September 1999, Volume 5, Number 9

Cover Art By: *Darryl Dennis*

ON THE COVER

6 Outlook from Delphi — Bill Todd
Mr Todd shows us how to use Outlook as an Automation server to extract contact information, update a central database, add new contacts derived from other sources, create new folders, add items of any type, and more — all from the power and flexibility of Delphi as the Automation client.

FEATURES

10 Dynamic Delphi
Data at a Distance — Mike Riley
Mr Riley demonstrates Microsoft Remote Data Services (RDS), which enable developers to remotely access server-based ODBC and OLE DB data sources without an ODBC or BDE data connection.

15 Delphi at Work
Making a Hash of It — Gregory H. Deatz
Hash tables are useful for dealing with key/value pairs that require frequent and fast access. Mr Deatz presents an abstract class for designing just about any kind of hash table a Delphi developer could desire.

19 In Development
Low-level Delphi — Andre van der Merwe
Mr van der Merwe provides a step-by-step introduction to the Delphi 4 CPU window, an invaluable tool for getting "under the hood" to see the assembler code that the Delphi compiler generates.

23 OP Tech
MDI and ActiveX — Dan Miser
Like oil and water, MDI and ActiveX just don't mix. Or do they? Using smoke, mirrors, and Delphi's ActiveForm, Mr Miser demonstrates 10 simple steps for getting them to cooperate.

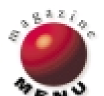
25 Columns & Rows
Inside Oracle Queries — Keith Wood
Mr Wood shares a utility program for viewing the plan the Oracle query engine will follow when executing a specified query. It also demonstrates the use of tree views and their limitations in Delphi.

REVIEWS

29 DBISAM 1.12
Product Review by Wes Peterson

DEPARTMENTS

2 Delphi Tools
5 Delphi News
33 File | New by Alan C. Moore, Ph.D.



Delphi TOOLS

New Products and Solutions

D C AL CODA Releases EditorPro 1.0

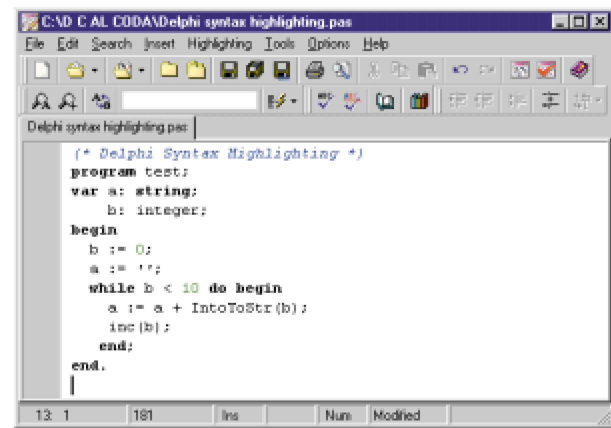
D C AL CODA released *EditorPro 1.0*, a full-featured, multi-tabbed text editor with syntax highlighting and advanced features for 10 languages and multiple file types.

EditorPro 1.0 centralizes your text editing needs, from simple text files to several programming languages: Object Pascal (Delphi), ANSI C (C/C++), HTML, INI files, Java, ObjectPal (Paradox 9), Perl, SQL, and Visual Basic. Benefits include fully customizable Editor Options, Color Syntax Highlighting, Code Templates, Key Assignments (keyboard shortcuts), unlimited Undo/Redo, Grouped Undo/Redo, Undo After Save, Bookmarks, Block Indent/Unindent, Print Preview, Active Spell, extensive Find and Replace with history, Regular Expressions, and Scope and direction.

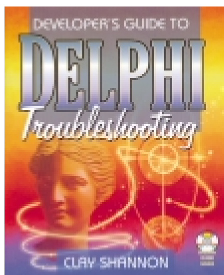
Spelling dictionaries (18 total) are included for Delphi, Paradox, American English, British English, Danish, Dutch, French, German, Italian, Latin, Math, Medical, Norwegian, Polish, Spanish, and Swedish. Additionally, the 30,000-word *Roget's Thesaurus* and 160,000-word Definitions Dictionary provide instant access to synonyms and antonyms, as well as the meaning and context of words.

EditorPro Suite adds a System Tray-based application, Tray Suite, which offers instant access to on-the-fly spell checking, Definitions Dictionary, *Roget's Thesaurus*, and EditorPro. Tray Suite can analyze any word, paragraph, or document from any application, text editor, HTML editor, or database.

D C AL CODA
Price: EditorPro, US$29.95 for a single-user license; EditorPro Suite, US$49.95 for a single user license; network and site licenses are available.
Phone: (800) 656-5443
Web Site: http://www.dcalcoda.com



Developer's Guide to Delphi Troubleshooting
Clay Shannon
Wordware Publishing, Inc.



ISBN: 1-55622-647-0
Price: US$49.99
(495 pages, CD-ROM)
http://www.wordware.com

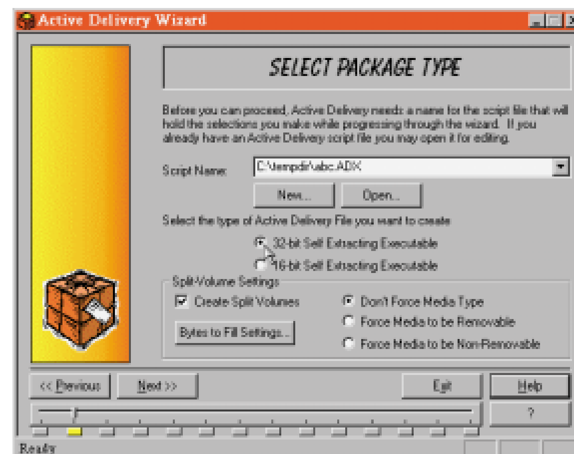
Inner Media Ships DynaZIP 4.0 and Active Delivery 2.0

Inner Media, Inc. began shipping the next generation of zip-compatible data compression toolkits/components. Included in this new release are *DynaZIP-Complete 4.0*, *DynaZIP-AX 4.0*, and *Active Delivery 2.0*.

DynaZIP-AX 4.0 and DynaZIP-Complete 4.0 provide zip-compatible data compression with a feature set compatible with a range of development tools and languages. New features include full multi-threaded operations, memory-to-memory operations, and more. They both lend themselves well to automated environments, such as Web servers, backup systems, etc., and are fully compatible with ASP.

DynaZIP-Complete 4.0 provides DLL, VBX, OCX, ActiveX, VCL, and DZ-Easy interfaces, supporting 16- and 32-bit applications. DynaZIP-AX 4.0 provides a pair of ActiveX components, one each for Zip and Unzip, and is designed for use with Delphi, VB, Access, Visual FoxPro, Visual C++, and any other 32-bit programming environment that supports ActiveX components. The components may be redistributed royalty-free, and include full documentation and a collection of sample source code.

Active Delivery 2.0, a self-extracting zip toolkit, is well-suited for use on Web sites where unattended delivery of custom data is required. As with prior releases, Active Delivery gives the programmer a path to creating self-extracting zip files that have many of the features of setup/install programs. The main difference is that these "packages" are created under program control, and are therefore extremely customizable, and may be created on-demand. In addition, Active Delivery 2.0 is fully multi-threaded.



Inner Media, Inc.
Price: DynaZIP-Complete 4.0, US$299; DynaZIP-AX 4.0, US$149; and Active Delivery 2.0, US$249.
Phone: (800) 962-2949
Web Site: http://www.innermedia.com

Delphi TOOLS

New Products and Solutions

DT Software Releases dtSearch 5.2

DT Software, Inc. introduced ***version 5.2 of dtSearch, dtSearch Web***, and ***dtSearch Text Retrieval Engine***. The new releases offer developers and large enterprises more flexible handling of data sources across PCs, networks, intranets, and the Internet.

With over two dozen text search options, dtSearch instantly searches through gigabytes of text. It includes built-in file and image viewers for popular file types, and automatically recognizes, searches, and displays documents with search hits highlighted. dtSearch 5.2 features more seamless support for heterogeneous language environments for international organizations and other mixed-language users.

dtSearch Web 5.2 adds an easier, frames-based user interface for navigating search results and retrieved documents. It also adds support for highlighting hits in PDF files through the browser interface, similar to the way dtSearch Web highlighted hits in HTML documents. Additionally, it includes a simple ASP interface that makes it easy to add customized search features to a Web site.

The dtSearch Text Retrieval Engine allows developers to embed dtSearch text retrieval technology in commercially distributed or in-house products for the PC, LAN, Internet, or an intranet. dtSearch Text Retrieval Engine 5.2 has more sample source code and enhanced Delphi, C++, Visual C++, and Visual Basic APIs, including ActiveX interfaces to data sources using SQL, ADO, MAPI, and more.

Search features in dtSearch products include natural language with relevancy-ranking by hit density and rarity; fuzzy, adjustable from 1 to 10 at the time of a search for OCR and typographical errors; thesaurus, including synonym or concept expansion based on a comprehensive built-in and/or user-added thesaurus; phonic; proximity; phrase; wildcard (two types); stemming; numeric range; database field; variable term weighting; filtering by filename, date, and size; query expand, narrow, and exclude; alphabet customization for non-English character sets; and a scrolling list of indexed words.

DT Software, Inc.
Price: dtSearch, from US$199; dtSearch Web and dtSearch Text Retrieval Engine, from US$999.
Phone: (800) IT-FINDS or (703) 413-3670
Web Site: http://www.dtsearch.com

Dart Announces New PowerTCP Tools

PowerTCP, **Dart Communications**' Internet toolkit for building TCP/IP applications, has been unbundled and re-engineered as the new ***PowerTCP Tool*** product line. Individually packaged, royalty-free ActiveX components, the all new PowerTCP Tools include the FTP Tool, Winsock Tool, Telnet Tool, Emulation Tool, and Server Tool. At press time, Dart's Web Tool was scheduled for release in late May 1999, and the Mail Tool and SNMP Tool were scheduled for release later in the summer.

PowerTCP Tools allow software developers to write Internet applications in a fraction of the time it would take to construct networking code from scratch. 32-bit ATL controls for TCP, FTP, POP3, SMTP, RAS, Telnet, SNMP, VT320, UDP, Server, and HTTP/S eliminate the need for researching, developing, and testing at the socket level. Included samples for Delphi, C++Builder, Visual Basic, Visual C++, and PowerBuilder allow developers to write high-performance Internet applications right out of the box.

Additional benefits of new PowerTCP Tools include a modular, multi-tier architecture for flexibility in application design, Winsock 2 compatibility, and the addition of blocking behavior for scripting applications.

Dart Communications
Price: From US$99 (PowerTCP FTP Tool).
Phone: (315) 431-1024
Web Site: http://www.dart.com

ABACO Releases DbCAD dev 1.5

ABACO srl Mantova released ***DbCAD dev 1.5***, a component library for GIS, MAPPING, GPS, and CAD applications, using Delphi, C++, VB, and other development tools.

DbCAD dev allows your application to manage an integrated graphic window (multi-instance available) with zoom, pan, entity pick, and dynamic overview commands with event control, by using the OCX component. DbCAD dev is available for most standard 2D vector and raster formats, including AutoCAD DWG (R14 or lower), DXF (R14 or lower), ESRI Shapefile, WMF, TIF, BMP, RLE, and RLC. It's also possible to load a custom installable driver to use the unsupported raster and vector formats within the DbCAD dev graphic window.

With DbCAD dev, you can overlay images by using transparency effects, and partially load drawings and images. Import, export, create, select, and edit all the 2D vector entities, such as lines, polylines, polygons, arcs, blocks, text, etc., including their properties. Filled polygons are available with various hatch styles (also user-defined) and transparency effects. Display and overlay real-time vector entity animation (GPS). TTFonts are supported.

In addition to graphic manipulation functions, DbCAD dev provides intelligent functions to link your database records with vector entities, and allows spatial analysis and query.

ABACO srl Mantova
Price: From US$210 for Project OCX/ActiveX.
Phone: +39 (0)376-222181
Web Site: http://www.dbcad.com

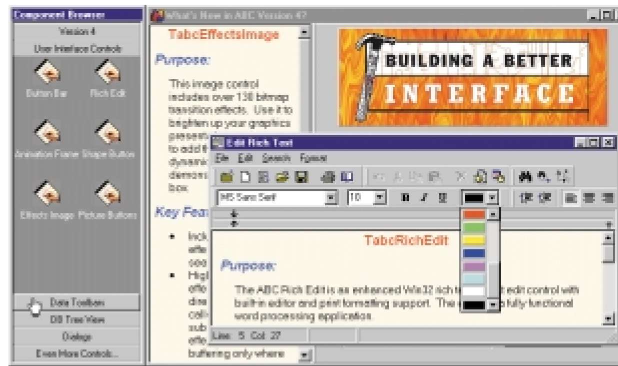
Delphi TOOLS

New Products and Solutions

Objective Announces Version 4.2 of ABC for Delphi

Objective Software Technology Pty Ltd. announced the release of *ABC for Delphi version 4.2*, a new version of its visual component library.

New components include a suite of custom menus with animation, bitmaps, and sound support, together with dockable toolbar menus and a customizable system menu. The new form control makes creating bold splash screens and custom applications easier. The new version also includes a new animation dialog box for adding custom AVI progress pop-ups, and an enhanced Outlook style button list control.

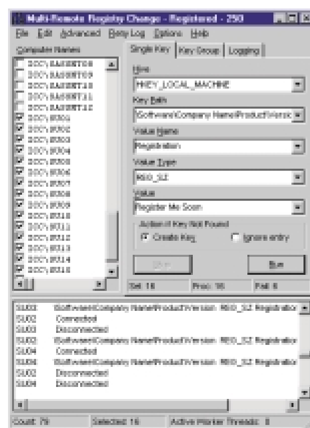


ABC for Delphi 4.2 contains 200 components and adds to ABC's array of time-saving data navigation, presentation, and exception-handling components. It ships on CD-ROM with 100MB of source code, sample programs, and run-time images for all versions of Delphi and C++Builder.

Objective Software Technology Pty Ltd.
Price: From US$149
Phone: +61 2 9955 3397
Web Site: http://www.obsof.com

Eytcheson Software Releases Multi-Remote Registry Change 3.0

Eytcheson Software released *Multi-Remote Registry Change 3.0*, a tool for remotely managing the registry on multiple Windows NT computers.



With support for adding, changing, and deleting every key and value type, as well as security and auditing, Multi-Remote Registry Change 3.0 allows you to remotely manage all areas of the registry. Several difficult management tasks, such as remotely changing user names and passwords for services, and modifying user rights on remote machines, are also supported. Multi-Remote Registry Change 3.0 lets you avoid buying and learning numerous command-line utilities and writing complicated, error-prone batch files.

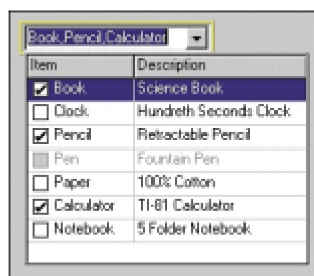
New ease-of-use enhancements, including drag-and-drop copying of keys and values from machine to machine, and drag-and-drop of information between operations, let you concentrate on the management task instead of on the spelling of the key or value. Multi-Remote Registry Change 3.0 provides the ability to change tens of thousands of keys per minute.

Eytcheson Software
Price: US$35 per administrator, plus US$2 per managed computer (discounts available for volume purchases).
Phone: (316) 332-4604
Web Site: http://www.eytcheson.com

Res-cue Offers Res-cue Mate 1.04

Res-cue (Resourceful Components for User Ease) announced *Res-cue Mate 1.04*, a package consisting of 26 components for Delphi.

Res-cue Mate components provide Focus technology, a programming technique that gives the developer the ability to show visual cues when a control receives focus. With Res-cue Mate, developers need not rely on Windows' selected highlighted text to show an edit control has focus.



Res-cue Mate 1.04 allows developers to customize properties for the focus rectangle, customize *TLabels* associated to a control, use Focus technology on existing controls (any descendant of *TWinControl* can have this ability by using the *TrscMasterFocus* component), and have custom visual cues.

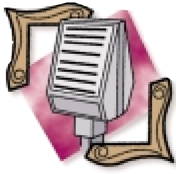
Edit Button technology gives the developer the ability to add multiple buttons in an edit control. This feature allows multiple buttons within the frame of the edit control, and enables developers to add and delete buttons at design time. All properties and events of the buttons added to an edit control can be edited at design time. Also, each button contains a *HotKey* property to assign keystrokes. Finally, each button contains a *Position* property, used to set the order in which the button appears in the edit control.

Res-cue
Price: Standard (Delphi 3 and 4), US$75; Professional (Delphi 3 and 4), US$125; Professional Site License, US$275.
E-Mail: sales@res-cue.com
Web Site: http://www.res-cue.com

News

LINE

September 1999



Inprise and Microsoft Confirm Commitment of Inprise Tools for Windows

Scotts Valley, CA — Microsoft Corp. and Inprise Corp. announced the completion of a set of strategic technology and licensing agreements that will be the foundation for a long-term alliance between the two companies. The announcement includes a US$25-million purchase by Microsoft of shares of Inprise preferred stock.

Key components of the arrangement include Inprise's commitment to support the Windows 2000 operating system, including the COM+ and the Windows DNA architecture; to license the latest version of the MFC; and to license the latest version of the Windows platform SDK.

In turn, Microsoft made a long-term commitment to provide Inprise with technologies related to the Windows platform and Internet technologies. Microsoft also paid Inprise US$100 million for the rights to use Inprise-patented technology in Microsoft products, and to settle a number of long-standing patent and technology licensing issues. The total value of the investment and payment to Inprise is US$125 million.

Inprise Stockholders Elect New Board Member

Scotts Valley, CA — Stockholders of Inprise Corp. elected C. Robert Coates, president and chief executive officer of Management Insights, Inc., a tax consulting firm, as a new member of the company's board of directors. Stockholders also re-elected Stephen J. Lewis, chief executive officer of All Bases Covered, an information technology consulting company, to the board. The votes to elect both men to three-year terms occurred at the company's annual stockholders' meeting.

During the annual meeting, Dale Fuller, who was named interim president and CEO in April, updated stockholders on the company's progress in formulating its long-term strategy. Fuller announced that the company's divisionalization has been put on hold as the company reviews its options. John Floisand, former president of the Borland division, has been named senior vice president of worldwide sales, a post he held at Inprise from 1996 through 1998. Fuller also announced that JoAnne M. Butler has been appointed vice president, general counsel, and secretary. Butler served as associate corporate counsel from 1993 to 1997, when she was promoted to corporate counsel. She replaces Hobart McK. Birmingham, who is currently serving as interim chief administrative officer.

Also at the annual meeting, stockholders approved an amendment to the 1997 Stock Option Plan to increase the number of common shares reserved for issuance by 1,500,000 shares; approved the 1999 Employee Stock Option Purchase Plan, including the reservation of 800,000 shares for the plan; and ratified the appointment of PricewaterhouseCoopers as independent auditors for fiscal year 1999.

Shaman Offers Complementary Solution to Microsoft SMS 2.0 Y2K Remediation

San Francisco, CA — Shaman Corp. announced the interoperability of Enterprise Shaman with Microsoft Systems Management Server (SMS) 2.0 to provide corporations with a comprehensive Y2K remediation solution.

Microsoft announced last week that they will help enterprises achieve Y2K compliance by offering a free 120-day trial of SMS 2.0. In line with Microsoft, Shaman announced features that enable SMS' electronic software distribution system to deliver Y2K compliance data, bug fixes, and updates directly from Shaman's comprehensive Knowledge-base.

The Shaman Knowledge-base contains Y2K information for over 200 software publishers, and is continually updated by Shaman's SRM analysts.

Enterprise Shaman supplies the IT manager with a complete software and hardware inventory of their network, informs them of the Y2K compliance status for their software, and delivers Y2K updates as they are released by software vendors. Enterprise Shaman also includes a testing feature that analyzes the BIOS on every desktop and delivers comprehensive compliance reports. For more information on Shaman, visit http://www.shamancorp.com.

Sun Selects VisiBroker Object Request Broker for CORBA Support

Atlanta, GA — Sun Microsystems selected Inprise's VisiBroker as the object request broker for its Solstice Enterprise Manager software. The integration of Common Object Request Broker Architecture and Solstice Enterprise Manager allows organizations to manage additional applications and devices on their network.

VisiBroker provides out-of-the-box tools for building distributed object computing applications that are transparent and platform independent. Linking VisiBroker with Solstice Enterprise Manager will enable customers to tie together disparate systems to a common management platform.

Solstice Enterprise Manager is focused on delivering scalable, reliable, and powerful network management primarily to telecommunication organizations and network service providers. Solstice Enterprise Manager is a key component of Sun's Service-Driven Network strategy. It can be extended to address specific customer requirements for the delivery and management of new and timely services. Out-of-the-box consolidated management supports protocols including SNMP, CMIP, RPC, and Java agents to enable a consistent approach to management policies and procedures. For more information, visit http://sun.com/sem.

table of CONTENTS

On the Cover

Delphi / Office 97 / Outlook 98 / COM

By Bill Todd

Outlook from Delphi

Automating Microsoft Outlook

Microsoft Office 97 appears to be five well-integrated applications. It is, in fact, much more. Office 97 was created using Microsoft's Component Object Model (COM). The Office applications are composed of a series of COM servers you can access from your Delphi applications using Automation (formerly known as OLE Automation). Beginning with Outlook 98, this article series will explore the object model of each of the Office applications — and how you can use them from Delphi.

The Outlook object model consists of objects and collections of objects (see Figure 1). The top-level object in Outlook 98 is the **Application** object. The **Application** object is the root of the object tree and provides access to all the other Outlook objects. The **Application** object is unique in that it's the only object you can gain access to by calling *CreateOleObject* from a Delphi (or any other) application. Next comes the **NameSpace** object, which provides access to a data source. The only available data source in Outlook 98 is the MAPI message store.

The **MAPIFolders** collection is just that — a collection of MAPI folders. You can think of collections as arrays of objects, somewhat like a Delphi *TList*. However, collection objects can be referenced by name or number. The **MAPIFolder** object in Figure 1 represents one of the folders in the **MAPIFolders** collection. Each **MAPIFolder** contains a **Folders** collection, and each of these contains an **Items** collection that contains the items appropriate to that folder. For example, the Contacts folder contains contact items.

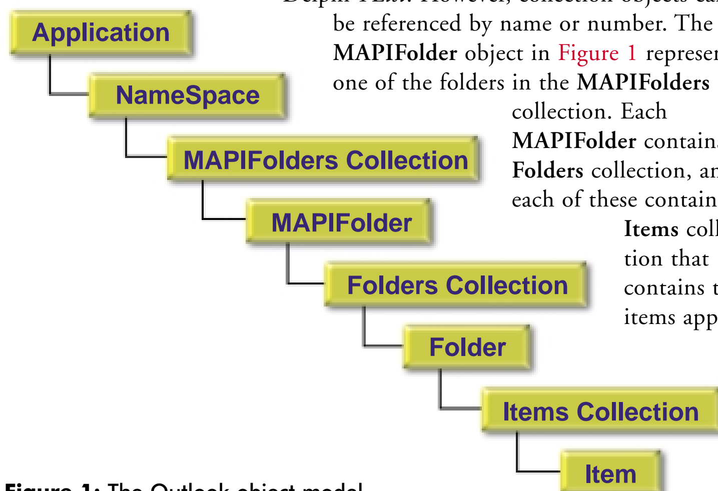


Figure 1: The Outlook object model.

Figure 2 shows the main form of a Delphi project that displays the **MAPIFolders** collection, the **Folders** collection of the MAPI Personal folder, and the **Items** in the Contacts folder. Listing One (on page 8) displays the code from the **Open Outlook** button's *OnClick* event handler.

The code in Listing One begins by declaring four Variant variables for use as references to various Outlook objects. The call to *CreateOleObject* loads the Outlook server and returns a reference to the **Application** object. The parameter passed to *CreateOleObject*, *Outlook.Application*, is the class name Outlook registers itself as when it's installed. Using the **Application** object, you can get a reference to any other Outlook object.

Calling the **Application** object's **GetNameSpace** method returns a reference to the **NameSpace** passed as a parameter. Using the MAPI **NameSpace** reference variable, *Mapi*, the code loops through the **MAPIFolders** collection and adds the name of each folder to the MapiList listbox. As with all objects in object-oriented programming, Outlook objects have properties, methods, and events. The **Count** property of the **Folders** collection is used to limit the number of times the **for** loop executes. All collections have a **Count** property to provide the number of objects in the collection. Each **Folder** in the **MAPIFolders** collection also has a **Name** property.

As you can see in Figure 2, the **MAPIFolders** collection contains two folders, Microsoft Mail Shared Folders and Personal Folders. The following statement gets a reference to the Personal Folders collection from the **MAPIFolders** collection. While the **for** loop that displayed the names of the MAPI Folders accessed the **MAPIFolders** collection by number, the statement:

```
Personal := Mapi.Folders('Personal Folders');
```

indexes the collection by name. The next **for** loop uses the reference to the Personal Folder to display the names of all the folders in its **Folders** collection in the second listbox in Figure 2. The code then gets a reference to the Contacts folder and uses it to loop through the Contacts folder's **Items** collection. One of the properties of a Contact item is **FullName**; this property is added to the third listbox to display the names of the contacts.

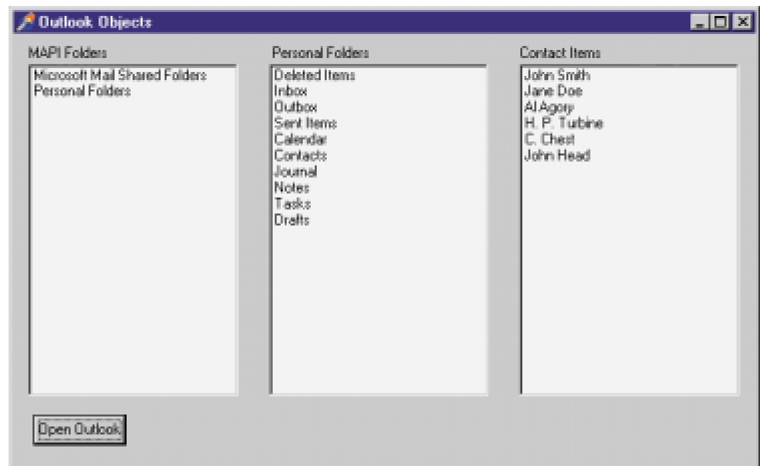


Figure 2: The **MAPIFolders** collection displayed in a Delphi form.

Clearly, the secret to working with Outlook 98 from your Delphi applications is understanding the Outlook object hierarchy and the properties, methods, and events of each object. Outlook 97 includes a Help file, VBAOUTL.HLP, that contains this information; however, I have been unable to find it on the Outlook 98 CD. Fortunately, very little has changed in Outlook 98. (Outlook 2000 is a different story, and will be the topic of a future article.)

Working with Contacts

Listing Two (beginning on page 8) shows the *OnClick* event handler from the LoadTbl project that accompanies this article (all examples referenced in this article are available for download; see end of article for details). This code demonstrates how to search the Outlook Contacts folder for the records you wish to select, and copy them to a database table.

As in the example shown in Listing One, this one begins by getting the **Application** object and the MAPI **NameSpace** object. Next, a reference is obtained using the statement:

```
ContactItems := Mapi.Folders('Personal Folders').
                 Folders('Contacts').Items;
```

This statement demonstrates how you can chain objects together using dot notation to get a reference to a low-level object without having to get individual references to each of the higher-level objects. In this case, five levels of intervening objects are specified to get to the **Items** object of the Contacts folder. These objects are:

- The MAPI **NameSpace** object
- The **Folders** collection
- The **Personal Folders** object
- The **Folders** collection
- The **Contacts** object

You can use this notation to get a reference to any Outlook object in a single statement. The next new feature of this method is the call to the **Find** method of the **ContactItems** collection. Almost all collection objects have a **Find** method you can use to locate a particular item in the collection using one or more of its properties. In this example, the statement:

```
CurrentContact := ContactItems.Find(' [CompanyName] = ' +
                   QuotedStr('Borland International'));
```

finds the first contact item where the value of the **CompanyName** property is equal to `Borland International`. If no matching item is found, the Variant *CurrentContact* will be empty. The **while** loop inserts a new record into the database table, and assigns each of the Contact item's properties to the corresponding field in the table. The **while** loop continues until *CurrentContact* is empty, indicating that no more items matching the search criteria can be found. At the end of the **while** loop, the call to **FindNext** finds the next matching record, if there is one. If no record is found, *CurrentContact* is set to empty and the loop terminates.

Creating new Contact folders and records is just as easy. Suppose you want to copy all your Contact records for Borland employees into a new folder. The code in Listing Three (on page 9) from the NewFolder sample project will do the job.

This method begins by getting the **Application**, MAPI **NameSpace**, and Contacts folder's **Items** object. Next, it uses a **for** loop to scan the **Folders** collection looking for the Borland Contacts folder. If the folder is found, its number is assigned to the *ToRemove* variable. The Borland Contacts folder is deleted by calling the **Folders** collection's **Remove** method and passing the *ToRemove* variable as the parameter.

Next, a call to the **Folders** collection's **Add** method creates the Borland Contacts folder. **Add** takes two parameters. The first is the name of the folder to be created. The second parameter is the folder type and can be *olFolderCalendar*, *olFolderContacts*, *olFolderInbox*, *olFolderJournal*, *olFolderNotes*, or *olFolderTasks*. To find the values of these and any other constants you need, search the VBAOUTL.HLP file for Microsoft Outlook Constants. The next statement gets a reference to the new Borland Contacts folder and stores it in the *BorlandContacts* variable.

A call to the Contacts folder's **Items** collection's **Find** method locates the first record for a Borland employee. The **while** loop is used to iterate through all the Borland employees in the Contacts folder. At the top of the loop, a new record is added to the Borland Contacts folder by calling the folder's **Items** collection's **Add** method.

Add takes no parameters; it simply inserts a new empty record and returns a reference to the new record, which is saved in the *NewContact* variable. The statements that follow assign values

from the existing record to the new one. Finally, the new record's **Save** method is called. This is a critical step. If you don't call **Save**, no errors will be generated — but there will be no new records in the folder. When the **while** loop terminates, Outlook is closed by assigning the constant *Unassigned* to the *OutlookApp* variable.

Other Outlook Objects

The **Folders** collection of the **Personal Folder** object contains the following folders:

- Deleted Items
- Inbox
- Outbox
- Sent Items
- Calendar
- Contacts
- Journal
- Notes
- Tasks
- Drafts

You can work with the **Items** collection of any of these folders using the same code shown for working with Contacts. Only the properties of the items are different. Listing Four (on page 9) shows a method that copies to a Paradox table all appointments that are all-day events and whose start date is greater than 4/27/99. This example copies the *Start*, *End*, *Subject*, and *BusyStatus* properties to the table. Note that this example uses a more sophisticated *find* expression than previous examples. Find supports the >, <, >=, <=, = and <> operators, as well as the logical operators **and**, **or**, and **not**, which allows you to construct complex search expressions.

Conclusion

Delphi applications can easily act as Automation clients, allowing your applications to interact with the Microsoft Office Suite applications in any way you wish. Using Outlook you can extract contact information to update a central database, add new contacts derived from other sources, create new folders, and add items of any type. One of Outlook's limitations is its lack of a powerful reporting tool. With a Delphi application, you can provide much more powerful reporting capabilities for Outlook data. With a basic understanding of the Outlook object model and a copy of the VBAOUTL.HLP help file, you are well on your way. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\SEP\DI9909BT.

Bill Todd is president of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is a Contributing Editor of *Delphi Informant*, co-author of four database-programming books and over 60 articles, and a member of Team Borland, providing technical support on the Borland Internet newsgroups. He is a frequent speaker at Borland Developer Conferences in the US and Europe. Bill is also a nationally known trainer and has taught Paradox and Delphi programming classes across the country and overseas. He was an instructor on the 1995, 1996, and 1997 Borland/Softbite Delphi World Tours. He can be reached at bill@dbginc.com, or (602) 802-0178.

Begin Listing One — Displaying Outlook objects

```
procedure TForm1.OpenBtnClick(Sender: TObject);
var
  OutlookApp,
  Mapi,
  Contacts,
  Personal:   Variant;
  I:           Integer;
begin
  { Get the Outlook Application object. }
  OutlookApp := CreateOleObject('Outlook.Application');
  { Get the MAPI NameSpace object. }
  Mapi := OutlookApp.GetNameSpace('MAPI');
  { Loop through the MAPI Folders collection and add the
    Name of each folder to the listbox. }
  for I := 1 to Mapi.Folders.Count do
    MapiList.Items.Add(Mapi.Folders(I).Name);
  { Get the Personal folder from the MAPI folders
    collection. }
  Personal := Mapi.Folders('Personal Folders');
  { Loop through the Personal Folders Collection and add
    the name of each folder to the listbox. }
  for I := 1 to Personal.Folders.Count do
    PersonalList.Items.Add(Personal.Folders(I).Name);
  { Get the Contacts folder from the Personal Folders
    collection. }
  Contacts := Personal.Folders('Contacts');
  { Loop through the Contacts folder's Items collection
    and add the FullName property of each Item
    to the listbox. }
  for I := 1 to Contacts.Items.Count do
    ContactsList.Items.Add(Contacts.Items(I).FullName);
  { Close Outlook. }
  OutlookApp := Unassigned;
end;
```

End Listing One

Begin Listing Two — Searching for contacts

```
procedure TLoadTableForm.LoadBtnClick(Sender: TObject);
var
  OutlookApp,
  Mapi,
  ContactItems,
  CurrentContact: Variant;
begin
  { Get the Outlook Application object. }
  OutlookApp := CreateOleObject('Outlook.Application');
  { Get the MAPI NameSpace object. }
  Mapi := OutlookApp.GetNameSpace('MAPI');
  { Get the Items collection from the Contacts folder. If
    you don't do this, FindNext will not work. }
  ContactItems := Mapi.Folders('Personal Folders').
                    Folders('Contacts').Items;
  { Load Contacts into table. }
  with ContactTable do begin
    EmptyTable;
    Open;
    DisableControls;
    CurrentContact :=
      ContactItems.Find('[CompanyName] = ' +
        QuotedStr('Borland International'));
    while not VarIsEmpty(CurrentContact) do begin
      Insert;
      FieldByName('EntryId').AsString :=
        CurrentContact.EntryId;
      FieldByName('LastName').AsString :=
        CurrentContact.LastName;
      FieldByName('FirstName').AsString :=
        CurrentContact.FirstName;
      FieldByName('CompanyName').AsString :=
        CurrentContact.CompanyName;
      FieldByName('BusAddrStreet').AsString :=
        CurrentContact.BusinessAddressStreet;
```

```
      FieldByName('BusAddrPOBox').AsString :=
        CurrentContact.BusinessAddressPostOfficeBox;
      FieldByName('BusAddrCity').AsString :=
        CurrentContact.BusinessAddressCity;
      FieldByName('BusAddrState').AsString :=
        CurrentContact.BusinessAddressState;
      FieldByName('BusAddrPostalCode').AsString :=
        CurrentContact.BusinessAddressPostalCode;
      FieldByName('BusinessPhone').AsString :=
        CurrentContact.BusinessTelephoneNumber;
      Post;
      CurrentContact := ContactItems.FindNext;
    end; // while
    EnableControls;
  end; // with
  { Close Outlook. }
  OutlookApp := Unassigned;
end;
```

End Listing Two

Begin Listing Three — Creating a Contacts folder and new contacts

```
procedure TCreateFolderFrom.CreateBtnClick(Sender: TObject);
const
  olFolderContacts = 10;
  olContactItem = 2;
var
  OutlookApp,
  Mapi,
  NewContact,
  BorlandContacts,
  ContactItems,
  CurrentContact:  Variant;
  I,
  ToRemove:        Integer;
begin
  { Get the Outlook Application object. }
  OutlookApp := CreateOleObject('Outlook.Application');
  { Get the MAPI NameSpace object. }
  Mapi := OutlookApp.GetNameSpace('MAPI');
  { Get the Items collection from the Contacts folder. If
    you don't do this, FindNext will not work. }
  ContactItems := Mapi.Folders('Personal Folders').
                    Folders('Contacts').Items;
  { Remove the test folder. }
  ToRemove := 0;
  for I := 1 to Mapi.Folders('Personal Folders').
                Folders.Count do
    if Mapi.Folders('Personal Folders').Folders(I).Name =
         'Borland Contacts' then
      begin
        ToRemove := I;
        Break;
      end; // if
  if ToRemove <> 0 then
    Mapi.Folders('Personal Folders').
      Folders.Remove(ToRemove);
  { Create a new folder. }
  Mapi.Folders('Personal Folders').
    Folders.Add('Borland Contacts', olFolderContacts);
  BorlandContacts := Mapi.Folders('Personal Folders').
                       Folders('Borland Contacts');
  { Load Contacts into new folder. }
  CurrentContact := ContactItems.Find('[CompanyName] = ' +
                      QuotedStr('Borland International'));
  while not VarIsEmpty(CurrentContact) do begin
    { Add a new item to the folder. }
    NewContact := BorlandContacts.Items.Add;
    { Assign values to the fields in the item record. }
    NewContact.FullName := 'John Doe';
    NewContact.LastName := CurrentContact.LastName;
    NewContact.FirstName := CurrentContact.FirstName;
    NewContact.CompanyName := CurrentContact.CompanyName;
    NewContact.BusinessAddressStreet :=
      CurrentContact.BusinessAddressStreet;
    NewContact.BusinessAddressPostOfficeBox :=
      CurrentContact.BusinessAddressPostOfficeBox;
    NewContact.BusinessAddressCity :=
      CurrentContact.BusinessAddressCity;
    NewContact.BusinessAddressState :=
      CurrentContact.BusinessAddressState;
    NewContact.BusinessAddressPostalCode :=
      CurrentContact.BusinessAddressPostalCode;
    NewContact.BusinessTelephoneNumber :=
      CurrentContact.BusinessTelephoneNumber;
    { Save the new record. }
    NewContact.Save;
    { Find the next record in the Contacts folder. }
    CurrentContact := ContactItems.FindNext;
  end; // while
  { Close Outlook. }
  OutlookApp := Unassigned;
end;
```

End Listing Three

Begin Listing Four — Reading Calendar folder

```
procedure TLoadTableForm.LoadBtnClick(Sender: TObject);
var
  OutlookApp,
  Mapi,
  ApptItems,
  CurrentAppt: Variant;
begin
  { Get the Outlook Application object. }
  OutlookApp := CreateOleObject('Outlook.Application');
  { Get the MAPI NameSpace object. }
  Mapi := OutlookApp.GetNameSpace('MAPI');
  { Get the Items collection from the Contacts folder. If
    you don't do this, FindNext will not work. }
  ApptItems := Mapi.Folders('Personal Folders').
                 Folders('Calendar').Items;
  { Load Contacts into table. }
  with ApptTable do begin
    EmptyTable;
    Open;
    DisableControls;
    CurrentAppt := ApptItems.Find('[Start] > ' +
                     '"4/27/99" and [AllDayEvent] = True');
    while not VarIsEmpty(CurrentAppt) do begin
      Insert;
      FieldByName('Start').AsDateTime := CurrentAppt.Start;
      FieldByName('Subject').AsString :=
        CurrentAppt.Subject;
      FieldByName('End').AsDateTime := CurrentAppt.End;
      FieldByName('Busy').AsBoolean :=
        CurrentAppt.BusyStatus;
      Post;
      CurrentAppt := ApptItems.FindNext;
    end; // while
    EnableControls;
  end; // with
  { Close Outlook. }
  OutlookApp := Unassigned;
end;
```

End Listing Four

By Mike Riley

Data at a Distance

Using Microsoft Remote Data Services

Microsoft Remote Data Services (RDS) enables Windows developers to remotely access server-based ODBC and OLE DB data sources without requiring the client to manually configure and establish an ODBC or BDE data connection. Seamless connectivity with server-based data sources over the Internet, even through firewalls, provides Windows developers with the ability to build rich user interfaces to data sources located anywhere HTTP will take them. RDS communication can even be established over HTTPS to ensure secure data transmissions over the Internet, which might be the case with an extranet Windows-based application.

As with many technologies, RDS has its advantages and disadvantages; the "pros and cons" of RDS in its current form include those listed here.

Pros:

- "Thin" client allows for easy updates to a light-weight core executable.
- Stateless connection provides efficient use of network bandwidth, and minimizes server load and resources in a highly distributed network environment, i.e. the Internet.
- LAN-like client/server data access inside compiled applications via standard HTTP communication, allowing for rich user interface elements, and data access over the Internet — even through firewalls.
- Easy primer for more advanced uses of DCOM over HTTP (a feature enabled in Windows NT Service Pack 4).

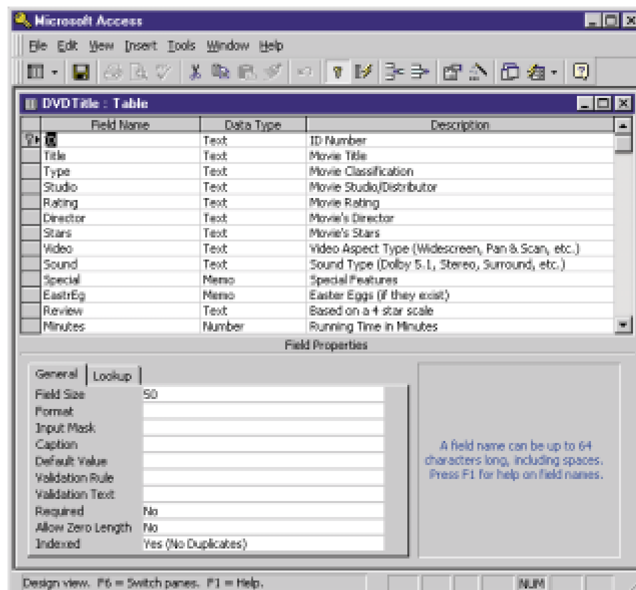


Figure 1: The demonstration database.

Cons:

- "Fat" client computing in terms of the overwhelming component installation. Application updates require a full replacement or binary patching of the client-based executables and DLLs.
- Stateless connections provide disconnected recordsets and all-or-nothing updates.
- Direct database access over HTTP may pose a security risk for inadequately monitored internal networks.
- RDS may be redundant and unnecessary now that DCOM over HTTP is possible.

RDS Requirements

Enabling easy, inexpensive data access does have its consequences. Following is a list of the client and server requirements and the URLs (where applicable) of where to obtain the components as of this writing.

Client requirements:

- DCOM 2.1 or higher for Windows 95 (Windows 98 users already have DCOM installed). DCOM for Windows 95 can be located at http://www.microsoft.com/com/dcom.asp.
- Internet Explorer 4.01 with Service Pack (SP) 1.
- MDAC (Microsoft Data Access Components), which includes Active Data Objects 2.0 and

RDS 2.0. MDAC Redistributables and SDK can be located at http://www.microsoft.com/data/download2.htm.

Server requirements:
- Windows NT 4.0 Workstation or Server with SP3 and NT Option Pack. If Windows NT 4.0 SP4 has been applied, MDAC 2.0 SP1 is required, and can be located at http://www.microsoft.com/data/mdac2.htm.
- Internet Information Server 4.0.
- Microsoft MDAC SDK (includes Visual C++ and Visual Basic client demonstrations that can be easily adapted to Delphi).
- MDAC-compliant data source (Microsoft SQL Server 6.5, ODBC, or OLE DB-compliant data source).

Development requirements:
- Delphi 2 or higher (Delphi 4 Professional or higher recommended).

Like most Microsoft technology updates, apply the requirements in the order in which Microsoft released them. For the client, this means installing Internet Explorer 4.01 and applying the IE4 service pack, followed by the MDAC install. Any deviation may result in DLL conflicts.

RDS is definitely a Windows 32-bit-only technology, just as Delphi 4 is definitely a Windows 32-bit-only development tool. If data distribution needs to exist in a heterogeneous client/server environment, look to Java, JDBC, CORBA, and IIOP instead. Quite frankly, the configuration and client maintenance of virtual machines and object request brokers will be almost as troublesome as RDS component installations, and the expensive licenses for such CORBA components will keep this option a costly corporate enterprise-wide-only solution, at least for the near future.

Luckily, installation of MDAC 2.0 and its dependencies is a straightforward and transparent process. Testing for a successful RDS installation on the client and server can be performed using Microsoft Internet Explorer to retrieve an Active Server Page test page designed to fetch data from an advworks.mdb Microsoft Access 97 data source. The default path to this .ASP test page is http://*server*/msadc/samples/adctest.asp, where *server* is the default name of the Web server running RDS. Naturally, the RDS SDK must be installed on the server for this URL to exist.

Experience has proven that the third-party ActiveX demonstration grid may not populate with return results even though RDS is correctly installed. There are a number of reasons for this, one of which might be that data source permissions haven't been set to allow the IIS Internet User account (typically IUSER_MACHINENAME, where MACHINENAME is the name of the server running IIS). The most elusive and disappointing one is that RDS doesn't support virtual Web server addresses. RDS servers must run the service from a primary default server name/IP address per machine. Hopefully, Microsoft will resolve this annoying limitation with the next MDAC service pack.

Configuring the Server Data Source

The first objective of any RDS project is to determine what the data source will be, and what fields need to be retrieved and/or modified. If the data schema needs to be created, use an ODBC-compliant desktop database product like Microsoft Access that can be upsized into a SQL database, such as Oracle or Microsoft SQL Server if necessary. RDS has been optimized for use with Microsoft SQL Server

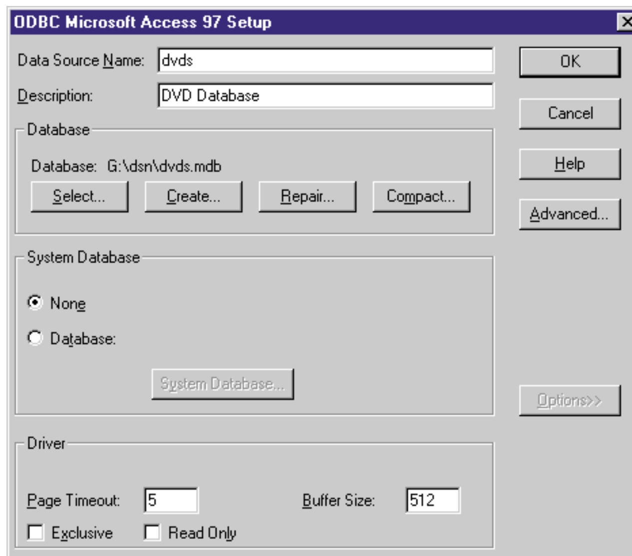


Figure 2: Connecting an Access 97 data source.

Figure 3: Successful RDS connection results.

6.5 or higher, and other OLE DB-compliant data sources. This is hardly surprising, considering the 100 percent Microsoft 32-bit environment that RDS requires.

After the data source has been defined as a system DSN (configured using the server's ODBC Control Panel applet), connection verification can be performed using the adctest.asp page and changing the DSN default value from the AdvWorks data source to the name of the DSN being tested. Remember to grant database access to the IIS Internet User account. The grid should populate with the table's field names and existing records if successful.

For this article, I created a DVD demonstration database (see Figure 1). Connecting this Access 97 data source via the ODBC Control Panel applet was elementary (see Figure 2). Once the data source was defined on the server, the RDS connection was tested using the adctest.asp page with successful connection results (see Figure 3).

Coding the Client Interface

Launch Delphi, begin a new project, and import the ADO and RDS 2.0 type libraries using the **Project | Import Type Libraries** menu option. Locate and add the Microsoft ActiveX Data Objects Recordset 2.0 Library (Version 2.0) and Microsoft Remote Data Services 2.0 (Version 1.5) type libraries to the project. These will import as ADODB_TLB.pas and RDS_TLB.pas units respectively (available for download; see end of article for details). If these type libraries don't exist, make sure the RDS SDK or RDS redistributables have been correctly installed by pointing the client's IE browser at the adctest.asp page on the server.

Once the type libraries have been imported, populate the form receiving the data with standard Windows controls. Note that RDS doesn't use the BDE, so Delphi data-aware controls won't be effective in dealing with RDS data. (If BDE-connected data over the Internet is a requirement, check out Dalco Technologies' dbOvernet at http://www.dbovernet.com.) Once the number of controls matches the number of display fields desired, connect each control to the appropriate RDS field. Refer to Listing One (beginning on page 13) for the example used for the DVD Access database.

Notice that the RDS objects are created, and a connection to the server is established, during the *FormCreate* event. Data isn't retrieved until the RDS *Refresh* method is called. The RDS *SQL* property must contain a query before calling the *Refresh* method. Otherwise, an empty set will be returned. The returned recordset is then passed to the ADO *Recordset* object where it can be further manipulated using the *MoveFirst*, *MoveLast*, *MoveNext*, *MovePrevious*, *Update*, *AddNew*, and *Delete* methods.

Note that none of the recordset client changes are replicated, nor permanently affect the original source data on the server, until the *SubmitChanges* method is called by the client. This is because RDS is a stateless connection and no data is transacted unless the client explicitly fires the events to do so.

Here's a summary of the client development steps:

- Import the ADO and RDS type libraries.
- Create the UI using standard Windows controls or other Delphi visual components (data-aware/data-bound controls don't work; use Dalco Technologies' dbOvernet for accessing BDE-enabled data sources and populating Delphi data-aware controls over the Internet; it works so well that I wrote an NT service demonstration for Dalco that helps capture the benefits of server-based RDS).
- Declare the ADO session, recordset, and RDS connection objects, and assign them to variables of type Variant.
- Supply the RDS connection object with an active RDS server and data source, and connect.
- Retrieve the recordset by passing a SQL query to the RDS server-based data source.
- Populate the client form's controls with the recordset results.
- If the application warrants read-write access, post any data changes via a batch update process to the RDS server.
- When finished with the session, close the connection and release the objects.

Running the Application

Assuming the server has been properly configured and successfully tested, running the client should produce similar results to those displayed in the DVD client screenshot shown in Figure 4. The client program can add, update, and delete records, as well as

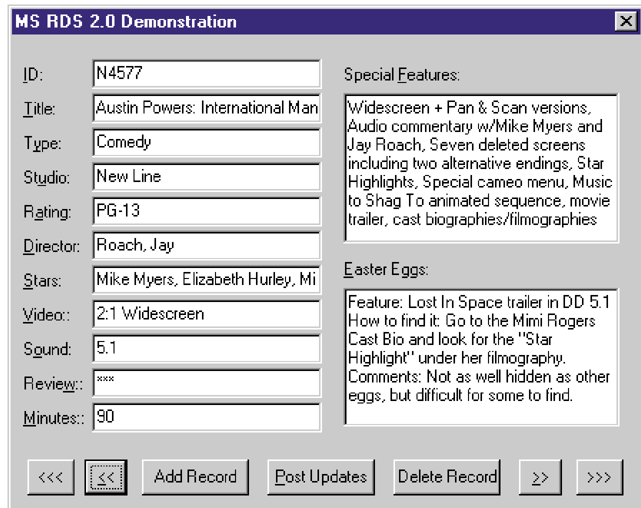


Figure 4: The demonstration DVD client at run time.

move between records in DBNavigator-like style. Exiting the application without first clicking the **Post Updates** button will discard any changes.

Conclusion

The future of RDS is unclear. While Microsoft has publicly committed to continuing support for the MDAC developer community, the effort required to make Windows 95 clients RDS-aware is substantial. In addition, MDAC version updates will perpetuate fat-client computing in a thin-client world. While Windows 98 and Windows 2000 clients benefit from having IE4, RDS, and DCOM pre-installed, RDS needs more revisions to overcome its current limitations, especially if it's intended to act like a traditional stateful client/server data connector over the Internet.

With DCOM over HTTP now available in Windows NT 4.0 SP4 and Windows 2000, ADO and RDS components that were once required to be installed and registered on the client can now be created as DCOM objects on the server, eliminating the need to redistribute bulky MDAC client components. Of course, allowing DCOM object instantiation of server objects could be a more troublesome security risk for Internet-exposed internal networks, because this capability can potentially expose for remote automation any registered object (including objects that can format hard drives or distribute passwords) on the server, not just data objects.

In the meantime, RDS provides Windows COM developers an inexpensive and exciting way to access, process, and store data anywhere in the world using a wealth of rich, well-established, visual components. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\SEP\DI9909MR.

Mike Riley is the Director of Internet Application Development for RR Donnelley & Sons, North America's largest printer. He actively participates in the company's Internet, intranet, and extranet strategies using a wide variety of Web-enabled technologies, including Delphi 4. Mike can be reached via his spam-shielded e-mail address, mikeriley@hotmail.com.

Begin Listing One — RDS 2.0 Demonstration

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls,ADOR_TLB, RDS_TLB;

type
  TMainForm = class(TForm)
    editID: TEdit;
    lblID: TLabel;
    Label1: TLabel;
    editTitle: TEdit;
    Label2: TLabel;
    editType: TEdit;
    Label3: TLabel;
    editStudio: TEdit;
    Label4: TLabel;
    editRating: TEdit;
    Label5: TLabel;
    editDirector: TEdit;
    Label6: TLabel;
    editStars: TEdit;
    Label7: TLabel;
    editVideo: TEdit;
    Label8: TLabel;
    editSound: TEdit;
    Label9: TLabel;
    editReview: TEdit;
    Label10: TLabel;
    editMinutes: TEdit;
    memoFeatures: TMemo;
    Label11: TLabel;
    Label12: TLabel;
    memoEggs: TMemo;
    btnNext: TButton;
    btnPrevious: TButton;
    btnRefresh: TButton;
    btnAdd: TButton;
    btnDelete: TButton;
    btnFirst: TButton;
    btnLast: TButton;
    procedure FormCreate(Sender: TObject);
    procedure btnNextClick(Sender: TObject);
    procedure btnPreviousClick(Sender: TObject);
    procedure btnRefreshClick(Sender: TObject);
    procedure RefreshRDSData;
    procedure UpdateRDSData;
    procedure AddRDSData;
    procedure DeleteRDSData;
    procedure btnFirstClick(Sender: TObject);
    procedure btnLastClick(Sender: TObject);
    procedure btnDeleteClick(Sender: TObject);
    procedure btnAddClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  end;

var
  MainForm: TMainForm;
  rdsDC: DataControl;  // Define RDS Datacontrol object.
  rstRS: Recordset;    // Define RDS Recordset object.

implementation

{$R *.DFM}

// Populate the MainForm's visual controls with the current
// record values. Note: <null> values are not checked and
// will cause a type cast error if one is encountered. For
// this demo, all <null> values in the demo database were
// converted to a 'None' string. To make this routine more
// robust, <null> values should be detected and values
// substituted accordingly before displaying the results in
// the text or memo boxes. Lastly, the Refresh, AddNew,
// Update and DeleteRDSData routines could be further
// optimized by assigning the Item string to the respective
// control's Tag property and iterate through each
// participating control in the form's collection, saving the
// explicitly redundant code in these procedures. In an
// effort to help new RDS programmers understand how RDS
// works and keep the demo as simple to understand as
// possible, this optimization opportunity was rejected.
procedure TMainForm.RefreshRDSData;
begin
  editID.Text := rstRS.Fields.Item['ID'].Value;
  editTitle.Text :=  rstRS.Fields.Item['Title'].Value;
  editType.Text := rstRS.Fields.Item['Type'].Value;
  editStudio.Text := rstRS.Fields.Item['Studio'].Value;
  editRating.Text := rstRS.Fields.Item['Rating'].Value;
  editDirector.Text := rstRS.Fields.Item['Director'].Value;
  editStars.Text := rstRS.Fields.Item['Stars'].Value;
  editVideo.Text := rstRS.Fields.Item['Video'].Value;
  editSound.Text := rstRS.Fields.Item['Sound'].Value;
  editReview.Text := rstRS.Fields.Item['Review'].Value;
  editMinutes.Text := rstRS.Fields.Item['Minutes'].Value;
  memoFeatures.Text := rstRS.Fields.Item['Special'].Value;
  memoEggs.Text := rstRS.Fields.Item['EastrEg'].Value;
end;

// Save any modifications made to the current record. As
// with the RefreshRDSData procedure, error checking for
// <null> values should be added to make the UpdateRDSData
// procedure more robust.
procedure TMainForm.UpdateRDSData;
begin
  with rstRS do begin
    Update('ID', editID.Text);
    Update('Title', editTitle.Text);
    Update('Type', editType.Text);
    Update('Studio', editStudio.Text);
    Update('Rating', editRating.Text);
    Update('Director', editDirector.Text);
    Update('Stars', editStars.Text);
    Update('Video', editVideo.Text);
    Update('Sound', editSound.Text);
    Update('Review', editReview.Text);
    Update('Minutes', editMinutes.Text);
    Update('Special', memoFeatures.Text);
    Update('EastrEg', memoEggs.Text);
  end;
end;

// Clear the form and populate the fields with spacer data
// so RDS won't complain when a user attempts to post
// invalid or <null> field data to the database.  Again,
// this should be optimized to replace null fields with
// valid data being inserted into the database with the
// SubmitChanges command.
procedure TMainForm.AddRDSData;
begin
  editID.Text := ' ';
  editTitle.Text :=  ' ';
  editType.Text := ' ';
  editStudio.Text := ' ';
  editRating.Text := ' ';
  editDirector.Text := ' ';
  editStars.Text := ' ';
  editVideo.Text := ' ';
  editSound.Text := ' ';
  editReview.Text := ' ';
  editMinutes.Text := '0';
  memoFeatures.Text := 'None';
  memoEggs.Text := 'None';
  with rstRS do begin
    AddNew('ID', editID.Text);
    with Fields do begin
      Item['ID'].Value := editID.Text;
      Item['Title'].Value := editTitle.Text;
      Item['Type'].Value := editType.Text;
      Item['Studio'].Value := editStudio.Text;
      Item['Rating'].Value := editRating.Text;
      Item['Director'].Value := editDirector.Text;
      Item['Stars'].Value := editStars.Text;
```

```
      Item['Video'].Value := editVideo.Text;
      Item['Sound'].Value := editSound.Text;
      Item['Review'].Value := editReview.Text;
      Item['Minutes'].Value := editMinutes.Text;
      Item['Special'].Value := memoFeatures.Text;
      Item['EastrEg'].Value := memoEggs.Text;
    end;
    Update('ID', editID.Text);
  end;
  rstRS.MoveLast;
  btnNext.Enabled := False;
  btnPrevious.Enabled := True;
end;

// Delete the current record. The deleted record will not
// be reflected in the server's database until the
// SubmitChanges event occurs.
procedure TMainForm.DeleteRDSData;
begin
  rstRS.Delete(adAffectCurrent);
  rstRS.MoveFirst;
  RefreshRDSData;
  btnNext.Enabled := True;
  btnPrevious.Enabled := False;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  try
    // Create RDS Datacontrol and Recordset objects.
    rdsDC := CoDataControl.Create;
    rstRS := CoRecordset.Create;
    // Execute the queries synchronously to aid debugging.
    // Switch to adcExecAsync to improve UI mechanics.
    rdsDC.ExecuteOptions := adcExecSync;
    rdsDC.FetchOptions := adcFetchBackground;
    // Replace 'server' with address of RDS server with
    // 'dvds' data source.
    rdsDC.Server := 'http:// server';
    // Connect string must contain DSN, can also contain
    // uid and pwd parameters.
    rdsDC.Connect := 'DSN=dvds';
    // Fetch the recordset and store the results in the
    // rstRS object.
    rdsDC.SQL := 'SELECT * FROM DVDTitle';
    rdsDC.Refresh;
    rstRS := rdsDC.Recordset as RecordSet;
    // Move client-side cursor to the first record in
    // the recordset.
    rstRS.MoveFirst;
    // Call RefreshRDSData procedure and populate text/memo
    // fields with the first record in the recordset.
    RefreshRDSData;
  except
    ShowMessage('Error in creating RDS objects.');
  end;
end;

procedure TMainForm.btnNextClick(Sender: TObject);
begin
  if rstRS.AbsolutePosition = rstRS.RecordCount-1 then
    begin
      btnNext.Enabled := False;
      btnLastClick(Application);
      ShowMessage('Last record in recordset reached.');
    end
  else
    begin
      btnPrevious.Enabled := True;
      // Save changes and move to the next record.
      UpdateRDSData;
      rstRS.MoveNext;
      RefreshRDSData;
    end;
end;

procedure TMainForm.btnPreviousClick(Sender: TObject);
begin
  if rstRS.AbsolutePosition = 2 then
    begin
      btnPrevious.Enabled := False;
      btnFirstClick(Application);
      ShowMessage('First record in recordset reached.');
    end
  else
    begin
      btnNext.Enabled := True;
      // Save changes and move to the previous record.
      UpdateRDSData;
      rstRS.MovePrevious;
      RefreshRDSData;
    end;
end;

procedure TMainForm.btnRefreshClick(Sender: TObject);
begin
  UpdateRDSData;
  rdsDC.SubmitChanges;
  rdsDC.Refresh;
  ShowMessage(
    'Recordset changes have been posted to the database.');
end;

procedure TMainForm.btnFirstClick(Sender: TObject);
begin
  // Save changes and go to the first record.
  UpdateRDSData;
  rstRS.MoveFirst;
  RefreshRDSData;
  btnPrevious.Enabled := False;
  btnNext.Enabled := True;
end;

procedure TMainForm.btnLastClick(Sender: TObject);
begin
  // Save changes and go to the last record.
  UpdateRDSData;
  rstRS.MoveLast;
  RefreshRDSData;
  btnNext.Enabled := False;
  btnPrevious.Enabled := True;
end;

procedure TMainForm.btnDeleteClick(Sender: TObject);
begin
  if Application.MessageBox(
       'Are you sure you want to delete this record? ' +
       Chr(10) + Chr(13) + 'The record will not be ' +
       'deleted from the database until the Post ' +
       'Updates button is selected.',
       'Delete Current Record',
       MB_OKCANCEL + MB_DEFBUTTON1) = IDOK then
    begin
      DeleteRDSData;
      ShowMessage('Record deleted from current recordset.');
    end;
end;

procedure TMainForm.btnAddClick(Sender: TObject);
begin
  UpdateRDSData;
  AddRDSData;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  // Release the Datacontrol and Recordset objects.
  rdsDC := nil;
  rstRS := nil;
end;

end.
```

End Listing One

By Gregory H. Deatz

Making a Hash of It

Setting Up an Abstract Hash Table

One of the primary things that application developers do is manage lists. We need to create a list of things and access the items in the list in some fashion.

More often than not, it's entirely adequate to access items by their index in the list. However, it's frequently desirable to access an item not by its integer index, but by some other key. To access elements of generic lists by key, we're forced to search for the element. Sometimes the list is unsorted, so we have to scan the entire list. Sometimes the list is sorted, so we can perform a binary search. In both cases, we can only expect a linear search (in the case of unsorted lists), or a logarithmic search (in the case of the sorted list).

There's another way to access elements of a list by key. It's called *hashing*. The notion of hashing recognizes the fact that a sorted list isn't necessarily what we need; we simply need virtually instant access to the element by its key. Theoretically speaking, a hash table comes in handy when the "universe" of possible keys is substantially larger than the number of elements that will actually be in the list. In essence, we want to extend the notion of an array to include the concept of an arbitrary key, instead of a "plain Jane" integer.

Hashing

There is quite a bit of theory behind hash tables, but the implementation of them is simple. The gist is that we create an array of buckets, and each bucket will contain a linked list of key/value pairs. To determine if a key/value pair belongs in a particular bucket, we use a hashing function. A hashing function takes a key and computes an integer index based on the key. This index is the index of a bucket.

The next most important consideration in a hash table is its size, or the number of buckets the hash table will contain. If we use too few buckets (i.e. the size of the array is too small), the hash table will perform no better than a linked list, and we'll only get linear-search performance. On the other hand, if we use too many buckets, the hash table will take up an unnecessary amount of space. So, a given hash function must be tailored directly to its associated hash table size. A hash function that returns a range of results from 0..1 would be useless to a table whose size is 256. Likewise, a hash function whose range is 0..255 could cause some major problems in a table whose size is 16.

In addition to tailoring the hash function to its associated hash table size, you must tailor the hash function to its expected input. That is, if one is hashing on integers, one's hash function will probably look decidedly different from a function based on strings.

Abstract Hash Table

Aside from the hash function and the size of the table itself, the rest of the hash table is generic. We need to be able to access a value by its associated key; delete a value from the table by its associated key; clear the hash table itself; determine if a given key exists in the table; determine how many elements are stored in the hash table; know how many elements are in a given bucket; etc.

It's time to write some code! Take a look at Hash.pas in Listing One (beginning on page 17). The following sections will provide instructions for the more important supporting objects and procedures.

THashItem

The *THashItem* object is never seen by the user of a hash table derived from *TCustomHashTable*; rather, it's a supporting structure that allows the hash table to store a linked list of all items in the table (*FNext*, *FPrev*), as well as a linked list of all items in its specific bucket (*FHNext*, *FHPrev*).

Each hash item is responsible for knowing its index in its hash table's hashing array (*FHashIndex*), and each hash item stores its associated *Key* and *Value*. Notice that the *Key* and *Value* properties are of type Variant, thus allowing our implementation of hash tables to be based on keys of any type, and store values of any type.

TCustomHashTable

The key methods in *TCustomHashTable* are listed below. Some are followed by a brief description.

This protected *FindItem* method returns the *THashItem* associated with a given key. If an item isn't found, *bQuiet* instructs the method to either be quiet and return a **nil** value, or complain loudly and raise an exception. A *HashVal* can be passed if the hashing function has already been computed:

```
function FindItem(const Key: Variant; bQuiet: Boolean;
  HashVal: Integer): THashItem
```

Another protected method, *HashError*, raises an *EHashError* exception with the passed error message:

```
procedure HashError(const ErrMsg: THashErrMsg);
```

Given a *Key*, find the appropriate entry in the table and get or set its value, respectively:

```
function GetValue(const Key: Variant): Variant;
```

```
procedure SetValue(const Key: Variant; Value: Variant);
```

How many buckets does the hash table contain?

```
function HashSize: Integer; virtual; abstract;
```

The virtual constructor/destructor for the class is as follows:

```
constructor: Create; virtual;
```

```
destructor: Destroy; override;
```

Given a *Key* value, compute the hash function, returning the "bucket" in which *Key* belongs:

```
function HashFunc(Key: Variant): Integer;
  virtual; abstract;
```

Add and remove a key/value pair to and from the table, respectively, as follows:

```
procedure AddItem(Key, Value: Variant);
```

```
procedure RemoveItem(Key: Variant);
```

Clear the hash table, as follows:

```
procedure Clear
```

Does an entry for *Key* exist in the table?

```
function KeyExists(Key: Variant): Boolean;
```

Given a specified bucket (either by index or key), return the number of items in the bucket. This function is particularly useful when testing a given hash function on an expected domain of inputs:

```
function BucketCountByIdx(const Idx: Integer): Integer;
```

```
function BucketCountByKey(const Key: Variant): Integer;
```

```
TStringKeyHashTable = class(TCustomHashTable)
protected
  function HashSize: Integer; override;
public
  function HashFunc(Key: Variant): Integer; override;
  property Count;
  property Size;
  property Value; default;
end;
```

Figure 1: Implementing a table to hash strings.

```
function TStringKeyHashTable.HashSize: Integer;
begin
  Result := 256;
end;

function TStringKeyHashTable.HashFunc(Key: Variant):
  Integer;
var
  st: string;
  i: Integer;
begin
  st := Key;
  Result := 0;
  for i := 1 to Length(st) do
    Inc(Result, Integer(st[i]));

  Result := Result mod Size;
end;
```

Figure 2: The hashing routines.

TCustomHashTable Properties

There are three properties exposed for our custom hash table:

- *Count* returns the current number of items stored in the hash table.
- *Size* is a "property hook" to the *HashSize* function.
- *Value* allows the developer to treat a hash table as an array of values, indexed by key, whatever that key may be.

Implementing a String Hash Table

We've discussed the generic functionality of a customizable hash table. Now it's time to look at a concrete example descended from *TCustomHashTable*. Not surprisingly, our example implements a table that hashes strings (see Figure 1).

Note that we are required to implement only two functions, *HashSize* and *HashFunc*, and we've made the three protected properties public. The implementations of the hashing routines are shown in Figure 2.

This example creates a hash table with 256 buckets, more than enough for most applications that require a hash table with strings. The hash function operates by summing up the ordinal values of each character in the string. Finally, to guarantee the hash function returns a valid value for the hash table size, it performs remainder arithmetic to get a value within the range of 0-255.

A Quick Example

That was easy! Let's watch it at work. Load the example application Example provided with this article (see end of article for download details) and run it.

Enter some key/value pairs by entering text in the appropriate text boxes. By clicking on keys in the list box, you can automatically search for keys that have already been entered. To verify that the hash function performs well, you can also see how many items are in each bucket.

Don't Take This Too Far

The theory behind hash tables demonstrates that a properly matched hash function and hash table size will produce, on average, constant time results. This means that, on average, a hash table performs much better than a sorted list. However, it's important to note the phrase "on average." In the worst case, a hash table will perform just as badly as an unsorted list of elements.

Hash tables are incredibly useful tools, but generally only useful when the developer has some knowledge of the domain of possible keys. Are the keys random? Are the keys based on English? What are the most commonly entered keys? These questions must be addressed at some level when designing a hash table.

Even more interesting is that a hash table performs very well when its size is substantially larger than the actual number of elements it will contain. Obviously, if we had a list of 1,024 or more key/value pairs, our *TStringKeyHashTable* would not perform in constant time. And, as the size of the list grows, a simple binary search of a sorted list would outperform it.

Conclusion

This article has presented an abstract class for designing just about any kind of hash table a developer could desire. Hash tables are extremely useful tools when dealing with a set of key/value pairs to which we require frequent and fast access.

We demonstrated a concrete example of a hash table based on string keys, and we also indicated that hash tables are not solve-all solutions; rather, they need to be custom-built for each particular case. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\SEP\DI9909GD.

Gregory Deatz is a senior programmer/analyst at Hoagland, Longo, Moran, Dunst & Doukas, a law firm in New Brunswick, NJ. His current focus is legal billing and case management applications. He is the author of FreeUDFLib, a free UDF library for InterBase written entirely in Delphi, and FreeIBComponents, a set of native InterBase components for use with Delphi 3.0. These tools can be found at http://www.interbase.com/downloads. He can be reached via e-mail at gdeatz@hlmdd.com, by voice at (732) 545-4717, or by fax at (732) 545-4579.

Begin Listing One — Hash.pas

```
unit Hash;

interface

uses
  SysUtils, Classes;

type
  THashItem = class(TObject)
  private
    FHashIndex: Integer;
    FNext, FPrev,                        // Global item list.
    FBucketNext, FBucketPrev: THashItem;  // Bucket list.
    FKey, FValue: Variant;
  public
    property HashIndex: Integer read FHashIndex;
    property Key: Variant read FKey;
    property Value: Variant read FValue write FValue;
  end;

  TBucket = record
    Count: Integer;
    FirstItem: THashItem;
  end;

  THashArray = array[0..0] of TBucket;
  PHashArray = ^THashArray;

  EHashError = class(Exception);

  THashErrMsg =
    (hKeyNotFound, hKeyExists, hIndexOutOfBounds);

  TCustomHashTable = class(TObject)
  private
    FItems: THashItem;
    FHash: PHashArray;
    FHashCount: Integer;
    function FindItem(const Key: Variant; bQuiet: Boolean;
      HashVal: Integer): THashItem;
    procedure HashError(const ErrMsg: THashErrMsg);
    function GetValue(const Key: Variant): Variant;
    procedure SetValue(const Key: Variant;
      Value: Variant);
  protected
    function HashSize: Integer; virtual; abstract;
    property Count: Integer read FHashCount;
    property Size: Integer read HashSize;
    property Value[const Key: Variant]: Variant
      read GetValue write SetValue;
  public
    constructor Create; virtual;
    destructor Destroy; override;
    function HashFunc(Key: Variant): Integer;
      virtual; abstract;
    procedure AddItem(Key, Value: Variant);
    procedure RemoveItem(Key: Variant);
    procedure Clear;
    function KeyExists(Key: Variant): Boolean;
    function BucketCountByIdx(const Idx: Integer): Integer;
    function BucketCountByKey(const Key: Variant): Integer;
  end;

  TStringKeyHashTable = class(TCustomHashTable)
  protected
    function HashSize: Integer; override;
  public
    function HashFunc(Key: Variant): Integer; override;
    property Count;
    property Size;
    property Value; default;
  end;

implementation

const
  HashErrMsgs: array[THashErrMsg] of string =
    ('Hash key not found',
     'Hash key already exists',
     'Bucket index out of bounds');

// TCustomHashTable
constructor TCustomHashTable.Create;
begin
  FItems := nil;
  FHashCount := 0;
  GetMem(FHash, Size * SizeOf(TBucket));
  // Ensure that the buckets are zero-initialized.
  FillChar(PChar(FHash)^, Size * SizeOf(TBucket), #0);
end;

destructor TCustomHashTable.Destroy;
begin
```

```
  Clear;
end;

function TCustomHashTable.FindItem(const Key: Variant;
  bQuiet: Boolean; HashVal: Integer): THashItem;
begin
  Result := nil;
  if HashVal < 0 then
    HashVal := HashFunc(Key);
  if (HashVal < 0) then
    begin
      if (not bQuiet) then
        HashError(hKeyNotFound);
    end
  else
    begin
      Result := FHash[HashVal].FirstItem;
      while (Result <> nil) and (Result.Key <> Key) do
        Result := Result.FBucketNext;
      if (Result = nil) and (not bQuiet) then
        HashError(hKeyNotFound);
    end;
end;

procedure TCustomHashTable.HashError(
  const ErrMsg: THashErrMsg);
begin
  raise EHashError.Create(HashErrMsgs[ErrMsg]);
end;

function TCustomHashTable.GetValue(
  const Key: Variant): Variant;
begin
  Result := FindItem(Key, False, -1).Value;
end;

procedure TCustomHashTable.SetValue(
  const Key: Variant; Value: Variant);
var
  p: THashItem;
begin
  p := FindItem(Key, True, -1);
  if p <> nil then
    p.Value := Value
  else
    AddItem(Key, Value);
end;

procedure TCustomHashTable.AddItem(Key, Value: Variant);
var
  i: Integer;
  hi: THashItem;
begin
  i := HashFunc(Key);
  if FindItem(Key, True, i) <> nil then
    HashError(hKeyExists);
  hi := THashItem.Create;
  hi.FKey := Key;
  hi.FValue := Value;
  hi.FHashIndex := i;
  // Insert hi at the beginning of the items list.
  if (FItems <> nil) then
    FItems.FPrev := hi;
  hi.FNext := FItems;
  FItems := hi;
  // Insert hi at the beginning of its hash bucket.
  if (FHash[hi.FHashIndex].FirstItem <> nil) then
    FHash[hi.FHashIndex].FirstItem.FBucketPrev := hi;
  hi.FBucketNext := FHash[hi.FHashIndex].FirstItem;
  FHash[hi.FHashIndex].FirstItem := hi;
  Inc(FHashCount);
  Inc(FHash[hi.FHashIndex].Count);
end;

procedure TCustomHashTable.RemoveItem(Key: Variant);
var
  hi: THashItem;
begin
  hi := FindItem(Key, False, -1);
  // Remove hi from the items list.
  if (hi.FNext <> nil) then
    hi.FNext.FPrev := hi.FPrev;
  if (hi.FPrev <> nil) then
    hi.FPrev.FNext := hi.FNext
  else
    FItems := hi.FNext;
  // Remove hi from its hash bucket.
  if (hi.FBucketNext <> nil) then
    hi.FBucketNext.FBucketPrev := hi.FBucketPrev;
  if (hi.FBucketPrev <> nil) then
    hi.FBucketPrev.FBucketNext := hi.FBucketNext
  else
    FHash[hi.FHashIndex].FirstItem := hi.FBucketNext;
  Dec(FHashCount);
  Dec(FHash[hi.FHashIndex].Count);
  // Finally, free hi from memory.
  hi.Free;
end;

procedure TCustomHashTable.Clear;
var
  p: THashItem;
begin
  FHashCount := 0;
  FillChar(PChar(FHash)^, Size * SizeOf(TBucket), 0);
  // Walk FItems and destroy all items.
  p := FItems;
  while (p <> nil) do begin
    FItems := FItems.FNext;
    p.Free;
    p := FItems;
  end;
end;

function TCustomHashTable.KeyExists(Key: Variant): Boolean;
begin
  Result := (FindItem(Key, True, -1) <> nil);
end;

function TCustomHashTable.BucketCountByIdx(
  const Idx: Integer): Integer;
begin
  if (Idx < 0) or (Idx >= Size) then
    HashError(hIndexOutOfBounds);
  Result := FHash[Idx].Count;
end;

function TCustomHashTable.BucketCountByKey(
  const Key: Variant): Integer;
begin
  Result := FHash[HashFunc(Key)].Count;
end;

// TStringKeyHashTable
function TStringKeyHashTable.HashSize: Integer;
begin
  Result := 256;
end;

function TStringKeyHashTable.HashFunc(
  Key: Variant): Integer;
var
  st: string;
  i: Integer;
begin
  st := Key;
  Result := 0;
  for i := 1 to Length(st) do
    Inc(Result, Integer(st[i]));
  Result := Result mod Size;
end;

end.
```

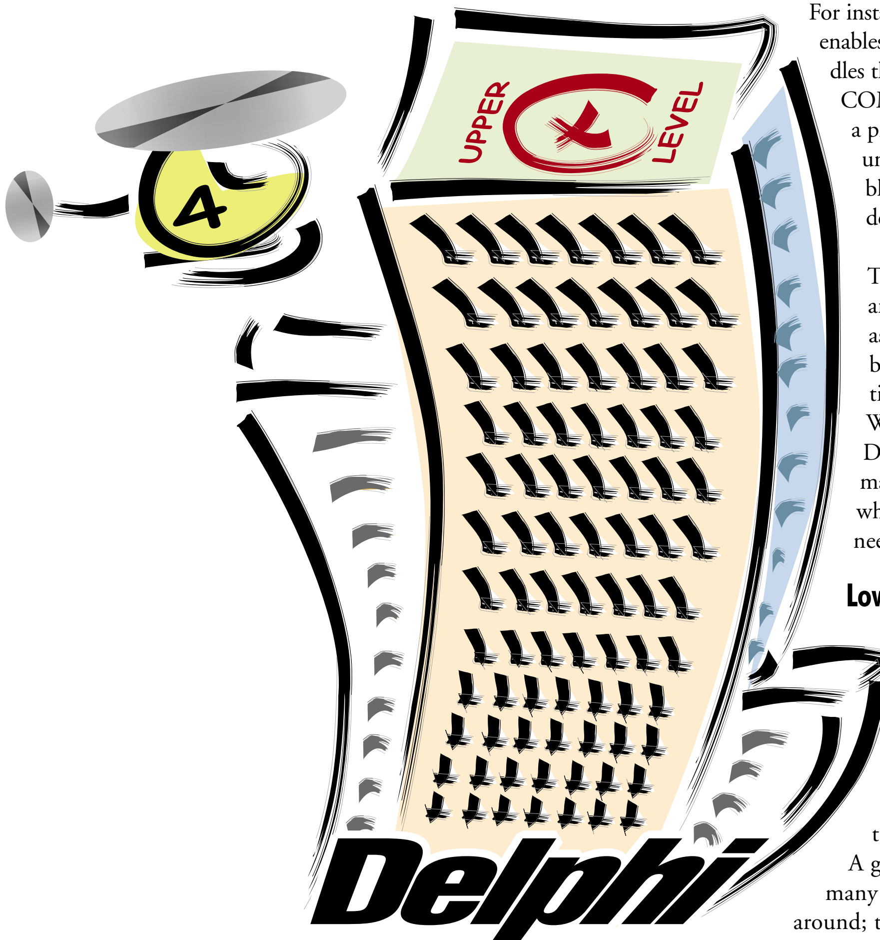
End Listing One

By Andre van der Merwe

Low-level Delphi

An Introduction to the Delphi 4 CPU Window

Working with Delphi at or near the machine-code level is not everyone's idea of fun. Why would you want to see how Delphi generates machine code? For some, understanding how it works is reason enough. There are other more practical reasons as well, some of which might be more applicable to you.



For instance, understanding the machine code enables you to figure out exactly how Delphi handles things normally hidden from view, such as COM reference counting. Creating or evaluating a protection scheme also requires a low-level understanding of your code. Lastly, and possibly most practically, sometimes low-level debugging is the only way to find a bug.

To understand the material presented in this article, you'll need a basic understanding of assembler. If you've previously used assembler in DOS, you'll soon (with a bit of practice) realize that using assembler under Windows is much easier than it was under DOS. This is because Windows provides many basic functions (such as *MessageBox*), whereas in DOS, each application would need to have its own message box routine.

Low-level Programming Requirements

This article will demonstrate how to use Delphi (specifically Delphi 4) to do basic low-level work, and explain some low-level structures. You don't need any application other than Delphi. However, there are a few tools that can make your life a lot easier if you're going to be doing much low-level programming. A good hex editor is invaluable. And there are many great shareware and freeware hex editors around; take a look at http://www. winfiles.com and http://www.nonags.com.

For a fantastic introduction to assembler programming, I recommend *The Art Of Assembly Language Programming* by Randall Hyde [1996]. An online version of this manuscript can be found at http://webster.cs.ucr.edu/. It's truly one of the best references you'll find, and you can download it for free. There are a few other books on assembler out there, but they're not always easy to find.

Delphi 4's CPU Window

Although Delphi 3 has a CPU window, it's undocumented. It's also not nearly as powerful as the Delphi 4 CPU window. For this reason, it's assumed that you're using Delphi 4 for the rest of this article.

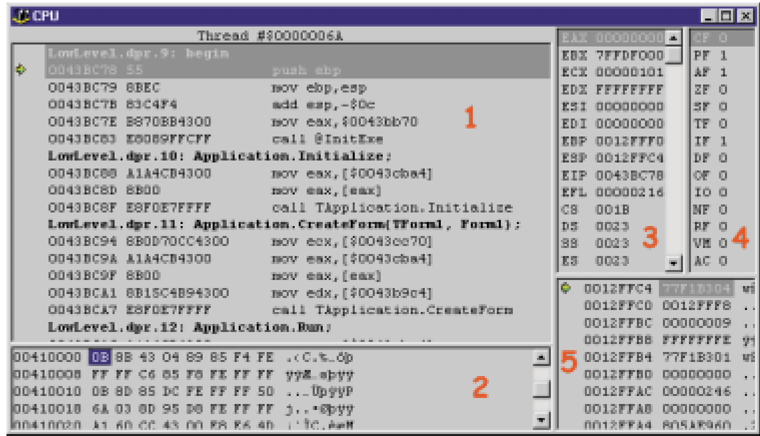


Figure 1: The Delphi 4 CPU window is split into five panes.

Figure 1 shows the Delphi 4 CPU window, which is split into five panes:

1) Disassembly pane
2) Memory dump pane
3) Register pane
4) CPU flags pane
5) Stack pane

To see the CPU window, you need to be debugging an application. While in debugging mode, press Ctrl Alt C or select **View | Debug Windows | CPU**.

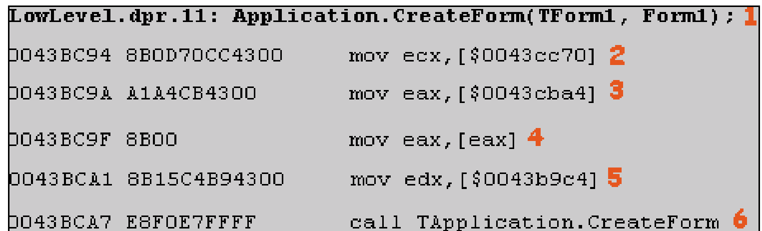


Figure 2: A section from the disassembly pane.

The Disassembly Pane

Figure 2 shows a section from the disassembly pane. This pane shows a line of Delphi code (bold to line 1) followed by its corresponding assembler code (lines 2 through 6).

The following is a brief description of what the code does. I'll explain this in more detail later. Lines 2 and 3 move *Form1* and *TForm1* pointers into registers. These are the two parameters in the Delphi call displayed on line 1. Line 5 moves the *Self* pointer (referred to as the "this" pointer by C/C++ programmers) into the EDX register. Line 6 calls the *TApplication.CreateForm* procedure.

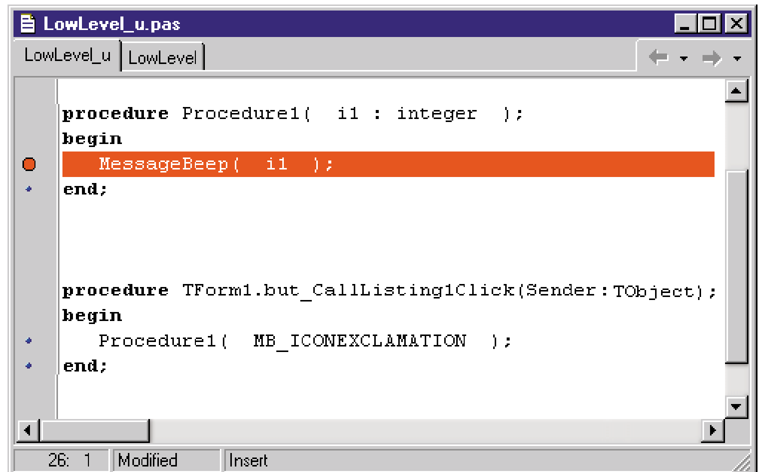


Figure 3: Setting the breakpoint.

Procedures

The easiest way to see how something will look in assembler is to write it in Delphi and view the disassembly. The example project, LowLevel.dpr, has a procedure named *Procedure1*, and an *OnClick* event that calls it:

```
procedure Procedure1(i1: Integer);
begin
  MessageBeep(i1);
end;

procedure TForm1.but_CallListing1Click(Sender: TObject);
begin
  Procedure1(MB_ICONEXCLAMATION);
end;
```

As you can see from the previous example, *Procedure1* will be called and will execute *MessageBeep(MB_ICONEXCLAMATION)*. If you place a breakpoint in the *Procedure1* procedure (see Figure 3), then debug the application, you'll be able to view the CPU window. The disassembly pane will look like the one shown in Figure 4.

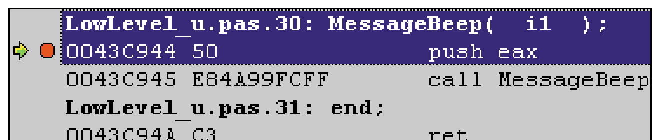


Figure 4: The disassembly pane after setting the breakpoint and debugging.

The entire procedure consists of three lines of assembler:

```
push EAX
call MessageBeep
ret
```

Remember that this procedure is called with a single parameter. This parameter is then passed to the *MessageBeep* function. Knowing this, we can assume the parameter for the *MessageBeep* function has been passed by the calling procedure/function in

the EAX register. Figure 5 confirms this. (If you don't feel this is an obvious assumption, see the section titled "Calling Conventions" later in this article. It should shed some light on the subject.) So, all this procedure is doing is pushing the parameter that was passed to it and calling the *MessageBeep* function.

Figure 6 shows a flow diagram for the *MessageBeep* function.

From this example, you can see there are at least two ways of passing parameters to functions/procedures. The first is by putting parameters in registers, as used by procedures/functions calling *Procedure1*. The second is by pushing parameters onto the stack, e.g. used when calling *MessageBeep*.

There are advantages and disadvantages to both methods. There are a limited number of registers, and only a few of them can be used for passing parameters. Stack space isn't unlimited, but it offers a great deal more space than registers. However, using registers is a lot faster than using the stack. This is why Delphi uses registers for *Procedure1*.

Functions

Functions are the same as procedures, except they return a value. The following shows a function and an event handler that calls it:

```
LowLevel_u.pas.36: Procedure1(  MB_ICONEXCLAMATION  );
0043C94C B830000000          mov eax,$00000030
0043C951 E8EEFFFFFF          call Procedure1
LowLevel_u.pas.37: end;
0043C956 C3                  ret
```

Figure 5: This confirms that the parameter for the *MessageBeep* function has been passed by the calling procedure/function in the EAX register.

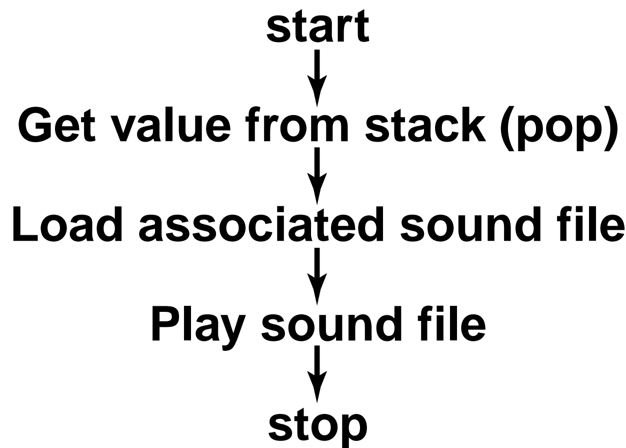


Figure 6: A flow diagram for the *MessageBeep* function.

```
LowLevel_u.pas.43: Result := MB_ICONEXCLAMATION;
0043C958 B830000000          mov eax,$00000030
LowLevel_u.pas.44: end;
0043C95D C3                  ret
```

Figure 7: The assembly code for the function ...

```
LowLevel_u.pas.49: MessageBeep(  Function1  );
0043C960 E8F3FFFFFF          call Function1
0043C965 50                  push eax
0043C966 E82999FCFF          call MessageBeep
LowLevel_u.pas.50: end;
0043C96B C3                  ret
```

Figure 8: ... and the event handler that calls it.

```
function Function1: Integer;
begin
  Result := MB_ICONEXCLAMATION;
end;

procedure TForm1.but_Function1Click(Sender: TObject);
begin
  MessageBeep(Function1);
end;
```

Figure 7 shows the assembly code for the function, and Figure 8 shows the event handler that calls it.

The function itself is very simple. The return value is simply put into EAX. The event handler is only slightly more complex:
1) It calls the function, *Function1*, which returns the value in EAX;
2) it then pushes EAX (the value returned),
3) calls the *MessageBeep* function, and
4) returns.

Calling Conventions

Figure 9 shows the calling conventions that Delphi can use.

The default for Delphi is the **register** calling convention. The **stdcall** calling convention is used by WinAPI calls, e.g. *MessageBeep*.

Parameter order. If a function was called with three parameters, say *Bla(p1, p2, p3)*, left-to-right order would mean that *p1* was pushed, then *p2* was pushed, and finally, *p3* was pushed. For a right-to-left order, the sequence of pushes would be reversed.

Routine/caller performs cleanup. This indicates if the routine being called removes parameters from the stack, or if the function doing the calling cleans up the stack.

As you can see, only with the **cdecl** calling convention (used by C/C++) does the caller do the clean up. This is what allows C++

register	Left-to-right parameter order. Routine performs clean-up. Parameters passed in registers.
pascal	Left-to-right parameter order. Routine performs clean-up. Parameters not passed in registers.
cdecl	Right-to-left parameter order. Caller performs clean-up. Parameters not passed in registers.
stdcall	Right-to-left parameter order. Routine performs clean-up. Parameters not passed in registers.
safecall	Right-to-left parameter order. Routine performs clean-up. Parameters not passed in registers.

Figure 9: The calling conventions that Delphi can use.

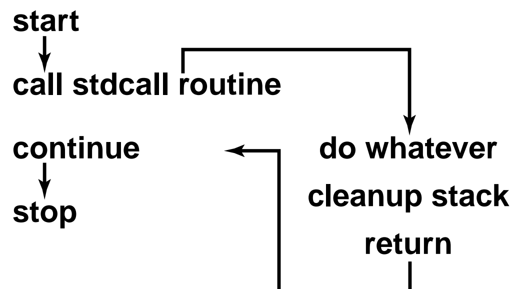


Figure 10: A routine where the called routine does the clean-up.

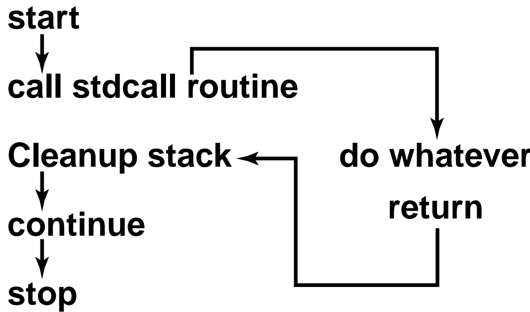


Figure 11: A routine where the caller does the clean-up.

```
LowLevel_u.pas.58: self.Caption := '123';
0043C9A3 BAC8C94300        mov edx,$0043c9c8
0043C9A8 8BC3              mov eax,ebx
0043C9AA E83D2BFEFF        call TControl.SetText
LowLevel_u.pas.59: Caption := '123';
0043C9AF BAC8C94300        mov edx,$0043c9c8
0043C9B4 8BC3              mov eax,ebx
0043C9B6 E8312BFEFF        call TControl.SetText
```

Figure 12: Type in `Caption := '123'`, or `Self.Caption := '123'`, and you get the same result.

```
LowLevel_u.pas.74: MessageBeep(  Function1  );
0043CA03 E8B0FFFFFF        call Function1
0043CA08 50                push eax
0043CA09 E88698FCFF        call MessageBeep
LowLevel_u.pas.77: MessageBeep(  ClassGetSound  );
0043CA0E 8BC3              mov eax,ebx
0043CA10 E8E3FFFFFF        call TForm1.ClassGetSound
0043CA15 50                push eax
0043CA16 E87998FCFF        call MessageBeep
```

Figure 13: The disassembly for the event handler.

functions to take a variable number of parameters. Figure 10 shows a routine where the called routine does the clean-up, e.g. **pascal**. Figure 11 shows a routine where the caller does the clean-up, e.g. **cdecl**.

If you look at these two figures, it's clear why the **cdecl** calling convention (see Figure 11) allows a variable number of parameters, and the others make it more difficult (see Figure 10). In Figure 10, the called routine does the clean-up, which means it must "know" how many parameters to remove from the stack. In Figure 11, the called routine does not have to know how many parameters were passed. The calling function knows how many functions it passed, so it knows how many to remove from the stack during the clean-up.

That is not to say that the **cdecl** calling convention is better. It forces the clean-up code to be put after every call to a function. The other calling conventions only require that the clean-up code be in one place: the function itself. As with most things, there is a trade-off.

"Parameters (not) passed in registers" indicates whether the parameters are passed in the registers, or on the stack.

The *Self* Pointer

In the discussion of Figure 2, there was a reference to the *Self* pointer. This pointer is passed to functions and procedures that are members of a class. Basically, the *Self* pointer is a pointer to the current instance of the class.

For example, in an event handler in *TForm*, you can type `Caption := '123'`, which will change the form's caption. You can also type `Self.Caption := '123'`, which is the same thing. The demonstration application does exactly this. You can see from the disassembly (see Figure 12) that there is no difference.

The *Self* pointer allows the same code, in a class function or procedure, to act on different instances of that class at run time. You can have multiple instances of a *TForm*, and the *Self* pointer will uniquely identify each of them.

The following code shows a class function that does the same thing as *Function1* (see Figure 7): return a sound to be played by the *MessageBeep* function. It also shows an event handler that calls *Function1* and the class function just described:

```
function TForm1.ClassGetSound : Integer;
begin
  Result := MB_ICONSTOP;
end;

procedure TForm1.but_CallClassClick(Sender: TObject);
begin
  // Normal function.
  MessageBeep(Function1);
  // Function is a member of a class.
  MessageBeep(ClassGetSound);
end;
```

Figure 13 shows the disassembly for the event handler. You can see that there is an extra step:

```
mov eax, ebx
```

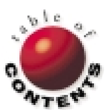
when the class member is called. This extra step is the passing of the *Self* pointer in the EAX register.

Conclusion

If you've made it this far, you're well on your way to understanding low-level programming. In this article, I covered some basics that will enable you to fiddle around with Delphi, and see how it works internally. Start with a simple procedure and disassemble it (with the CPU window). Then start moving on to more complex procedures and functions. You will be amazed how much you can learn. Enjoy! ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\SEP\DI9909AV.

Andre van der Merwe is a professional Delphi programmer and a big fan of assembler. You may reach Andre via e-mail at dart@pobox.com.



By Dan Miser

MDI and ActiveX

Oil and Water *Can* Mix

Multiple document interface (MDI) applications have been a part of Windows for a long time. Unfortunately, ActiveX has no concept of MDI. This can be easily demonstrated by creating a simple project that has MDI forms. Figure 1 shows the result.

This makes it more difficult to convert your existing MDI applications to be ActiveX compatible through the use of Delphi's ActiveForms. To make the transition from an MDI application to an ActiveForm, you have two options: Don't use MDI; and take control of the form's creation process to circumvent MDI behavior.

The first option is reminiscent of the patient who goes to his doctor and says: "It hurts when I do this." To which the doctor replies, "Then don't do that." It may be fine advice, but seldom is it that easy to avoid "doing that." The second option provides these, and other, benefits: You can have an ActiveX *and* a stand-alone version of your program; or you share the code base among an ActiveForm and an MDI application.

Getting Started

When creating an MDI application that will be hosted in an ActiveForm, you cannot rely on, or use, anything related to MDI. This includes references to the Visual Component Library (VCL) properties *ActiveMDIChild*, *MDIChildCount*, and *MDIChildren*, or any API-level MDI messages. None of these will work properly in an ActiveX control. If you absolutely must use MDI-processing, you'll have to isolate these calls by checking if the application is being run in ActiveX mode or executable mode. The easiest way to accomplish this is to use a Boolean flag, which we'll cover later.

There are two form styles in the VCL that deal with MDI forms: *fsMDIForm* and *fsMDIChild*. When you set the *FormStyle* property of a form to one of these styles, the VCL performs some magic to use the Windows API to create the MDI forms.

Figure 1: You receive this error message if you try to use MDI in an ActiveX control.

Figure 2 shows the pertinent excerpt from *TCustomForm.CreateWindowHandle*.

To allow your MDI forms to work under ActiveX, set the form to *fsNormal*. Because the *CreateWindowHandle* method creates the form based on the current value of *FormStyle*, we'll override this method and modify *FormStyle* before calling the inherited method. We can control whether we force the style to *fsNormal* by using a flag variable. This variable is placed in the form's **interface** declaration, and will be set by calling a new constructor instead of the standard *Create* constructor. Add the code shown in Figure 3 to each child form that will be displayed in the ActiveForm.

When changing the form's style to *fsNormal*, however, the child form will no longer respect the client space of the main form, but will instead use all of the main form's space. This is troublesome if you have controls like toolbars and status bars on your parent form. To prevent this behavior, we'll set the parentage of the child forms to a Panel component that will be placed on the parent form.

One last technique to help transition your MDI application to an ActiveForm is the use of a proxy form. This method allows you to create one Delphi form that is responsible for all the other Delphi forms. If you create a typical ActiveForm, you're actually creating a Delphi form that serves as a placeholder for your visual interaction between the ActiveForm and Internet Explorer. Because this is a Delphi form, you can use it as a parent to other "child" forms you create. A perfect example would be the MDI-less forms we just created. The *FormCreate* method for the main ActiveForm is shown in Figure 4. Conrad Herrmann first made this technique available on his Web site shortly after the release of Delphi 3. He also hosts other Frequently Asked Questions for ActiveX problems; for more information visit http://pw2.netcom.com/~cherrman/daxfaqs.htm.

```
procedure TCustomForm.CreateWindowHandle(
  const Params: TCreateParams);
var
  CreateStruct: TMDICreateStruct;
begin
  if (FormStyle = fsMDIChild) and
     not (csDesigning in ComponentState) then
    begin
      if (Application.MainForm = nil) or
         (Application.MainForm.ClientHandle = 0) then
        raise EInvalidOperation.Create(SNoMDIForm);
      with CreateStruct do begin
        szClass := Params.WinClassName;
        szTitle := Params.Caption;
        hOwner  := HInstance;
        X       := Params.X;
        Y       := Params.Y;
        cX      := Params.Width;
        cY      := Params.Height;
        style   := Params.Style;
        lParam  := Longint(Params.Param);
      end;
      WindowHandle := SendMessage(
        Application.MainForm.ClientHandle,
        WM_MDICREATE, 0, Longint(@CreateStruct));
      Include(FFormState, fsCreatedMDIChild);
    end
  else
    begin
      inherited CreateWindowHandle(Params);
      Exclude(FFormState, fsCreatedMDIChild);
    end;
end;
```

Figure 2: Relevant code from *TCustomForm.CreateWindowHandle*.

```
constructor TfrmStep1.CreateNoMDI(AOwner: TComponent);
begin
  FNoMDI := True;
  inherited Create(AOwner);
end;

procedure TfrmStep1.CreateWindowHandle(
  const Params: TCreateParams);
begin
  if FNoMDI then begin
    FormStyle := fsNormal;
    Visible   := False;
  end;
  inherited CreateWindowHandle(Params);
end;
```

Figure 3: Code used to remove MDI-specific attributes from a form.

```
procedure frmMain.FormCreate(Sender: TObject);
begin
  frmMain:=TfrmMain.Create(Self);
  frmMain.Parent         := Self;
  frmMain.Align          := alClient;
  frmMain.BorderStyle    := bsNone;
  frmMain.Visible        := True;
  frmMain.Panel1.Visible := True;

  frmContacts := TfrmContacts.CreateNoMDI(Self);
  frmContacts.Parent := frmMain.Panel1;
  frmContacts.Show;
end;
```

Figure 4: The *FormCreate* method for the main ActiveForm.

Not So Fast

Now that we have the basic functionality implemented, there are a couple of caveats to ActiveForm application development you should know about. First, the *OnActivate* and *OnDeactivate* events don't get called when using Internet Explorer 4.0. You can either move the contents of these procedures to another method, such as *OnCreate* or *OnShow*, or you can manually call the events as needed. Second, the ActiveForm doesn't auto-create any forms. You must take control of this process by creating secondary forms manually. A good place to do this is in the ActiveForm's *OnCreate* event. Also, items such as Data Modules don't get created automatically, so they must be created manually. You should double-check the order that forms were auto-created in your stand-alone application; if you try to use a form that hasn't been created from another form's *OnCreate* event, an access violation will occur.

10 Steps to ActiveForm Conversion

1) Create a new ActiveForm using **File | New | ActiveX | ActiveForm**.
2) Save this form in a sub-directory under your existing MDI application.
3) Add the existing MDI forms to the ActiveForm project using the Project Manager.
4) Add an invisible Panel component to your MDI parent form to act as parent to the MDI child forms.
5) Add the code in Figure 3 to all MDI children.
6) Add the code in Figure 4 to the ActiveForm.
7) Make sure you create all the Data Modules and forms.
8) Ensure you aren't using MDI-related properties or messages.
9) Ensure you aren't using *OnActivate* and *OnDeactivate* events for the ActiveForm.
10) Compile, deploy, and enjoy!

— Dan Miser

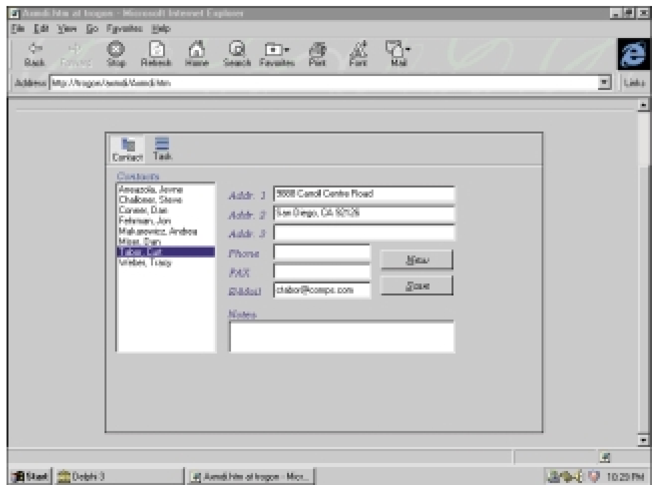


Figure 5: An MDI application in an ActiveX control.

Conclusion

Using the techniques presented here, you can get an MDI application converted for use within an ActiveForm in record time, and with minimal problems (see Figure 5). By using an ActiveForm with the functionality of your existing application, you can capture a whole new segment of your market. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\SEP\DI9909DM.

Dan Miser is a long-time Delphi programmer and consultant, specializing in multi-tier application design using MIDAS. He is active in the Borland newsgroups, where he serves as a proud member of TeamB (http://www.teamb.com). Dan also finds time to write for *Delphi Informant* and speak at Borland conferences. You can visit his Web site at http://www.execpc.com/~dmiser, or contact him at dmiser@execpc.com.

By Keith Wood

Inside Oracle Queries

A Delphi Utility Gets Oracle to Explain Itself

Access to client/server databases is easy with Delphi. We can use SQL to retrieve and manipulate just about any database on the market. Although just throwing together a few SQL statements may provide the functionality we require, to make the application perform in the real world, we need to understand what the SQL database engine is doing with each statement.

With Oracle, it's relatively easy to gain insight into its internal processing by asking it to explain how it gets our records. The utility described in this article provides an interface to this ability and makes it easier to see what's happening. It allows us to test alternate strategies for data retrieval, and provides easy access to the structure of the database itself.

Oracle Internals

When Oracle processes a SQL statement, it strings together a series of internal operations to efficiently achieve the required effect. To determine which steps to follow, it first decides which optimizer to use. Oracle 7 has three settings available:

- a rule-based optimizer (RBO) that works from syntactical rules;
- a cost-based optimizer (CBO) that consults statistics generated by the ANALYZE command to find the best access; and
- a Choose option that uses the cost-based optimizer if the tables have been analyzed, and the rule-based optimizer if not.

The selected optimizer parses the SQL statement and determines which tables to access in what order, how to access them, and how to combine the resulting records. The operations are divided into several categories, including table, index and view access, combining index scans, ordering and grouping rows, and joining tables. The more common operations, and the situations in which they might be used, are shown in Figure 1.

Operation	Purpose	Situation
TABLE ACCESS FULL	Sequential full-table scan.	no WHERE clause is used.
TABLE ACCESS BY ROWID	Retrieve rows by internal row ID (very efficient).	accessing through indexes.
INDEX UNIQUE SCAN	Retrieve rows via a unique index.	a unique index can be used for a WHERE clause.
INDEX RANGE SCAN	Retrieve rows via an index.	a non-unique index can be used, or a range of uniquely indexed values is required.
AND-EQUAL	Combine multiple index scans.	accessing via indexes connected by AND.
CONCATENATION	Combine multiple index scans.	accessing via an index on multiple values.
SORT ORDER BY	Sort rows on column values.	an ORDER BY clause is used.
SORT UNIQUE	Sort rows, eliminating duplicates.	the DISTINCT keyword is used.
SORT JOIN	Sort rows.	preparing for a join.
SORT AGGREGATE	Sort rows and group together.	MAX, MIN, or COUNT are used.
SORT GROUP BY	Sort rows and group together.	a GROUP BY clause is used.
UNION ALL	Combine two sets of records.	the UNION keyword is used.
MINUS	Find the difference between two sets of records.	the MINUS keyword is used.
INTERSECTION	Find the rows in common between two record sets.	the INTERSECT keyword is used.
VIEW	Return records from a view.	a view is accessed and in subqueries.
FILTER	Eliminate rows from a view.	a WHERE clause is used with a view.
MERGE JOIN	Join two tables.	joining two tables without indexes available.
MERGE JOIN OUTER	Join two tables using an outer join.	an outer join is requested on two tables without indexed fields.
NESTED LOOPS	Join two tables.	joining two tables on indexed fields.
NESTED LOOPS OUTER	Join two tables using an outer join.	an outer join is requested on two tables with indexed fields.

Figure 1: Selected Oracle SQL operations.

Some operations (e.g. index scans) can return data to the user immediately while continuing to process the rest of the query. Others (e.g. sorting operations) require

the entire set of records be available before they can start. This means the result set can take longer to produce. Knowing which operations are being used, and how they function, allows us to tune the queries for a particular application.

Oracle Explains

To see what steps Oracle is taking when it processes a SQL statement, it helpfully provides the EXPLAIN PLAN command:

```
EXPLAIN PLAN SET STATEMENT_ID = :statement_id
FOR <SQL statement>
```

where *statement_id* is a string literal used to identify this explanation, and *SQL statement* is any valid SQL statement. This command causes records to be inserted into a special table, whose fields describe the operations applied to process the SQL statement. The supplied ID is attached to each record belonging to this explanation to uniquely identify them.

The structure of the table that Oracle uses is widely published, and can usually be generated by running the utlxplan.sql script that comes with Oracle (see Figure 2). By default, the table is named PLAN_TABLE.

Once the explanation is placed into this table, we can retrieve it and view the results with the following SELECT statement:

```
    SELECT DISTINCT
           ID, OBJECT_TYPE, OPTIMIZER,
           SUBSTR(LPAD(' ', 2 * LEVEL - 2) ||
                  OPERATION || ' ' || OPTIONS || ' ' ||
                  OBJECT_NAME, 1, 120) AS PLAN
      FROM PLAN_TABLE
     WHERE STATEMENT_ID = :statement_id
START WITH ID = 0
CONNECT BY PRIOR ID = PARENT_ID
       AND STATEMENT_ID = :statement_id
```

where *statement_id* is set to the value used in the original EXPLAIN command. This statement makes use of a couple of features of Oracle SQL to order and format the explanation so that its structure can be easily seen.

These two SQL statements, the EXPLAIN and subsequent SELECT, form the basis of the utility program described in this article, allowing us to peer inside the Oracle database and its SQL engine.

Program Structure

The utility program's screen is divided into two areas (see Figure 3). The top panel allows us to enter the SQL query to be explained, and an identifier to retrieve that. The query text can be copied in from any text source via the Clipboard. Any value can be entered for the identifier, and this can be set to automatically increment by checking the box next to it. Doing this causes a trailing numeric to be added to the identifier, and to be incremented each time the **Explain** button is pressed. In a multi-user environment, you should perhaps use your initials as part of the identifier to avoid conflicts with other users of the database. Below this area is a tabbed notebook showing various details about the query or the database as a whole.

The first tabbed page displays the explanation of the query processing. It lays out the operations that Oracle performs to produce the desired output. The grid is filled when the **Explain** button is pressed. The indentation of the execution plan steps indicates the relationships between the various elements. The innermost operations are performed first, combined, and further manipulated by the outer steps.

The second page allows the entered SQL to be executed, displaying the resulting record set (if any), the time taken to retrieve this output, and the number of records returned (see Figure 4). This page allows us to check that the query performs what we want, and to empirically compare the time it takes to finish. However, subsequent executions of the query may return reduced times once the records are cached.

Database information is shown on the remaining two pages: Tables, which shows a list of tables, their fields and indexes (see Figure 5); and Views,

Name	Type
STATEMENT_ID	VARCHAR2(30)
TIMESTAMP	DATE
REMARKS	VARCHAR2(80)
OPERATION	VARCHAR2(30)
OPTIONS	VARCHAR2(30)
OBJECT_NODE	VARCHAR2(128)
OBJECT_OWNER	VARCHAR2(30)
OBJECT_NAME	VARCHAR2(30)
OBJECT_INSTANCE	NUMERIC
OBJECT_TYPE	VARCHAR2(30)
OPTIMIZER	VARCHAR2(255)
SEARCH_COLUMNS	NUMERIC
ID	NUMERIC
PARENT_ID	NUMERIC
POSITION	NUMERIC
OTHER	LONG

Figure 2: Oracle's plan explanation table.

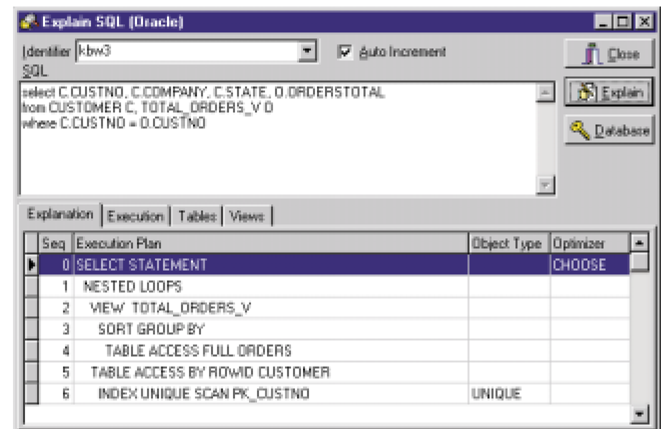


Figure 3: The Explain SQL utility in action.

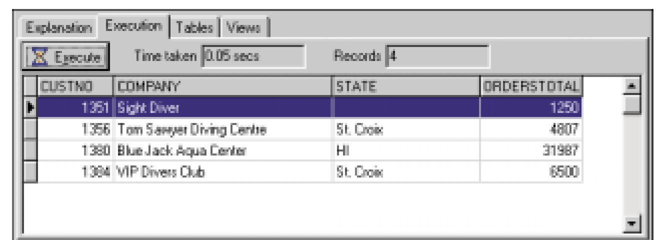


Figure 4: The Execution page.

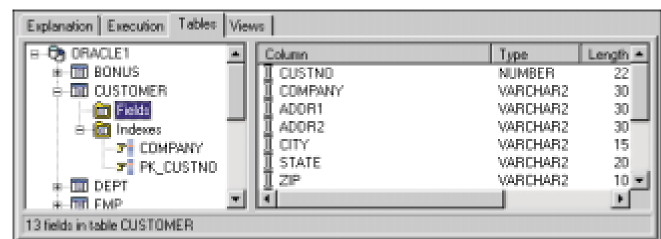


Figure 5: The Tables page.

which shows a list of views on these tables (see Figure 6). These two pages allow us to peruse the database structures to see what fields and indexes are available for our use. In general, using an index greatly enhances the performance of a query, so it's useful to know how we can access the data efficiently. Note that only the objects owned by the user are shown.

To make it easier to work through a query explanation, the grid containing the operations is linked to the Tables and Views pages. Double-clicking on a step that references a table, index, or view immediately takes us to that item on the appropriate page. So, for example, we don't need to search through the entire table structure to find out what index SYS_C001234 is composed of.

On start up, the program requests login details from us. These are stored in the Windows registry for use the next time the application is run. A connection with the database is then established and loading of the database structure commences. Once completed, we can enter SQL statements of interest and have them explained to us.

Programming Efficiencies

There are several areas of the program that run quite slowly without additional tuning. In particular, the tree views seem to slow down exponentially as their size grows when they are sorted. Similarly, searching through the hierarchy of a tree view for a particular item isn't rapid either. To overcome these deficiencies and speed up the program, follow the steps described below.

Sorting within the tree views is unnecessary. Because the data they contain is only loaded once and doesn't change, it's easier and much faster to have the database order the records being loaded in the first place. Then, we only have to read them sequentially and populate the tree view.

To search the tree views for particular table, index, or view names, we'd normally step through each node, and then recursively, through all its children, until we arrived at the correct location. But a much faster solution uses the abilities of a *StringList* to perform binary searches on its elements and associate an object with each string value. As we load the table, index, and view names, we also enter them into two *StringLists* (one for each tree view). Attached to these entries are references to the tree nodes that display them in the hierarchy.

When the time comes to jump to a particular table, index, or view, we only have to ask the appropriate *StringList* to find the value in question. Once located, we now have a pointer back to the corresponding tree node and can make it the current node by setting the *Selected* property of the tree view to it. To ensure it's visible within the tree view, we also set the *TopItem* property to scroll the entire hierarchy. Finally, the page containing the view is given focus. The code in Figure 7 shows this processing for the tables and indexes.

Data on Demand

One final area of increasing the performance of the program involves retrieving and processing data only when, and if, it's required. For example, the program can display the fields for every table. But most of the time, only a few tables would be looked at. It wastes time and memory to load all the field definitions for every table on the slight chance that someone might want to see them.

Instead, the program only loads the field definitions for those tables the user specifically requests. Initially, the tree view is set up to contain only the nodes representing each table's fields and indexes, without storing all the subsequent details in memory. When the Field entry is clicked in the tree view, we respond to the selection by checking whether we have already loaded the field definitions for this table. If so, their descriptions are contained in a *StringList* referenced by the *Data* property of the tree node.

If the *Data* property points to nothing, we go back to the database and request the details about the fields for this table. From the returned records, we generate the *StringList* and attach it to the tree node via its *Data* property (see Figure 8). Again, we use the ability of a *StringList* to associate an object with each value as we store the various details for each field.

In either case, we now have a list of the fields and their details, and can display them in the list view to the right of the tree. We establish the appropriate column headings, widths, and alignments before adding the sub-items from the *StringList*'s associated objects.

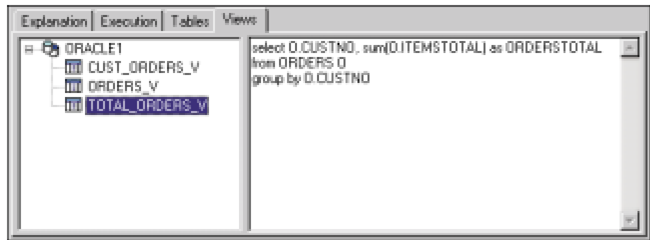


Figure 6: The Views page.

```
// Jump to the nominated table/index in the indexes view.
procedure TfrmExplain.JumpToIndex(sIndex: string);
var
  i: Integer;
begin
  i := slsIndexes.IndexOf(sIndex);
  if i > -1 then
    with trvIndexes do begin
      Selected := TTreeNode(slsIndexes.Objects[i]);
      if Selected.Level = 3 then  { Index. }
        TopItem := Selected.Parent.Parent
      else                          { Table. }
        TopItem := Selected;
      pgcOutcomes.ActivePage := tabTables;
    end;
end;
```

Figure 7: Locating and displaying the requested table or index.

```
// Load column details for the current table.
procedure LoadColumns(trnFields: TTreeNode);
begin
  with trnFields, qryColumns do begin
    Data := TStringList.Create;
    ParamByName('TABLE').AsString := Parent.Text;
    Open;
    while not EOF do begin
      // Add column data.
      TStringList(Data).AddObject(
        FieldByName('COLUMN_NAME').AsString,
        TTableColumnDetails.Create(
        FieldByName('DATA_TYPE').AsString,
        FieldByName('DATA_LENGTH').AsInteger,
        FieldByName('DATA_PRECISION').AsInteger,
        FieldByName('DATA_SCALE').AsInteger,
        FieldByName('NULLABLE').AsString));
      Next;
    end;
    Close;
  end;
end;
```

Figure 8: Load field details into a StringList structure on demand.

Similar processing is performed when viewing the fields that make up an index. In addition, the first two entries in the nodes' extra details are reserved for the uniqueness and current status of the index.

Object-oriented Persistence

Between sessions in the utility, various parameters are stored in the registry: the login details (alias, user ID, and password) and positioning values for the form and its panels. To avoid having to code references to the registry in several places, all access is encapsulated into an object, *TExplainParameters*, which provides the required values as properties. This allows us to hide the internal workings of these parameters and to shield the rest of the program from their idiosyncrasies. If we later want to alter how the values are stored, we can do this while retaining the same interface for the other modules. All the registry keys and entry names can be placed in this unit and made invisible to the outside world (that doesn't need to know anyway).

This encapsulation also provides some security over manipulating the values to be stored. One important example of this is the password used to access the database. Storing this as plain text could be a security problem, so the *TExplainParameters* object encrypts and decrypts this value when transferring it to and from the permanent storage. Again, encapsulation allows us to hide how this encoding is done from other modules. (A simple substitution coding is used in this program, but could easily be modified to invoke a more secure algorithm.)

To ensure the rest of the program has immediate access to these parameters, an object of type *TExplainParameters* is created automatically when the program starts. We must also free the object upon termination. Add code to the **initialization** and **finalization** sections at the end of the unit (invoked before and after other processing in the program, respectively). A global variable, *expParams*, is declared in the **interface** section of the unit to allow access to the object from other modules:

```
initialization
  // Create global parameters object.
  expParams := TExplainParameters.Create;
finalization
  // Release global parameters object.
  expParams.Free;
end.
```

Consider an example involving customers and their orders. These two tables are linked via a customer number field. To retrieve the total value of the orders for each customer, we might create a view that performs this calculation on the orders table:

```
CREATE VIEW TOTAL_ORDERS_V
         AS
     SELECT O.CUSTNO, SUM(O.ITEMSTOTAL) AS ORDERSTOTAL
       FROM ORDERS O
   GROUP BY O.CUSTNO
```

This can then be combined with data from the customers table:

```
SELECT C.CUSTNO, C.COMPANY, C.STATE, O.ORDERSTOTAL
  FROM CUSTOMER C, TOTAL_ORDERS_V O
 WHERE C.CUSTNO = O.CUSTNO
```

Submitting this construct to Oracle results in the following steps: 1) SELECT statement; 2) Nested loops; 3) View TOTAL_ORDERS_V; 4) Sort GROUP BY; 5) Table access full ORDERS; 6) Table access by row ID CUSTOMER; 7) Index unique scan PK_CUSTNO.

This shows that the ORDERS table is accessed first by reading every record (table access full). The records are then sorted into groups to compute the required sum (sort GROUP BY). This forms the output of the view TOTAL_ORDERS_V (view), which is combined in a join (nested loops) that accesses the CUSTOMER table via the unique index on the CUSTNO field (index unique scan), and hence to the physical record (table access by row ID).

The use of a unique index in step 6 means direct access to the customer's details for each record in the orders view, with the nested loop operation returning each combined record as it's matched. But the entire orders table must be scanned before its records can be grouped and summed. Table access via indexes, or the row ID, provides the best performance (for online users), with nested loops joining tables rapidly on further indexes. Full table scans and merge join operations take longer.

For the program to run against any database it's directed at, the particular table structure that Oracle uses for its explanations must be present. Embedded in the application is a query with the SQL to create this table. During the logon process, the program checks to see whether the table exists under its default name of PLAN_TABLE. If it's not already there, the query is executed to generate it before proceeding with the rest of the program. As we submit more SQL statements to be explained, the PLAN_TABLE grows. To avoid having the table expand to ungainly proportions, we must delete the entries we've placed in it. The application does this when it's closing and when we change from one database to another.

The program remembers its position and size on the screen between sessions, and recalls the locations of the divider bars between the various panels within the application. These are saved in the registry when the program is closed, but are reloaded on startup.

A Help file has been built and integrated with the application. An introduction to the various Oracle internal operations is included. Each page or panel in the program has been assigned a topic number, allowing context-sensitive help to be shown when requested by pressing F1. Each of the internal Oracle operations that appear in the explanation grid are directly tied to their appropriate description in the Help file. To achieve this, the program responds to changes in the user's position in the grid. The **Execution Plan** field is tokenized (split into separate words) and used to decide which section of Help to display. The corresponding topic number is then set into the *HelpContext* property for the page.

Conclusion

Developing client/server applications is easier with Delphi. But to get the best performance out of the database, we need to tune it and the SQL statements we use to access it. The utility presented in this article provides access to Oracle's ability to explain the execution path it uses. Combining the results from this utility with an understanding of the database itself can increase the performance of our programs. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\SEP\DI9909KW.

Keith Wood is an analyst/programmer with CCSC, based in Atlanta. He started using INPRISE's products with Turbo Pascal on a CP/M machine. Occasionally working with Delphi, he has enjoyed exploring it since it first appeared. You can reach him via e-mail at kwood@ccsc.com.

table of CONTENTS

By Wes Peterson

DBISAM 1.12

Elevate Software Rewrites the Rules of the Delphi Database Game

Since its introduction, Delphi's superb integration with databases has been one of its leading attractions. However, the Borland Database Engine (BDE) and Borland's desktop (or "local") databases (dBASE and Paradox) have left a large niche in the market for third-party databases. Most of these non-Borland efforts have aimed to eliminate the need for the BDE with dBASE and/or to allow important non-Borland databases, like Btrieve and Microsoft's Jet, to be used with Delphi. Like everything else in computing, each of these options offers opportunities — and a set of problems.

Some third-party solutions continue to use the aging dBASE file format, and thus perpetuate the need to define indexes on expressions, to deal with deleted records that aren't really deleted, and to recover deleted record space. The developer who chooses a Btrieve or Jet alternative is still faced with the licensing requirements imposed by their respective vendors.

With DBISAM, Elevate Software has rewritten the rules of the Delphi database game. DBISAM is far more than another "me too" BDE replacement. This review will detail several of DBISAM's most interesting features (see the sidebar "DBISAM Features" for a complete list):

- DBISAM compiles into your application's EXE. There are no DLLs or other vendor dependencies to be considered in your distribution.
- DBISAM's footprint is small. It adds only about 300KB to your application — far smaller than the overhead of some user interface or reporting libraries.
- DBISAM is configuration-free and alias-free. Every networked database application needs an aliasing mechanism, but such a facility needn't be as burdensome as that imposed by the BDE.
- DBISAM includes native, engine-level support for the SQL SELECT statement — a real boon when it comes to report writing.
- DBISAM features DBSYS, a Database Explorer-like utility. DBSYS is the DBISAM "database administrator" application used to: define, restructure, copy, rename, and empty DBISAM tables; search, examine, and edit data; repair corrupted tables; write, test, save, and load SQL statements; and document database structures.

Over the years, we've put up with some pretty clunky database "admin" utilities. DBSYS is refreshingly different. The user interface is clean and intuitive, and the absence of bugs is refreshing. Empty Table is a nice touch. Often, while developing an application, we populate the database with test data. Before distribution, however, we want a set of pristine, empty tables. This isn't easy with some other databases. It's just a mouse-click away in DBSYS.

DBISAM's file format is proprietary. While it borrows best-of-class ideas from Paradox, FoxPro, and others, a totally new format was the best way to eliminate the baggage that attends each of the "traditional" formats. Thoughtfully, DBISAM includes a utility to transfer data from/to dBASE, FoxPro, and Paradox files.

Unlike Jet, DBISAM doesn't use a monolithic file structure. Each table is stored in two or three operating system files. Data records and indexes are stored in DAT and IDX files, respectively. If the table includes BLOb fields, they're stored in a BLB file. Using discrete files for each table has its advantages and disadvantages. With discrete files, database repair can be limited to the affected table(s) and thus, be much faster than with a monolithic file. Repair, by the way, is offered both in DBSYS and at the application level with the *RepairTable* method of *TDBISAMTable*.

The main disadvantage of a discrete table format is the complexity it adds to distribution. DBISAM has another thoughtful feature that radically simplifies this issue: Reverse Engineering, a DBSYS option, examines an existing DBISAM table and generates the Object Pascal code to create and restructure that table. You copy the resulting code into your project, and your application can create its own database at run time. How's that for simplifying distribution?

It seems no database application is ever finished. Enhancement requests inevitably require changes to your database definition. Migrating existing databases to newer versions has always been problematic, but since DBISAM tables can also include developer-defined version information, your application upgrades can examine that version information and use Reverse Engineering-generated code to upgrade existing tables as needed.

In-memory Tables

In-memory tables are often overlooked, but provide an incredibly powerful solution to several knotty database application issues. Frequent, fast lookups are always needed in a modern database user interface. Instead of your lookups going to disk for their data, you can load the data once into an in-memory table. From that point forward, your user interface can access that data with never another peek at the disk.

In-memory tables are a great way to handle the conflicting requirements of time-consuming user data-entry processes and the need for short database transaction cycles. When users are performing data entry on a master-detail form, you usually don't want to write changes to the database until the user has finished and accepted the entire form. Loading detail records into an in-memory table, performing the edits there, then writing to disk only when the user finishes, allows for very brief transactions — or none if the user cancels the operation.

Less important (now that DBISAM includes SQL), but still useful, is the ability to use in-memory tables to simulate joins and to provide sort orders not present in the indexes defined in your database. You can load your "result set" from several on-disk tables into an in-memory table that includes the desired index, then base reports on the in-memory table. DBISAM's implementation of in-memory tables is complete and elegant, differing from the BDE's in several important ways.

The BDE's in-memory tables are cursor-based, meaning they exist only as a cursor and cannot be shared or used by more than one *TTable* component. DBISAM's in-memory tables aren't cursor-based, but rather exist similar to a disk-based table in a virtual file system attached to each session. You refer to DBISAM's in-memory tables just as a regular disk-based table. Multiple *TDBISAMTable* components can reference the same in-memory table.

BDE in-memory tables don't let you use BLObs and other features normally available in disk-based tables. DBISAM doesn't distinguish between a disk-based table and an in-memory table in this regard. You can even use transactions against in-memory tables, query them using a SELECT statement, or generate an in-memory table as the result set of a SELECT statement. You can also stream in-memory tables to any destination stream, which means you can store in-memory tables inside BLObs in a DBISAM disk-based table. DBISAM's in-memory tables can be shared across multiple threads and are completely thread-safe. In-memory tables are easily created with DBISAM's Reverse Engineering.

DBISAM's in-memory tables are so useful that we employ them in some of our non-DBISAM projects. For example, our work sometimes involves legacy databases over which we have no control of database design or available indexes. Yet we need to extract and report data in ways not foreseen by the original vendor. For such tasks, DBISAM's in-memory tables, by themselves, justify the price of the entire package.

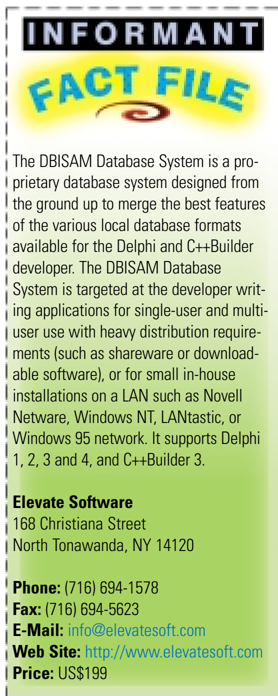
Using DBISAM

Using DBISAM is as easy, if not easier, than using other BDE-like databases. The *TDBISAMDatabase*, *TDBISAMTable*, and *TDBISAMQuery* components are the virtual equivalents of their Delphi namesakes. Each is used in a manner identical to its "native" counterpart, and DBISAM works seamlessly with the stan-

DBISAM Features

- Compiles into the application's EXE
- Small footprint, around 300KB
- Available for Delphi 1, 2, 3, 4 and C++Builder 3 with the same set of functionality across all versions (Delphi 1 and 2 are "replacements" for the VCL DB units and hook into the data-aware controls while Delphi 3 and 4, and C++Builder 3 are *TDataset* descendants)
- Does not pre-allocate large chunks of memory, only uses memory as needed and frees the memory when done
- Transparent single-user and multi-user usage, no special setup needed to support multi-user use
- Native, engine-level SQL SELECT support
- Performance is exceptional with optimization of filtering and SQL
- Built-in repair facilities
- Provides utility for transferring data from Paradox, dBASE, and FoxPro formats
- Provides utility for browsing, restructuring, updating, and searching data files
- Complete BLOb support, including configurable block sizes
- Buffered transactions that allow data files to survive unexpected client workstation power-downs with little, or in most cases, no corruption
- Primary and secondary indexes
- Complete filter support
- In-memory data files with support for streaming
- Partial index key searches and ranges
- Ranges with accurate record counts
- Auto-increment fields
- Logical sequence numbers
- Complete NULL support
- Min/max and required validity checks
- Default values
- Complete free-space recycling
- Index key compression
- Read-only open support for CD-ROMs
- Complete BCD support
- Case-insensitive indexes
- Password-protected encryption of data files
- Unique secondary indexes
- Complete international support for over 100 language variations
- User-defined version numbering of data files
- User-defined semaphore locking

— Wes Peterson



The DBISAM Database System is a proprietary database system designed from the ground up to merge the best features of the various local database formats available for the Delphi and C++Builder developer. The DBISAM Database System is targeted at the developer writing applications for single-user and multi-user use with heavy distribution requirements (such as shareware or downloadable software), or for small in-house installations on a LAN such as Novell Netware, Windows NT, LANtastic, or Windows 95 network. It supports Delphi 1, 2, 3 and 4, and C++Builder 3.

Elevate Software
168 Christiana Street
North Tonawanda, NY 14120

Phone: (716) 694-1578
Fax: (716) 694-5623
E-Mail: info@elevatesoft.com
Web Site: http://www.elevatesoft.com
Price: US$199

dard Delphi data-aware controls, as well as with other third-party components that connect to a *TDataSource* (Orpheus, ReportBuilder, etc.).

Where differences exist, they almost always reflect additional DBISAM capabilities, rather than roadblocks to your usual practices. DBISAM's *RestructureTable* method, for example, has extra parameters for user-defined version information.

Stability in DBISAM appears very good. As with any shared file database, though, improper practices can result in corruption. Resetting a machine during a database update is, naturally, a recipe for disaster (that includes using Program Reset in the IDE). DBISAM's buffered transactions can greatly reduce your application's exposure to corruption. We've been using DBISAM since its beta early last year and have never experienced data loss, even though we've repeatedly violated the proscription regarding Program Reset in the IDE. Fortunately, DBISAM's repair facilities are effective, but no database vendor has ever crafted a repair facility that can replace a rigorous backup regimen.

Performance with DBISAM is very good. We haven't done any formal benchmarks, but our informal benchmark (how does it feel in a real-world application?) gives DBISAM high marks. Indexed searches, even on filtered tables, appear to be as fast as with any other database we've used. Record navigation in our user interfaces is crisp and snappy, even on tables containing large image BLOb fields, and even on "Dog Tester," an aging 486/100 box we keep alive just to see how bad our applications can be.

DBISAM uses bitmapped-filtering optimizations similar to those of FoxPro's Rushmore technology, but with an additional performance-enhancing twist.

Documentation, Installation, and Support

Documentation for DBISAM consists of an extensive Help file, the usual "ReadMe" example projects, and FAQs and tips articles on the Elevate Web site. There is no hard-copy manual. As of this writing, we rate DBISAM's documentation somewhere between adequate and good, but not complete. This isn't because a hard-copy manual is lacking (we've come to prefer online docs), but the recent addition of SQL happened, seemingly, overnight and the online Help doesn't yet reflect the changes. Given our previous experiences with Elevate, however, we'll be surprised if this isn't corrected by the time this review goes to print.

Installation of DBISAM is straightforward, and the required steps for each version of Delphi and BCB are thoroughly documented in the ReadMe file.

Support for DBISAM is via e-mail only. We actually prefer e-mail support, as long as the vendor is responsive. Forced to write out a support inquiry, we often draft a more concise and ordered question than might be posed over the phone. Like the old adage says, "Want a better answer? Ask a better question." The potential weak link in e-mail-only support is, of course, vendor responsiveness. This is another area where Elevate is different. Virtually all our questions have been answered the same day, often on weekends (although weekend support isn't promised). We've frequently received answers within the hour.

To make things even better, DBISAM includes a "Customer Support System." This is a DBISAM database application that includes all bug reports (and description of fixes). You cannot only search the database before posing your inquiry, but you can use this program to submit your problem reports to Elevate. Elevate keeps the Customer Support database up-to-date and downloadable on their Web site, making it easy to stay abreast of changes and answers.

What's Missing, What's Wrong

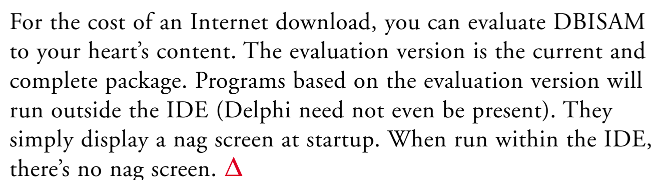
As good as DBISAM is, nothing in computing is perfect. We don't think DBISAM suffers from any fatal flaws, but it's only fair to mention the significant bumps. In its brief year of public life, DBISAM has matured quickly, and continues to be improved. Elevate has been very responsive to enhancement requests, and to the inevitable bug reports. Some of the enhancements have required file format changes and, thus, required developers to convert existing databases. Code change requirements for the user, however, have been minimal.

We expect, in fact welcome, DBISAM's continued evolution, and accept that this will, from time to time, force us to make adjustments on our end. Not all users will be comfortable with this. As of now, DBISAM lacks stored procedures, batch moves, cached updates, and support for referential integrity. Also, the SQL in DBISAM is not a complete implementation. But it is adequate for reporting needs.

Conclusion

Is DBISAM for you? If you're looking for a small, fast, inexpensive, royalty-free, single-user, and shared-file multi-user database — one that eliminates the BDE and configuration hassles — DBISAM merits your consideration.

For the cost of an Internet download, you can evaluate DBISAM to your heart's content. The evaluation version is the current and complete package. Programs based on the evaluation version will run outside the IDE (Delphi need not even be present). They simply display a nag screen at startup. When run within the IDE, there's no nag screen. Δ



Wes Peterson is President of LexCraft, a consultancy based in Carmichael, CA. LexCraft specializes in business database solutions with an emphasis on trade and professional associations. LexCraft has been using DBISAM since its beta release in the Spring of 1998. You can contact Wes at wpeterson@lexcraft.com.

The Delphi Toolbox: Testing and Debugging

Testing and debugging applications is important, regardless of the programming language. But what about testing and debugging components? To test a component you must first build a test project that uses it, then try different combinations of property settings. This can involve a lot of work. Fortunately, there's a tool that can help automate this process: reAct from Eagle Software. This month I'll address testing and debugging components using reAct and Raize Software Solutions' CodeSite.

For testing, I chose one of the VCL components, *TStringGrid*. This is how simple the process was. First, I clicked on **New Test Program** (a reAct item) on the **Component** menu. When the Select Class for Test Program dialog box came up, I checked **Component is already on the palette** and selected **TStringGrid** in the **Name** combo box. (Note: you can also test components that have not been installed on the palette. However, since reAct depends on RTTI, it will not be able to provide as much information as it can for an installed component.) Then I clicked **OK** to display the Properties/Events to Watch page, and made sure everything was checked (the default). That's all there was to creating the test program.

Now to run the program (see Figure 1). (I'm including the test executable so you can get an idea of what it looks like; see end of article for download details. Keep in mind that you won't be able to use the reAct Breakpoint Manager in the stand-alone test program.) There are three windows in the reAct test program: the test form itself (at first without the component on it); the Component Inspector to the right, which enables you to create the component and set properties; and an event log below, which enables you to automatically keep track of various events as you test a component. You have several options with logging: You can log testing results to the reAct log window, to a file, or even to CodeSite.

Now let's see what kind of interesting behavior we can expose. On the Component Inspector, click the **Create** button and experiment with the *TStringGrid*'s *Options* property of the newly created grid. To change the various sub-properties, double-click the *Options* property to bring up the reAct property editor. Find anything interesting yet? OK, under *Options*, try setting *RowSelect* and *ColMoving* to True and move some of the columns with the mouse; be sure to move the right-most column to the left and the left-most column to the right. *RowSelect* and *ColMoving* work fine independently of each other, but when you try putting them together it's a nasty brew.

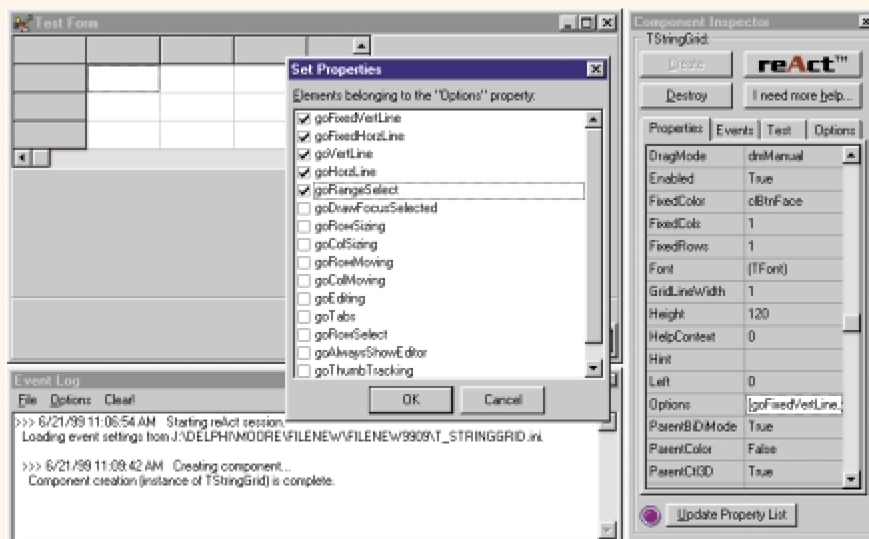


Figure 1: The test program with the reAct property editor displayed.

We've located a bug, so what next? If it's in one of our own components we'll want to find the source and fix it. Let's go down that path. First copy the unit containing the *TStringGrid* component (Grids.pas) to the project folder. To get the Grids.pas file linked to your code, you'll probably need to do a full build and restart Delphi. You'll want to set some breakpoints and trace into the component unit. You might consider tracing into the *MouseMove* or *MouseUp* methods. At first I thought the problem might be in the *MoveAndScroll* method called in the

MouseMove method. I set several CodeSite messages within this method, but found nothing problematic.

I then checked out the *MoveColumn* method: I put a *CodeSite.EnterMethod* at the beginning, a *CodeSite.ExitMethod* at the end, and tracked the values of *ToIndex* and *FromIndex*. While I was getting close to finding the problem, I still wasn't there. At least I had eliminated some code that wasn't problematic — a good debugging approach in itself. Next, I searched for instances of *goRowSelect* and found the following statement in an earlier method:

```
if goRowSelect in Options then
  FCurrent.X := FixedCols;
```

There was no such similar check in this method, so I added it immediately before the following lines, adding appropriate conditional statements and enclosing the existing code within a **begin..end** block:

```
if goRowSelect in Options then
  FCurrent.X := FixedCols
else begin
  MoveAdjust(FCurrent.X, FromIndex, ToIndex);
  MoveAdjust(FAnchor.X, FromIndex, ToIndex);
  MoveAdjust(FInplaceCol, FromIndex, ToIndex);
end;
```

This simple change fixed the cosmetic bug. Without reAct I would have been unable to find the bug so easily. Without CodeSite I would have been unable to eliminate certain areas of concern so quickly. With reAct, once you've found a problematic combination of property settings, you can save those settings for use in the next testing session. To find out more about the powerful features of CodeSite, see my review in the January, 1999 issue of *Delphi Informant*. In future columns we'll return to the Delphi Toolbox and examine additional Delphi tools and techniques, such as profiling and memory management. Next month we'll present an interview with Ray Konopka. Until then ... ∆

— Alan C. Moore, Ph.D.

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\SEP\DI9909AM.

Alan Moore is a Professor of Music at Kentucky State University, specializing in music composition and music theory. He has been developing education-related applications with the Borland languages for more than 10 years. He has published a number of articles in various technical journals. Using Delphi, he specializes in writing custom components and implementing multimedia capabilities in applications, particularly sound and music. You can reach Alan on the Internet at acmdoc@aol.com.