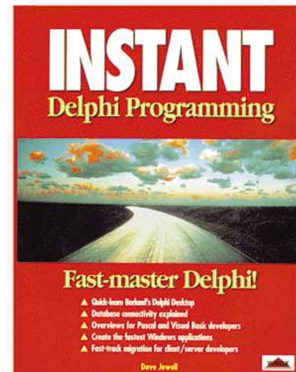
Conclusion

Developers Visual Suite Deal from Visual Tools is exceptional. It takes custom applications to a new level. Even the most mundane of the included tools is worth the price of the package. Getting Formula One and First Impression at this price almost makes the buyer feel guilty.

ImageStream will be sorely missed by many developers, but even without it, the Developers Suite Deal is well worth the money. It brings much needed functionality to Delphi applications with minimal programming. Buy this package and use it. The competition certainly will. ∆

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

Instant Will Score with Windows Programmers



Despite the fact that it was the first Delphi-specific book on the market, *Instant Delphi Programming* by Dave Jewell (Wrox Press Inc.) does not suffer from beta screen shots and the editorial gaffes typical of a quick-to-market effort. Instead, *Instant* is an excellent introduction to Delphi programming for an experienced developer. (Conversely, *Instant* would be a poor choice for a beginning programmer; it assumes a good deal of programming knowledge.)

Jewell is obviously very comfortable with Delphi and Windows programming, and liberally peppers *Instant* with Windows programming savvy. It's full of unexpected treats, such as a section of tips for keeping your Object Pascal code ready for a port to the 32-bit Win'95. This will be especially appealing to developers coming from a Windows programming background who are curious about what Delphi is doing behind the scenes as it interacts with Microsoft Windows.

The coding examples are easy to follow and Jewell breaks them up with small excursions into related tidbits of information. He carries this off very well and succeeds at spicing up the material rather than going off on tangents. For example, immediately after a demonstration of creating a graduated blue backdrop à la Windows Help (itself, a nifty trick), Jewell makes the important point that many Object Pascal Canvas methods (e.g. *FillRect* and *LineTo*) have the same names as their Windows API counterparts. The Object Pascal equivalents are much easier to use, but this can be confusing.

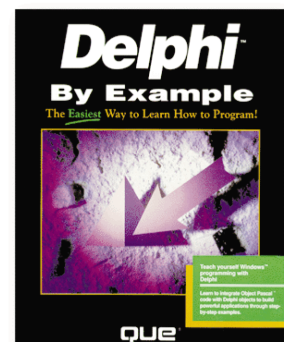
Instant covers the ground you would expect from an introductory book: the Delphi IDE, the various pieces of a Delphi project (e.g. .PAS and .PRJ files, etc.), managing projects, customizing the Delphi development environment, programming menus, building MDI applications, touring the Component Palette, etc. *Instant* features a particularly good introduction to event handlers with — again — an eye towards the Windows details. There's also a solid discussion of the Delphi debugging environment, and a particularly strong introduction to Delphi graphics programming (i.e. working with the Canvas object). Another stand-out section describes how to create custom components, using descendants of the *TOpenDialog* and *TGraphicControl* classes.

Instant finishes with a short-but-clear introduction of using Delphi as a database front-end, and two appendices with tips for programmers coming from Pascal or Visual Basic. Each chapter ends with some additional exercises, so *Instant* would lend itself to a classroom setting.

For the lazy or slow-of-hand, an accompanying diskette contains just over 1MB of supporting Delphi files. (Note: The initial printing of *Instant* contained no diskette and had

"Instant" continued on page 41

Que Offers Beginners Imperfect *Example*



With many step-by-step examples of real-world programming tasks, Blake Watson's *Delphi By Example* has plenty to offer the beginning Object Pascal programmer. Unfortunately, this Que Corporation book is so badly flawed with errors and incomplete examples, that it's difficult to recommend. Especially to its target audience — it is beginners that will be most confused by *Example*'s lapses.

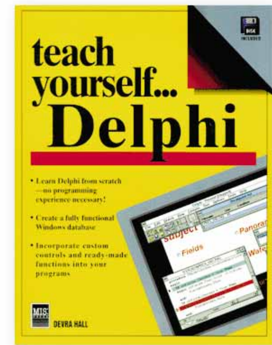
As it claims, *Example* is for beginners; accomplished programmers will find the pace maddening. However, new programmers will probably benefit from Watson's incremental approach. He warms up to topics slowly, taking the reader by the hand through each statement, and even providing a historical perspective on programming concepts such as loops, objects, and sub-routines. Again, however, the explanations are too often marred by missing or inaccurate instructions.

Also frustrating is that *Example* contains sections that are plainly taken from an earlier Pascal book. Statements such as "Borland's Delphi has evolved greatly until it has become as powerful as any tool available" reveal the cut-and-paste approach.

Having said all this, *Example* does have something to offer if you're willing to put up with its failings. The emphasis is on Object Pascal (although Watson insists on calling it Pascal throughout), an emphasis provided by no other beginning programming book now on the market. The other introductory books available (as of this early June writing) focus on visual programming. While such books are important, there's a real lack of language-based books such as *Example*.

"Example" continued on page 41

teach yourself ... Will Confound Target Audience



Devra Hall's *teach yourself ... Delphi* (MIS Press) was the second Delphi book available in bookstores, and it shows. It was obviously based on a pre-release version of Delphi and contains beta screen shots throughout. This is particularly inexcusable, since the first Delphi book on the market — the far more ambitious *Instant Delphi Programming* (also reviewed in this issue) — doesn't have this problem.

First, *teach yourself* is a slight book. With large print and oversize figures, it's really shorter than its 309 pages. (In fact, the font size selected for the code examples is so large that many Object Pascal statements are chopped up to such a degree that they're hard to read.) There's nothing wrong with this per se, but there are other books that do a better job covering the same material. Essentially an introduction to visual programming with Delphi, *teach yourself* attempts to cover the same ground as IDG's *Delphi Programming for Dummies* (reviewed in last month's *Delphi Informant*), although it doesn't have the broad scope of *Dummies* — and none of its panache.

The initial four chapters of *teach yourself* introduce the Delphi programming environment — from the Code Editor and Object Inspector, to customizing event handlers. These introductory chapters suffer the most from inaccuracies, beta problems, and typos. This is followed by a long-but-shallow chapter on variables and data types. The next chapter goes on to introduce the Canvas object. It's an odd specialty topic in this context, considering the book neglects so many basic areas of Delphi. For example, there is no methodical exploration of the Component Palette.

On the other hand, *teach yourself* does a particularly good job of describing the construction of a database form that includes lookup help. An entire chapter is devoted to the Database Desktop and demonstrates creating and populating a Paradox table. However, the book incorrectly limits Delphi's database support to Paradox and dBASE tables, omitting any of the supported SQL servers, and — most surprisingly — InterBase.

The book ends with a perfunctory chapter on ReportSmith that features a two-page description of the Query component. The final chapter on "Debugging and Error Handling" is inappropriate for the reader level and cursory in any case. For the record, an accompanying diskette contains just over 2MB of example files and includes two Visual Basic controls from Sheridan Software.

teach yourself might be appropriate if you're looking for a book that takes a very slow approach. If you find yourself in over your head with *Dummies*, *teach yourself* might be the book you're looking for to help you get to square one with Delphi. Unfortunately, the numerous errors and beta version material will present an additional hurdle between you and Delphi.

The unpleasant truth is that few will benefit from this book. Leave *teach yourself ... Delphi* in the bookstore. Even in the fledgling Delphi-book marketplace, you can do much better.

— *Jerry Coffey*

teach yourself ... Delphi by Devra Hall, MIS Press, 115 West 18th Street, New York, NY 10011; (800) 488-5233.

ISBN: 1-55828-390-0
Price: US$27.95
309 pages, diskette

"Instant" [cont. from page 40]

a confusing text omission early in the book, page 38, to be exact. Wrox Press corrected the error in subsequent printings and included the diskette. You can contact Wrox to get the diskette if you have this early version of the book.)

Dave Jewell's *Instant Delphi Programming* is a pithy, tip-studded excursion into Delphi development that shows a remarkable level of understanding — especially remarkable for *the* first book about a new product.

— *Jerry Coffey*

Instant Delphi Programming by Dave Jewell, Wrox Press Inc., 2710 West Touhy Avenue, Chicago, IL 60645-3008; Phone: (312) 465-3559; Fax: (312) 465-4063.

ISBN: 1-874416-57-5
Price: US$24.95
446 pages, diskette

"Example" [cont. from page 40]

After a quick tour of the Delphi interface, *Example* begins its exploration of Object Pascal by introducing variables, sub-routines, and data conversion. This is followed by a discussion of Object Pascal's branching and looping structures, and advanced data types.

The next section tackles ASCII text and typed files. Conspicuously absent is any discussion of working with database files. The final section of *Examples* discusses objects and OOP, demonstrates the **as** and **is** operators, and offers a quick guide to approaching a development project.

Closing each chapter with review questions and exercises, *Example* is well suited to a classroom situation. In fact, with its blemishes, this is probably the optimal environment for this book — one with an expert to help cut the wheat from the chaff.

Despite its serious flaws, Blake Watson's *Delphi By Example* is the only Object Pascal primer available just now. If you're satisfied with your visual programming skills and need an introduction to Delphi programming with plenty of down-and-dirty coding examples, this could be the one for you.

— *Jerry Coffey*

Delphi By Example by Blake Watson, Que Corporation, 11711 North College Avenue, Suite 140, Carmel, IN 46032; (800) 428-5331.

ISBN: 1-56529-757-1
Price: US$29.99
517 pages