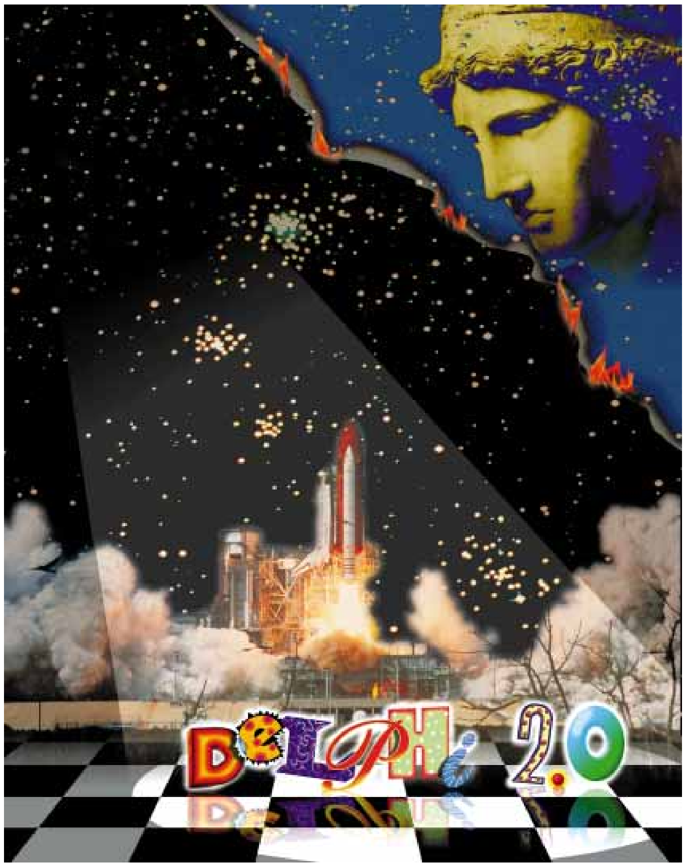
February 1996 - Volume 2, Number 2



Cover Art By: *Tom McKeith*

Serendipity!

Delphi 2.0 Makes Its Debut

Symposium

Straight line exists between me and the good thing.
I have found the line and its direction is known to me.
— David Byrne, "The Good Thing"
Talking Heads, *More Songs about Buildings and Food*

Visual programming without walls. Delphi 1.0 — and only Delphi 1.0 — makes it possible for the Windows 3.x environment. Now Delphi 2.0 offers it for Windows 95 development.

Just as it did the first time I fired it up, Delphi continues to dazzle me with its myriad qualities: speed, power, elegance, ease-of-use, extensibility, sound design, and — something else. Let's call it extravagance. Yes, Delphi 2.0's feature set is nothing short of extravagant. How else can you refer to a product which — just for starters — offers over 90 ready-made components?

With no real head-to-head competition for Delphi 1.0, and a less-than-hurried acceptance of Windows 95 in the business world, Borland decided to spend a little extra time with Delphi 2.0 and add even more features. The result is impressive, and is described by *DI* Contributing Editor, Richard Wagner, beginning on page 7.

But this isn't Richard's first look at Delphi 2.0. He initially previewed it for us in October of last year. This month's offering is really an update that describes the changes made since then. In case you missed that issue, let me give you a quick run down.

All that is necessary to turn a Delphi 1.0 application into a fully-functional, 32-bit, Windows 95 application, is to recompile it under Delphi 2.0 (unless, of course, you make an exotic API call, i.e. one that has changed or is no longer supported in Windows 95). It's also important to note that Windows 95 does not support VBXes, thus Delphi 2.0 does not support them. This could be painful if you're relying on a VBX that hasn't made the metamorphosis into an OCX, although most have.

There are other changes that are integral with a move to Windows 95. As Mr Wagner pointed out in October, there is no more 64KB limit, and Integer and Cardinal values are now 32-bit. Also, Borland has been careful to comply with all Windows 95 UI standards.

And okay, despite all the nice things I said earlier, Delphi 1.0 — or more properly, Object Pascal 1.0 — does have one annoying characteristic: its meshugeh, 255-character string type. But that's been taken care of with Delphi 2.0. You can have strings as long as you like. Better yet, they behave as Pascal strings *and* as C (null-terminated) strings, which will sure make those API and DLL calls easier. Besides the greatly enhanced string type, Object Pascal 2.0 offers two entirely new data types: the *variant* type can store a string, integer, or floating-point value and is particularly useful for OLE automation; and the *ANSIChar* type has been added to support Unicode.

Test (loops/sec.)	PowerBuilder 3.0 (16-bit)	Visual Basic 3.0 (16-bit)	Delphi 1.0 (16-bit)	Delphi 2.0 (32-bit)
Sieve	0.22	11.95	52.77	179.37
Whetstone	0.04	1.41	4.70	15.53
File write	0.05	0.42	0.74	2.89
File read	0.05	0.33	1.75	5.28

Delphi 2.0 also sports a greatly enhanced compiler that delivers executables even faster than those of Delphi 1.0 (see table). Moreover, the compiler now shares the same back-end as Borland's C++ product, making it possible, for example, to compile .OBJ files directly into your Delphi applications. In short, Delphi 2.0 is an impressive upgrade to an already awesome product, and a testament to the solid foundation upon which it is built.

Moving to another topic, your response to our new Web site has been overwhelmingly positive. Here's a sample:

Dear Editor,
Thanks for the web site! I'm an avid reader of Delphi Informant and very glad you've added a new dimension to your publishing. Now I'm able to locate files ... spotlighted in the monthly articles. You don't know how useful that can be.

Keep up the good work, and the web site.

Your loyal reader,
Wai Chong

And speaking of the Informant Communications Group Web site — at www.informant.com, natch — don't forget to cast your ballot for the First Annual *Delphi Informant* Reader's Choice Awards. You can also have your say on our CompuServe forum (GO ICGFORUM), or by just mailing in the Official Ballot you'll find later in this issue. And about that ballot. It contains two farshtunken errors: 1) the results will appear in the April issue, and 2) the voting deadline is February 20, 1996. The winners will be announced, for the first time, in late March at Software Development 96 West at Moscone Center in San Francisco.

Thanks for reading,

[illegible]

Jerry Coffey, Editor-in-Chief

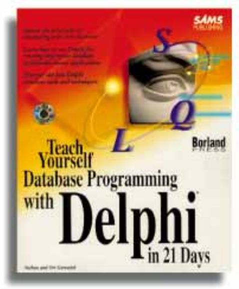
Internet: jcoffey@informant.com
CompuServe: 70304,3633
Fax: 916-686-8497
Snail: 10519 E. Stockton Blvd., Ste. 142,
Elk Grove, CA 95624

Delphi
T O O L S

New Products and Solutions

New Delphi Book

Teach Yourself Database Programming with Delphi in 21 Days
Nathan and Ori Gurewich
Sams Publishing



ISBN: 0-672-30851-7
Price: US$39.99, Canada $53.99 (569 pages, CD-ROM)
Phone: (800) 428-5331

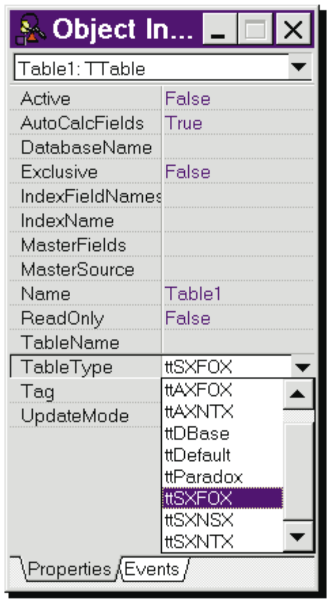
Now Shipping: Apollo 2.0 For Delphi

SuccessWare International, of Temecula, CA, is shipping *Apollo 2.0*, an alternative to Borland's Database Engine (BDE) in Delphi. Apollo 2.0 is designed to be smaller and faster than the current XBase DBMS engine, without requiring the BDE or ODBC.

The Replacement Database Engine (RDE) technology allows for record-based (xBASE) data-table navigation and management syntax during program development, with minimal concern for the ultimate format of the database. As a "no-code" replacement for the BDE, Apollo 2.0 can be installed into an existing application with no source code changes. This allows for multi-user access of xBASE data files from legacy applications. All concurrent-access record/file-locking is compatible with existing FoxPro and Clipper applications. Apollo 2.0 supports CA-Clipper (NTX), FoxPro2.x (IDX/CDX), and HiPer-SIx (NSX) systems.

Apollo 2.0 introduces new features to the Delphi marketplace, including Freeform-Text Searching (allows text documents to be embedded in data files with instantaneous retrieval); VariField support (fixed-length fields that automatically expand to meet the input requirements); image/BLOb support in memo fields (storage and retrieval without intermediate files); conditional indexes (only adds the records meeting a query/FOR condition); index SCOPES (instant index filters for control over the database views); and record-level data encryption.

Price: Apollo 2.0, US$179; Apollo 2.0 Pro (includes Delphi VCL source code), US$269. Both are royalty-free, and include 30 days of free voice and unlimited fax, CompuServe, or BBS technical support.



Active	False
AutoCalcFields	True
DatabaseName	
Exclusive	False
IndexFieldNames	
IndexName	
MasterFields	
MasterSource	
Name	Table1
ReadOnly	False
TableName	
TableType	ttSXFOX
Tag	
UpdateMode	

ttAXFOX
ttAXNTX
ttDBase
ttDefault
ttParadox
ttSXFOX
ttSXNSX
ttSXNTX

Contact: SuccessWare International, 27349 Jefferson Ave., Ste. 101, Temecula, CA 92590
Phone: (800) 683-1657 or (909) 699-9657
Fax: (909) 695-5679
BBS: (909) 964-6891
E-Mail: CIS: 74774,2240
CIS Forum: GO SWARE

Announcing BSS Business Systems Software

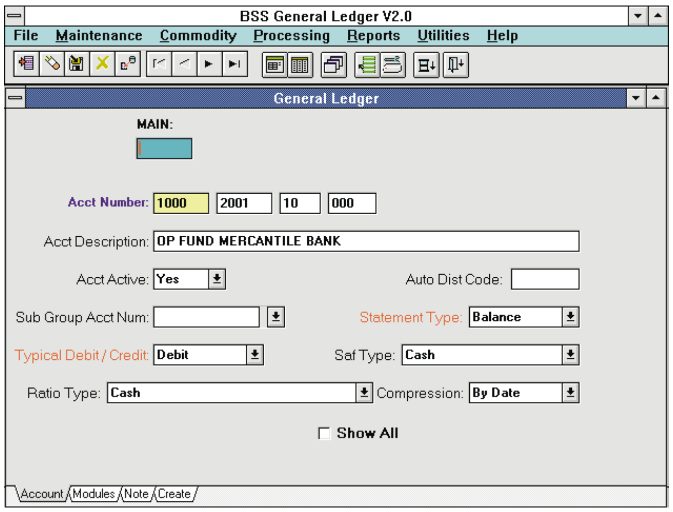
Business Software Systems, Inc., of Fairfax Station, VA, will release ***BSS Business Systems***, a line of accounting, distribution, and business management 32-bit GUI client/server software based on Delphi 2.0, Oracle and InterBase databases, and Crystal Reports PRO.

DB Manager, General Ledger, Accounts Payable, Accounts Receivable, and Bank Book will be available through an Early Experience Program (EEP) in early 1996.

According to the company, medium to large companies ready to upgrade their internal systems to a Windows-based client/server platform can select the modules required.

Source code is available for developers to modify any of the modules, and an application framework is available to assist in writing corporate-specific modules.

The BSS Application Framework includes a Single Document Interface (SDI) menu panel with custom DBNavigator VCL, registration, database login, table maintenance, transaction processing, and setup screens. It also features a special development project to ease team development.



MAIN:

Acct Number: 1000 2001 10 000
Acct Description: OP FUND MERCANTILE BANK
Acct Active: Yes
Auto Dist Code:
Sub Group Acct Num:
Statement Type: Balance
Typical Debit / Credit: Debit
Saf Type: Cash
Ratio Type: Cash
Compression: By Date
Show All

Price: Standard Edition starts at US$995 per module for a 5-user license; Enterpise Edition starts at US$5,000 per module for a 10-user license. BSS Application FrameWork with TurboPower's Orpheus controls 1.0, US$495.

Contact: Business Software Systems, Inc., P.O. Box 7416, Fairfax Station, VA 22039
Phone: (703) 503-5600
Fax: (703) 503-7901
E-Mail: CIS: 76652,2065

Delphi
TOOLS

New Products and Solutions

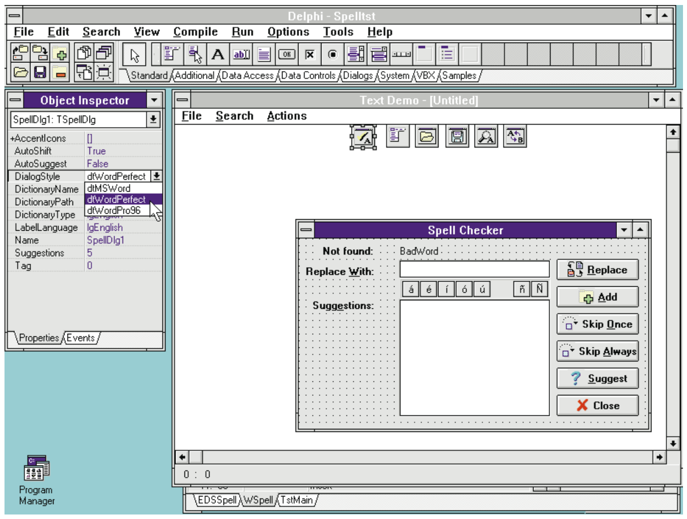
Add Spell Checking to Delphi Applications with EDSSpell

Eminent Domain Software of Laredo, TX recently released version 2.0 of its ***EDSSpell Checking Component for Delphi***. With EDSSpell, developers can add a spell checking engine to any Delphi application. Much like an OpenDialog or PrintDialog component, developers can drop a SpellDlg component onto their form. They can then spell check a document with one line of code:

```
SpellDlg1.CheckMemo(Memo1);
```

The EDSSpell component publishes several properties, giving developers control over language, dialog style (WordPerfect, Microsoft Word, or WordPro96), and more. EDSSpell ships with several languages including 125,000+ word English and 380,000+ word Spanish dictionaries. Other new features include support for TurboPower's Orpheus components, and word/unique word count.

Version 2.0 is a native Delphi component and requires no additional DLLs. There are no run-time royalties for distributing applications with EDSSpell.



SpellDlg1: TSpellDlg

+AccentIcons	[]
AutoShift	True
AutoSuggest	False
DialogStyle	dtWordPerfect
DictionaryName	dtMSWord
DictionaryPath	dtWordPerfect
DictionaryType	dtWordPro96
LabelLanguage	lgEnglish
Name	SpellDlg1
Suggestions	5
Tag	0

Properties Events

Not found: BadWord
Replace With:
Suggestions:

Price: US$99; upgrade from EDSSpell Version 1.0, US$20.

Contact: Eminent Domain Software, 413 Gale St., Laredo, TX 78041
Phone: (800) 246-5757 or (210) 729-0123
Fax: (210) 729-0011
E-Mail: Internet: easalgad@icsi.net or CIS: 70600,3451

Stylus Announces Telephony Support For Delphi

Stylus Innovation, Inc., of Cambridge, MA has announced the release of its telephony toolkit, ***Visual Voice for Delphi***, which allows Delphi developers to add telephony features to their applications.

Visual Voice is a suite of VBX controls and a graphical workbench for creating telephony and fax applications that include interactive voice response (e.g. touch-tone banking), fax-on-demand, and voice mail. Typical business systems built with Visual Voice include 24-hour customer order services, benefits enrollment hotlines, and brochure fax-on-demand systems.

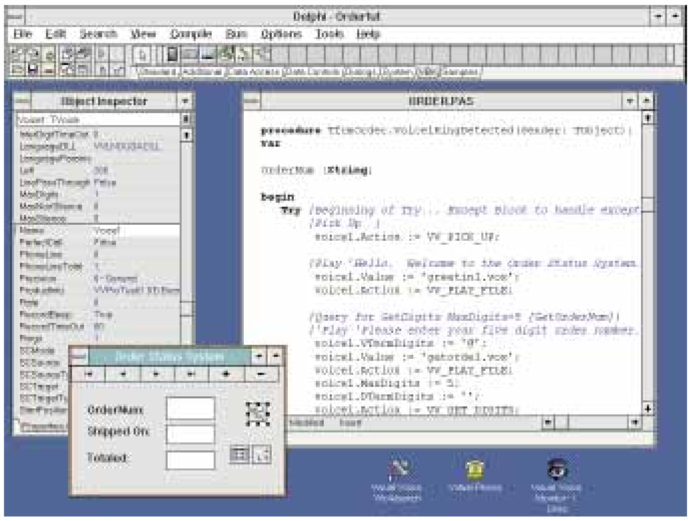
Visual Voice also features modules for text-to-speech and voice recognition support. In addition, it provides interfaces to the following functions: answering inbound calls, placing outbound calls, prompting for touch-tone input when calls are established, playing and recording voice files, sending and receiving faxes, and integrating with a PBX to perform call control functions such as transferring and conferencing.

Included in the Visual Voice toolkit is the Voice Workbench, a graphical tool to define the telephony components of an application and generate the corresponding application logic. The Workbench is used to create and maintain all telephony objects, including voice prompts, menus, and files.

Stylus' 32-bit OCX version of Visual Voice is currently in beta.

Price: Visual Voice for Windows 3.x begins at US$495. There are no run-time fees required.

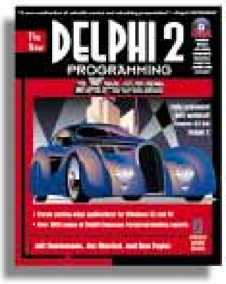
Contact: Stylus Innovation, Inc., One Kendall Square, Building 300, Cambridge, MA 02139
Phone: (617) 621-9545
Fax: (617) 621-7862
E-Mail: Internet: info@stylus.com
Web Site: http://www.stylus.com



```
procedure TTimeCodec.Voice1RingDetected(Sender: TObject);
var

OrderNum :String;

begin
  Try {Beginning of Try ... Except block to handle excep
      {Pick Up }
       voice1.Action := VV_PICK_UP;

      {Play 'Hello.  Welcome to the Order Status System.
       voice1.Value := 'greetin1.vox';
       voice1.Action := VV_PLAY_FILE;

      {Query for GetDigits MaxDigits=5 (GetOrderNum)}
      {'Play 'Please enter your five digit order number.
       voice1.VTermDigits := '#';
       voice1.Value := 'getordel.vox';
       voice1.Action := VV_PLAY_FILE;
       voice1.MaxDigits := 5;
       voice1.DTermDigits := '';
       voice1.Action := VV_GET_DIGITS;
```

New Delphi Book

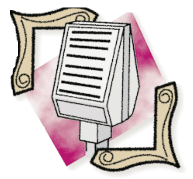
Delphi 2 Programming EXplorer
Jeff Duntemann, Jim Mischel, & Don Taylor
The Coriolis Group



ISBN: 1-883577-72-1
Price: US$44.99, Canada $62.99
(1000 pages, CD-ROM)
Phone: (800) 410-0192

News LINE

February 1996



VCL Contest for Delphi 2.0 Developers

Sams Publishing and Borland Press are holding a VCL competition for all Delphi 2.0 developers. The contest categories include: Interface Components, Internet/Communications, Database, Multimedia, and more. The top five winning entries in each category will receive US$100 and a copy of Sams' *Delphi 2.0 Developer's Guide, 2nd edition*. The grand prize is US$1,000 and a copy of *Delphi 2.0 Developer's Guide, 2nd edition*. The deadline for all entries is February 19, 1996. For complete contest rules, visit Borland Online at http://www.borland.com.

***Delphi Informant* Reader's Choice Deadline Extended**

The deadline for submitting ballots for the 1996 *Delphi Informant* Reader's Choice Awards has been extended to February 20, 1996. To vote, complete and submit the ballot included with this issue. Readers may also vote online using VOTE_DI.TX, an electronic version of the ballot available from our Web site (http://www.informant.com) or CompuServe forum (GO ICGFORUM).

Borland Ships Three Versions of Delphi 2.0

Scotts Valley, CA — Borland International has released three versions of its 32-bit Delphi application development tool for Windows 95 and Windows NT: Delphi 2.0 Desktop, Delphi 2.0 Developer, and Delphi 2.0 Client/Server.

Delphi 2.0 Desktop features a 32-bit optimizing native code compiler, over 85 standard components, a suite of Windows 95 common controls, data-aware components to build local database applications, and data modules for centralized data integrity and business rules.

It also has an object-oriented, extensible component architecture, visual inheritance and form linking, Windows 95 support for OLE automation and OCXes, as well as 32-bit development with Win API support, a Database Explorer to create and modify tables, aliases and indices, and an object repository to store and reuse objects and forms. Version 2.0 can create reusable DLLs and standalone EXEs, and can still be used for 16-bit Windows 3.1 development. It ships with Object Pascal documentation.

Delphi 2.0 Developer includes all the features found in the Delphi 2.0 Desktop edition, plus a scaleable data dictionary, Multi Object Grid, approximately 100 VCL components, advanced data aware components, a 32-bit version of ReportSmith, Borland Database Engine low-level API support and Help files, ODBC support, a single-user Local InterBase Server, and InstallShield Express.

Designed for developers, it also has additional experts and templates, Winsight32 for monitoring Windows messaging, expanded Open Tools API, team development interface (requires Intersolv's PVCS), and the Visual Component Library source code and printed reference.

Delphi 2.0 Client/Server includes all the features of the Delphi 2.0 Desktop and Developer editions, plus 32-bit SQL Link native drivers with unlimited deployment license, SQL Database Explorer, SQL Monitor, a 2-user InterBase NT license, new 16-bit Sybase System 10 and DB2 Native Drivers, a Data Pump Expert, Integrated Intersolv PVCS Version Control, ReportSmith SQL edition, Visual Query Builder, and cached updates. This client/server edition also includes integration with CASE tools, and client/server documentation.

Pricing for these three versions of Delphi 2.0 are: Delphi 2.0 Desktop US$499.95; Delphi 2.0 Developer US$799.95; and Delphi 2.0 Client/Server US$1999.95. For more details, see Richard Wagner's article "Delphi 2.0" beginning on page 14. Additional product information is available from Borland by calling (800) 331-0877 or (408) 461-9195, or by visiting their Web site at http://www.borland.com.

Delphi Informant on the Web

Pioneers In Product-Specific Technical Magazine Publishing

- Informant Magazines
- Informant Product Guides
- Informant CD-ROMs
- Files
- Advertising Information
- News from ICG
- Informant Apparel (Under Construction)
- About Informant Communications Group, Inc.

Copyright © 1995 Informant Communications Group, Inc. Elk Grove, CA

This page last modified December 1, 1995

[illegible]

Elk Grove, CA — Informant Communications Group, Inc. (ICG), publisher of *Delphi Informant*, *Paradox Informant*, and *Oracle Informant* magazines, has launched an Internet Web site at http://www.informant.com. From the initial Web page, users can choose from several sections including magazines, catalogs, CDs, files to download, advertising, ICG news, ICG apparel, and general information.

ICG also supports a CompuServe forum (GO ICGFORUM), and an FTP site (ftp.informant.com).

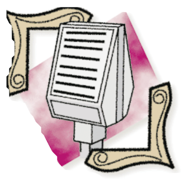
Companies and user groups interested in linking their Web site to the ICG Web site should contact Carol Boosembark at 72702.1274@compuserve.com, or call (916) 686-6610 ext. 16.

Borland Announces Agreement with Rios Corporation

Scotts Valley, CA — Borland International Inc. has signed an agreement with RIOS Corporation for the marketing and distribution rights of InterBase for UNIX in Japan. RIOS, the largest system integrator in Japan, chose InterBase because of its ability to scale in a client/server environment, and its support of UNIX and Windows-based applications.

According to the RIOS Corporation, they have been selling InterBase in Japan since 1992. In addition, RIOS has adopted Delphi and InterBase as their standard development tools for custom applications.

The RIOS Corporation has already begun marketing InterBase. Currently, InterBase 4.0 for Sun OS and HP-UX platforms is shipping. InterBase 4.0 for Solaris and AIX is slated for release in Japan in early 1996.



Borland Developers Conference Heads to Anaheim
Borland International Inc. has announced that the 7th Annual Borland Developers Conference will be held July 27-31, 1996 at the Anaheim Convention Center in Anaheim, CA. Conference speakers will be announced in March 1996. For more information contact Borland at (408) 431-1000 or http://www.borland.com.

Borland Online Set to Focus on Products
Borland International's World Wide Web site, Borland Online, will be redesigned for 1996. According to Borland, they are changing their Web site to better serve application developers, moving from a corporate-centric to a product-centric focus. From the home page, developers can select a product and be directed to Web pages specific to that product. Borland Online will also continue to upgrade and increase the links from its Java World page, and increase content associated with the Legal Toolbox.

Borland Endorses JavaScript

Mountain View, CA — Borland International Inc. is one of 28 companies to endorse JavaScript, released by Sun Microsystems, Inc. JavaScript is an open, cross-platform object scripting language for creating applications on enterprise networks and the Internet.

The JavaScript language complements Java, Sun's object-oriented, cross-platform programming language.

In addition, America Online, Inc., Apple Computer, Inc., Architext Software, Attachmate Corporation, AT&T, Brio Technology, Inc., Computer Associates, Inc., Digital Equipment Corporation, Hewlett-Packard Company, Iconovex Corporation, Illustra Information Technologies, Inc., Informix Software, Inc., Intuit, Inc., Macromedia, Metrowerks, Inc., Novell, Inc., Oracle Corp., Paper Software, Inc., Precept Software, Inc., RAD Technologies, Inc., The Santa Cruz Operation, Inc., Silicon Graphics, Inc., Spider Technologies, Sybase, Inc., Toshiba Corp., Verity, Inc., and Vermeer Technologies, Inc., have endorsed JavaScript as an open standard object scripting language, and intend to provide it in future products.

JavaScript is an object scripting language designed for creating online applications that link objects and resources on both clients and servers. While Java is used by programmers to create new objects and applets, JavaScript is designed for use by HTML page authors and enterprise application developers to script the behavior of objects running on either the client or the server. JavaScript is designed specifically for the Internet and creating network-centric applications. It also integrates with Java and HTML.

Java is available to developers free of charge. The Java Compiler and Java Developer's Kit as well as the HotJava browser and related documentation are available from Sun's Web site at http://java.sun.com. In addition, the Java source code can be licensed for a fee. Details on licensing are also available via the java.sun.com Web page.

To date, Sun has licensed Java to a number of leading technology companies, including Borland, Macromedia, Mitsubishi, Netscape, Oracle, Silicon Graphics, Spyglass, and Toshiba. Sun's Workshop for Java Toolkit is scheduled for release in Spring 1996.

Netscape and Sun plan to propose JavaScript to the W3 Consortium (W3C) and the Internet Engineering Task Force (IETF) as an open Internet scripting language standard. JavaScript will be an open, freely licensed proposed standard available to the entire Internet community. Existing Sun Java licensees will receive a license to JavaScript.

In addition, Sun and Netscape intend to make a source code reference implementation of JavaScript available for royalty-free licensing.

Additional information on Sun Microsystems is available on the Internet at http://www.-sun.com or, for Java information, http://java.sun.com. For more information on Netscape products, visit http://home.-netscape.com, send an e-mail to info@netscape.com, or call (415) 528-2555.

Database & Client/Server World Set for Boston in March

Andover, MA — Scheduled for March 26-28, 1996, Database and Client/Server World will feature over 800 exhibits at the Hynes Convention Center in Boston, MA.

With over 25,000 MIS professionals expected to attend, Database & Client/Server World is an exposition of databases, tools, warehousing, and application development products.

The event features database and client/server experts Dr E.F. Codd, Jeff Tash, Richard Finkelstein, Chris Date, Ronald G. Ross, Shaku Atre, Herb Edelstein, Ken Orr, Paul Harmon, Roger Burlton, Larry R. DeBoever, Max Dolgicer, Richard Winter, Ken Lownie, Aaron Zornes, Michael Stonebraker, and Thomas Lipscomb.

In addition, Database & Client/Server World will feature nine conferences: Client/Server Tools and Application Development, Data Warehouse and Repositories, Systems Management for the Client/Server Environment, Parallel Databases and VLDBs, Data and Object Modeling, Object-Oriented Technologies, Middleware, Groupware Application Development, and Multi-Tier Architectures and Application Partitioning.

For more information visit DCI's Web site at http://www.dciexpo.com.

table of CONTENTS

By Richard Wagner

Delphi 2.0

The Serendipitous Tool Makes Its 32-Bit Debut

Serendipity. This is the word that came to mind when I first saw the latest field test of Delphi 2.0. The enhancements from earlier field tests were both wonderful and unexpected. If you recall, in the October 1995 *Delphi Informant*, we previewed Delphi 2.0 during its initial beta release. At that time, it already sported some amazing enhancements over the 16-bit version of the product, including: a 32-bit optimizing compiler, OCX support, Windows 95 UI components, multithreading support, a new 32-bit database engine, new data types, closer coupling with C++, and an enhanced IDE.

The Delphi 2.0 R&D team was obviously not content with the status quo. In subsequent builds, there has been a steady influx of major new features that, when combined, make Delphi 2.0 one of the most extensive version upgrades I've ever seen. This article will focus on many of the new features that have been added to the 32-bit product since our October survey.

New UI Controls

In addition to the eight Windows 95 UI components we discussed in October, there are three new ones: *TListView*, *THotKey*, and *TStatusBar*. First, the *TListView* control will enable you to use one of the fundamental Win95 user interface elements in your applications. Anywhere you go in Windows 95 — such as My Computer, Control Panel, or Windows Explorer — you work with a ListView control (see Figure 1). It's an intuitive way of displaying list information in a variety of styles (Large Icon, Small Icon, List, or Details).

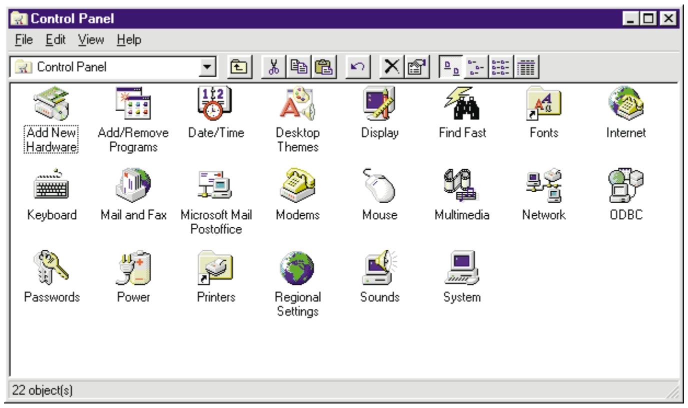


Figure 1: Windows Control Panel uses a ListView to present options to the user.

Using a *TListView* component, you can use this method of presentation in your Delphi applications as well. For example, Figure 2 shows a sample application called Resource Explorer that ships with the current Delphi 2.0 build. Using *TTreeView* and *TListView* components, it effectively presents resource data in a metaphor similar to the Windows Explorer. Given the

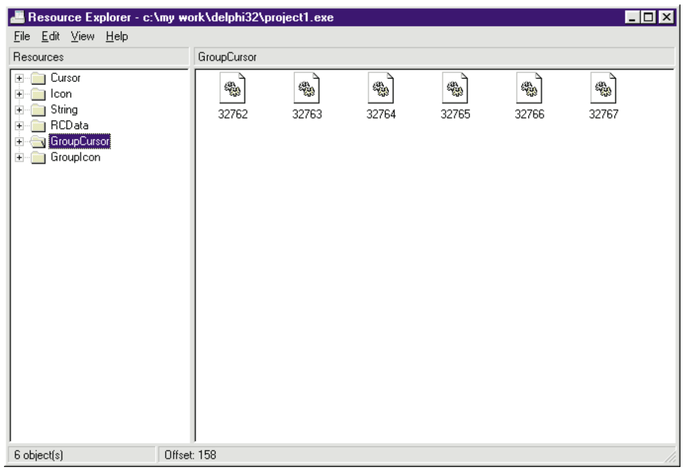


Figure 2: This sample Delphi application employs TreeView and ListView components.

success of the Windows 95 user interface, you'll probably want to incorporate TreeView and ListView components into your application UIs.

The *TStatusBar* component provides a simple way for developers to give their applications a standard Windows 95 status bar, while the *THotKey* component allows the user to quickly perform a specified action using a combination of keys (such as Ctrl C). All the Windows 95 UI components available with Delphi 2.0 are briefly described in the sidebar on page 12.

Multithreading

New UI controls may change the way you present information to the user, but a new feature called *multithreading* could fundamentally change your approach to application development.

Using Delphi 2.0, you can create threaded applications. It will be important for developers to resist the urge of going "thread crazy" and create multiple threads in even the tiniest of applications. With this caveat in mind, when multithreading is used wisely, it can be extremely powerful, and is perhaps the most convincing factor for developing Win32 applications instead of creating programs for the Windows 3.1 16-bit environment.

In our October preview, we had a high-level discussion on Delphi 2.0's multithreading support. In this installment, let's take a closer, interactive look at multithreading by building a basic two-thread application. (Note that documentation for multithreading was minimal at press time.)

Suppose you want to execute two queries simultaneously in an application, but would like to prioritize the queries. To do this, you could place the queries into separate threads. Figure 3 shows the UI for this example. The priority of the two threads is dependent on the position of the TrackBars shown on the left of the form, and the results of the queries are shown in the two *TDBGrid* controls.

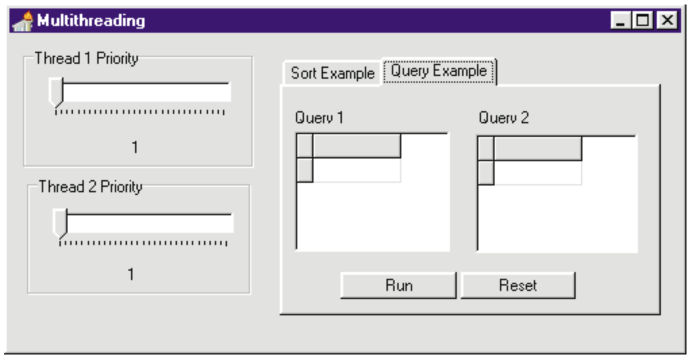


Figure 3: A sample application executing queries in different threads.

Diving inside the code, we must first separate the two queries into different routines that we'll call *Query1Thread* and *Query2Thread*. The first query is executed when *Query1Thread* is called:

```
function Query1Thread(parms: pointer) : LongInt; far;
begin
  Form1.LongQuery.Open;
end;
```

Then, the second query is executed when *Query2Thread* is called:

```
function Query2Thread(parms: pointer) : LongInt; far;
begin
  Form1.ShortQuery.Open;
end;
```

Now we can add the multithreading code. The *Button2Click* procedure uses the *createThread* method to create threads for the queries and then set a thread's priority based on the position of its associated TrackBar (note that the T3 and T4 variables are of type *THandle*):

```
procedure TForm1.Button2Click(Sender: TObject);
var
  ThreadID : dWord;
begin
  T3 := createThread(nil,0,@Query1Thread,nil,0,threadID);
  setThreadPriority(T3,FirstTrackBar.Position);
  T4 := createThread(nil,0,@Query2Thread,nil,0,threadID);
  setThreadPriority(T4,SecondTrackBar.Position);
end;
```

Many articles on multithreading (especially on how to best employ it) are surely in the works. But even when running this rudimentary demonstration, you can visibly see the results of what threading can do for you.

Database Enhancements

Delphi 2.0 now provides several new enhancements that database application developers will definitely appreciate. These include cached updates, *TField* lookup fields, table filters, and two enhanced data controls.

Cached Updates. An important part of any client/server application is how the client works with the SQL server to process data. In Delphi 1.0, every client data event is individually sent to the database as a separate SQL transaction. This process is known as *navigational updating*.

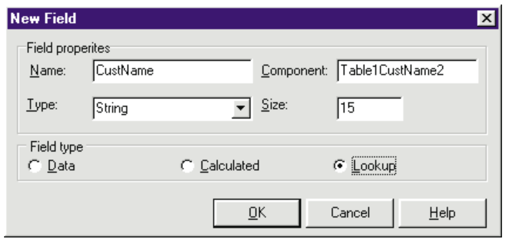
Delphi 2.0 continues to support navigational updates, but adds a new transaction mechanism called *cached updates*. Rather than sending each data update as a separate SQL transaction, cached updates gather multiple client transactions and send them in a batch to the server within a single SQL transaction.

Cached updates have several uses, but can be particularly effective in situations where network traffic and record contentions are major concerns. However, keep in mind that both navigational and cached update models have advantages and disadvantages. Therefore, the developer is responsible for employing the proper transaction method within a given context. Not only do cached updates work on SQL databases, but they also work on local databases such as Paradox and dBASE.

To enable cached updates, set the *CachedUpdates* property of a *TTable* object to *True*. Then, any updates made by the user will be displayed on screen, but will not actually be sent to the database until the *TTable* method *ApplyUpdates* is executed. You can cancel the current record's update using the *CancelCurrentUpdate* method or cancel all updates within the current transaction using *CancelUpdates*.

Lookup Support for *TField*. In addition to providing simply calculated field support, *TField* objects in Delphi 2.0 also furnish support for lookup fields. This capability will allow you to specify lookup fields to receive a value from another table. Lookup fields are created similarly to calculated fields using the Fields editor. A New Field dialog box is displayed in which you can specify its field type (see Figure 4). You then set four properties — *LookupTable*, *LookupMasterFields*, *LookupDetailFields*, and *LookupResultField* — for the read-only lookup field. Figure 5 shows a lookup field in a *TDBGrid*.

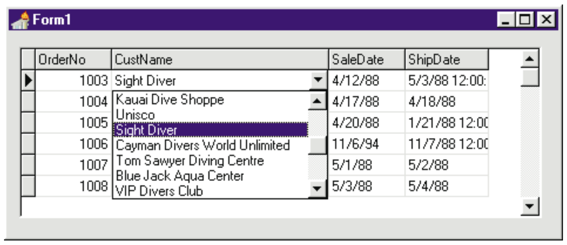
Table Filters. In developing database applications, developers periodically want to use both queries and filters to display a limited set of data to the user. To use filters in Delphi 1.0, you must use a third-party component or create your own because *TTable* does not provide built-in filtering.


New Field

Field properites
Name: CustName
Component: Table1CustName2
Type: String
Size: 15

Field type
( ) Data ( ) Calculated [✓ (•) Lookup]

Figure 4: The New Field dialog box.


Form1

	OrderNo	CustName	SaleDate	ShipDate
▶	1003	Sight Diver	4/12/88	5/3/88 12:00:
	1004	Kauai Dive Shoppe	4/17/88	4/18/88
	1005	Unisco	4/20/88	1/21/88 12:00
	1006	Sight Diver	11/6/94	11/7/88 12:00
	1007	Cayman Divers World Unlimited	5/1/88	5/2/88
	1008	Tom Sawyer Diving Centre	5/3/88	5/4/88

Kauai Dive Shoppe
Unisco
Sight Diver
Cayman Divers World Unlimited
Tom Sawyer Diving Centre
Blue Jack Aqua Center
VIP Divers Club

Figure 5: A lookup field is shown in the enhanced *TDBGrid* component.

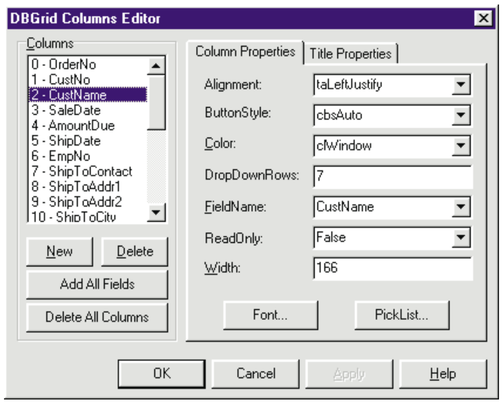
Delphi 2.0 makes filtering much easier by adding filter support to the basic *TTable* component. By setting the *TTable's Filtered* property to *True*, each record is evaluated when it's retrieved by the data set. You can configure a filter to "block out" all records that do not match your criteria by placing the data filter in the *OnFilterRecord* event handler for the *TTable* object.

For example, suppose you want to filter out all records in a Customer table for people not living in Massachusetts. To do this, you would write the following filter:

```
procedure TForm1.CustomerTblFilterRecord(
  DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['State'] <> 'MA';
end;
```

Enhanced Data Controls. Data presentation just got much easier in Delphi 2.0 with two new/enhanced data controls: a much improved *TDBGrid* component and a new *TDBCtrlGrid* component. Both of these allow you to simultaneously display multiple records, but in different ways.

Many people consider the *TDBGrid* component one of the weaker components in Delphi 1.0. You have little control over the columns and cannot easily create lookup field support within the grid itself. These deficiencies alone probably sold many third-party replacements for *TDBGrid*. Conversely, the new *TDBGrid* has many improvements, the two most important of which include a Columns Editor that allows you to set an array of properties for a given column (see Figure 6), and the ability to display *TField* lookups (see Figure 7).

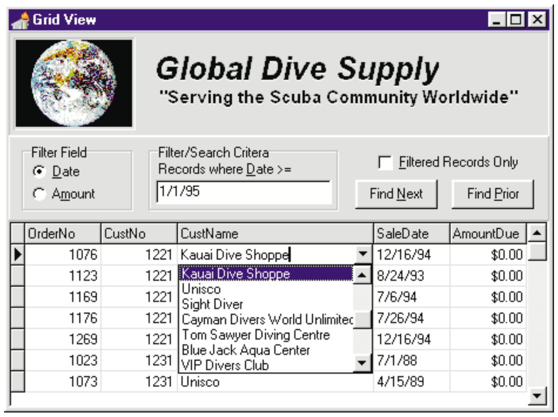

DBGrid Columns Editor

Columns
0 - OrderNo
1 - CustNo
2 - CustName
3 - SaleDate
4 - AmountDue
5 - ShipDate
6 - EmpNo
7 - ShipToContact
8 - ShipToAddr1
9 - ShipToAddr2
10 - ShipToCity

Column Properties | Title Properties

Alignment: taLeftJustify
ButtonStyle: cbsAuto
Color: clWindow
DropDownRows: 7
FieldName: CustName
ReadOnly: False
Width: 166

Figure 6: The DBGrid component's Columns Editor allows you to customize the look of each column in the grid.


Grid View

Global Dive Supply
"Serving the Scuba Community Worldwide"

Filter Field
(•) Date
( ) Amount

Filter/Search Critera
Records where Date >=
1/1/95

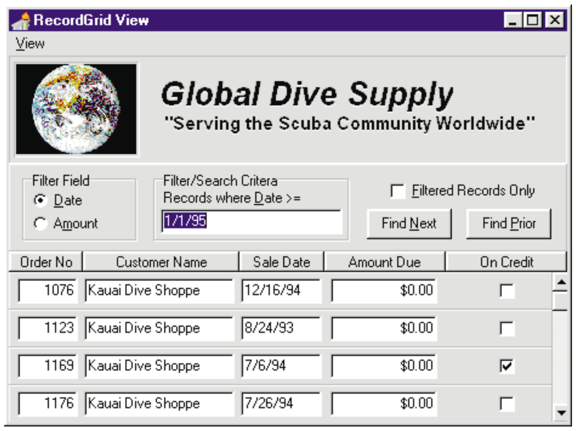
[ ] Filtered Records Only

	OrderNo	CustNo	CustName	SaleDate	AmountDue
▶	1076	1221	Kauai Dive Shoppe	12/16/94	$0.00
	1123	1221	Kauai Dive Shoppe	8/24/93	$0.00
	1169	1221	Unisco	7/6/94	$0.00
	1176	1221	Sight Diver	7/26/94	$0.00
	1269	1221	Cayman Divers World Unlimited	12/16/94	$0.00
	1023	1231	Tom Sawyer Diving Centre	7/1/88	$0.00
	1073	1231	Unisco	4/15/89	$0.00

Kauai Dive Shoppe
Unisco
Sight Diver
Cayman Divers World Unlimitec
Tom Sawyer Diving Centre
Blue Jack Aqua Center
VIP Divers Club

Figure 7: New and improved *TDBGrid* supports combination boxes and other data controls.

The *TDBCtrlGrid* is a powerful new component for displaying data. Like *TDBGrid*, *TDBCtrlGrid* allows you to display multiple records of a table, but in a more flexible manner. Not only can you display any data aware control in it, but you can also specify how many rows and columns of records to display. If you have ever used Paradox for Windows, you will find that the *TDBCtrlGrid* is essentially the equivalent of Paradox's multi-record object. Figure 8 shows an example of a *TDBCtrlGrid*.



Filter Field: (● Date) Amount

Records where Date >= 1/1/95

Order No	Customer Name	Sale Date	Amount Due	On Credit
1076	Kauai Dive Shoppe	12/16/94	$0.00	☐
1123	Kauai Dive Shoppe	8/24/93	$0.00	☐
1169	Kauai Dive Shoppe	7/6/94	$0.00	✓
1176	Kauai Dive Shoppe	7/26/94	$0.00	☐

Figure 8: The new DBCtrlGrid component gives you greater flexibility in presenting multiple records of data.

New IDE Features

The development environment of Delphi 2.0 continues to mature with the addition of visual form inheritance, data modules, visual form linking, and an Object Repository.

Visual Form Inheritance. One of the biggest shortfalls of Delphi 1.0 was the inability to truly subclass *TForm* objects. Delphi 2.0 answers this deficiency with *visual form inheritance*. This functionality allows you to create descendant forms from an original base form, all without writing any code. Unlike Delphi 1.0's version of "form templates" (that were really just copies of your favorite forms), Delphi 2.0 allows you to create true "abstract" forms within and across projects. When you update an ancestor form, all its descendants are updated at the same time. You can even view this simultaneous update on screen.

To illustrate, we'll look at a demonstration application called Gdsdemo. It's an example of presenting the same data using both a *TDBGrid* and a *TDBCtrlGrid*. However, it also shows the power of inheritance.

Gdsdemo uses a base form called *GDSStdForm* (see Figure 9) that has the company logo header used by two forms. A descendant form of *GDSStdForm*, called *StdDataForm* (also shown in Figure 9), adds filtering controls onto the base form.

Neither of these forms are ever seen by the user — they are simply abstract objects being used by their children: *GridViewForm* and *RecGridForm* (again, see Figure 9). Both of these forms inherit the properties, events, and methods from their two ancestor forms. Therefore, if you change the bitmap of *GDSStdForm*, all other forms in the application would be updated. Additionally, if you change the event handler for the Find Next button on the *StdDataForm*, the associated handler in *GridViewForm* and *RecGridForm* will be updated as well.

The importance of visual form inheritance is clear for object-oriented programmers. In Delphi 2.0, you can extend the OOP boundary to now include forms, as well as data modules (which we'll discuss below).

Data Modules and Visual Form Linking. When developing multi-form database applications in Delphi 1.0, I find the form-specific nature of data access components to be frustrating. Because of this limitation, if you use a table across 16 different forms, you must place a *TTable* component on each of the 16 forms if you linked that table to a UI object. Additionally, if you need to synchronize the data across multiple forms, you are forced to write the code to perform the synchronization.

Thankfully, Delphi 2.0 added data modules to combat this unnecessary chore. A *data module* serves as a container for non-visual database components (see Figure 10). Using a data module, you can manage your database access code from a centralized facility. You can also keep all your database logic in this single location rather than spread throughout the application.

A data module is form-like, so you can work with data modules quite similarly as you do with Delphi forms. However, a data module does not have a UI and thus will never appear to the end user of your application.

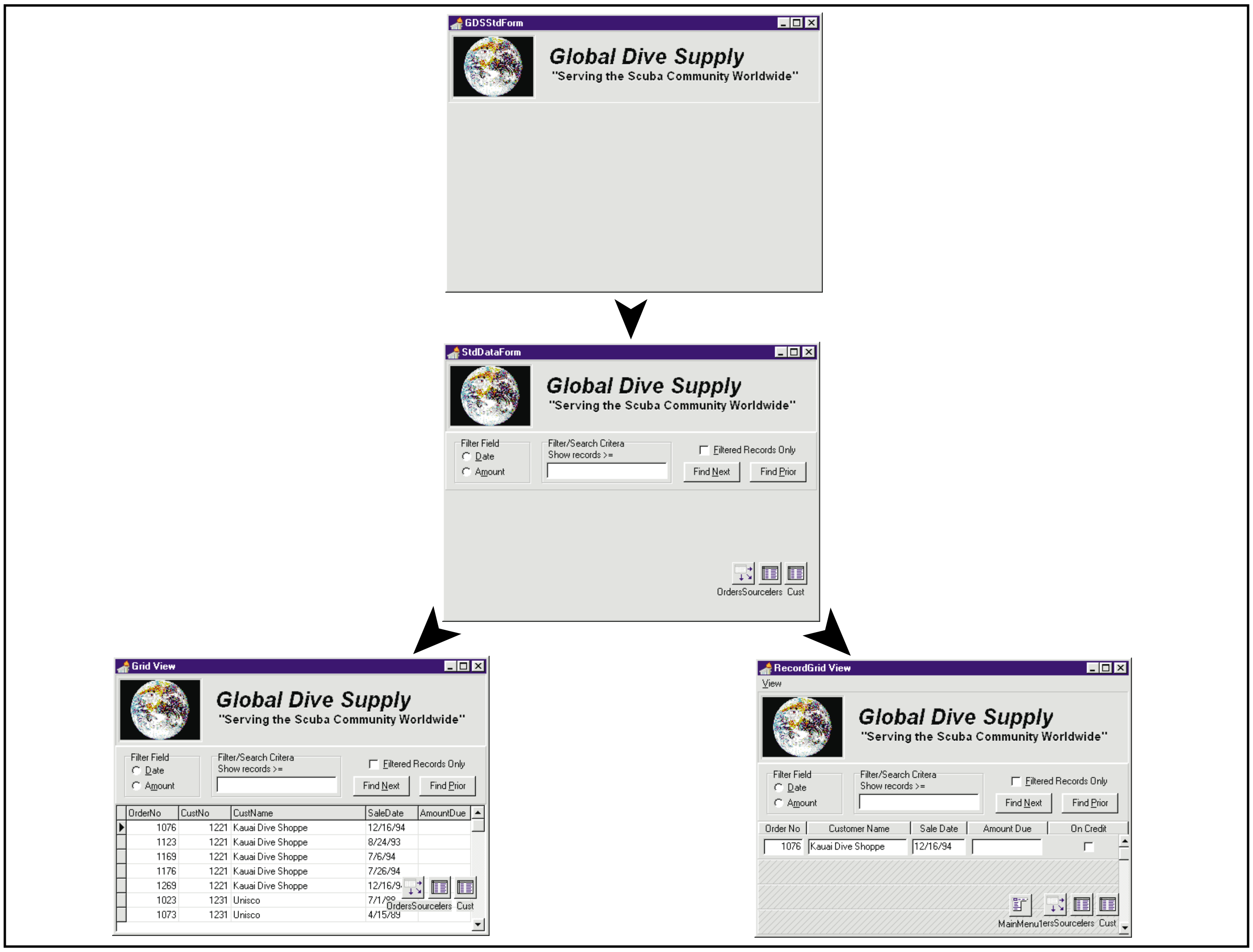
Having a centralized data house is possible through *visual form linking*. With this functionality, you can link a data component on one form to a data aware component on another form or data module. Delphi 2.0 links forms and data modules by linking their associated unit files. Therefore, once you declare a form's unit in the **uses** clause of another form, you can access any of its data components. Figure 11 shows the dot notation reference in the Object Inspector.

Object Repository. Delphi 2.0 uses the Object Repository as an organizing tool to manage form objects, data modules, project experts, and project templates. The Object Repository is really an enhanced version of Delphi 1.0's Gallery. After adding an object to this facility, you can inherit, reference, or copy the object in future applications. You'll notice the contents of the Object Repository are displayed when you select the File | New from Delphi's menu (see Figure 12). In multi-developer environments, you can maintain a single network-based Object Repository and reference it from every developer workstation.

Noteworthy Changes

There are many miscellaneous changes that are noteworthy (including, but not limited to):

- The File | Use Unit command adds the **select** unit to the **uses** clause of the current unit file (or associated unit file of the active form).



	OrderNo	CustNo	CustName	SaleDate	AmountDue
▶	1076	1221	Kauai Dive Shoppe	12/16/94	
	1123	1221	Kauai Dive Shoppe	8/24/93	
	1169	1221	Kauai Dive Shoppe	7/6/94	
	1176	1221	Kauai Dive Shoppe	7/26/94	
	1269	1221	Kauai Dive Shoppe	12/16/9	
	1023	1231	Unisco	7/1/	
	1073	1231	Unisco	4/15/89	

Order No	Customer Name	Sale Date	Amount Due	On Credit
1076	Kauai Dive Shoppe	12/16/94		☐

Figure 9: Delphi 2.0 features visual form inheritance. In this example, the form StdDataForm (second from the top) inherits all the properties and behavior of GDSStdForm. In turn, GridViewForm and RecGridForm (at the bottom of the figure) inherit from StdDataForm.

- A new comment symbol (`//`). In addition to previous comment delimiters (`{}`) and (`(* *)`), all words to the right of `//` are considered comments.
- The Thread Status window will allow you to monitor threads running in your application.
- *TSession* is now a component on the Component Palette.
- Non-visual components are now easier to distinguish. You can select the **Show Component Captions** option to see their labels in design mode.
- Delphi 2.0 supports unit aliasing. You can use an alias to refer to another unit file.
- The Fields editor now provides drag-and-drop support. You can drag a field from the Fields editor onto any form and a field object will automatically be created and set to that field. This enhancement will make form creation much easier.
- The Client/Server edition of Delphi 2.0 will provide built-in PVCS version control support. [For a complete description of the three new editions of Delphi 2.0 that are available, see the news item "Borland Ships Three Versions of Delphi32" on page 5.]

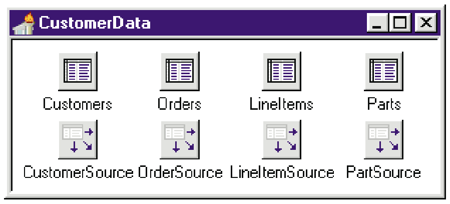
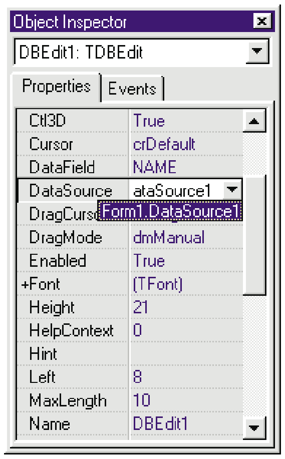


Figure 10: Delphi 2.0 supports centralized, reusable holders for nonvisual database components called *data modules*. They're specialized Delphi forms that can be modified just as other forms.



Ctl3D	True
Cursor	crDefault
DataField	NAME
DataSource	ataSource1
DragCurs	Form1.DataSource1
DragMode	dmManual
Enabled	True
+Font	(TFont)
Height	21
HelpContext	0
Hint	
Left	8
MaxLength	10
Name	DBEdit1

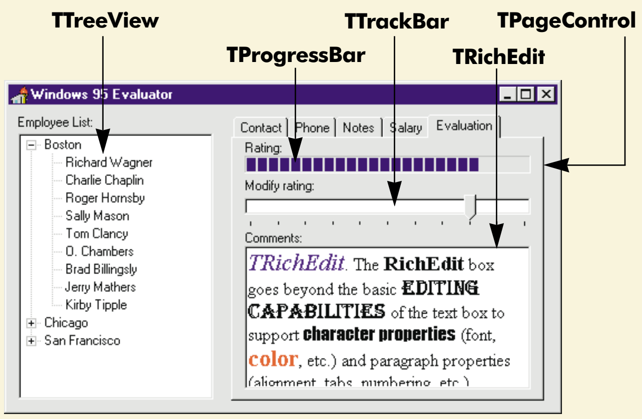
Figure 11: The new *visual form linking* feature allows you to access data controls in other forms using dot notation.

Conclusion

No matter how you use Delphi 2.0, it's proving to be a convincing upgrade. Database developers will be flooded with a host of new database tools, components, data modules, and database engine enhancements. General

Windows 95 Common UI Controls

Delphi 2.0 provides new Windows 95 common UI controls on its Component Palette. Figure A illustrates the use of several of these controls.



TRichEdit. The RichEdit box goes beyond the basic EDITING CAPABILITIES of the text box to support character properties (font, color, etc.) and paragraph properties (alignment, tabs, numbering, etc.)

TTabControl. The TabControl allows you to create a set of tabs. If you want the tabs associated with "pages," then use the PageControl component.

TPageControl. PageControl is used to create Win95-style tabbed dialog boxes. With this component, you can add pages more easily than with its Delphi 1.0 predecessor (TabbedNotebook) by right-clicking and choosing **New Page** from the menu. Another ease-of-use feature is the ability to activate a page by clicking its tab with your mouse rather than having to change the *ActivePage* property in the Object Inspector.

TTreeView. TreeView is a Win95 version of the Outline component in Delphi 1.0. The TreeView component is used quite often in Windows 95. Look at the Windows Explorer or Delphi 2.0's Database Explorer (see Figure 7 earlier in the article) for other examples of a TreeView.

TListView. The ListView control encapsulates the UI capabilities exemplified by Windows 95 Control Panel.

TTrackBar. The TrackBar is a "slider like" component used to adjust values that fall within a continuous range.

TProgressBar. The ProgressBar is a Win95 "percentage meter" that enables you to show the percentage remaining in a lengthy process.

THeaderControl. Enhancing the capabilities of the 16-bit Header component, the HeaderControl is used to display headings above columnar data. You can divide the header into multiple sections when you need to place a heading above multiple columns of text or numbers.

TStatusBar. The StatusBar allows you to add standard Windows 95 status bars to your applications.

TRichEdit. The RichEdit box goes beyond the basic editing capabilities of the text box to support character properties (font, color, etc.) and paragraph properties (alignment, tabs, numbering, etc.).

TUpDown. Used in conjunction with an edit box, the UpDown control is typically used to create a circular loop of input values.

THotKey. The HotKey component performs a specified action using a combination of keys (e.g. Ctrl C).

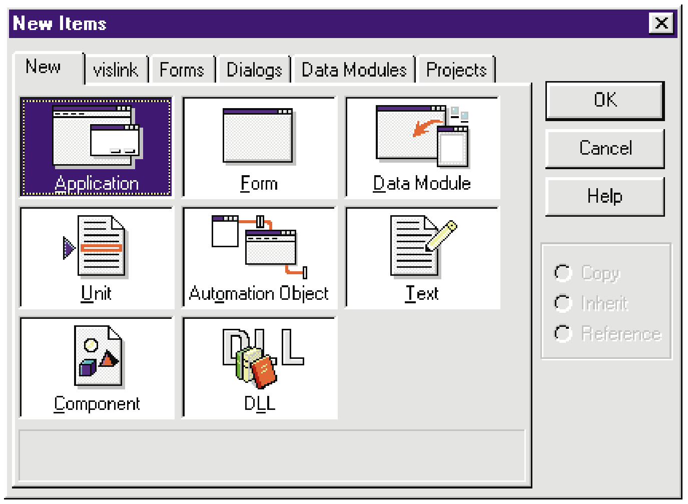


Figure 12: An enhanced version of the Delphi 1.0 Gallery, the New Item dialog box is displayed when you select **File | New** from the Delphi 2.0 menu.

and OOP programmers will definitely welcome its closer C++ integration, optimizing backend compiler, ability to create OLE servers, and multithreading support. And everyone will enjoy a more sophisticated IDE with its visual form inheritance, Object Repository, and other new features.

As much as I love Delphi 1.0, I recognize that it has some missing pieces. Quite serendipitously, many of the gaps in Delphi 1.0 have been filled in its 32-bit successor. ∆

This article is based on a prerelease version of Delphi 2.0 and may describe features that differ or are entirely absent from the shipping version.

Richard Wagner is a Chief Technical Officer for Acadia Software (formerly IT Solutions/Boston) located in the Boston, MA area. He is author of several Paradox, Windows, and CompuServe/Internet books, and is also a member of Team Borland on CompuServe. Richard can be reached on CompuServe at 71333,2031 or via the Internet at rwagner@cis.compuserve.com.

By Jeff Chant

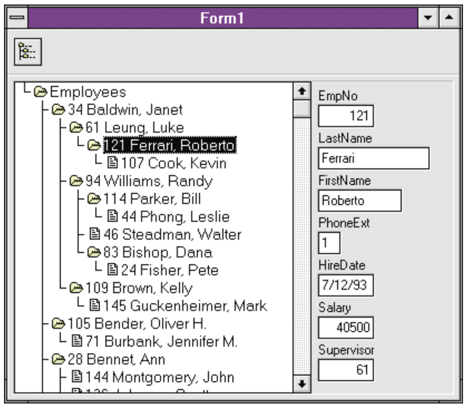
DBOutline

Using TOutline to Manage Hierarchical Data

Delphi's *TOutline* component is a versatile tool for viewing hierarchically-organized data. One tremendous example of its power is its ability to manipulate the hierarchy of data through drag-and-drop. *TOutline* can even be used to navigate a DataSet and update physical data records.

Most developers will appreciate the value of this feature when confronting a table that refers to itself for a parent record. Such *self-referencing* tables are numerous in the business world: employee files referring to themselves for a supervisor; address masters referring to themselves for a parent address; and business segment tables referring to themselves for a parent segment. These recursive relationships are often many levels deep, and the deeper the nesting becomes the more difficult it is for end-users to manage them with normal grid- or form-based interfaces. Solving this problem demands a tool capable of editing *hierarchically*-related data. The *TOutline* component — a hierarchy-oriented tool — fulfills this requirement perfectly.

Figure 1 shows a sample application that accesses a company's employee data. Employees are nested under their respective supervisors. Clicking on an outline node (an employee) causes the *TDBEdit* fields to display the information associated with that node. For example, if employee 105 is dragged and dropped on employee 121, then 105 becomes a child node of 121, and the Supervisor field is updated to reflect the new parent record. This application can be used to maintain the hierarchy of any recursive table with only minor changes to the source code.



```
Employees
  34 Baldwin, Janet
    61 Leung, Luke
      121 Ferrari, Roberto
        107 Cook, Kevin
    94 Williams, Randy
      114 Parker, Bill
        44 Phong, Leslie
      46 Steadman, Walter
      83 Bishop, Dana
        24 Fisher, Pete
    109 Brown, Kelly
      145 Guckenheimer, Mark
  105 Bender, Oliver H.
    71 Burbank, Jennifer M.
  28 Bennet, Ann
    144 Montgomery, John
```

EmpNo	121
LastName	Ferrari
FirstName	Roberto
PhoneExt	1
HireDate	7/12/93
Salary	40500
Supervisor	61

Figure 1: This application provides a hierarchic view of a *self-referencing* employee table. The user can manipulate the hierarchy (i.e. the contents of the Supervisor field) using drag-and-drop.

Producing the sample application is a multi-stage process. First, make a self-referencing table. Next, create the form and add the components. Third, construct an algorithm to populate the *TOutline* component with information from the self-referencing table. Once the algorithm is complete, implement drag-and-drop procedures to allow the manipulation of the outline nodes. Finally, establish a simple procedure to keep the current record of the *TTable* synchronized with the currently selected *TOutlineNode*.

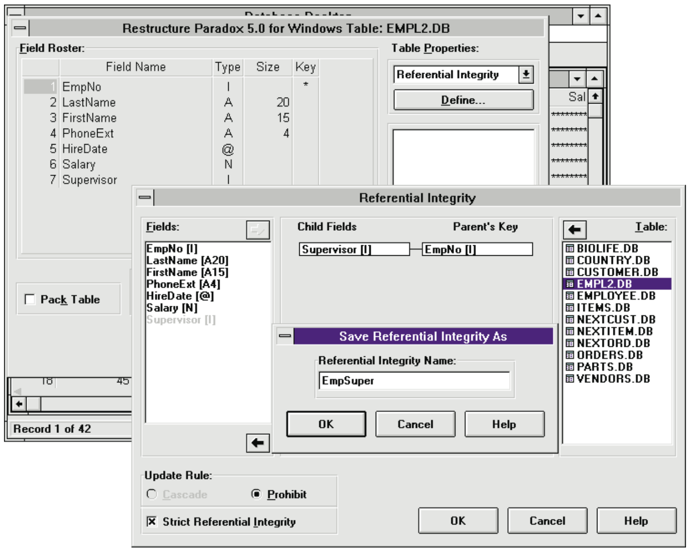
Creating a Self-Referencing Table

Before the DBOutline program can be created, a self-referencing table must be constructed. The example in Figure 1 uses a modified version of the Employee (EMPLOYEE.DB) table that ships with Delphi. It's accessed using the default alias, DBDEMOS. Using the Database Desktop, the modified table is produced as follows:

- Select **File | Working Directory**.
- Select the Alias `DBDEMOS`, then click on **OK**.
- Select **Utilities | Copy**.
- In the Copy dialog box, select `DBDEMOS` for **Drive (or Alias)**.
- Select `EMPLOYEE` for **Copy File From**, and enter `EMPL2` in **To**.
- Select **OK** to perform the copy.

This procedure creates a new Paradox table named EMPL2.DB that is a replica of the Employee table. To make the table self-referencing, a new field is added to hold the employee number of each employee's supervisor. To add this field and define the self-referencing relation to the table, proceed as follows:

- Use **File | Open | Table** to open EMPL2, and select **Table | Restructure Table** to modify it.
- Add a field named Supervisor of Type I (i.e. Long Integer).
- Save the changes to the table, then select **Table | Restructure Table** to modify it again.
- Select the **Table Properties** combo box and select **Referential Integrity**. Select **Define** to create a new reference. Create the reference (see Figure 2), and name the reference `EmpSuper`.



Restructure Paradox 5.0 for Windows Table: EMPL2.DB

Field Roster:

	Field Name	Type	Size	Key
1	EmpNo	I		*
2	LastName	A	20	
3	FirstName	A	15	
4	PhoneExt	A	4	
5	HireDate	@		
6	Salary	N		
7	Supervisor	I		

Table Properties: Referential Integrity

Pack Table

Record 1 of 42

Referential Integrity

Fields: EmpNo [I], LastName [A20], FirstName [A15], PhoneExt [A4], HireDate [@], Salary [N], Supervisor [I]

Child Fields	Parent's Key
Supervisor [I]	EmpNo [I]

Table: BIOLIFE.DB, COUNTRY.DB, CUSTOMER.DB, EMPL2.DB, EMPLOYEE.DB, ITEMS.DB, NEXTCUST.DB, NEXTITEM.DB, NEXTORD.DB, ORDERS.DB, PARTS.DB, VENDORS.DB

Save Referential Integrity As

Referential Integrity Name: EmpSuper

Update Rule: ○ Cascade [● Prohibit]

☒ Strict Referential Integrity

Figure 2: Using the Database Desktop to create EMPL2.

The table is now self-referencing. Note that the new Supervisor field must be saved before the referential integrity is added. This is because adding a self-referencing referential integrity name must be performed with no other modifications to the table.

Creating the Form

The Employee form can be built using the Database Form Expert. First, create a new project and remove the default form using the Project Manager. Then select **Help | Database Form Expert** and make these selections within the Expert:

- Create a simple form.
- Create a form using *TTable* objects.
- Table Name EMPL2.DB in DBDEMOS.
- Select all **Available Fields**.
- Vertical field layout.
- Top label placement.
- Generate a main form.

The Database Form Expert has created a new main form for the project, automatically adding and initializing the *TTable* and *TDataSource* components, as well as a *TDBEdit* component for each field in the table. Our example application, however, does not require the *TDBNavigator* component, so delete it from the form.

In addition, make these changes:

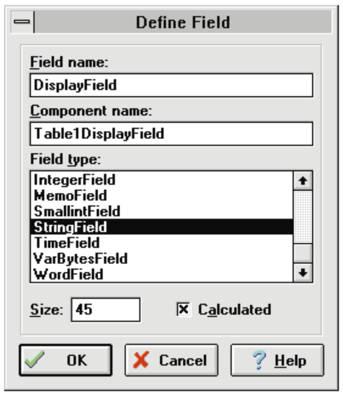
- Change the *TScrollBox* component's *Align* property to *alRight*.
- Drag the left border of the *TScrollBox* component to the right, leaving the *TScrollBox* just wide enough to view all the fields contained within (see Figure 1).
- Add a *TSpeedButton* component to *Panel1*, with the glyph DIRECTRY.BMP (located in \DELPHI\IMAGES\BUTTONS) assigned to its *Glyph* property. This speedbutton is used to execute the algorithm that loads the table data into the outline.
- Add a *TOutline* component to *Panel2*, and change its *Align* property to *alClient*.
- Add a *TTimer* component to *Panel1*. Set its *Enabled* property to *False*, and its *Interval* property to 50. The timer facilitates drag-and-drop, as we'll see later.
- Set the *Enabled* property of all *TDBEdit* fields to *False*.

When completed, the form should resemble the sample shown in Figure 1.

Prior to Calculations

The information shown in the sample application includes the last and first names of the employees. Rather than explicitly populating the outline with the string values of both the LastName and FirstName fields, we'll use a single calculated field generated by the Fields editor.

Access the Fields editor by double-clicking on the form's *TTable* component. Before defining the calculated field, add all fields by selecting **Add**, then **OK**. Pick **Define** to create the calculated field definition. Give the calculated field a Field name of `DisplayField`, a Field type of `StringField`, and a Size of `45` (see Figure 3).



Define Field

Field name: DisplayField

Component name: Table1DisplayField

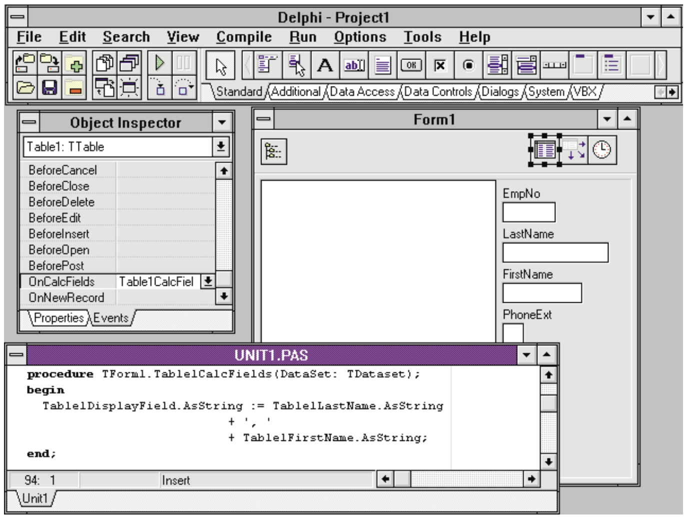
Field type: IntegerField, MemoField, SmallintField, [✓ StringField], TimeField, VarBytesField, WordField

Size: 45

☒ Calculated

Figure 3: Creating a calculated field.

Using *Table1*'s *OnCalcFields* event, compute the value of



```
procedure TForm1.Table1CalcFields(DataSet: TDataset);
begin
  Table1DisplayField.AsString := Table1LastName.AsString
                               + ', '
                               + Table1FirstName.AsString;
end;
```

Figure 4: The completed form design, showing the event handler for *OnCalcFields*.

the calculated field. To access the event's handler, select the *TTable* component, then select the Events page of the Object Inspector. Double-click the *OnCalcFields* event to create the handler and enter the code shown in Figure 4.

At this point, the shell of the sample application is complete. Now let's add the extras that make it easy to modify the application to access other self-referencing tables. The sample assigns the various field and index names to constant values, then references the constants throughout the code so the application can be made to access a different *TTable* by performing only three steps:

1) Modify the *OnCalcFields* event of *Table1* to create DisplayField from the new field(s).
2) Assign the new field/index names to the constants.
3) Change the form's *TDBEdit* components to access the new fields, adding/deleting *TDBEdit* components as necessary.

The constants are declared just before the **implementation** section as:

```
const
  { Index to sort the table by }
  SortIndexName  = 'ByName';
  { Field to display after SeekField }
  DisplayField   = 'DisplayField';
  { The key field of the self-reference }
  SeekField      = 'EmpNo';
  { The field referencing the key field }
  RecursiveField = 'Supervisor';
```

Although the sample application's source code accesses SeekField as a surrogate name to EmpNo, and RecursiveField as a surrogate to Supervisor, the article will continue to address these fields by their true names to ensure clarity.

The next task is to populate the *TOutline* component from the data in *Table1*.

Populating the Outline

The example application uses a nested algorithm to populate the outline. To keep the code modular and easy to maintain, the algorithm is divided among six procedures: the *OnClick* event handler of the *TSpeedButton* component, and user-defined *ParentLoad*, *ChildLoad*, *ChildReIterate*, *ProcessChildRecord*, and *AddNodeText*. Figure 5 illustrates how these procedures are nested.

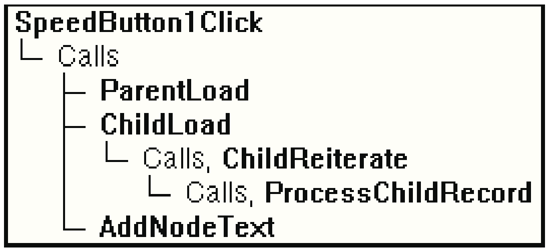


Figure 5: Sequence of procedural calls used to populate the outline.

The user-defined procedures are intended to be called only in sequence and only by the *OnClick* event of SpeedButton1. Therefore, declare them in the **private** section of UNIT1, as shown in the full source in Listing One on page 19.

Here is the methodology behind the algorithm:

- Use the *Add* method of *TOutline* to add a master node containing the text `Employees`. Load the EmpNo values of records in which the Supervisor field is blank into the outline using the *AddChild* method, as children of the master node Employees. These are the top-level parent records since they are not children of any other records.
- Load the EmpNo values of all remaining records (i.e. Supervisor is *not* blank) into the outline, using the *AddChild* method, as children of the outline node that corresponds to Supervisor. Reiterate through the table until these records have been added.
- Beginning at *Table1.First*, read each record, search the outline for the node representing the record using the *GetTextItem* method, and replace the node's *Text* property with a string consisting of EmpNo concatenated with DisplayField. This procedure simply adds the employee's name to the node, which formerly contained only the string equivalent of EmpNo.

The *OnClick* event of SpeedButton1 first calls the *ParentLoad* procedure that loads the key fields of all top-level parent records. Then *OnClick* calls the *ChildLoad* procedure, which calls *ChildReIterate* and *ProcessChildRecord* to load the key fields of all other records. Finally, *OnClick* calls the *AddNodeText* procedure that adds the text of calculated field DisplayField to the appropriate *TOutlineNode*.

Understanding the Algorithm

To understand the algorithm, it's important to know how the *TOutline* component assigns and maintains the indexes of its nodes. Each item in a *TOutline* is contained in a *TOutlineNode* object and has a unique *Index* number corresponding to its position in the outline. The first *TOutlineNode* has an *Index* of 1, and the last has an *Index* equal to the number of items in the

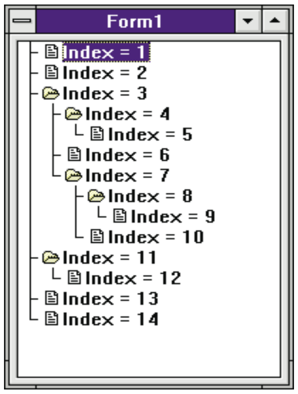


Figure 6: The *Indexes* of the *TOutlineNodes* are numbered sequentially from top to bottom.

outline. *Index* numbering progresses sequentially regardless of nested level (see Figure 6).

Throughout the algorithm, which *TOutlineNode* a child should be added to is indicated by these indexes. To determine the *Index* property of a node that is not currently selected, the sample application uses *TOutline*'s *GetTextItem* and *GetItem* methods. *GetTextItem* searches for a text string in the outline and returns the *Index* of the first node whose *Text* property is a match. *GetItem* returns the *Index* of the node at the X, Y coordinates of the mouse pointer.

In Listing One, notice that the *ParentLoad* procedure first creates a master parent node called Employees by calling *TOutline*'s *Add* method. As the first node in the outline, the master parent node has an *Index* of 1. All other valid employee records are nested underneath the Employees node to facilitate the movement of employee nodes via drag-and-drop. (By nesting all employees under the Employee's node, a child record can be dragged and dropped on the Employee's node to raise it to a top-level parent position. We'll discuss this more later.)

Once the master parent node is created, *ParentLoad* simply loads all top-level parent records (those that have a zero value in the Supervisor field) into the *TOutline* component using the *AddChild* method. The nodes are added as children to node *Index*1, the master parent node.

The next procedure, *ChildLoad*, drives the addition of all records that are not top-level parents (those that have a non-zero value in the Supervisor field) into the *TOutline* component. *ChildLoad* loops through the entire table repeatedly until there are no more records that must be added to the outline (see Figure 7).

An Iteration

ReIterate is a Boolean variable that, if *True*, indicates that there are still employee records to add to the outline. The value is changed in the *ProcessChildRecord* procedure if records are outstanding. If *ReIterate* is *True*, *ChildLoad* will continue to loop through *Table1* attempting to add the outstanding employee records.

It will probably take several iterations of the table before all records are successfully added to the outline. An employee record cannot be added to the outline if its parent has not yet been added. This can happen if the employee's parent record is the child of another record, and the employee's parent record appears after the employee record within the indexed table. If the inner **while not**

```
procedure TForm1.ChildLoad(Sender: TObject);
begin
  ReIterate := True;
  IterateWithoutAdd := False;
  while ReIterate do
    begin
      ReIterate   := False;
      ChildAdded  := False;
      while not Table1.eof do
        begin
          ChildReIterate;
          Table1.Next;
        end;
      Table1.First;
      if ReIterate and
         not ChildAdded then
        IterateWithoutAdd := True;
    end;
end;
```

Figure 7: The *ChildLoad* procedure adds all records with a non-zero value in the Supervisor field. The **while** loop steps through the table until there are no more records to add to the outline.

eof loop returns with a *ReIterate* of *True*, the outer **while** *ReIterate* loop will reprocess the table.

In the next iteration, the parent record of the employee that could not previously be included may (depending on the level of nesting) have been added, allowing the employee record to be attached as its child. This repeated looping causes the outline's population to become slower as the nesting becomes deeper and more numerous. However, the repeated looping does have an escape clause (which we'll discuss later).

Child's Play

The *ChildReIterate* procedure that is called by *ChildLoad* has a simple function. It examines all non-top-level records (`Supervisor <> 0`) and, if they do not exist in the outline, calls procedure *ProcessChildRecord*. It uses the *GetTextItem* method of *TOutline* to determine if the record has previously been added. For example, if *GetTextItem* returns an *Index* of 0, a *TOutlineNode* containing the text does not exist, and the record should be processed by *ProcessChildRecord*.

ProcessChildRecord must determine if the parent record of the employee exists in the outline (see Figure 8). *GetTextItem* searches for and returns the *Index* of a node containing the string equivalent of the record's Supervisor field. If the *Index* returned is not zero, it adds the employee, using the *AddChild* method, to the parent record's *TOutlineNode* via the returned *Index*. If the *Index* returned is 0, it assigns a value of *True* to *ReIterate*.

Within *ProcessChildRecord* is the escape clause to the repeating loop — if the previous loop added no new records to the outline (i.e. Boolean variable, *IterateWithoutAdd*, is *True*), the current loop will add any remaining records as children of node *Index* 0, indicating the parent records could not be found. Orphaned records will never occur if the self-referencing referential integrity name was added to

```
procedure TForm1.ProcessChildRecord;
var
  NodeIndex: integer;
begin
  with Table1 do
    begin
      NodeIndex := Outline1.GetTextItem(
        FieldByName(RecursiveField).AsString);
      if NodeIndex = 0 then
        begin
          ReIterate := True;
          if IterateWithoutAdd = True then
            begin
              Outline1.AddChild(0,
                FieldByName(SeekField).AsString);
            end;
        end
      else
        begin
          ChildAdded := True;
          Outline1.AddChild(NodeIndex,
            FieldByName(SeekField).AsString);
        end;
    end;
end;
```

Figure 8: The *ProcessChildRecord* procedure determines if the parent record is in the outline.

the Paradox table as instructed earlier. The code is added simply as a safety precaution. For instance, if the application is modified to access a table with no recursive relation defined (or a table unable to accept or enforce one), the handling of orphaned records will prevent the algorithm from entering an endless loop.

One More Loop

As a final step, *AddNodeText* loops once more through the table, locating the *TOutlineNode* that corresponds to each record, and replacing the node's text:

```
(EmpNo) with EmpNo + ' ' + DisplayField
```

This step can't be performed as the nodes are added because of the way the *GetTextItem* method operates. The text of the node must exactly match the text being searched to find a match.

If nodes were added with:

```
EmpNo + ' ' + DisplayField
```

then before seeking a node corresponding to the Supervisor field of the current record, the record pointer would have to be moved to the record whose EmpNo field contained the value of Supervisor. Here, the DisplayField value would have to be retrieved for use in the search. This would create at least one, and usually several, *SetBookMark*/*FindKey*/*GotoBookMark* combinations for each child record — a much greater performance hit than one loop through the table.

Drag and Drop

With the *TOutline* populated, the task remains to enable drag-and-drop of the *TOutlineNodes*. To accomplish this, use the *OnMouseMove*, *OnDragOver*, and *OnDragDrop* events of the *TOutline* component. Note that *OnMouseMove* triggers the dragging state rather than the *OnMouseDown* event that is generally used in Borland's drag-and-drop examples. The reason for this will become clear.

To enable drag-and-drop, enter this code into the *MouseMove* event handler:

```
with Outline1 do
  if not Dragging then
    if (Shift = [ssLeft]) then
      BeginDrag(False);
```

Running the application with this code, however, highlights a problem: a node can only be dragged and dropped onto another node that is currently visible in the outline. It can't be dropped on a node that has scrolled off the viewing area. If drag mode is *not* enabled, dragging from the outline to an area above the control makes the outline scroll downward. Likewise, dragging to an area below the control scrolls upward. However, after drag mode is enabled, this feature of the *TOutline* control is disabled.

This problem is solved by using the *OnMouseMove* event to trigger *BeginDrag*. Code can be added to the *MouseMove* event handler that will note, if the outline is currently dragging, whether the mouse pointer has moved beyond the outline's borders. If so, the outline will be scrolled in the appropriate direction. However, the *OnMouseMove* event shouldn't actually cause the scrolling, or scrolling will stop when the mouse ceases moving. Instead, the *OnMouseMove* event should enable the *TTimer* component, and *TTimer*'s *OnTimer* event will handle the scrolling. The completed *MouseMove* handler is shown in Figure 9.

The X and Y coordinates of the mouse pointer are compared to the *TOutline* top and left coordinates (0) and height to determine if the *TOutline* should be scrolled and, if so, in which direction. The Boolean variables *ScrollUp*, *ScrollDown*, *ScrollLeft*, and *ScrollRight* have their values set depending on the scrolling direction. These variables instruct the *TTimer* component after it is enabled. Note that *TTimer* is disabled if the outline should not be scrolled. This stops *TTimer* from continuing to scroll after the mouse has been moved back within the borders of the *TOutline*.

The *OnTimer* event of *TTimer* evaluates the current dragging state and the Boolean variables (*ScrollUp*, *ScrollDown*, *ScrollLeft*, and *ScrollRight*), and it sends the appropriate message to the Outline control's window. If *TOutline* is not in a dragging state, *TTimer* is disabled. This ensures that the timer is stopped if the left button is released while the mouse is still outside the boundaries of the *TOutline* (see Figure 10).

WM_VSCROLL and WM_HSCROLL are the messages sent to a window when its vertical and horizontal scroll

```
with Outline1 do
  begin
    if not Dragging then
      begin
        if (Shift = [ssLeft]) then
          BeginDrag(False);
      end
    else
      begin
        if (Y < 0) or (Y > Height) or
           (X < 0) or (X > Height) then
          begin
            ScrollUp    := False;
            ScrollDown  := False;
            ScrollLeft  := False;
            ScrollRight := False;
            if Y < 0 then
              ScrollUp := True
            else if Y > Height then
              ScrollDown := True;
            if X < 0 then
              ScrollLeft := True
            else if X > Height then
              ScrollRight :=True;
            Timer1.Enabled := True;
          end
        else
          Timer1.Enabled := False;
      end;
end;
```

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  with Outline1 do
    begin
      if Dragging then
        begin
          if ScrollUp then
            SendMessage(Handle, WM_VSCROLL, SB_LINEUP,
                        SB_THUMBTRACK)
          else if ScrollDown then
            SendMessage(Handle, WM_VSCROLL, SB_LINEDOWN,
                        SB_THUMBTRACK);
          if ScrollLeft then
            SendMessage(Handle, WM_HSCROLL, SB_LINEUP,
                        SB_THUMBTRACK)
          else if ScrollRight then
            SendMessage(Handle, WM_HSCROLL, SB_LINEDOWN,
                        SB_THUMBTRACK);
        end
      else
        Timer1.Enabled := False;
    end;
end;
```

Figure 9 (Top): The completed *MouseMove* handler.
Figure 10 (Bottom): Setting boundaries with the *Timer* procedure.

bars (respectively) are clicked. SB_LINEUP and SB_LINEDOWN indicate the direction of scroll (with UP as left and DOWN as right on the horizontal bar). SB_THUMBTRACK indicates that the scroll box should move to the corresponding position. When many records with extensive scrolling are involved, it might be worthwhile to expand the evaluation of mouse movements to include SB_PAGEUP and SB_PAGEDOWN messages if the mouse moves above 0 - 30, or below Height + 30.

Once dragging and scrolling are functional, the *DragOver* handler must be created. All this handler must do is check that the source of the *OnDragOver* event is the *TOutline* control and, if so, accept a drop if it occurs.

The *OnDragDrop* event must trigger the move of the dragged node to be a child of the dropped-on node (moving all its children with it). In addition, the *OnDragDrop* event must update the dragged node's corresponding record in *Table1* to reflect the new Supervisor. This is accomplished in four steps:

1) Extract the EmpNo values from the dragged outline node and the dropped-on outline node.
2) Move the *TTable* record pointer to the dragged outline node's employee number using *TTable's FindKey* method.
3) Call *TTable*'s *Edit* method, move the dropped-on outline node's employee number into the record's Supervisor field, and call *TTable*'s *Post* method.
4) Call *TOutline*'s *MoveTo* method to move the dragged node and its children to the dropped-on node.

The handler checks to ensure that a parent is not being dragged to one of its own children before accepting the drop. This prevents the creation of a cyclical reference (i.e. employee 102 is the supervisor of employee 54, and employee 54 is the supervisor of employee 102). If this check were not put in place, and a cyclical reference was created, the node would not actually move until the outline was cleared and reloaded. During reload, the records would be added as orphaned records.

Notice that, if the dragged node is dropped on the master parent Employees, the extracted EmpNo value is blank. Hence the purpose of the master parent — drop a node on it and the node becomes a top-level parent.

Keeping the Table Synchronized

To keep the contents of the *TDBEdit* fields synchronized with the currently selected node on the outline, the record pointer of the *TTable* must be moved whenever the currently selected node changes. This is best accomplished through the *OnClick* event of the outline. The key field value is extracted from the outline node, and the *FindKey* method of *Table1* is called to reposition the record pointer.

To complement record synchronization, there are lines throughout the code that hide/show the *ScrollBox* (thereby hiding/showing the *TDBEdit* fields grouped within), depending on whether the EmpNo extracted from the currently selected *TOutlineNode* is found in the table. The scrollbox is hidden if the record cannot be found. This should only occur when the master node Employees is selected or, in a multi-user environment, if the record was deleted and the DataSet refreshed.

Conclusion

TOutline organizes layers of hierarchical data with ease — that's its power. With a simple algorithm, the outline can be loaded from a self-referencing database — opening the door to drag-and-drop manipulation of the data's hierarchy. Self-referencing tables are by no means the only relation type where *TOutline* can be used; one-to-many and one-to-many-to-many relations may also lend themselves to hierarchical manipulation. Of course, a completely new algorithm would need to be designed. ∆

The demonstration projects referenced in this article are available on the Delphi Informant Delphi Informant Works CD located in INFORM\96\FEB\DI9602JC.

Jeff Chant is the owner of Maelstrom Software, an Ontario-based company specializing in the design and construction of Delphi tools and applications. He is also a SYNON architect, working with large warehouse management systems on the AS/400. He can be reached on CompuServe at 71431,62.

Begin Listing One — Unit1

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    ScrollBox: TScrollBox;
    Label1: TLabel;
    EditEmpNo: TDBEdit;
    Label2: TLabel;
    EditLastName: TDBEdit;
    Label3: TLabel;
    EditFirstName: TDBEdit;
    Label4: TLabel;
    EditPhoneExt: TDBEdit;
    Label5: TLabel;
    EditHireDate: TDBEdit;
    Label6: TLabel;
    EditSalary: TDBEdit;
    Label7: TLabel;
    EditSupervisor: TDBEdit;
    Panel1: TPanel;
    DataSource1: TDataSource;
    Panel2: TPanel;
    Table1: TTable;
    SpeedButton1: TSpeedButton;
    Outline1: TOutline;
    Table1EmpNo: TIntegerField;
    Table1LastName: TStringField;
    Table1FirstName: TStringField;
    Table1PhoneExt: TStringField;
    Table1HireDate: TDateTimeField;
    Table1Salary: TFloatField;
    Table1Supervisor: TIntegerField;
    Table1DisplayField: TStringField;
    Timer1: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure Table1CalcFields(DataSet: TDataset);
    procedure SpeedButton1Click(Sender: TObject);
    procedure Outline1MouseMove(Sender: TObject;
                                 Shift: TShiftState;
                                 X, Y: Integer);
    procedure Outline1DragOver(Sender, Source: TObject;
                                X, Y: Integer;
                                State: TDragState;
                                var Accept: Boolean);
    procedure Outline1DragDrop(Sender, Source: TObject;
                                X, Y: Integer);
    procedure Timer1Timer(Sender: TObject);
    procedure Outline1Click(Sender: TObject);
  private
      { Private declarations }
      procedure ParentLoad;
      procedure ChildLoad;
      procedure ChildReiterate;
      procedure ProcessChildRecord;
      procedure AddNodeText;
    public
      { Public declarations }
    end;

var
  Form1: TForm1;
  ReIterate, IterateWithoutAdd,
  ChildAdded, NodeDrag, ScrollUp,
  ScrollDown, ScrollLeft,
  ScrollRight: Boolean;
const

  { Index to sort the table by }
  SortIndexName = 'ByName';
  { Field to display after SeekField }
  DisplayField   = 'DisplayField';
  { The key field of the self-reference }
  SeekField             = 'EmpNo';
  { The field referencing the key field }
  RecursiveField= 'Supervisor';

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Table1.Open;
end;

procedure TForm1.Table1CalcFields(DataSet: TDataset);
begin
  Table1DisplayField.AsString :=
    Table1LastName.AsString + ', ' +
    Table1FirstName.AsString;
end;

procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  { Hide the scrollbox and its TDBEdit fields }
  ScrollBox.Visible      := False;
  with Table1 do
    begin
      Screen.Cursor      := crHourglass;
      try
        Form1.ParentLoad;
        Form1.ChildLoad;
        Form1.AddNodeText;
      finally
        Screen.Cursor := crDefault;
      end;
    end;
end;

procedure TForm1.ParentLoad;
begin
  with Table1 do
    begin
      Outline1.Clear;
      { Add a Master Parent for all valid employees }
      Outline1.Add(0, 'Employees');
      { Index the table for outline display order }
      IndexName := SortIndexName;
      First;
      while not eof do
        begin

          { If the record is top-level (has no parent) }
          if FieldByName(RecursiveField).AsInteger = 0 then
```

```
            { Add the record as a child of the Master
              Parent created above (index=1) }
            Outline1.AddChild(1,
              FieldByName(SeekField).AsString);
            Next;
        end;

      { Set the index back to the primary key }
      IndexFieldNames := SeekField;
      First;
    end;
end;

procedure TForm1.ChildLoad;
begin
  ReIterate := True;
  IterateWithoutAdd := False;

{ While there are still child records that have not
  been added to a parent }
  while ReIterate do
    begin
      ReIterate         := False;
      ChildAdded        := False;
      while not Table1.eof do
        begin;
          Form1.ChildReIterate;
          Table1.Next;
        end;
      Table1.First;

      { If an iteration through the table was performed,
        and there were record(s) not yet added to the
        outline, but no records were added to the outline
        throughout the iteration }
      if ReIterate and
         not ChildAdded then
        IterateWithoutAdd := True;
    end;
end;

procedure TForm1.ChildReIterate;
var
  NodeIndex: integer;
begin
  with Table1 do
    begin
      { If record has parent (i.e. Supervisor not 0) }
      if FieldByName(RecursiveField).AsInteger <> 0 then
        begin
          NodeIndex := Outline1.GetTextItem(FieldByName
                           (SeekField).AsString);

          { If record was not added to the outline in a
            previous iteration (can't be found in outline) }

          if NodeIndex = 0 then
            Form1.ProcessChildRecord;
        end;
    end;
end;

procedure TForm1.ProcessChildRecord;
var
  NodeIndex: integer;
begin
  with Table1 do
    begin
      NodeIndex := Outline1.GetTextItem
                  (FieldByName(RecursiveField).AsString);

      { If node representing parent record does
        not exist in the outline }
      if NodeIndex = 0 then
        begin
            { Indicate that a reiteration will be
              necessary }
            ReIterate := True;
            { If this is the second iteration through the
              file with no new added child nodes, add the
              record as an orphan }

            if IterateWithoutAdd = True then
              begin
                Outline1.AddChild(0,FieldByName
                                (SeekField).AsString);
              end;
          end

      else
          { If node representing parent record does exist,
            add the record as a child }
          begin
            ChildAdded := True;
            Outline1.AddChild(NodeIndex,FieldByName
                            (SeekField).AsString);
          end;
    end;
end;

procedure TForm1.AddNodeText;
var
  NodeIndex: integer;
begin
  with Table1 do
    begin
      First;
      while not eof do
        begin
          NodeIndex := Outline1.GetTextItem(FieldByName
                           (SeekField).AsString);
          if NodeIndex <> 0 then
            Outline1[NodeIndex].Text :=
              FieldByName(SeekField).AsString + ' ' +
              FieldByName(DisplayField).AsString;
          Next;
        end;
    end;
end;

procedure TForm1.Outline1MouseMove(Sender: TObject;
                                Shift: TShiftState;
                                X, Y: Integer);
begin
  with Outline1 do
    begin
      if not Dragging then
        begin
          if (Shift = [ssLeft]) then
            BeginDrag(False);
        end
      else
        begin
          { If the mouse has been dragged outside the
            TOutline component }
          if (Y < 0) or (Y > Height) or
             (X < 0) or (X > Height) then
            begin
              ScrollUp := False;
              ScrollDown := False;
              ScrollLeft := False;
              ScrollRight := False;
              if Y < 0 then
                ScrollUp := True
              else if Y > Height then
                ScrollDown := True;
              if X < 0 then
                ScrollLeft := True
              else if X > Height then
                ScrollRight :=True;
              Timer1.Enabled := True;
            end

          else
            Timer1.Enabled := False;
        end;
    end;
end;

procedure TForm1.Outline1DragOver(Sender, Source: TObject;
                                  X, Y: Integer;
                                  State: TDragState;
                                  var Accept: Boolean);
begin
  if Source is TOutline then
    Accept := True;
end;

procedure TForm1.Outline1DragDrop(Sender, Source: TObject;
                                  X, Y: Integer);
var
  FieldFrom, FieldTo,
  NodeFrom, NodeTo,
  NodeText:         string;
  IndexFrom, IndexTo,
  ParentIndex:      integer;
begin
  if (Source is TOutline) and
     (Outline1.GetItem(X, Y) <> Outline1.SelectedItem) then
   begin
     IndexFrom := Outline1[Outline1.SelectedItem].Index;
     NodeFrom := Outline1[Outline1.SelectedItem].Text;

     { Extract the key field from the fromnode }
     FieldFrom := Copy(NodeFrom, 1, Pos(' ', NodeFrom)-1);
     IndexTo := Outline1[Outline1.GetItem(X,Y)].Index;
     NodeTo := Outline1[Outline1.GetItem(X,Y)].Text;

     { Extract the key field from the tonode }
     FieldTo := Copy(NodeTo, 1, Pos(' ', NodeTo)-1);

     { Verify that FromNode is not an ancestor of
       ToNode before moving }
     ParentIndex := Outline1[IndexTo].Parent.Index;
     while (ParentIndex <> 0) and
           (ParentIndex <> IndexFrom) do
       ParentIndex := Outline1[ParentIndex].Parent.Index;

     { If FromNode is a direct ancestor of ToNode,
       send error message }
     if ParentIndex = IndexFrom then
       MessageDlg('Sorry, you have attempted to create' +
                  'a cyclical relation.',mterror,[mbOK],0)
     else
       begin
       if MessageDlg('Make [' + NodeFrom + '] a child of [' +
                     NodeTo + ']?',mtConfirmation,
                     mbOkCancel,0) = mrOk then
         begin
           { Indicate that a NodeDrag move will be
             performed--this will prevent OnClick from
             triggering an Index Out of Bounds exception }
           NodeDrag := True;
           with Table1 do
             begin
               if FindKey([Fieldfrom]) then
                 begin
                   Edit;
                   FieldByName(RecursiveField).AsString
                               := Fieldto;
                   Post;
                   Outline1[Outline1.SelectedItem].MoveTo
                      (Outline1.GetItem(X,Y),oaAddChild);
                 end;
             end;
         end;
```

```
          { Move the record pointer to the currently
            selected node to update the TDBEdit fields }
          if Outline1.ItemCount >= 1 then
            begin
              NodeText :=
                Outline1[Outline1.SelectedItem].Text;
              NodeText :=
                Copy(NodeText, 1, Pos(' ', NodeText)-1);
              if Table1.FindKey([NodeText]) then
                ScrollBox.Visible := True
              else
                ScrollBox.Visible := False;
            end;
        end;
    end;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  with Outline1 do
    begin
      if Dragging then
        begin

          if ScrollUp then
            SendMessage(Handle, WM_VSCROLL, SB_LINEUP,
                        SB_THUMBTRACK)
          else if ScrollDown then
            SendMessage(Handle, WM_VSCROLL, SB_LINEDOWN,
                        SB_THUMBTRACK);
          if ScrollLeft then
            SendMessage(Handle, WM_HSCROLL, SB_LINEUP,
                        SB_THUMBTRACK)

          else if ScrollRight then
            SendMessage(Handle, WM_HSCROLL, SB_LINEDOWN,
                        SB_THUMBTRACK);
        end
      else

        Timer1.Enabled := False;
    end;
end;

procedure TForm1.Outline1Click(Sender: TObject);
var
  NodeText: string;
begin
  { Only perform if a node has not just been dragged
    to a new location }
  if not NodeDrag then
    begin
      if Outline1.ItemCount >= 1 then
        begin
         NodeText := Outline1[Outline1.SelectedItem].Text;
         NodeText := Copy(NodeText,1,Pos(' ',NodeText)-1);
          if Table1.FindKey([NodeText]) then
            ScrollBox.Visible := True
          else
            ScrollBox.Visible := False;
        end;
    end
  else
    NodeDrag := False;
end;

end.
```

End Listing One

OP TECH

DELPHI 1.0 / OBJECT PASCAL

By *Bill Todd*

Error Handling: Part I

A Primer for Handling Object Pascal Exceptions

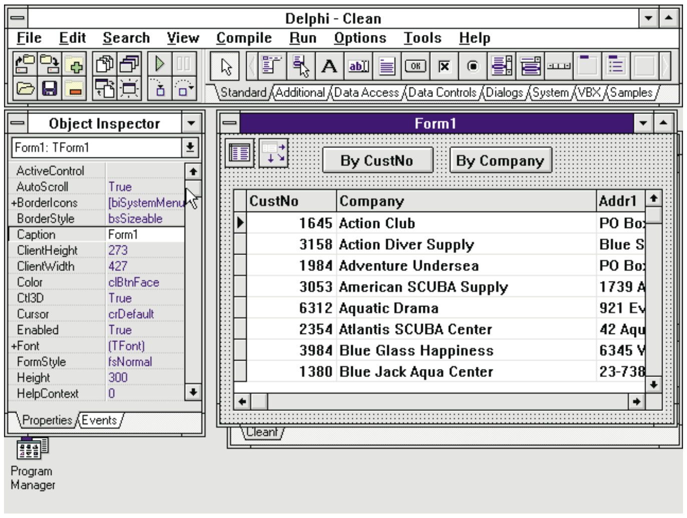
Handling run-time errors is a serious concern regardless of the language or environment in which you develop applications. Delphi is no different. It handles errors through an object called an exception. When a run-time error occurs, Delphi raises (creates) an exception. If your code does not handle it, an OnException event is raised.

In this two-part series, you'll learn how to handle exceptions, ensure that cleanup code is always executed, and create custom exceptions and exception handlers.

Handling Cleanup and Resource Allocations

Cleanup code is easy to implement in Delphi. Cleanup code is code that must execute to restore the system to a safe, stable state even if a run-time error occurs. For example, disabling an application's data-aware controls allows you to open and close a Table or Query component, or change a Table component's active index without annoying screen flicker.

To handle cleanup chores, Delphi provides the **try..finally** construct. The easiest way to understand it is to examine the code from the sample project, CLEAN.DPR (its form is shown in Figure 1). The form features Table, DataSource, and DBGrid components. The Table component is connected to the Customer table in the DBDEMOS database. It also contains two Buttons that let you change the active index from CustNo (the primary index) to ByCompany (the secondary index). This *OnClick* event handler switches the indexes:



ActiveControl	
AutoScroll	True
+BorderIcons	[biSystemMenu
BorderStyle	bsSizeable
Caption	Form1
ClientHeight	273
ClientWidth	427
Color	clBtnFace
Ctl3D	True
Cursor	crDefault
Enabled	True
+Font	(TFont)
FormStyle	fsNormal
Height	300
HelpContext	0

CustNo	Company	Addr1
1645	Action Club	PO Bo:
3158	Action Diver Supply	Blue S
1984	Adventure Undersea	PO Bo:
3053	American SCUBA Supply	1739 A
6312	Aquatic Drama	921 Ev
2354	Atlantis SCUBA Center	42 Aqu
3984	Blue Glass Happiness	6345 V
1380	Blue Jack Aqua Center	23-738

Figure 1: A form that changes the active index.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  with Table1 do
    begin
      DisableControls;
      IndexName := 'x';
      EnableControls;
    end;
end;
```

Unfortunately, there's a problem with this code. If the statement that changes the *IndexName* property fails because the index does not exist, an exception is automatically generated and the procedure's execution terminates. This means the *EnableControls* call will not execute. The result is that you will be in the form and the DBGrid will no longer be connected to the Table. Try it by running the NOCLN.EXE program and click the **By Company** button. Try to move around in the grid. Obviously, you don't want to leave the user in this condition.

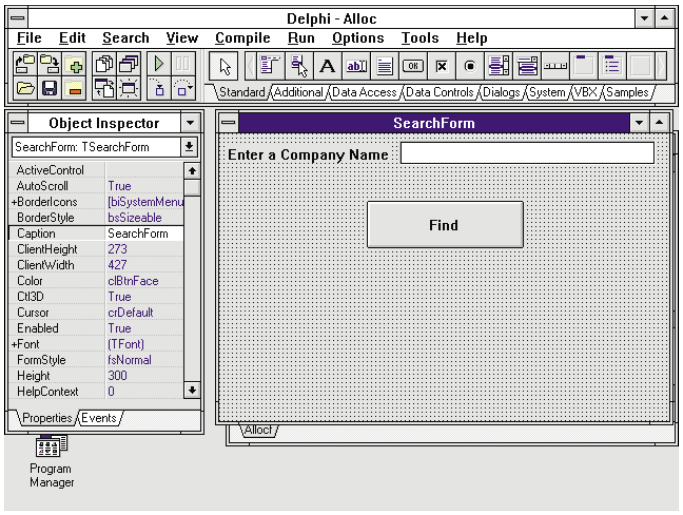
Fortunately, Delphi's solution to this problem is elegant because it allows you to guarantee that the cleanup code will execute in spite of any run-time errors that occur. For example, look at the *OnClick* handler of the **By Company** button in the CLEAN.DPR project:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  with Table1 do
    begin
      DisableControls;
      try
        IndexName := 'ByCompany';
      finally
        EnableControls;
      end;
    end;
end;
```

The code following the *DisableControls* method call is enclosed in a **try..finally** block. Code in the **finally** block executes if an exception occurs in any part between **try** and **finally**. Therefore, the *EnableControls* call will execute even if the change to the *IndexName* property fails. You can see this by changing *IndexName* to `X` and running this code. After clicking the **By Company** button and clearing the error dialog box, you can still move around in the grid.

In addition, it's critical that code executes when resources are allocated and you must ensure they are released. Memory, files, and Windows resources must be recovered if an error occurs. In an object-oriented environment, this is best illustrated by creating an instance of an object in your code.

Figure 2 shows the form from the sample project ALLOC.DPR. After a company name is entered, the form searches the customer table and reports if a match was found. The **Find** button's *OnClick* event handler declares a variable named *CustTbl* of type *TTable* that searches the customer table (see Figure 3). However, *CustTbl* must first be created by calling its constructor method, *Create*.



SearchForm: TSearchForm

ActiveControl	
AutoScroll	True
+BorderIcons	[biSystemMenu
BorderStyle	bsSizeable
Caption	SearchForm
ClientHeight	273
ClientWidth	427
Color	clBtnFace
Ctl3D	True
Cursor	crDefault
Enabled	True
+Font	(TFont)
FormStyle	fsNormal
Height	300
HelpContext	0

Figure 2: Dynamic object creation.

```
procedure TSearchForm.FindBtnClick(Sender: TObject);
var
  CustTbl: TTable;
begin
  { Create a TTable object. }
  CustTbl := TTable.Create(SearchForm);
  { Make sure the TTable's destructor gets called
    no matter what happens in the following code. }
  try
    with CustTbl do
    begin
      { Assign the DatabaseName and TableName. }
      DatabaseName := 'DBDEMOS';
      TableName    := 'customer.db';
      { Open the table. }
      Open;
      { Search for the requested name. }
      while EOF = False and
            ToFind.Text<>FieldByName('Company').AsString do
        Next;
      { Display a message. }
      if ToFind.Text = FieldByName('Company').AsString then
        MessageDlg('Found',mtInformation,[mbOK],0)
      else
        MessageDlg('Not Found',mtInformation,[mbOK],0);
    end;
  finally
    { Call the TTable's destructor to free its memory. }
    CustTbl.Free;
  end;
end;
```

Figure 3: The code from the **Find** button's *OnClick* handler.

Here's the important issue: Whenever you create a new instance of an object by calling its constructor, you are allocating memory. Therefore, you are responsible for deallocating that memory by calling the object's destructor method. Otherwise, memory will not be available until you restart Windows.

To ensure that *CustTbl*'s *Free* method is called, all the code in this procedure (following the call to *TTable's Create* method) is enclosed in a **try..finally** block. The call to the destructor, *Free*, follows the **finally** keyword so that it will execute even if an exception is raised by any of the code between **try** and **finally**. (Note that **try.. finally** blocks can be nested to any depth.)

The code from the FASTE.DPR project is an example of protecting both file and memory resources. Figure 4 shows the form for this project and Figure 5 is the **Create** button's *OnClick* event handler.

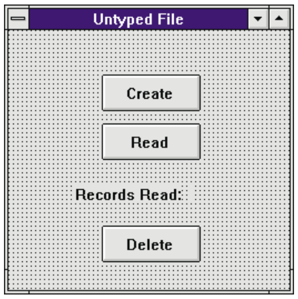


Figure 4: The FASTE project's main form.

This procedure opens a new file by calling the run-time library (RTL) procedure, *Rewrite*. Then after the file has been opened, the code ensures that the file is closed by encasing the subsequent statements in a **try..finally** block. If an error occurs, the call is made to this statement in the **finally** block:

```
System.Close(addrFile)
```

```
procedure TForm1.CreateBtnClick(Sender: TObject);
const
  MaxRecs = 100;
var
  buff: array[1..MaxRecs] of TAddress;
  addrFile: File;
  i, count: Word;
begin
  AssignFile(addrFile,'addr.dat');
  Rewrite(addrFile,SizeOf(TAddress));
  try
    { Put 100 records into the buffer. }
    for i := 1 to MaxRecs do
      with buff[i] do
      begin
        PasToArray('John Doe',name);
        PasToArray('123 East Main Street',addr);
        PasToArray('New York',city);
        PasToArray('NY',state);
        PasToArray('55555-5555',zip);
      end;
      { Write 100 buffers (10,000 records). }
      for i := 1 to 100 do
        BlockWrite(addrFile,buff,MaxRecs,count);
    finally
      System.Close(addrFile);
    end;
end;
```

```
procedure TForm1.ReadBtnClick(Sender: TObject);
const
  MaxRecs = 500;
type
  Tbuff = array[1..MaxRecs] of TAddress;
var
  buff:      ^Tbuff;
  addrFile:  File;
  total,
  count:     Word;
begin
  New(buff);
  try
    AssignFile(addrFile,'addr.dat');
    Reset(addrFile,SizeOf(TAddress));
    try
      { Read the file, MaxRecs records at a time. }
      total := 0;
      repeat
        count := 0;
        BlockRead(addrFile,buff^,MaxRecs,count);
        total := total + count;
      until count = 0;
    finally
      System.Close(addrFile);
    end;
    ReadCount.Caption := IntToStr(total);
  finally
    Dispose(buff);
  end;
end;
```

Figure 5 (Top): The code attached to the **Create** button.
Figure 6 (Bottom): Code from the **Read** button.

The **Read** button's *OnClick* event handler (see Figure 6) has two resource allocations that must be protected and uses nested **try..finally** blocks.

Memory is the first resource that is allocated when *New* is called:

```
New(buff);
```

This assigns the buffer array into which the records are read. Then, the **try** keyword begins a block that ends with the call to *Dispose* after the **finally** keyword:

```
Dispose(buff)
```

Within this outer **try..finally** block, the call to *Reset* opens a file:

```
Reset(addrFile, SizeOf(TAddress));
```

Next, the **try** keyword initiates a **try..finally** block that protects the file resource by ensuring the following command is called in the **finally** section:

```
System.Close(addrFile);
```

If a run-time error occurs in this code, neither global heap memory nor file handles will be lost.

Trapping RTL Exceptions

Calling RTL procedures and functions can generate exceptions. RTL exceptions fall into one of these categories:

- Conversion
- Floating point math
- Hardware
- Heap (memory allocation)
- Input/Output
- Integer math
- Typecast

Before delving into the details of RTL exception handling, look at the sample INTERR.DPR project for an overview of the process. Its form has a Label, Panel, and three Buttons (see Figure 7). Here is the **MathError** button's *OnClick* event handler:

```
procedure TForm1.MathErrorClick(Sender: TObject);
var
  i,j,k,l,m: Integer;
begin

  i := 23;
  j := 0;
  l := 2;
  m := 4;

  { Make a calculation. }
  k := i div j * (l div m);
  { Display the result. }
  Result.Caption := IntToStr(k);

end;
```

As you can see, the code declares five integer variables and assigns values to all of them except k. When this expression is evaluated:

```
k := i div j * (l div m);
```

an RTL exception is raised because the value of j is zero and division by zero is not allowed. In this case, Delphi's default exception handler will manage the error by displaying an error message in a dialog box and ending execution of this procedure.

The **MathErrorHandled** button's *OnClick* handler (see Figure 8) will not display the normal error dialog box that appears when an exception is raised. In addition, your code does not

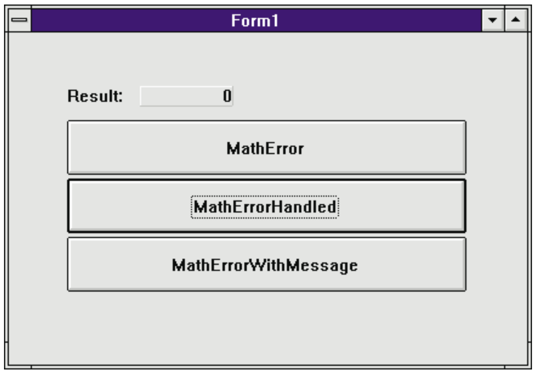


Figure 7: The INTERR project's main form.

```
procedure TForm1.MathErrorClick(Sender: TObject);
var
  i,j,k,l,m: Integer;
begin

  i := 23;
  j := 0;
  l := 2;
  m := 4;

  try
    k := i div j * (l div m);
  except
    on EIntError do k := 0;
  end;

  { Display the result. }
  Result.Caption := IntToStr(k);
end;
```

Figure 8: The *OnClick* handler for the **MathErrorHandled** button.

Exception	Description
EDivByZero	An attempt to divide by zero.
ERangeError	The number or expression result is beyond the range of the integer type.
EIntOverflow	A mathematical operation caused an integer overflow.

Exception	Description
EInvalidOp	The processor encountered an invalid instruction. This usually means the processor is trying to execute data due to a pointer error.
EZeroDivide	Attempt to divide by zero.
EOverflow	A floating point operation overflowed.
EUnderflow	A floating point operation underflowed.

Exception	Description
EFault	The base exception object for all faults.
EGPFault	General protection fault. The most common cause of GPFs is an uninitialized pointer.
EStackFault	Illegal access to the stack segment.
EPageFault	The Windows memory manager could not access the swap file.
EInvalidOpCode	The processor encountered an undefined instruction. This is usually caused by trying to execute data.
EBreakpoint	The program generated a breakpoint interrupt.
ESingleStep	The program generated a single step interrupt.

Figure 9 (Top): Integer math exceptions. **Figure 10 (Middle):** Floating point math exceptions. **Figure 11 (Bottom):** Hardware exceptions.

handle it because the calculation is enclosed in a **try..except** block. The **except** code sets the result to zero and that's the end of the exception. Clearly, this is much easier than handling the possibility of division by zero without **try..except**. To detect if j or m is zero would require:

```
if (j = 0) or (m = 0) then
  k := 0
else
  k := i div j * (l div m);
```

If the computation were more complex, testing for all possible integer math errors requires a lot of code. Note that when you run a program in the IDE, it runs under the control of the Interactive Debugger. You will always see its exception dialog box whether your code handles the exception or not. To view what the user will see, run the application from Program Manager.

When an exception is raised in a **try..except** block, Delphi checks if it's listed in the **except** section. If so, then the code for that exception is executed. In this example, if any integer math exception occurs then the value of k is set to zero.

Understanding the RTL Exception Hierarchy

Before continuing, you must understand how the seven classes of RTL exceptions are organized. Three of the categories — integer math, floating point math, and hardware exceptions — have a hierarchy:

- *EIntError* is a generic integer math exception. You can test for it, or for the specific integer math errors shown in Figure 9.
- For floating point math operations, the generic exception is *EMathError*. The individual floating point errors are listed in Figure 10.
- The generic hardware exception is *EProcessorException* and its specific descendants are listed in Figure 11.

With the exception of *EGPFault*, you should never encounter or need to worry about any of the hardware exceptions. They only occur if a serious hardware or operating system failure happens or if you are running under a debugger. The remaining four categories of exception do not have a generic exception. You must test for each specific exception that you want to handle:

- For input/output errors, there is a single exception, *EInOutError*. It has a field named *ErrorCode* containing the operating system error code for the error that occurred.
- *EInvalidCast* is a single typecast exception that occurs anytime you attempt a typecast using the AS operator and the typecast fails.
- All conversion exceptions raise the *EConvertError* exception. For example, if you call *StrToInt* and the string cannot be converted, then *EConvertError* is raised.
- Two heap exceptions can occur when using dynamic memory. They are listed in Figure 12.

Exception	Description
EOutOfMemory	An attempt to allocate memory on the heap failed.
EInvalidPointer	An attempt was made to dispose of a pointer that points to an address outside of the heap.

```
procedure TForm1.MathErrorWithMessageClick(Sender:
                                             TObject);
var
  i,j,k,l,m: Integer;
begin
  i := 23;
  j := 0;
  l := 2;
  m := 4;
  try
    k := i div j * (l div m);
  except
    on EDivByZero do begin
      k := 0;
      MessageDlg('Divide by zero error',mtError,[mbOK],0);
      end;
    on EIntError do begin
      k := 0;
      MessageDlg('Integer math error.',mtError,[mbOK],0);
      end;
  end;
  { Display the result. }
  Result.Caption := IntToStr(k);
end;
```

Figure 12 (Top): Heap exceptions. **Figure 13 (Bottom):** The *OnClick* event handler for the **MathErrorWithMessage** button.

Handling Multiple Exceptions

The third button on the INTERR.DPR form, MathErrorWithMessage, shows how to handle multiple exceptions in a single **try..except** block (see Figure 13).

If you look at the **except** clause, you'll notice that it contains checks for two exceptions, *EDivByZero* and *EIntError*. In this case, a specific message is displayed for *EDivByZero* and another message for all other integer math exceptions. The order of the tests in this code is critical. If you first test for *EIntError*, you will never see the *EDivByZero* message. This is because *EIntError* includes all integer math exceptions including *EDivByZero*. This construct lets you handle as many exceptions as needed in a single **try..except** block.

Conclusion

This article has presented the basics of Delphi's exception handling mechanism. You have seen how to handle resource allocations to ensure that the resources are freed if a run-time error occurs. You have also seen how to handle exceptions raised by run-time library procedures. The second part of this series explores using the exception object, silent exceptions, writing your own custom exception handler, and more. ∆

This article was adapted from material from *Delphi: A Developer's Guide* by Bill Todd and Vince Kellen [M&T Books, 1995 — 800-488-5233].

The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\FEB\DI9602BT.

Bill Todd is President of The Database Group, Inc., a Phoenix area consulting and development company. He is co-author of *Delphi: A Developer's Guide* [M&T Books, 1995], *Creating Paradox for Windows Applications* [New Riders Publishing, 1994], and *Paradox for Windows Power Programming*; Technical Editor of *Paradox Informant*; a member of Team Borland; and a speaker at every Borland database conference. He can be reached at (602) 802-0178, or on CompuServe at 71333,2146.

By Cary Jensen, Ph.D.

Filtering Tables: Part II

An Introduction to the Query Component

In last month's "DBNavigator" we discussed how to limit access to specific records in a table that match a range. We considered several techniques that involve using Table components. In this installment of "DBNavigator," we'll use a Query component to achieve a similar effect. Specifically, we'll discuss how to create a query that returns a subset of records, as well as how to control which records are selected at run time.

First, we'll compare Table and Query components for selecting subsets of records from a table. Then, we'll continue with three basic techniques for selecting records using queries. The reasons for displaying a subset of records from a table were discussed in last month's article, so they won't be repeated here.

The techniques in this article use Query components, and therefore involve SQL (Structured Query Language). While SQL itself is not necessarily difficult to use, there are many complex issues involved that are far beyond the scope of this article. Therefore, we'll only cover the issues that are relevant to filtering tables.

Query versus Table Components

Although there are some similarities between displaying subsets of a table's records using Table and Query components, there is one overriding difference. Due to the nature of queries, an index is not required. This is not true for Table components. Using a Table component, you can only define ranges for fields involved in an index. Likewise, creating linked tables requires an appropriate index. Furthermore, those indexes must have a specific structure. For example, you can set a range based on the values in a field named City, but only if there is at least one index where City is the first field.

A second major difference is that Query components do not have range-related methods (such as *SetRange*, *ApplyRange*, and so on). Instead, they have a *SQL* property that you use to define SQL statements. For example, to filter a table, a SELECT statement is used. This is SQL DML (Data Manipulation Language) statement, and is used to define which data will be returned as an answer set.

Selecting Records with Queries

There are two critical properties for using Query components. The first is the *SQL* property, which is a *StringList* property. You can, and usually do, enter this property at design time using the String List editor (the third technique demonstrated in this article shows you how to define this property at run time). The text that you enter into the *SQL* property defines which process the query will perform.

The second is the *DatabaseName* property that you must set to either a BDE alias or a subdirectory path. *DatabaseName* defines the location of the tables being queried. There is, however, one situation where *DatabaseName* is not required. If you include the alias name in the Query's SQL statement, *DatabaseName* can remain blank.

When selecting subsets of records, you'll use the SQL SELECT statement. It has two required parts or *clauses*: SELECT and FROM. In SELECT, you specify the fields to include in the answer set that is returned, and you use the FROM statement to specify the tables from which the fields are selected. Selecting subsets of records requires one additional clause, WHERE, for identifying the appropriate records. There are additional clauses that you can use with the SELECT statement, but they are outside the scope of this article. (For information on these other clauses, select **Help | Topic Search** from Delphi's menu and enter `SQL Statements` in the Search All dialog box.)

All the queries in this article will be single-table queries, since Delphi 1.0 only supports editing of single-table queries. Let's consider the SELECT statement. SELECT is followed by a comma-separated list of field names that you want to include in the answer set. The FROM clause includes a comma-separated list of the tables where these fields are found. For example, the following SQL statement will select the CustNo and Company fields from the CUSTOMER.DB table:

```
SELECT CustNo, Company FROM CUSTOMER
```

The above example assumes that the *DatabaseName* property has been set to `DBDEMOS`, an alias that points to the location of the CUSTOMER.DB table. (Delphi creates this alias during installation. If you do not have this alias already established, you must create it before trying these examples.)

Alternatively, you can include the alias in the table name. For example, the following SQL statement does not require the *DatabaseName* property to be assigned a value:

```
SELECT CustNo, Company FROM ':DBDEMOS:CUSTOMER.DB'
```

If the *DatabaseName* property has been set to `DBDEMOS`, the following SQL statement has the same effect as the preceding one:

```
SELECT CustNo, Company FROM CUSTOMER
```

To select all fields from the specified table, you can replace the individual field names with an asterisk. For example, the following statement selects all fields from the CUSTOMER.DB table (again, assuming the *DatabaseName* property has been set to `DBDEMOS`):

```
SELECT * FROM CUSTOMER
```

While the SELECT clause specifies which fields (or *columns* in SQL vernacular) will be included in the answer set, the WHERE clause specifies which records (or *rows*) will be included. For example, the following statement will select all fields from each record where the CustNo field is equal to 1221:

```
SELECT * FROM CUSTOMER
WHERE CustNo = 1221
```

Non-numeric comparison values must be enclosed in quotes. For example:

```
SELECT StateName FROM STATES
WHERE StateCode = 'AZ'
```

Also, any comparison operators, including >, <, >=, and so forth, can be used as a comparison operator in a WHERE clause.

A WHERE clause can also include multiple conditions, using the AND and OR operators. For instance, assuming the *DatabaseName* property is set to `DBDEMOS`, the following query will select all fields from the ORDERS table where CustNo is 1221 and the SaleDate is greater than 1/1/94:

```
SELECT * FROM ORDERS
WHERE CustNo = 1221 AND
      SaleDate > '1/1/94'
```

Using these basic rules, it's now possible to demonstrate three ways of selecting subsets of records using Query components.

Linked Queries

The easiest, although least flexible, way to select a subset of records with a query is to use a *linked* query. In a linked query, the subset of records is defined by values in another DataSource, such as a Table. The WHERE condition(s) use one or more fields in the DataSource's *DataSet* to select specific records to display.

To do this, you set the Query component's *DataSource* property to the name of the DataSource that contains the field(s) used in the Query's WHERE clause. Then, within the WHERE clause, you include the field name(s) from the DataSource's DataSet in comparisons. The only trick to this is that the field names must be preceded by colons (:) so the Query can distinguish them from static conditions.

This technique is difficult to describe, but easy to demonstrate with an example. Follow these instructions to create a linked query:

1) Create a new project. On the new form, add two Label components, two DBEdit components, a DBGrid, two DataSources, a Button, a DBNavigator, a Table, and a Query component. Your form should resemble Figure 1.
2) Set *Form1*'s *Caption* property to `Linked Form Example`, and its *Position* property to *poScreenCenter*.
3) Set the *Caption* property for *Label1* to `Company:` and the *Caption* property for *Label2* to `Customer Number:`. Next, set the *DataSet* property of *DataSource1* to *Table1*, and the *DataSet* property of *DataSource2* to *Query1*. Set the *DatabaseName* property of *Table1* to `DBDEMOS`, and the

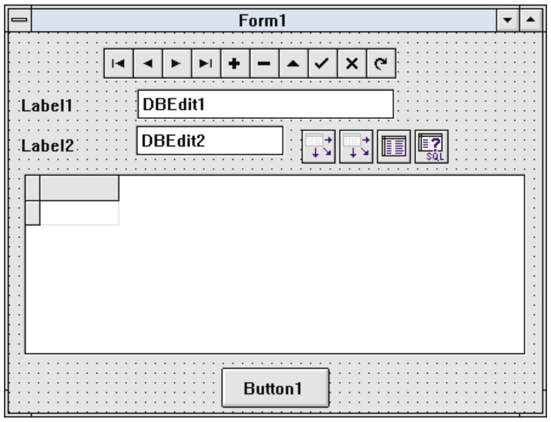


Figure 1: A new form for the LINKQRY project.

TableName property to `CUSTOMER.DB`. Finally, set the *Active* property of *Table1* to *True*.

4) Set the *DataSource* properties of both *DBEdit1* and *DBEdit2* to *DataSource1*. Next, set the *DataField* property of *DBEdit1* to *Company*, and the *DataField* property of *DBEdit2* to *CustNo*.
5) Set the *DataSource* property of *DBGrid1* to *Query1*. Set the *DataSource* property of *DBNavigator1* to *Table1*.
6) Set the *Caption* property of *Button1* to `&Close`. Next, double-click this button to create an *OnClick* event handler, and enter the procedure *Close*. This event handler should resemble:

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

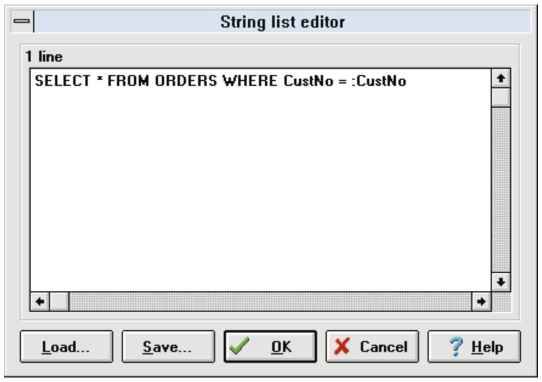
7) Now it's time to modify the Query component. Select *Query1* and set its *DatabaseName* property to `DBDEMOS`. Next, open the property editor for the *SQL* property and enter the SQL statement shown in Figure 2. Notice that the CustNo field name in the WHERE condition is preceded by a colon. CustNo is a field in CUSTOMER.DB that is the DataSet pointed to by *DataSource1*.
8) Set the *Active* property for *Query1* to *True*. Now run the form. Your screen should resemble Figure 3. Notice that each time you move to a new record in CUSTOMER.DB, the contents of the DBGrid update with the results of the linked query.

Parameterized Queries

Parameterized queries share many similarities with linked queries. The *SQL* property is typically defined at run time, and the WHERE clause includes values that change. In linked queries, these values are based on values in a DataSet pointed to by the Queries' DataSource. In parameterized queries, you control these values through code.

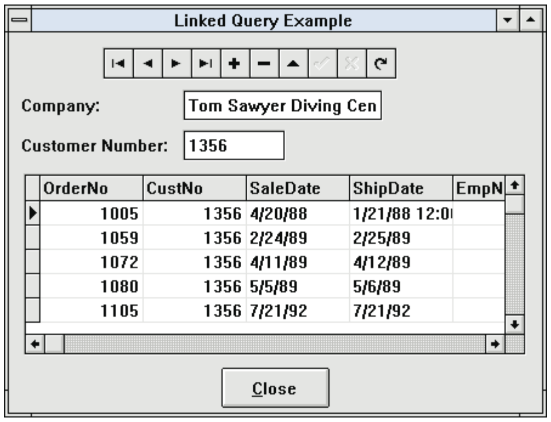
The following is an example of a parameterized SQL statement. Notice that in the WHERE clause, `CustNumber` is preceded by a colon.

```
SELECT * FROM ORDERS
WHERE CustNo = :CustNumber
```



```
SELECT * FROM ORDERS WHERE CustNo = :CustNo
```

Figure 2: A SQL SELECT statement for a parameterized query.



Company: Tom Sawyer Diving Cen
Customer Number: 1356

OrderNo	CustNo	SaleDate	ShipDate	EmpN
1005	1356	4/20/88	1/21/88 12:0	
1059	1356	2/24/89	2/25/89	
1072	1356	4/11/89	4/12/89	
1080	1356	5/5/89	5/6/89	
1105	1356	7/21/92	7/21/92	

Figure 3: The completed query example.

In a parameterized query, the colon designates `CustNumber` as a parameter. A parameterized query can have as many parameters as necessary. The only restriction is that you must define the value for a parameter before making the query active. This can be done at design time using the property editor for the *Params* property of the Query component. However, the only reason for defining a parameter at design time is to activate the query at design time. In most cases, the parameters are defined and the query is activated (by setting the *Active* property to *True* or by calling the *Open* method) at run time.

Figure 4 shows the Parameters dialog box. This is the property editor for a Query component's *Params* property. In this dialog box you can define default values, as well as define the data type of one or more of the parameters you have included in a query.

Defining a parameter at run time requires the use of either the *Params* property or the *ParamByName* method. The *Params* property is a zero-based array and its elements correspond to the Query parameters based on order of inclusion in the SQL statement. In the case of the preceding parameterized SQL statement, *CustNumber* is the only parameter, so it will correspond with the first element, `0`, of the *Params* array. Here is a sample statement that assigns the value `1221` to the `CustNumber` parameter using the *Params* property:

```
Query1.Params[0].AsInteger := 1221;
```

If the query includes more than one parameter, the first one that will appear in the SQL statement is associated with element `0` of the *Params* property; the second is associated with element `1`, and so on.

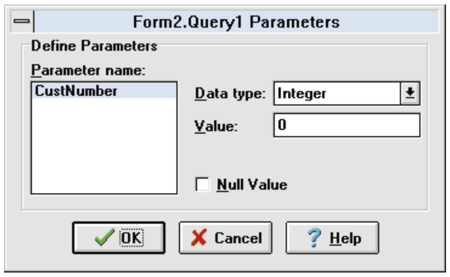
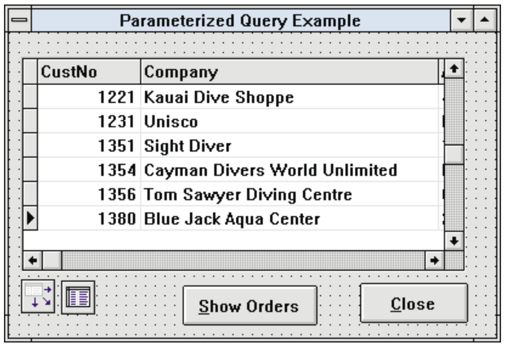


Figure 4: The Query component's *Params* property editor.

The *ParamByName* method allows you to set a parameter's value based on the parameter's name. For example, this statement uses the *ParamByName* method, and is equivalent to the preceding one that uses the *Params* property:

```
Query1.ParamByName('CustNumber').AsInteger := 1221;
```

Figure 5 is the main form of the PARAM.DPR project that demonstrates parameterized queries. This form contains a DBGrid, two Buttons, a DataSource, and a Query. The Table is associated with the CUSTOMER.DB table in the directory that the DBDEMOS alias points to.



CustNo	Company
1221	Kauai Dive Shoppe
1231	Unisco
1351	Sight Diver
1354	Cayman Divers World Unlimited
1356	Tom Sawyer Diving Centre
1380	Blue Jack Aqua Center

Figure 5: *Form1* of the project PARAM.DPR.

Figure 6 is the second form in this project. It contains a DBGrid, a Button, a DataSource, and a Query. This Query's *SQL* property is associated with the parameterized SQL statement shown earlier.

Here is the critical code in this example:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Form2.Query1.Close;
  Form2.Query1.ParamByName('CustNumber').AsInteger :=
    Table1.FieldByName('CustNo').AsInteger;
  Form2.Query1.Open;
  Form2.Caption := 'Orders for ' +
    Table1.FieldByName('Company').AsString;
  Form2.ShowModal;
end;
```

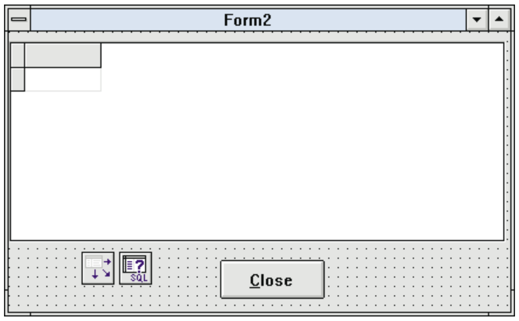


Figure 6: *Form2* of the project PARAM.DPR.

This code is associated with the *OnClick* event handler for the **Show Orders** button on *Form1*.

First, the code closes the Query component on *Form2* (in case it's already open — you cannot change the parameters of an open query). Next, the *CustNumber* parameter is assigned a value using the *ParamByName* method. The value assigned to this parameter is based on the current record of *Table1*.

Next, the query is opened, which executes the SELECT statement. The *Caption* property of *Form2* is then assigned an appropriate title. Finally, the *ShowModal* method is used to display *Form2*. The result is that the DBGrid on *Form2* displays only those records associated with the Customer that the user has selected on *Form1*.

This example is somewhat more complicated since the Query, whose properties are being modified, appears on another form. Remember, to refer to *Form2* and its objects from *Form1*, the unit associated with *Form2* (in this case, it's *ParamU2*) must be listed in a **uses** clause in the unit associated with *Form1* (*ParamU* in this example). The *ParamU* unit is shown in Listing Two on page 32. Listing Three on page 32 lists all the code for *ParamU2*.

Changing the *SQL* Property at Run Time

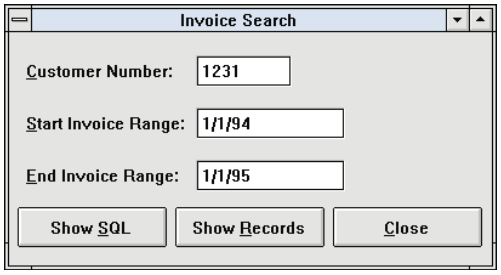
Parameterized queries are great when the fields you need to use to select a subset of records are known. However, to create a query where you sometimes select all records (i.e. when there is no WHERE clause), and sometimes select a subset based on one or more parameters, a parameterized query is not an option.

Likewise, if the table being queried is not known until run time, a parameterized query cannot be used because a parameter cannot appear in the FROM clause. Instead, you must modify the contents of the *SQL* property at run time. This permits you to create flexible queries.

Since the *SQL* property is a *StringList* property, it can be modified using the methods of the *TStringList* class. Among the most useful methods are *Add* and *Clear*. *Add* inserts a new line into a *StringList* in the last position, whereas *Clear* empties a *StringList*. The following is an example of how these two statements can be used to define a new SQL statement at run time:

```
Query1.Close;       { Close the query if it's open }
Query1.SQL.Clear;   { Remove old SQL statements }
Query1.SQL.Add('SELECT * FROM ORDERS');
Query1.SQL.Add(' WHERE CustNo = 1211');
Query1.SQL.Open;    { Open the new query }
```

This technique is demonstrated in the SQL.DPR project (its main form, *Form1*, is shown in Figure 7). This form contains three Labels, Edits, and Buttons. When the user presses the **Show SQL** or **Show Records** button, a *StringList* is constructed based on the values entered in the three Edit components (this is done with a custom procedure named *BuildQueryString*). If the Edit components are empty, the *StringList* will not contain

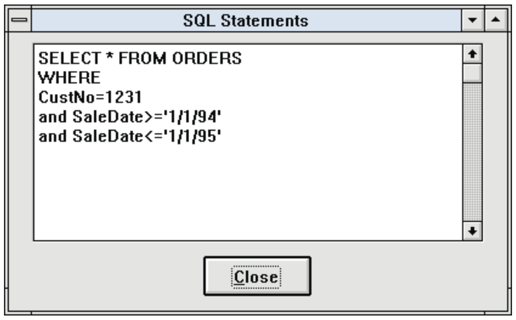


Customer Number:	1231
Start Invoice Range:	1/1/94
End Invoice Range:	1/1/95

Figure 7: *Form1* of the project SQL.DPR.

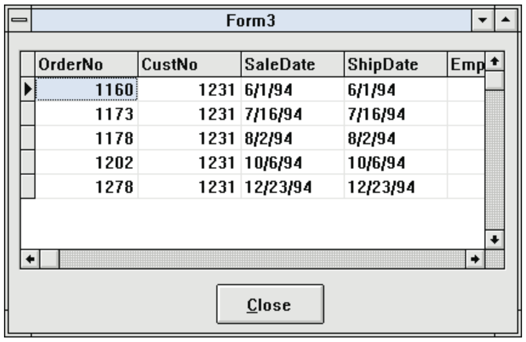
a WHERE clause. Otherwise, the *StringList* contains selection conditions based on the entered data.

When the user presses **Show SQL**, the constructed *StringList* is assigned to the *Lines* property of a Memo component on *Form2* (associated with the *SQLU2* unit). This form is then displayed (see Figure 8) and enables the user to view the SQL statements. When the user presses **Show Records**, *StringList* is assigned to the *SQL* property of a Query object on *Form3* (associated with *SQLU3*). This query is then opened and returns the appropriate subset of records (see Figure 9).



```
SELECT * FROM ORDERS
WHERE
CustNo=1231
and SaleDate>='1/1/94'
and SaleDate<='1/1/95'
```

Figure 8: The SQL statements generated in this example are based on user-entered criteria.



OrderNo	CustNo	SaleDate	ShipDate	Emp
1160	1231	6/1/94	6/1/94	
1173	1231	7/16/94	7/16/94	
1178	1231	8/2/94	8/2/94	
1202	1231	10/6/94	10/6/94	
1278	1231	12/23/94	12/23/94	

Figure 9: An answer set based on the criteria shown in Figure 8.

The code for SQL.DPR is shown in Listings Four, Five, and Six beginning on page 33. Note that this example was simplified by defining custom procedures and functions for those routines requiring repeated calls.

Other Query Issues

This article's purpose is to describe the basics of creating queries that display subsets of a table's data. However, there are many important query-related issues that are not directly related to this topic. If you are new to generating queries, you should read the online help concerning SQL queries, as well as the Query component. In particular, you should study the *RequestLive* and *UpdateMode* properties of the Query component. These properties allow you to edit the query result. In addition, you may want to examine the Database component. It allows you to explicitly control transactions when you are editing the results of a live query.

Conclusion

Queries provide you with an alternative to Table components when you must display a subset of records from a table. Using linked queries, you can have the Query component automatically update itself based on values in the linked DataSource. When you want to control the displayed records using code, parameterized queries offer a simple alternative. When you need the greatest flexibility in your SQL statements, editing the *SQL* property at run time is the best solution. ∆

The demonstration forms referenced in this article are available on the Delphi Informant Delphi Informant Works CD located in INFORM\96\FEB\DI9602CJ.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, and is Contributing Editor of *Paradox Informant* and *Delphi Informant*. Cary is this year's Chairperson of the Paradox Advisory Board for the upcoming Borland Developers Conference. He has a Ph.D. in Human Factors Psychology, specializing in human-computer interaction. You can reach Jensen Data Systems at (713) 359-3311, or through CompuServe at 76307,1533.

Begin Listing Two — The Paramu Unit

```
unit Paramu;

interface

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

type
  TForm1 = class(TForm)
    Table1: TTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
 private
  { Private declarations }
 public
  { Public declarations }
 end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses
  paramu2;

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

procedure TForm1.Button2Click(Sender: TObject);
begin
  Form2.Query1.Close;
  Form2.Query1.ParamByName('CustNumber').AsInteger :=
    Table1.FieldByName('CustNo').AsInteger;
  Form2.Query1.Open;
  Form2.Caption := 'Orders for '+
    Table1.FieldByName('Company').AsString;
  Form2.ShowModal;
end;

end.
```

End Listing Two

Begin Listing Three — The Paramu2 Unit

```
unit Paramu2;

interface

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

type
  TForm2 = class(TForm)
    DBGrid1: TDBGrid;
    Button1: TButton;
    DataSource1: TDataSource;
    Query1: TQuery;
 private
  { Private declarations }
 public
  { Public declarations }
 end;

var
  Form2: TForm2;

implementation

{$R *.DFM}

end.
```

End Listing Three

Begin Listing Four — The Sqlu1 Unit

```
unit Sqlu1;

interface

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

type
  TForm1 = class(TForm)
    CustomerNumber: TEdit;
    BeginDate: TEdit;
    EndDate: TEdit;
    ShowRecords: TButton;
    CloseButton: TButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    ShowSQL: TButton;
    procedure ShowRecordsClick(Sender: TObject);
    procedure ShowSQLClick(Sender: TObject);
    procedure BuildSQLString(var TheQuery: TStringList);
    procedure CloseButtonClick(Sender: TObject);
 private
  { Private declarations }
 public
  { Public declarations }
 end;

function IsDate(Source: TEdit) : Boolean;
procedure AddCondition(var TheQuery: TStringList;
  const field, comparison, value: string);

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses sqlu2,sqlu3;

procedure AddCondition(var TheQuery: TStringList;
  const field, comparison, value: string);
begin
  if TheQuery.Count > 2 then
    { When Count > 2, there is already at
      least one condition }
    TheQuery.Add('and '+field+comparison+value)
  else
    TheQuery.Add(field+comparison+value);
end;

function IsDate(Source: TEdit) :Boolean;
{ This function returns True if a TEdit contains a date }
begin
  try
    StrToDate(TEdit(Source).Text);
    result := True
  except
    result := False;
  end;
end;

procedure TForm1.BuildSQLString(var TheQuery: TStringList);
begin
  if BeginDate.Text <> '' then
    if not IsDate(BeginDate) then
      begin
        BeginDate.SetFocus;
        raise Exception.Create('Date value expected');
      end;

  if EndDate.Text <> '' then
    if not IsDate(EndDate) then
      begin
        EndDate.SetFocus;
        raise Exception.Create('Date value expected');
      end;

  TheQuery.Add('SELECT * FROM ORDERS') ;
  if not ((CustomerNumber.Text = '') and
     (BeginDate.Text = '')            and
     (EndDate.Text = ''))             then
    begin
      TheQuery.Add('WHERE');
      if CustomerNumber.Text <> '' then
        AddCondition(TheQuery,'CustNo','=',
                     CustomerNumber.Text);
      if BeginDate.Text <> '' then
        AddCondition(TheQuery,'SaleDate','>=',
                     #39+BeginDate.Text+#39)
      if EndDate.Text <> '' then
        AddCondition(TheQuery,'SaleDate','<=',
                     #39+EndDate.Text+#39);
    end;
end;

procedure TForm1.ShowRecordsClick(Sender: TObject);
var
  TheQuery: TStringList;
begin
  TheQuery := TStringList.Create;

  try
    BuildSQLString(TheQuery);
    { Query is done. Process it }
    Form3.Query1.SQL := TheQuery;
    Form3.Query1.Open;
    Form3.ShowModal;
  finally
    TheQuery.Free;
  end;

end;

procedure TForm1.ShowSQLClick(Sender: TObject);
var
  TheQuery: TStringList;
begin
  TheQuery := TStringList.Create;

  try
    BuildSQLString(TheQuery);
    { Query is done. Process it }
    Form2.SQLStatements.Lines := TheQuery;
    Form2.ShowModal;
  finally
```

```
    TheQuery.Free;
  end;
end;

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

end.
```

End Listing Four

Begin Listing Five — The Sqlu2 Unit

```
unit Sqlu2;

interface

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

type
  TForm2 = class(TForm)
    Button1: TButton;
    SQLStatements: TMemo;
private
  { Private declarations }
public
  { Public declarations }
end;

var
  Form2: TForm2;

implementation

{$R *.DFM}

end.
```

End Listing Five

Begin Listing Six — The Sqlu3 Unit

```
unit Sqlu3;

interface

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

type
  TForm3 = class(TForm)
    DataSource1: TDataSource;
    Query1: TQuery;
    DBGrid1: TDBGrid;
    Button1: TButton;
 private
  { Private declarations }
 public
  { Public declarations }
 end;

var
  Form3: TForm3;

implementation

{$R *.DFM}

end.
```

End Listing Six

Delphi 1.0 / Object Pascal / OLE

By Gary Entsminger

Creating an OLE Object Manager

State of the Object Art: Part II

Don't you hate it when someone breaks into an article before it even gets rolling? I do. But I can't help myself. I must warn you that the project we develop in this article behaves a bit differently on systems running Windows 3.1 than those running Windows 95. And worse yet. I'm not sure why. So… if you're a Windows 95 user, you'll experience all the thrills and chills of our project. If you're using Windows 3.1, compile the project and let me know what happens.

A Delphi application can be the "glue" that allows users to manipulate a group of objects within a single application or document. These objects can be ones you create in Delphi, or those controlled by another Windows application. For example, you can control or manipulate a Paradox table with Paradox for Windows, or the Database Desktop that ships with Delphi. Typically, to edit or modify a table, you open one of these applications to handle the work. But this is not the only way to do it.

Let's say that you want to create an application manager program that runs a set of applications to display a corresponding set of data files on a single screen or page of text. Imagine a scenario where you could:

- obtain the values of your favorite stocks from an Internet provider
- view the data in a table
- manipulate the data and display results with a Delphi model application
- store the results in a second Paradox table

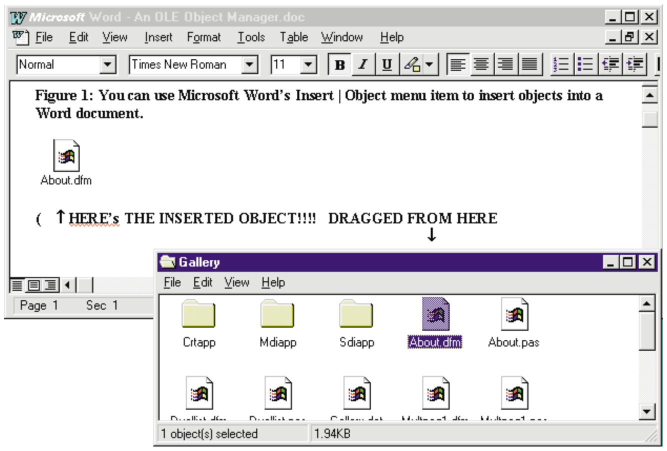


Figure 1: You can use Microsoft Word's Insert | Object menu item to insert objects into a Word document.

About.dfm

(↑HERE's THE INSERTED OBJECT!!!! DRAGGED FROM HERE ↓

Figure 1: You can use Microsoft Word's **Insert | Object** menu item to insert objects into a Word document.

In addition, all these applications could be open on the screen simultaneously.

Alternatively, you could display several kinds of data in a single document using OLE (Object Linking and Embedding). For example, in Microsoft Word you can use the **Insert | Object** command, or its OLE drag-and-drop capability to insert objects into a Word document (see Figure 1).

This article begins an exploration of Delphi's OLE capability. You'll create an OLE object manager application called *Objman* that uses two forms to maintain a group of MDI child windows and the object that each child window contains. In the Objman project, its main form, *OLEFrameForm*, is an MDI form that manages the group of MDI child windows based on the form class described in

OLEOBJ.PAS. This is the unit for the form, *OLEObjectForm*, that manages the objects.

Each time a user selects **File | New Object Window** from the application's menu, a child window is created to contain a new object. Alternatively, users can drop an object onto *OLEFrameForm* and automatically create a new child window containing the dropped object. Additionally, *OLEFrameForm* has event procedures for cascading and tiling child windows and arranging child window icons.

Note that any child window will use *OLEFrameForm* for basic window management. Therefore, a child window must reference *OLEFrameForm* in its unit's **implementation** section to avoid a circular reference. This is necessary because *OLEFrameForm* uses the child's unit, OLEOBJ.PAS, in the **interface** section of its unit.

You can use the Objman application to launch sets of applications and documents, as a model for a more complex File Explorer-type application, or as a file or application organizer. But before we get into the project details, let's briefly review a few central aspects of OLE.

OLE Servers, Containers, and Objects

Basically, OLE is a way for applications to share data. It differs from DDE (Dynamic Data Exchange) by allowing the "glue" application to connect to the OLE server application that created the data. [For more information on DDE, see Gary Entsminger's article "Great Journeys, Single Steps" in the January 1996 *Delphi Informant*.]

In DDE, you simply receive a copy of data, but not the application that created it. Additionally, you cannot edit the original data. In OLE, you get both — the data and the application that created it. Therefore, you can use the application to manipulate the data.

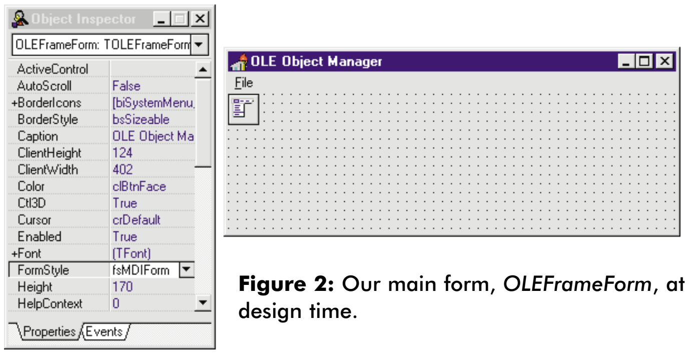
The application that creates an OLE object is called the *OLE server* and you use it to edit the object. For example, you can use Microsoft Excel as an OLE server to edit spreadsheets. Likewise, the Borland Database Desktop can be used as an OLE server to edit tables.

In short, in an OLE interaction, one application acts as the server and another as an *OLE container* that contains the OLE objects. The *OLE object* is the data — or a representation of the data — that the OLE server creates and maintains (e.g. a spreadsheet, table, .TXT file, .DOC file, etc.).

In the Objman project, you'll use Delphi to create an OLE container application that can interact with many OLE server applications.

Creating the Project

As mentioned, Objman uses two forms (an MDI container and an OLE object container) to allow users to add new MDI children that contain OLE objects. Each window can contain one OLE object. In one step, the user can drag an object onto the main form to create an MDI child and a link to the



OLEFrameForm: TOLEFrameForm

ActiveControl	
AutoScroll	False
+BorderIcons	[biSystemMenu,
BorderStyle	bsSizeable
Caption	OLE Object Ma
ClientHeight	124
ClientWidth	402
Color	clBtnFace
Ctl3D	True
Cursor	crDefault
Enabled	True
+Font	(TFont)
FormStyle	fsMDIForm
Height	170
HelpContext	0

Properties / Events

Figure 2: Our main form, *OLEFrameForm*, at design time.

dragged object. Alternatively, in two steps, the user can create a child window and paste an object from the Clipboard into a child window.

First, we'll create the user interfaces of both forms, and then we'll write the "glue" code. To begin, create a new project by selecting **File | New Project** from Delphi's menu. Delphi will automatically create a form and its corresponding unit. Then select **File | Save Project As** and save the form's unit as OLEFRAME.PAS. This will be the main form. Save the project as OBJMAN.DPR.

In the Object Inspector, change the form's *FormStyle* property to *fsMDIForm*. This establishes the form as an MDI "parent." Now change the form's *Caption* property to `OLE Object Manager`, and change its *Name* property to `OLEFrameForm`. Figure 2 shows our form in design view.

From the Standard page of the Component Palette, select the MainMenu component and add it to the form. Double-click on the MainMenu component to display the Menu Designer. Add a menu item (`&File`), and a sub-menu item (`&New Object Window`) as shown in Figure 3.

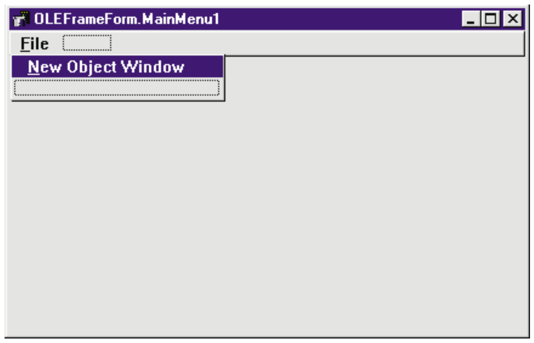


Figure 3: Use the Menu Designer to add a menu item (`&File`) and a sub-menu item (`&New Object Window`).

Merging MDI Menus

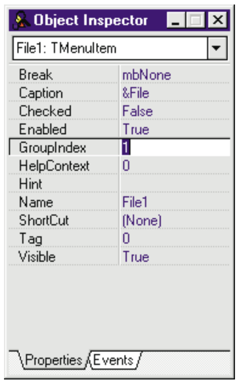
MDI applications always merge the menus of child windows with the main menu of the parent MDI window (i.e. form). However, you can control how this merging occurs.

All menu items have a *GroupIndex* property that is used to specify whether menu items are inserted or replaced in a menu. By default, all items in a menu have the same *GroupIndex* value, but you can explicitly change these values according to a group of menu merging rules.

For instance, any menu item with a *GroupIndex* property of 1, 3, or 5 will be replaced by the menu items with corresponding index values in the MDI child (the OLE server). Merging the menus of the OLE server and its container is called *in-place activation*.

Note that in non-MDI applications, the main menu's *AutoMerge* property determines if the menus (*TMainMenu*) of forms other than the main form merge with the principal form's main menu in non-MDI applications at run time.

To complete the visual aspect of *OLEFrameForm,* set the *GroupIndex* property of the *File1* menu item to 1 (see Figure 4). We'll add code after we design the OLE container child window.


Object Inspector

File1: TMenuItem

Break	mbNone
Caption	&File
Checked	False
Enabled	True
GroupIndex	1
HelpContext	0
Hint	
Name	File1
ShortCut	(None)
Tag	0
Visible	True

Figure 4: Set the *GroupIndex* property of the *File1* menu item to 1.

Creating the Child

OLEObjectForm is the child form that will contain OLE objects. Each child window features one OLEContainer component (the OLE container) to hold OLE objects. When a new MDI child window is created, it's responsible for managing the object contained in its OLE container.

The OLE container can receive an object from the Clipboard, or as the result of a drag-and-drop event. Once an object is contained by the OLE container, we can then use the container's built-in capability to edit the object or the object package. In addition, we can copy objects to the Clipboard as well as paste them from the Clipboard into the OLE container.

To create our second form, select **File | New Form** from Delphi's menu, and save the unit as OLEOBJ.PAS. This will be the blueprint for a child form. Change the form's *FormStyle* property to *fsMDIChild* to establish the form as a child window. Change the form's *Name* property to `OLEObjectForm`. From the System page of the Component Palette, select the OLEContainer component and add it to the form.

From the Standard page of the Component Palette, select a MainMenu component and place it on the form. Use the Menu Designer to add the menu items, `&File`, `&Edit`, and `&Window`, and implement their sub-menu items as follows:
- Under `&File`, add `&New Object Window` and `E&xit`.
- Under `&Edit`, add `&Copy Object to Clipboard`, `&Paste Object from Clipboard`, and `&Object`.
- Under `&Window`, add `&Cascade`, `&Tile`, and `Arrange &Icons`.

In the Object Selector, select the `&Object` menu item and set its *Enabled* property to *False*. The **Edit | Object** command remains dormant until the OLE container holds an object. When the container has an object, **Edit | Object** becomes active and allows the user to access the object. The user can then open, edit, or convert the object within the OLE container. Although you always add this menu item when using OLE containers, you never write event code for it. OLE handles it.

To enable each child to merge its *File1* menu items with *OLEFrameForm*'s *File1* menu, set the *GroupIndex* property of the *File1* menu item to 1 (to match the *OLEFrameForm*'s *File1* menu item). If you're in an experimental mood, try leaving either the main menu's *File1 GroupIndex* property or the child's set to 0. Either way the second *File1* menu item is added to the first, not replaced.

Figure 5 shows *OLEObjectForm* at design time.

Adding Object Pascal

Let's begin with our main form, *OLEFrameForm*. First, you must write a routine to create and show a child form. The *CreateMDIChild* function creates a new child form of type *TOLEObjectForm*:

```
function TOLEFrameForm.CreateMDIChild: TOLEObjectForm;
begin
  { Create a new MDI child }
  Result := TOLEObjectForm.Create(Self);
  { Show the child }
  Result.Show;
end;
```

This is the form type declared and implemented in OLEOBJ.PAS. However, note that other children could be created just as easily with this pattern.

Dragging and Dropping Objects

Before attempting to paste (e.g. drag-and-drop) objects into an OLEContainer, you must register Clipboard formats for the type of object you want to drag-and-drop (e.g. linked or embedded OLE objects). For example, Delphi uses these formats to initialize the array passed in the *Fmts* parameter of the *PasteSpecialDlg* function:

```
if PasteSpecialEnabled(Self,OLEFrameForm.Fmts) then
```

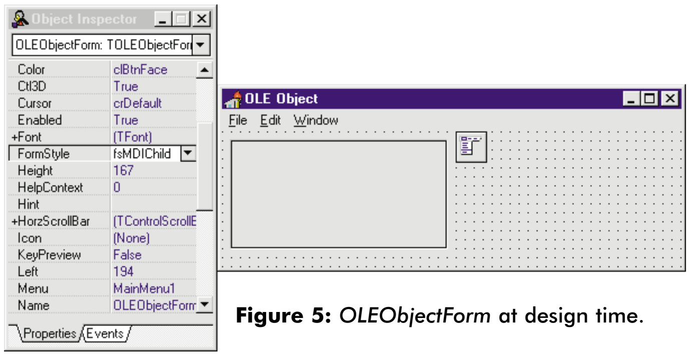
and in the *Fmts* parameter of the *RegisterFormAsOLEDropTarget* procedure:

```
RegisterFormAsOleDropTarget(Self,Fmts)
```

Now, we'll register the Clipboard formats. First, declare a *FLinkClipFmt* variable of type *Word* in the **private** section of the *OLEFrameForm* class:

```
private
  { Private registration declarations }
  FLinkClipFmt: Word;
```

Now, declare an array of type *BOLEFormat* with one element for each object format to be processed. For example, to allow dragging and dropping of linked objects only, declare a one-element array. Likewise, to allow dragging and dropping of linked *and* embedded objects declare a two-element array.



OLEObjectForm: TOLEObjectFor

Color	clBtnFace
Ctl3D	True
Cursor	crDefault
Enabled	True
+Font	(TFont)
FormStyle	fsMDIChild
Height	167
HelpContext	0
Hint	
+HorzScrollBar	(TControlScrollE
Icon	(None)
KeyPreview	False
Left	194
Menu	MainMenu1
Name	OLEObjectForm

Figure 5: *OLEObjectForm* at design time.

In this project, let's simplify the process by registering only linked objects. Thus, we need a single-element array. Declare this array in the **public** section of the *OLEFrameForm* class:

```
public
  { Public registration info for drag-and-drop --
    a single-element array to allow linking only }
  Fmts: array[0..0] of BOLEFormat;
```

To continue, initialize the *fmtId*, *fmtMedium*, *fmtIsLinkable*, *fmtName*, and *fmtResultName* fields of each *BOLEFormat* element of the array. Use the Object Pascal *BOLEMediumCalc* function to calculate the value of the *fmtMedium* field that corresponds to the value of the *fmtId* Clipboard format.

BOLEMediumCalc returns the *BOLEMedium* value to use with the Clipboard format ID passed in the *fmtId* parameter. *BOLEMedium* is the type of the *fmtMedium* field of the *BOLEFormat* record. In this project, we handle the registration process when we create the *OLEFrameForm* (see Figure 6).

OLEFrameForm's *FormDragDrop* event procedure is triggered each time an object is dropped onto the form (see Figure 7). After each drop, the *TOLEDropNotify* object is used to evaluate the *Source* object that's been dropped onto the form. If *Source* is of type *TOLEDropNotify*, then a new child window will be created to contain the dropped object. The new child window is created by the *NewObject1Click* procedure:

```
procedure TOLEFrameForm.NewObject1Click(Sender: TObject);
var
  NewMDIChild: TOLEObjectForm;
begin
  { Create a new MDI child }
  NewMDIChild := CreateMDIChild;
end;
```

Control then passes to the new child window when it's displayed on screen (see the *CreateMDIChild* procedure). The complete code for OLEFRAME.PAS is shown in Listing Seven beginning on page 40.

OLEOBJ.PAS

Next, let's proceed to the code for the child form, *OLEObjectForm*. Since *OLEFrameForm* is the MDI manager, menu items on child windows must ask *OLEFrameForm* to handle window management when the user requests it.

```
procedure TOLEFrameForm.FormCreate(Sender: TObject);
begin
  { Register this form to allow drag-and-drop }
  FLinkClipFmt := RegisterClipboardFormat('Link Source');

  { Initialize the registration array }
  Fmts[0].fmtId         := FLinkClipFmt;
  Fmts[0].fmtMedium     := BOLEMediumCalc(FLinkClipFmt);
  Fmts[0].fmtIsLinkable := True;
  StrPCopy(Fmts[0].fmtName, '%s');
  StrPCopy(Fmts[0].fmtResultName, '%s');

  { Register the form }
  RegisterFormAsOleDropTarget(Self, Fmts)
end;
```

```
procedure TOLEFrameForm.FormDragDrop(
  Sender,Source: TObject; X, Y: Integer);
var
  NewMDIChild: TOLEObjectForm;
begin
  { Is the dropped object usable? }
  if Source is TOLEDropNotify then
    begin
      { Then create a new child }
      NewMDIChild := CreateMDIChild;
      { Give the OLE container info about the object }
      with Source as TOLEDropNotify do
        NewMDIChild.OLEContainer.PInitInfo := PInitInfo
    end;
end;
```

Figure 6 (Top): Registering the form.
Figure 7 (Bottom): The *FormDragDrop* procedure.

First, the *TOLEObjectForm.NewObjectWindow1Click* event procedure tells *OLEFrameForm* to create a new child:

```
procedure TOLEObjectForm.NewObjectWindow1Click(
  Sender: TObject);
begin
  { Ask the OLEFrame to create a new child }
  OLEFrameForm.NewObject1Click(Sender)
end;
```

Next, the *TOLEObjectForm.Cascade1Click* event procedure sends a message to *OLEFrameForm* to cascade the child windows:

```
{ Ask the OLEFrame to handle child window behavior }
procedure TOLEObjectForm.Cascade1Click(Sender: TObject);
begin
  OLEFrameForm.Cascade
end;
```

The *TOLEObjectForm.Tile1Click* event procedure instructs *OLEFrameForm* to tile the child windows:

```
procedure TOLEObjectForm.Tile1Click(Sender: TObject);
begin
  OLEFrameForm.Tile
end;
```

At this point, the *TOLEObjectForm.ArrangeIcons1Click* event procedure informs *OLEFrameForm* to arrange any minimized child window icons:

```
procedure TOLEObjectForm.PasteObject1Click(Sender: TObject);
var
  ClipFmt: Word;
  DataHand: THandle;
  { Pointer to the object's info }
  Info: Pointer;
begin
  if PasteSpecialEnabled(Self, OLEFrameForm.Fmts) then
    { Show the user a PasteSpecial dialog to permit the
      user to decide the specifics of the Object paste }
    if PasteSpecialDlg(Self, OLEFrameForm.Fmts, 0,
                       ClipFmt, DataHand, Info) then
      begin
        OLEContainer.PInitInfo := Info; {the object's info }
        ReleaseOLEInitInfo(Info);
      end;
end;
```

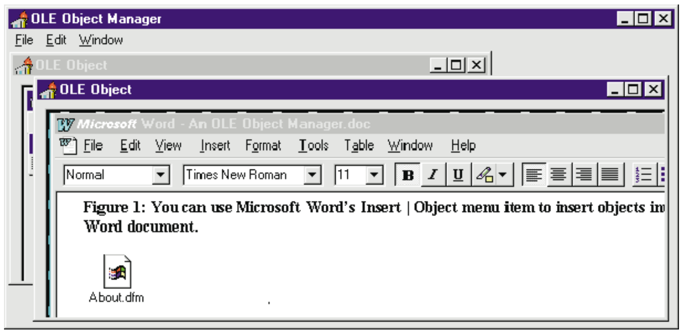
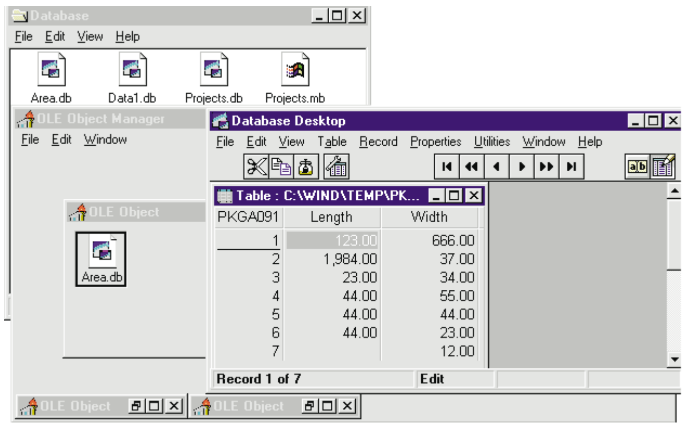

OLE Object Manager
File Edit Window
OLE Object
OLE Object
Microsoft Word - An OLE Object Manager.doc
File Edit View Insert Format Tools Table Window Help
Normal
Times New Roman
11
About.dfm

Figure 1: You can use Microsoft Word's Insert | Object menu item to insert objects in Word document.



PKGA091	Length	Width
1	123.00	666.00
2	1,984.00	37.00
3	23.00	34.00
4	44.00	55.00
5	44.00	44.00
6	44.00	23.00
7		12.00

Record 1 of 7 Edit

Figure 8 (Top): The *TOLEObjectForm.PasteObject1Click* event procedure. **Figure 9 (Middle):** Here, Objman is shown with Microsoft Word in a child window. **Figure 10 (Bottom):** In this running example of Objman, the user is entering information into a table with the Database Desktop.

```
procedure TOLEObjectForm.ArrangeIcons1Click(Sender: TObject);
begin
   OLEFrameForm.ArrangeIcons
end;
```

The *TOLEObjectForm.Edit1Click* event procedure keeps track of the Clipboard's contents. Each time the user selects **Edit** from the menu, the **Edit** item's sub-menus appear after a check for a pasteable object is made. If a pasteable object is detected in the Clipboard, *PasteObject1* is enabled:

```
{ If there's no pasteable object in the Clipboard,
  disable the PasteObject menu item. }
procedure TOLEObjectForm.Edit1Click(Sender: TObject);
begin
  PasteObject1.Enabled :=
    PasteSpecialEnabled(Self, OLEFrameForm.Fmts)
end;
```

Copy and Paste Procedures

The *TOLEObjectForm.CopyObject1Click* event procedure copies the OLEContainer's object in the active child window to the Clipboard:

```
procedure TOLEObjectForm.CopyObject1Click(Sender: TObject);
begin
  OLEContainer.CopyToClipboard(False);
end;
```

Finally, the *TOLEObjectForm.PasteObject1Click* event procedure (see Figure 8) pastes the OLEContainer's object from the Clipboard into the OLE container in the active child window. Figures 9 and 10 show Objman at run time. Feel free to explore and modify the application. (The complete code for OLEOBJ.PAS is shown in Listing Eight on page 40.)

Conclusion

In the next installment, we'll continue our OLE discussion as we implement additional functionality in our OLE Object Manager. For example, to allow a user to selectively close child windows, a parent window needs to keep track of its children (in an array, for example), then match a user's selection from a menu (or equivalent) to the corresponding child window. Deleting a child is equivalent to deleting an object. In Objman2, we'll get to that. See you then. ∆

The demonstration project referenced in this article is available on the Delphi Informant Delphi Informant Works CD located in INFORM\96\FEB\DI9602GE.

Gary Entsminger is the author of *The Way of Delphi* [Prentice-Hall, 1996], *The Tao of Objects* [M&T Books, 1995], *Secrets of the Visual Basic Masters* [Sams, 1994], and *Developing Paradox Databases* [M&T Books, 1993].

Begin Listing Seven — OLEFRAME.PAS

```
unit OLEFrame;

{ Description: An OLE main MDIform to contain MDI child
  windows. Use this form to launch and manage MDI child
  windows.

  In the project, OBJMAN.DPR, in this article, OLEFRAME.PAS
  launches MDI child windows described in OLEOBJ.PAS.

  This OLEOBJ.PAS unit is responsible for managing
  the objects.

  The OLEFrame is the main form for the OLE Object Manager.
  Each time a user selects New Object Window from its main
  menu, it creates a child window to contain a new object.
  Alternatively, users can drop an object onto this form to
  create a new child window containing the dropped object.

  This form is an MDI window. It contains event procedures
  to cascade and tile child  windows and arrange child
  window icons.

  Note that any child window using this MDI container,
  OLEFrame, must reference this form in the implementation
  section of its unit to avoid a circular reference.

  This is necessary because this unit uses its child unit,
  OLEOBJ.PAS, in the interface part of this unit.

  Code Patterns:
  Derive a new form from TForm (inheritance).
  Merge Parent and Child MDI menus.
  Register a form to allow drag-and-drop. }

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, Menus, ExtCtrls,
  BOLEDefs, TOCtrl,
  { This is the child unit for this project };
  OLEObj

type
  TOLEFrameForm = class(TForm)
  MainMenu1: TMainMenu;
  File1: TMenuItem;
  NewObject1: TMenuItem;

  { Menu item event procedures }
  procedure NewObject1Click(Sender: TObject);
  procedure Exit1Click(Sender: TObject);

  { Form event procedures }
  procedure FormCreate(Sender: TObject);
  procedure FormDragDrop(
    Sender, Source: TObject; X, Y: Integer);

private
  { Private registration declarations }
  FLinkClipFmt: Word;

public
  { Public registration info to drag-and-drop
    a single-element array to allow linking only }
  Fmts: array[0..0] of BOleFormat;

  { Public created MDI child that can be
    accessed from child forms }
  function CreateMDIChild: TOLEObjectForm;
end;

var
  OLEFrameForm: TOLEFrameForm;

implementation

{$R *.DFM}

procedure TOLEFrameForm.FormCreate(Sender: TObject);
begin
  { Register this form to allow drag-and-drop }
  FLinkClipFmt := RegisterClipboardFormat('Link Source');

  { Initialize the registration array }
  Fmts[0].fmtId  := FLinkClipFmt;
  Fmts[0].fmtMedium  := BOLEMediumCalc(FLinkClipFmt);

  Fmts[0].fmtIsLinkable  := True;
  StrPCopy(Fmts[0].fmtName, '%s');
  StrPCopy(Fmts[0].fmtResultName, '%s');
  { Register the form }
  RegisterFormAsOleDropTarget(Self, Fmts)
end;

function TOLEFrameForm.CreateMDIChild: TOLEObjectForm;
{ Create a child form of TOLEObjectForm type }
{ Note that other children could also be created
  using this pattern }

begin
  { Create a new MDI child }
  Result  := TOLEObjectForm.Create(Self);
  { Show the child }
  Result.Show;
end;

procedure TOLEFrameForm.Exit1Click(Sender: TObject);
begin
  Close
end;

{ Create a new child for a dropped object.
  The new child will contain the dropped object.}
procedure TOLEFrameForm.FormDragDrop(
  Sender, Source: TObject; X, Y: Integer);
var
  NewMDIChild: TOLEObjectForm;
begin
  { Is the dropped object usable? }
  if Source is TOLEDropNotify then
```

```
    begin
      { Then create a new child }
      NewMDIChild := CreateMDIChild;
      { Give the OLE container info about the object }
      with Source as TOLEDropNotify do
        NewMDIChild.OLEContainer.PInitInfo := PInitInfo
    end
end;

procedure TOLEFrameForm.NewObject1Click(Sender: TObject);
var
  NewMDIChild: TOLEObjectForm;
begin
  { Create a new MDI child }
  NewMDIChild := CreateMDIChild;
end;

end.
```

End Listing Seven

Begin Listing Eight — OLEOBJ.PAS

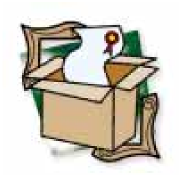
```
unit OLEobj;

{ Description: A child form to contain OLE objects.
  In the project, OBJMAN.DPR, OLEFRAME.PAS launches
  MDI child windows described in this unit, OLEOBJ.PAS.


  This OLEOBJ.PAS unit is responsible for managing the
  objects. This form manipulates an object contained by a
  TOLEContainer component.

  The OLE container can get an object from the Clipboard or
  by a drag-and-drop operation. This allows good object
  linking capability and keeps the application simple and
  easy to understand, but does not allow objects to be
  inserted directly into the OLE container without the
  use of a drag/drop event or the Clipboard.

  * It allows basic editing of the object or the object
    package contained by a TOLEContainer component.
  * It can copy an object to the clipboard or paste an
    object from the clipboard.
  * It allows the user to manipulate child windows
    (Create, Tile, Cascade windows and Arrange Icons) by
    using  the OLEFrame form class described in the
    OLEFRAME.PAS unit.
  * The OLEFrame is the main form for the OLE Oject
    Manager.
  * It creates a child window for each object and manages
    the child windows.

  Code Patterns:
  Derive a new form from TForm (inheritance).
  Use the OLEContainer component.
  Use the Clipboard to copy and paste objects. }

interface

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

type
  TOLEObjectForm = class(TForm)
  OleContainer: TOleContainer;

  { Menu system }
  MainMenu1: TMainMenu;
  File1: TMenuItem;
  NewObjectWindow1: TMenuItem;
  Exit1: TMenuItem;
  N1: TMenuItem;  { A line for looks }
  Edit1: TMenuItem;
  CopyObject1: TMenuItem;
  PasteObject1: TMenuItem;
  OLEObjectMenuItem: TMenuItem;
  Window1: TMenuItem;
  Cascade1: TMenuItem;
  Tile1: TMenuItem;
  ArrangeIcons1: TMenuItem;

  { Menu item click event procedures }
  procedure NewObjectWindow1Click(Sender: TObject);
  procedure Exit1Click(Sender: TObject);
  procedure Edit1Click(Sender: TObject);
  procedure CopyObject1Click(Sender: TObject);
  procedure PasteObject1Click(Sender: TObject);
  procedure Cascade1Click(Sender: TObject);
  procedure Tile1Click(Sender: TObject);
  procedure ArrangeIcons1Click(Sender: TObject);
private
   { Private declarations }

public
   { Public declarations }
end;

var
  OLEObjectForm: TOLEObjectForm;


implementation

{ Uses the MDI container, OLEFrame, in the implementation
  section to avoid a circular reference }
uses OLEFrame;

{$R *.DFM}

procedure TOLEObjectForm.NewObjectWindow1Click(
  Sender: TObject);
begin
  { Ask the OLEFrame to create a new child }
  OLEFrameForm.NewObject1Click(Sender)
end;

procedure TOLEObjectForm.Exit1Click(Sender: TObject);
begin
  OLEFrameForm.Exit1Click(Sender)
end;
```

```
{ Ask the OLEFrame to handle child window behavior }
procedure TOLEObjectForm.Cascade1Click(Sender: TObject);
begin
  OLEFrameForm.Cascade
end;

procedure TOLEObjectForm.Tile1Click(Sender: TObject);
begin
  OLEFrameForm.Tile
end;

procedure TOLEObjectForm.ArrangeIcons1Click(
  Sender: TObject);
begin
  OLEFrameForm.ArrangeIcons
end;


{ If there's no pasteable object in the Clipboard,
  disable the PasteObject menu item. }
procedure TOLEObjectForm.Edit1Click(Sender: TObject);
begin
  PasteObject1.Enabled :=
    PasteSpecialEnabled(Self, OLEFrameForm.Fmts)
end;


{ Clipboard Copy and Paste object procedures }
procedure TOLEObjectForm.CopyObject1Click(Sender: TObject);
begin
  OLEContainer.CopyToClipboard(False);
end;


procedure TOLEObjectForm.PasteObject1Click(
  Sender: TObject);
var
  ClipFmt: Word;
  DataHand: THandle;
  Info: Pointer;
begin
  if PasteSpecialEnabled(Self, OLEFrameForm.Fmts) then

    { Show the user a PasteSpecial dialog to permit the
      user to decide the specifics of the Object paste }
    if PasteSpecialDlg(Self, OLEFrameForm.Fmts, 0,
                       ClipFmt, DataHand, Info) then
      begin
        OLEContainer.PInitInfo := Info;
        ReleaseOLEInitInfo(Info);
      end;
end;

end.
```

End Listing Eight

NEW & USED

BY ROBERT VIVRETTE



HyperTerp Pro

HyperAct's Scripting Language for Delphi

HyperTerp Pro is, simply put, a scripting language. It enables developers to implement user-definable programming scripts into their applications. To many of us, the real need for such a capability may not be clear, so hopefully in this review I can explain enough about HyperTerp to clarify this. But first, let me provide a brief definition of a scripting language.

What Is a Scripting Language?

A Delphi programmer writes a program using Object Pascal, which is then converted into machine code by the Delphi compiler. However, suppose that you want your program's users to also have some control over the code in the application. You can't ship a Delphi compiler to each user, can you? Besides, I don't think you want them fiddling with your code.

Instead, you can provide a scripting ability *within* your program. The user could write scripts, or macros (whatever you want to call them), to perform some user-defined — rather than application-defined — task. These scripts could do pretty much anything that you might think of. For example:

- **A charting program that graphs mathematical formulas.** The user could define the formulas, as well as define how the graph will appear. I am not just talking about "pick a graph type from one of the X number available." Rather, the user could write an actual script or sub-program that would take the formula results and then draw, plot, or print the data in a specified format.
- **An automation system.** The user could define various chores that would be automated through the use of scripts. This may include a complex file or database maintenance, or even an "agent" system that manages various programs on your computer (e.g. retrieving e-mail, automated report printing, examining network devices, etc.).

Many of us are familiar with DOS batch files. These are actually a rudimentary form of a scripting language. UNIX provides a much more sophisticated scripting language. A scripting language such as HyperTerp is similar to these, but more tailored to a Windows environment. Whereas the first two examples run within an operating system, DOS and UNIX respectively, HyperTerp runs within your application.

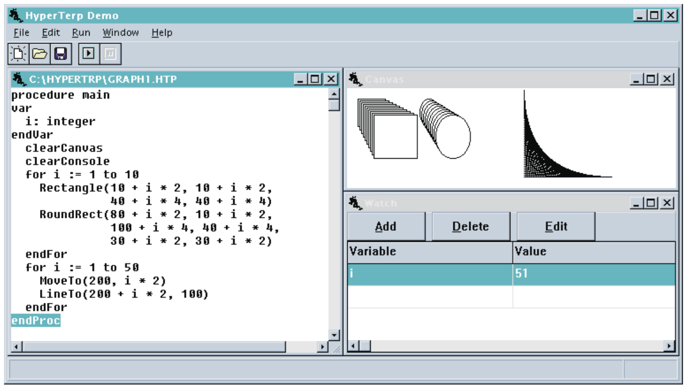
There are two basic limitations to the use of a scripting language. The first is that the scripting language itself will almost always be a subset of a more powerful language. Simply put, there may be some commands in a language that, for various reasons, simply do not transfer into a scripting language. The second limitation is speed — scripting languages are almost always interpreted rather than compiled; there is a scripting interpreter that takes the user's commands and executes them one at a time.

HyperTerp for Delphi

HyperTerp is a scripting language for Delphi, and there are also versions that support C++ and Borland Pascal. Some of HyperTerp's roots extend back to a similar product called PasterP that some readers will no doubt be familiar with.

The language used in HyperTerp has a "Pascal-ish" structure with a bit of BASIC thrown in for good measure. Figure 1 shows the demonstration program that can load and execute all the sample scripts included with HyperTerp. This script is a short program that draws various graphic shapes on a canvas and should give you a good idea of how the language is structured.

Incorporating the HyperTerp scripting language into your application is fairly straightforward. The installation steps add a component to Delphi's Component Palette. Placing one of these components on a form (typically the main form) attaches the scripting interpreter to your program. The interpreter "object" is responsible for performing all the HyperTerp script processing and will add about 150K of overhead to the application.



```
procedure main
var
  i: integer
endVar
  clearCanvas
  clearConsole
  for i := 1 to 10
    Rectangle(10 + i * 2, 10 + i * 2,
              40 + i * 4, 40 + i * 4)
    RoundRect(80 + i * 2, 10 + i * 2,
              100 + i * 4, 40 + i * 4,
              30 + i * 2, 30 + i * 2)
  endFor
  for i := 1 to 50
    MoveTo(200, i * 2)
    LineTo(200 + i * 2, 100)
  endFor
endProc
```

Variable	Value
i	51

Figure 1: The demonstration program included with HyperTerp showing some of its graphics capabilities.

After dropping the interpreter component onto your project, you will need to write a couple of event handlers to provide links between your program and the scripting engine. The demonstration program includes about four or five event handlers of about six to eight lines of code each. If you know what you want HyperTerp to do, it won't take long to make these connections.

The HyperTerp script files are ASCII text files, and it's a simple matter to create a quick program that loads/saves and executes these scripts. (Actually, the sample program described in the documentation does just this.)

Inside HyperTerp

As mentioned, HyperTerp is a subset of Pascal. The Standard function library includes 19 math, 10 string, six file, and two memory functions and procedures. The Extended library adds six more file functions and six system functions (primarily date/time stuff). The Windows library adds 27 of the more common Windows API routines, allowing scripts to create windows, dialog boxes, add and manipulate controls, access profile strings from .INI files, and so on.

There are also some procedures and functions unique to its language. For example, the *ForEachFile* procedure allows the user to automatically traverse all the files that match a valid DOS wildcard mask in the current directory (and optionally in all the subdirectories) and perform a common operation on each of these files.

Because HyperTerp is a subset of a more powerful language, users may occasionally need a function or procedure that is not currently supported. In the past, HyperAct has released updated versions that support additional programming functions and procedures, and this policy will surely continue.

Due to its interpreted nature, HyperTerp probably won't break speed records, but it is nonetheless a capable, solid system. What it does, it does very well. I don't need the features it provides, but I know there are many developers who would welcome HyperTerp's power and flexibility.

Documentation

My one key complaint with HyperTerp regards its documentation. Its accompanying manual is a flimsy, 20-page affair that covers installation of the software, a simple tutorial, a four-page reference on the three variations of the script interpreter, and a few more pages of some extensions to the example code. As someone new to scripting languages, the documentation creates more questions than it answers.

Fortunately, the package includes two help files. One is (for the most part) an electronic form of the printed material, while the other is a fairly complete language reference file. I would have liked to see some more detailed discussion of some of the commands, but it's sufficient as a reference on the scripting language. You'll find that you'll be working quite close to this language reference as you get up to speed with HyperTerp's scripting syntax. The inclusion of this reference went a long way toward taking the bad taste out of my mouth caused by the printed "manual."

HyperTerp comes in two flavors. The standard version includes the .DCU files for the component objects and sells for US$149. The professional version adds all the source code files as well and sells for US$395. There are no royalties for distributing the scripting engine. The enclosed literature indicates that the scripting engine will work within a DLL if desired. Both versions are completely native Delphi objects — no DLLs or VBXes. HyperAct has provided a demonstration version of HyperTerp in library 22 (3rd Party Products) of the Borland Delphi forum on CompuServe (GO DELPHI). The file name is PTRPDEMO.ZIP. They also have a Web site at http://www.hyperact.com/.

Conclusion

If you need the ability to provide a scripting system within your Delphi applications, you need look no further than HyperTerp Pro. Printed documentation aside, the whole package is stable and capable. ∆

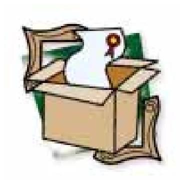
INFORMANT FACT FILE

HyperTerp Pro by HyperAct, Inc. is a scripting language that programmers can include in their Delphi applications. With HyperTerp, users can implement custom scripts (or macros) to add user-defined functionality to the program. This can be done by adding a HyperTerp component to the Component Palette, which in turn automatically attaches a scripting interpreter to the program. Although the documentation is lacking, HyperTerp includes a solid online language reference to assist you with the scripting syntax. If your system requires scripting, HyperTerp Pro is the tool for you.

HyperAct, Inc.
3437 335 Street
West Des Moines, IA 50266
Phone: (515) 987-2910
Fax: (515) 987-2909
E-Mail: CIS:76350,333 or Internet: rloewy@panix.com
Web Site: http://www.hyperact.com/
Price: Standard version, including the .DCU files, US$149; Professional version that includes all source code files, US$395. There are no royalties for distributing the scripting engine.

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

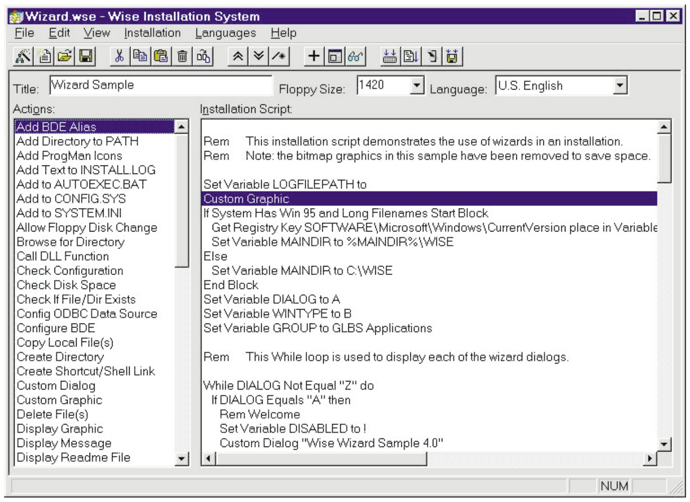
BY MICAH J. BLEECHER



WISE Installation System

Create Distribution Diskettes Quickly and Easily

If you're in the market for a powerful, easy-to-use Windows application installation system, then waste no time and order the WISE Installation System Version 4.0 by Great Lakes Business Systems (GLBS) today. In my search for a solid installation utility, no other package could compare to the combination of power, performance, and ease-of-use provided by WISE. The other packages I examined either lacked crucial features or required that I learn another programming language just to create an installation program. WISE beautifully combines the best of both worlds.



Wizard.wse - Wise Installation System
File Edit View Installation Languages Help
NUM

```
Title: Wizard Sample        Floppy Size: 1420        Language: U.S. English

Actions:                      Installation Script:
Add BDE Alias
Add Directory to PATH         Rem   This installation script demonstrates the use of wizards in an installation.
Add ProgMan Icons             Rem   Note: the bitmap graphics in this sample have been removed to save space.
Add Text to INSTALL.LOG
Add to AUTOEXEC.BAT           Set Variable LOGFILEPATH to
Add to CONFIG.SYS             Custom Graphic
Add to SYSTEM.INI             If System Has Win 95 and Long Filenames Start Block
Allow Floppy Disk Change        Get Registry Key SOFTWARE\Microsoft\Windows\CurrentVersion place in Variable
Browse for Directory            Set Variable MAINDIR to %MAINDIR%\WISE
Call DLL Function             Else
Check Configuration             Set Variable MAINDIR to C:\WISE
Check Disk Space              End Block
Check If File/Dir Exists      Set Variable DIALOG to A
Config ODBC Data Source       Set Variable WINTYPE to B
Configure BDE                 Set Variable GROUP to GLBS Applications
Copy Local File(s)
Create Directory              Rem   This While loop is used to display each of the wizard dialogs.
Create Shortcut/Shell Link
Custom Dialog                 While DIALOG Not Equal "Z" do
Custom Graphic                  If DIALOG Equals "A" then
Delete File(s)                    Rem Welcome
Display Graphic                   Set Variable DISABLED to !
Display Message                   Custom Dialog "Wise Wizard Sample 4.0"
Display Readme File
```

Figure 1: The WISE screen layout showing all the script actions.

Most people believe that installation utilities are reserved for developers of commercial or vertical software. However, WISE (the acronym for Windows Self-Installing Executable), is so easy to use that it's also an excellent choice to replace standard compression utilities for custom software developers to perform installations and upgrades.

The IDE

The IDE (Integrated Development Environment) comes in both 16- and 32-bit versions, although they both produce a 16-bit single installation file. It will create one large file for electronic deliveries or divide the file so it will span multiple floppy disks if necessary.

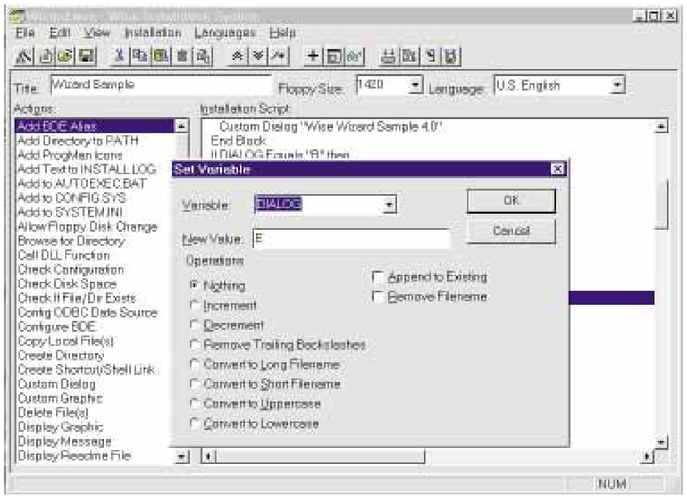
The screen layout is intuitive, as seen in Figure 1. The left side of the screen displays a list box containing 57 script actions and the right side displays a list of the selected script actions. To use a script action, simply highlight it and drag it to the selection window.

When you run the compiled installation program, each action is executed sequentially, much like a batch file. Although the WISE manual refers to these as script actions, the text of the selected action is not directly editable. Script actions must be edited by double-clicking on the appropriate line that displays a property editor.

In addition, a different property editor is available for each script action. Examine the Set Variable property editor in Figure 2. If you're a Borland product user, you will naturally want to right-click on a script action. It's important to note that the **Properties** choice on the resulting menu is the global properties for the project. Select **Edit** to modify the properties for the selected script action.

In addition to numerous examples, WISE has an Installation Expert that guides you through the entire process, and the resulting script can be edited as if you started from scratch. It's so easy that you can create a professional install script in a matter of minutes.

Programming and Control Structures. Although there are only 57 script actions, these are more than enough to do anything



Title: Wizard Sample Floppy Size: 1420 Language: U.S. English

Installation Script:
```
  Custom Dialog "Wise Wizard Sample 4.0"
End Block
If DIALOG Equals "B" then
```

Variable: DIALOG
New Value: E
Operations: [✓ Nothing]

Figure 2: The Set Variable property editor.

you want or need to model your installation logic. You can create an unlimited number of variables or use pre-defined system variables that can be referenced throughout your script and within custom dialog boxes.

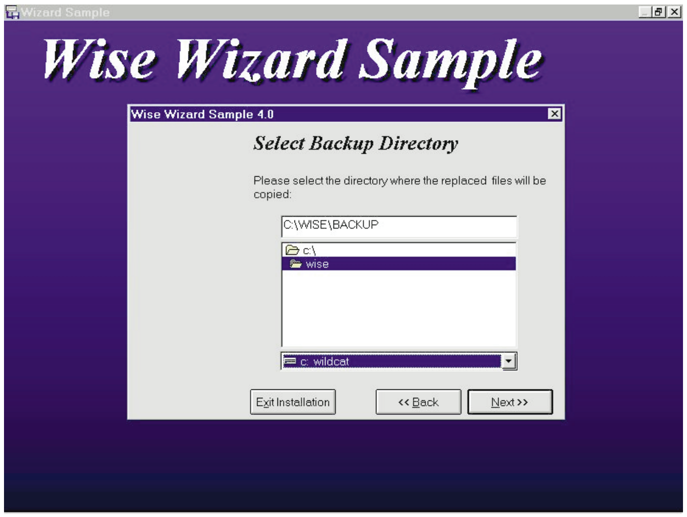
There are no GOTO actions, but you can accomplish your objectives by using the built-in IF, THEN, ELSE, and WHILE actions for conditional branching and looping. For special needs, a string parsing action is conveniently available if you are reading strings out of text files where only a portion of the string is needed.

System Access. WISE features numerous actions that allow you to interface with nearly every conceivable aspect of the operating system. These include: reading/writing files and .INI files, locating files, checking system configuration, video resolution, available memory and disk space, installed hardware capabilities and much more.

These features, along with the control structures, allow you to create a single install project that behaves differently in various operating systems. This functionality becomes extremely important with the advent of Windows 95 and the growing popularity of NT.

Custom Dialog Editor. To fully customize the user interface, you can create custom dialog boxes using a variety of built-in controls to present data and options to the user and then return selected values to your script. These dialog boxes can be saved to the disk so they can be shared between projects. There are numerous samples, an example script that uses custom dialog boxes to create an MS Wizard-like install program (see Figure 3), and a host of pre-written templates for a variety of functions.

Extendibility and Interoperability. If you cannot find what you need in the built-in script actions, you can easily call a function from within a DLL or run an executable file. This can be helpful when calling an external serializing module or migrating data during software upgrades. For unusual circumstances, your install program can even be controlled using DDE (Dynamic Data Exchange). Code samples of DLLs using Delphi and C are provided in addition to an explanation of how the DDE functionality works using Visual Basic.



Select Backup Directory

Please select the directory where the replaced files will be copied:

C:\WISE\BACKUP

c:\
wise

c: wildcat

Figure 3 : A custom dialog box that resembles a Microsoft Wizard.

Multi-Media Support. To add a little pizzazz to your installations, you can implement custom graphics and play .WAV files. WISE features a built-in graphics editor that is, while elementary, adequate for most needs. You can display graphics at any time during the installation process and simple animation effects, including fade-in, are supported. Most important however, is the ability to directly install fonts that are required by your application.

Windows 95 Install Support. A host of new issues must be considered when installing applications into Windows 95. The following script actions are included to interact with this operating system:

- Create shortcuts and shell links.
- Edit and retrieve registration database values.
- Self-registration of OCX and DLL modules.
- Support for and conversion between long and short filenames.
- Get Win32 system directory.
- Windows 95 shared DLL counter support.
- Complete support for uninstall.

Specific BDE Support. When working with Borland products it's important to install and configure the BDE (Borland Database Engine). Unfortunately, using the separate install package for the BDE that is provided with Delphi is simply awkward and unprofessional.

Fortunately WISE manages these tasks quite easily. A sample BDE configuration script is included that you can import into your projects. You can configure both BDE system parameters as well as add and configure aliases. You can see support for the 32-bit BDE that will be available shortly after Borland ships their 32-bit products.

INFORMANT FACT FILE

The WISE Installation System, by GLBS, Inc., is a solid installation utility for Windows-based applications. WISE's IDE is intuitive and creates 16-bit installation files. Script actions are executed as if they are batch files and are edited in individual property editors. An Installation Expert can also guide you as you create an installation script. WISE also supports system access, multi-media, Windows 95, the BDE, and much more. If you're looking for the right installation tool, WISE is a worthwhile investment.

WISE Installation System
Great Lakes Business Solutions, Inc.
2200 North Canton Center Road, Ste. 220
Canton, Michigan 48187
Phone: (800) 554-8565
E-Mail: CIS: GO WISEINSTALL
Web Site: http://www.glbs.com/
Price: US$199; Annual Technical Support Contract, US$95.

For those of you accessing data via ODBC, WISE easily installs and configures both 16- and 32-bit ODBC drivers as well.

Technical Support. The product comes with 30 days of free technical support. Although I needed to leave a message on voice mail, GLBS technical support staff promptly returned my call, was helpful, and was knowledgeable of the product. In addition, they have a section in CompuServe's Windows Third-Party H Forum. To access this forum, type WISE-INSTALL at the GO prompt and GLBS will answer questions online. They are good about making maintenance releases available from their BBS using passwords for registered users. An annual maintenance contract entitles the licensee to phone support beyond the first 30 days and free upgrades throughout the year.

Demonstration Versions. You can download a demonstration version of WISE from their CompuServe forum or from the GLBS Web site at http://www.glbs.com. The demonstration restricts you to using the resulting executable on the computer it was created on, but is otherwise fully functional.

Conclusion

WISE has too many features to mention in the space of this article. So don't hesitate to download a demonstration and try WISE for yourself. I am confident that you will be won over by this product as I was. My only complaint is that I would like to see more detail in the manual — not only regarding each of the features, but also program installation methodology in general. There are many new issues with Windows 95 and I would like to see some detailed information on the subject, even in the form of a help or supplemental text file. Nonetheless, the money I spent on WISE was well spent. I wish I could say that about all developer utilities I've purchased. ∆

Micah Bleecher is a partner in Datacraft Systems, Inc., a database consulting firm that serves the Northeast. They specialize in Delphi, Paradox, and Web database integration. Mr Bleecher can be reached at (609) 227-0202, e-mail: micahb@datacraft.db.com, or visit the Datacraft Systems Web site at: http: //www.datacraft.db.com.