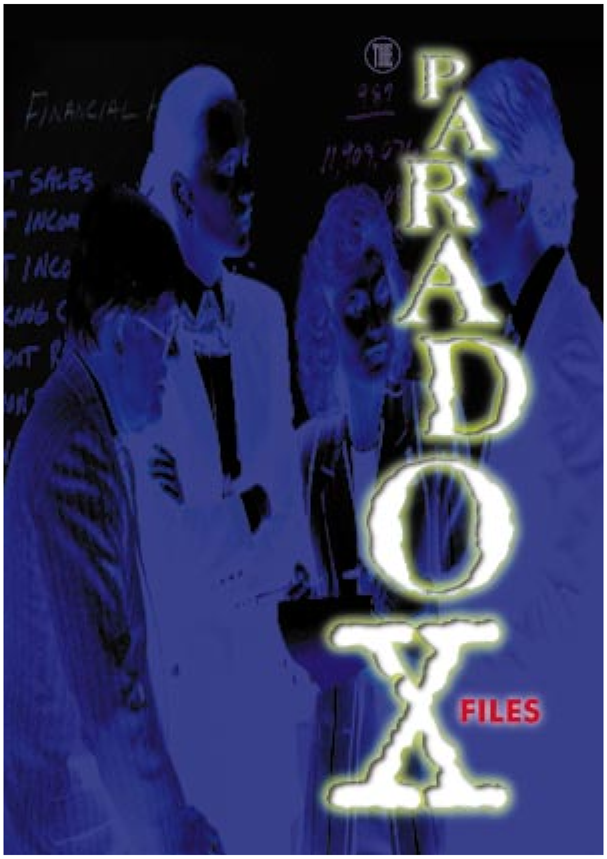
April 1997, Volume 3, Number 4

The Paradox Files

The Data Is Out There



Cover Art By: *Tom McKeith*

ON THE COVER

FEATURES

REVIEWS

DEPARTMENTS

Delphi TOOLS

New Products and Solutions

Videotex Launches Version 2.0 of T-BASE for Windows

Videotex Systems, Inc. of Dallas, TX has released *T-BASE 2.0* for Windows, an imaging library tool for Delphi, with added ActiveX controls.

Using the newest version of T-BASE, developers can image-enable any Windows 3.1, Windows 95, or Windows NT application that is ActiveX compatible or can call DLLs. In addition, controls and DLLs are available in 16- and 32-bit versions.

Version 2.0 can display, print, scan, and convert images, as well as play sound and video. Images are scanned using TWAIN support, enabling programmers to create applications that can control any scanner with a TWAIN driver.

T-BASE supports images in .JPG, .PCX, .TGA, .DCX, .GIF, .BMP, .DIB, and .TIF (including Group III/IV) file formats.

Using T-BASE 2.0, developers can create such imaging applications as ID badges, signature verifications, commercial real estate listings and floor plans, accounting records, and vehicle accident reports.

Price: T-BASE for Windows 2.0 components are priced separately from US$75 to US$595. Special pricing is available for registered users of earlier versions.
Contact: Videotex Systems, Inc.,
11880 Greenville Ave., Ste. 100,
Dallas, TX 75243
Phone: (800) 888-4336 or
(972) 231-9200
Fax: (972) 231-2420
E-Mail: software@videotexsystems.com
Web Site: http://www.videotex-systems.com

TurboPower Software Announces OnGuard for Delphi

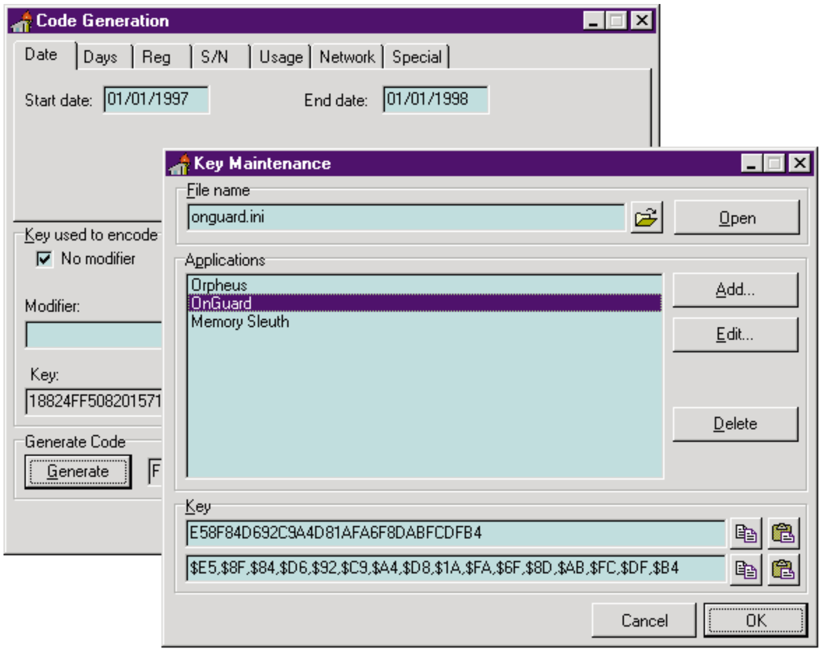
TurboPower Software Co. of Colorado Springs, CO has announced *OnGuard*, the newest product in its line of professional libraries and tools for Delphi 1 and 2.

OnGuard allows developers to create demonstration editions of their products for secure electronic distribution over the Internet. After sampling a downloaded program, customers can contact the developer to purchase a key code that unlocks other portions of the product.

OnGuard enables developers to choose from several protection mechanisms, including time-limited versions or versions that are limited to a particular number of executions. In addition, OnGuard lets developers add network metering to their programs, thereby limiting the number of simultaneous users of their programs in a network environment.

OnGuard ships with 16- and 32-bit support for Delphi 1 and 2. It is royalty free and includes full source code.

Price: US$199
Contact: TurboPower Software Co.,
P.O. Box 49009, Colorado Springs, CO
80949-9009
Phone: (800) 333-4160 or
(719) 260-9136
Fax: (719) 260-7151
E-Mail: info@tpower.com
Web Site: http://www.tpower.com



Start date: 01/01/1997 End date: 01/01/1998

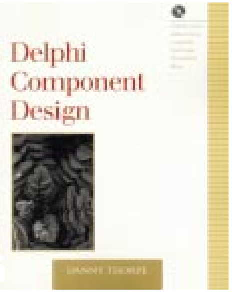
Key used to encode
☑ No modifier
Modifier:
Key: 18824FF508201571
Generate Code

File name: onguard.ini

Applications:
Orpheus
OnGuard
Memory Sleuth

Key:
E58F84D692C9A4D81AFA6F8DABFCDFB4
$E5,$8F,$84,$D6,$92,$C9,$A4,$D8,$1A,$FA,$6F,$8D,$AB,$FC,$DF,$B4

Delphi Component Design
Danny Thorpe
Addison-Wesley



ISBN: 0-201-46136-6
Price: US$36.95
(348 pages, CD-ROM)
Phone: (800) 822-6339

The Essential Guide to User Interface Design
Wilbert O. Galitz
John Wiley & Sons, Inc.

ISBN: 0-471-15755-4
Price: US$44.95
(626 pages)
Phone: (212) 850-6630

Perconti Ships New INI-Aware Components for Delphi 3

Perconti Data Systems, Inc. of St. Petersburg, FL has released *IAC's version 2.0*, a set of 19 INI-aware components for Delphi 3.

Although it's designed for Delphi 3, IAC for Delphi runs on all versions of Delphi. It allows developers to save user configuration options for any system developed.

The set includes the following INI-aware components: IASource, IAEdit, IALabel, IACheckbox, IAComboBox, IARadioGroup, IAListBox, IAEditCombo, IAMemo, IARadioButton, IASRButton, IASRBitBtn, IASRSpeedButton, IASaveDialog, IAOpenDialog, IADBComboBox, IASpinEdit, IAMaskEdit, and IAScrollBar.

Demonstration versions are available from Perconti's Web site.

All registered users of previous versions can receive a free upgrade.

Price: IAC's for Delphi version 2.0 (standard edition), US$69; IAC's for Delphi version 2.0, (professional edition with source code), US$99.
Contact: Perconti Data Systems, Inc.,
8601 Fourth Street North, Ste. 210,
St. Petersburg, FL 33702
Phone: (813) 576-7727
Fax: (813) 576-8033
E-Mail: info@perconti.com
Web Site: http://www.perconti.com

InstallShield Software Corp. Releases InstallShield5

InstallShield Software Corp. of Schaumburg, IL has released *InstallShield5 Professional*.

Using this upgrade, developers can create Windows installations through a visual interface. They can plan, script, compile, debug, and build in the same multi-paned development environment.

The new IDE features a built-in script editor window, which places each installation project into plain view. Functions, keywords, and other syntax elements are color-coded to help developers locate parts of the script when editing and debugging.

InstallShield5 Professional provides several wizards, including a Function Wizard, to assist developers in selecting a function and specifying its parameters.

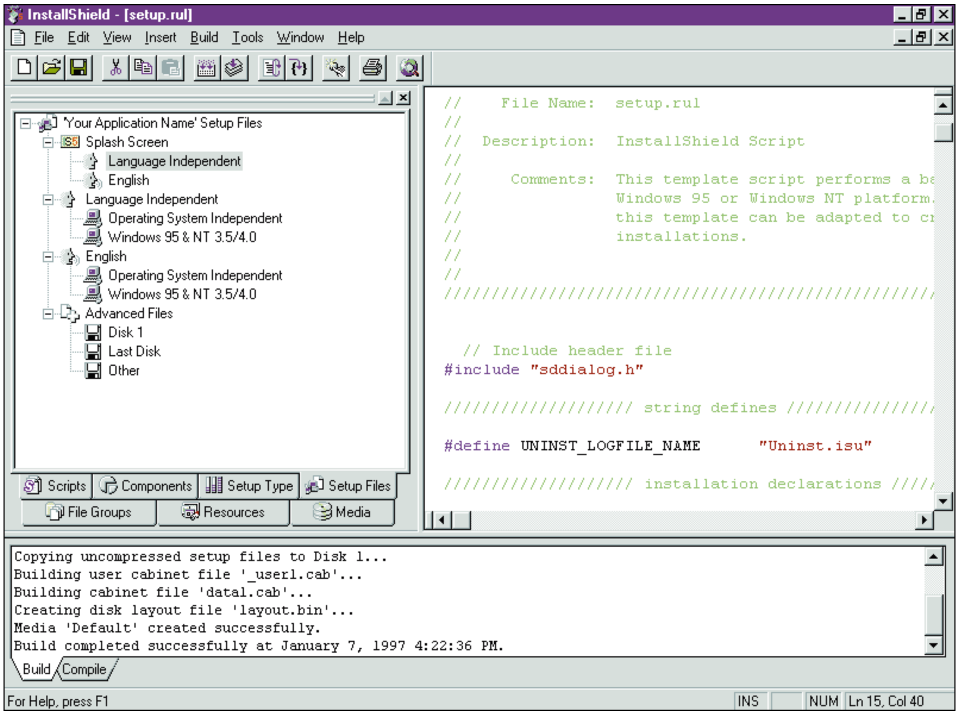
When creating the framework for a new installation project, developers can use the Project Wizard. It contains eight dialog boxes that allow developers to specify basic information about their applications and installations. The Wizard then builds and compiles a project, including a script, installation dialog boxes, and more.

Developers can also choose to create a Quick Build for testing purposes. In a Quick Build, files are referenced, but not included.

InstallShield5 Professional also allows developers to view and manage their file groups and components visually. No function calls are required to copy files. Developers can drag-and-drop to specify and group the files to be installed.

InstallShield5 Professional provides multimedia support for its application installations, including .AVI video, .WAV and MIDI sound, and 256-color bitmap images.

Price: US$795
Contact: InstallShield Software Corp.,
900 National Parkway,
Schaumburg, IL 60173
Phone: (800) 374-4353
Fax: (847) 240-9138
E-Mail: info@installshield.com
Web Site: http://www.installshield.com



```
//    File Name:  setup.rul
//
//  Description:  InstallShield Script
//
//     Comments:  This template script performs a ba
//                Windows 95 or Windows NT platform.
//                this template can be adapted to cr
//                installations.
//
//
//////////////////////////////////////////////////////


  // Include header file
#include "sddialog.h"

//////////////////// string defines ///////////////

#define UNINST_LOGFILE_NAME      "Uninst.isu"

//////////////////// installation declarations ////
```

```
Copying uncompressed setup files to Disk 1...
Building user cabinet file '_user1.cab'...
Building cabinet file 'data1.cab'...
Creating disk layout file 'layout.bin'...
Media 'Default' created successfully.
Build completed successfully at January 7, 1997 4:22:36 PM.
```

News LINE

April 1997

Oracle Licenses Borland's Java and C++ Development Tools

Scotts Valley, CA — Borland and Oracle Corp. have announced Oracle will license Borland's Java and C++ development technologies for use with Oracle database systems and application development tools.

Under the terms of the license agreement, Oracle will integrate and distribute Borland's C++Builder and JBuilder software tools with a number of Oracle's existing and future products, including Developer/2000, Designer/2000, and Oracle Power Objects (OPO).

The Oracle-Borland agreement is aimed at extending the tools available to Oracle customers for their Internet and intranet application development. As part of the license agreement, Borland and Oracle development teams will work together to promote integration and compatibility between technologies.

Borland C++Builder is a new object-oriented development tool that combines the C++ programming language with the RAD productivity of Delphi. A beta version of C++Builder is currently available to customers on Borland's Web site at http://www.borland.com.

JBuilder is Borland's upcoming Java application development tool. Scheduled to ship in the second quarter of FY 1998, JBuilder provides visual component-based development tools for cross-platform development. JBuilder is a development environment for projects such as Web-delivered applets, applications that require client/server database connectivity, and enterprise-wide distributed computing solutions.

"Oracle Licenses Borland's Java and C++ Development Tools" continued on page 5

Borland Announces Delphi/400 Client/Server Suite for IBM's AS/400

Scotts Valley, CA — Borland has announced Delphi/400 Client/Server Suite, a client/server development tool for the IBM AS/400 hardware platform.

The Delphi/400 Client/-Server Suite is based on Delphi Client/Server Suite 2.0 for Windows 95 and Windows NT, and AS/400-compatible connectivity and development technology recently licensed by Borland from TCIS of Paris, France. This RAD tool combines a visual component-based design, an optimized 32-bit native code compiler, and an open, scalable database architecture in an object-oriented environment.

With the addition of the ScreenDesigner/400 specialized object classes and ClientObject/400's APPC AS/400 connectivity, Delphi/400 Client/Server Suite delivers a secure and fast Windows interface for the AS/400.

In addition to this product launch, Borland announced it has joined IBM's AS/400 Partners in Development business program. Through this program, the two companies will work to provide Delphi-based software solutions to the AS/400 community.

Borland plans to demonstrate Delphi/400 Client/-Server Suite at three IBM trade shows this year: 1997 IBM Business Partner

"Delphi/400" continued on page 5

Searchable Delphi Knowledge Base Now on the Web

Marlboro, MA — Apogee Information Systems has released an online, interactive version of the DTopics database created by Mike Orriss. DTopics currently includes over 700 entries regarding Delphi programming topics.

Developed in IntraBuilder, the new Web application allows keyword- and description-based searches of the database. It's available free on Apogee's Web site at http://www.apogeeis.com.

Updates to the database will be made each time Orriss posts a new version on CompuServe. For more information call Apogee at (508) 481-1400.

[calendar 1997]

March

10-14 *Spring Internet World*, Los Angeles Convention Center, Los Angeles, CA. Visit http://www.-events.iworld.com.

19-21 *Internet World Asia '97*, Putra World Trade Center, Kuala Lumpur, Malaysia. Contact (800) 632-5537 or (203) 226-6967, or visit http://www.events.-iworld.com.

25-27 *DCI's Internet Expo*, London, England. Contact DCI at (508) 470-3870 or visit their Web site at http://www.DCIexpo.com.

April

1-3 *DCI's Web World*, Westin Swan, Orlando, FL. Contact DCI at (508) 470-3870 or visit their Web site at http://www.-DCIexpo.com.

2-4 *Software Development*, Moscone Convention Center, San Francisco, CA. Contact Miller Freeman at (415) 905-2784 or visit http://www.-sd97.com

8-10 *Network World Unplugged*, San Jose Convention Center, San Jose, CA. Contact DCI at (508) 470-3870, or visit their Web site at http://www.-DCIexpo.com.

8-11 *Internet World Argentina '97*, Buenos Aires Sheraton Hotel, Buenos Aires, Argentina. Contact (800) 632-5537 or (203) 226-6967, or visit http://www.-events.iworld.com.

8-11 *Internet World Brazil '97*, Ahembi Convention Center, Sao Paulo, Brazil. Contact (800) 632-5537 or (203) 226-6967, or visit http://www.events.iworld.com.

21-24 *Borland Developers Conference Europe*, London, England. Contact Borland at (408) 431-1000.

23-25 *Internet World Japan '97*, Makuhari Messe, Tokyo, Japan. Contact (800) 632-5537 or (203) 226-6967, or visit http://www.-events.iworld.com.

Southern California Delphi Developers' Conference Scheduled

Irvine, CA — The Orange County Delphi User Group (OCDUG), in association with the Chapman University chapter of the ACM, announced it will hold the second annual Southern California Delphi Developers' Conference at Chapman University in Orange on Saturday, May 10, 1997.

The conference will have various educational tracks aimed at both advanced and new Delphi developers.

The keynote presentation features Chuck Jazdzewski, co-developer and chief architect of Borland's Delphi. As with last year's conference, the date was selected to coincide with the release of a new version of Delphi.

The conference runs from 8 a.m. to 5 p.m. at Chapman University (287 North Center Street, Orange, CA 92666). Early registration costs US$99, but increases to US$149 after April 19. For more information, visit the OCDUG Web site at http://www.ocdelphi.org.

Borland Awards Software and Trademark for ReportSmith to Strategic Reporting Systems

Scotts Valley, CA — Borland announced it has awarded a world-wide software and trademark master license for the ReportSmith product line to Strategic Reporting Systems, Inc. of Peabody, MA.

As of December 31, 1996, Strategic Reporting assumed responsibility for worldwide service, maintenance, support, marketing, and sales of the ReportSmith family of products. Borland provided technical support for ReportSmith through January 31, 1997, and will continue to make it available to Delphi Client/Server Suite customers.

For more information, contact Strategic Reporting Systems at (508) 531-0905 or e-mail rsmith@sea-systems.com.

Oracle Licenses Borland's Java and C++ Development Tools (cont.)

Oracle products for developers include Designer/2000, Developer/2000, and OPO. Oracle's Designer/2000, a repository-based modeling tool, provides systems lifecycle support through business process re-engineering, systems analysis and design, and the automatic generation of client/server and Web applications.

Developer/2000 is the first Oracle tool to deliver scalable deployment of database applications in a network computing environment.

It incorporates application partitioning, drag-and-drop reusability, and through the use of Java, ensures low cost deployment on the Web.

OPO provides a drag-and-drop application development environment for BASIC programmers. It also eliminates the requirement to code database interaction while providing support for Windows, ActiveX, and other desktop standards.

For more information about Oracle products, call (415) 506-7000 or visit Oracle's Web site at http://www.oracle.com.

Delphi/400 (cont.)

Executive Conference; AS/400 User Group Conference; and AS/400 User Group Conference Europe. The Delphi/400 Client/Server Suite pricing starts at US$3,995.

For more information on Delphi/400, call (800) 233-2444.

The Borland Canada Developer's Conference

Canadian developers are planning to meet May 21-23 in Toronto to discuss the latest in software technologies and trends at the Borland Canada Developer's Conference. For more information, call (800) 265-1362, (416) 444-1071, e-mail webmaster@dbcenteral.com, or visit http://www.dbcentral.com.

table of CONTENTS

By Dan Ehrmann

The Paradox Files: Part I

The Table Format, Header, Family Files, Data Blocks, and Table Levels

Once there was a database program called Paradox. In the beginning, it knew no distinction between Windows and DOS; there was just Paradox, with its indivisible user interface, programming language, and file format. But when Borland introduced Paradox for Windows in January of 1993, it separated Paradox into two components: the user interface and the file format, connected by ODAPI, the Object Database API.

Over time, ODAPI became IDAPI, which in turn became the Borland Database Engine, or BDE. Paradox, the file format, took on a separate identity from Paradox, the desktop database tool. More recently, Borland sold the desktop-database Paradox to Corel, who will move it forward as part of their Office product. Now the Paradox file format is simply one of the native formats in the BDE, which is shipped with every one of Borland's developer tools, including Delphi, C++, C++Builder, JBuilder, and IntraBuilder.

Extension	Description
.DB	Table definition and all data except memo, graphic, and BLOb fields.
.MB	Memo, graphic, and BLOb data.
.PX	The index for the table's primary key.
.X*nn* and .Y*nn*	Each pair of files represents a separate secondary index. The naming convention for *nn* will be explained in the third article of this series.
.VAL	ValCheck, Table Lookup, and Referential Integrity definitions.

Figure 1: Paradox file format extensions.

Extension	Description
.TV	TableView settings (for Database Desktop and Paradox for Windows only).
.FAM	Linking file created with the .TV file (for Database Desktop and Paradox for Windows only).
.SET	TableView settings in Paradox for DOS.
.F and .F1-.F14	Forms attached to the table in Paradox for DOS.
.R and .R1-.R14	Reports attached to the table in Paradox for DOS.

Figure 2: Obsolete Paradox file format extensions.

Many Delphi developers use Paradox tables every day, but remarkably little information circulates in the Delphi community about how the Paradox file format really works. The manuals certainly explain almost nothing. Yet the Paradox file format is rich and robust, with many features to serve even the most demanding desktop applications.

This series of articles will explain how Paradox tables are structured internally, and what really happens when you add or remove a field from a table, create an index, pack a table, present a password, define referential integrity, or modify a record on a network.

Databases and Files

Microsoft Access defines a database as a single file with its own internal file system. Inside the typical .MDB file are tables, indices, relationships, forms, reports, queries, and code. Paradox takes the opposite approach; a *database* is loosely defined as a subdirectory containing related tables. Elements of each table are stored in separate files, which are loosely linked together as a *family*. You control the file name, and the BDE controls the extension. Figure 1 lists the different extensions for the Paradox file format.

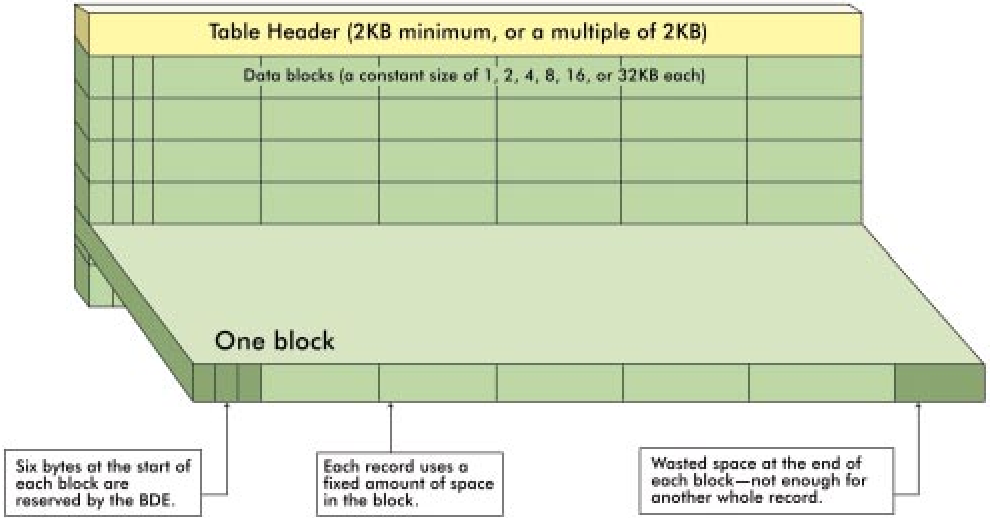


Table Header (2KB minimum, or a multiple of 2KB)
Data blocks (a constant size of 1, 2, 4, 8, 16, or 32KB each)
One block
Six bytes at the start of each block are reserved by the BDE.
Each record uses a fixed amount of space in the block.
Wasted space at the end of each block—not enough for another whole record.

Figure 3: Structure of a Paradox table.

(Older versions of the Paradox file format supported other files as part of the table's family. Figure 2 lists the extensions used by older versions of the Paradox file format and by the Database Desktop shipped with Delphi. These files don't need to be included with your tables when you deliver an application to a client.)

When you copy a table, you must copy all linked family members as well. Otherwise, the family may become obsolete, relative to the .DB file, creating all manner of problems when your application needs to read information from a family file.

The Table Format

Internally, the Paradox file format is known as a *clustered-key, VSAM-block, fixed-length record* file format, meaning that:

- Data is stored in records that occupy a fixed length in the table, irrespective of the number of characters in each field. For example, a 20-character alphanumeric field always occupies 20 characters inside the .DB file, even if many values are shorter than 20 characters, or are blank.
- Records are organized into physical blocks, which can be accessed sequentially or randomly. Each block contains one or more records. Records do not span blocks, often resulting in empty space at the end of each block not large enough to contain another record.
- Blocks — and the records within each block — are organized so that records are physically stored in primary key sequence. This feature significantly improves access speed when the primary key is used.

Figure 3 shows the structure of a table. It begins with a table header of at least 2KB, and may be a larger multiple of 2KB, depending on the information to be stored there, including the following (most of which will be explained in this and subsequent articles):

- The table level.
- The Structure ID which is used to keep family members synchronized. When the BDE opens one of the family files, it checks to see that the Structure ID in the header of that file is the same as the ID in the .DB file. If they are different, it means that the .DB was updated apart from its family members. Whenever you restructure a table, the Structure ID is refreshed in all family members.
- The header size in bytes.
- The record size in bytes.
- The block size, being one of the following values: 1KB, 2KB, 4KB, 8KB, 16KB, or 32KB.
- Whether the table is keyed or unkeyed.
- The number of fields in the table, from 1 to 255.
- If the table is keyed, the number of fields in the table's primary key.
- An array of field types and sizes, indexed by the field number.
- An array of field names, indexed by the field number.
- The number of data blocks in the table, a two-byte value that can track up to 64KB blocks.
- A pointer to the first data block, used to initialize the forward chain.
- A pointer to the last data block, used to initialize the backward chain.
- The number of free blocks in the table, also a two-byte value.
- A pointer to the first free block.
- The table language, used to define character translations and sort orders.
- The master password — itself encrypted — that was used to encrypt the data in the table. Note that the header is not encrypted, so it is always possible to read an encrypted table's structure and other information from the header.
- An encrypted list of auxiliary passwords defined for the table, together with the table and field rights defined for each password.
- The current value of the table's Autoincrement field (only one is allowed per table).

Some elements of the table's header support Paradox for DOS features (e.g. password rights for linked forms and reports, whether the table maps to a remote table on a database server). These areas of the header are ignored by the BDE.

Data Blocks

When you create a table, no data blocks are allocated. The table consists of the header only. When you then insert the first record, a data block is appended to the .DB file immediately following the header.

Data blocks can be 1KB, 2KB, 4KB, 8KB, 16KB, or 32KB in size; the default is 2KB, but this can be changed in the BDE Administration program. (Note that the 1KB size is considered obsolete, and is normally not used.) Block size does not vary within a table; when the table is created, the BDE picks a block size and will use that size for every data block appended to the .DB file.

Coupled with the 64KB limit in the number of data blocks, the maximum table size (excluding header and free blocks) is shown in Figure 4. In the real world, the Paradox file format grows unreliable, well before these theoretical limits are reached. For example, restructuring tables larger than 100 or 200MB can take hours, especially when updating many secondary indices.

The Paradox file format reserves six bytes at the beginning of each block for three internal pointer series of two bytes each, as follows:

- The sequential number of the next logical block. This series of pointers is known as the *forward chain*. BDE computes the physical starting address of the block by multiplying the block number by its size, and adding the size of the header.
- The sequential number of the previous logical block. This series of pointers is known as the *backward chain*.
- A counter to indicate how many active records are contained within that block. Data records are always stored at the front of the block, so this pointer, coupled with the record size stored in the header, tells the BDE exactly how much of the block contains active data.

When a block is completely empty, it's moved to the *free-block chain*. This is a separate chain of available blocks used when the BDE must allocate another data block for the table. Free blocks have the same three pointers at their beginning, although the "number of records" pointer is always zero for a block in this chain.

The BDE will use the block size specified in the Paradox driver section of your BDE configuration file, unless this block is too small for a single record — in the case of an unkeyed table — or three records in the case of a keyed table. If the specified block size is too small, the BDE uses the smallest block capable of holding that number of records, based on the table type. It will then fit as many records as it can within each block.

(The BDE uses a minimum of three records for a keyed table because modern disk controllers read a whole block from the hard disk, anyway. Research has determined that, when your program is searching for a specific record, it's invariably quicker to read a complete block into memory, then locate the record sequentially within the block.)

The BDE sometimes rounds the result of the record size calculation. For example, with a block size of 4KB (4096 bytes), you might think that the maximum record size for a keyed table would be (4096 - 6) / 3 = 1363 bytes.

Instead, in a holdover from the days when 4KB was the largest supported block size, the BDE considers 1350 bytes as the maximum record size for a keyed table using a 4KB block size. If your record is larger than 1350 bytes, the BDE uses an 8KB block for a keyed table.

Insert and Delete Records

Let's consider a keyed table with a record size of 204 bytes. (Don't worry about how we arrived at this number; you'll learn how to calculate record size in the next article.) Within a Paradox table, the BDE will use a 2KB block size, and will fit 10 records into each block, using 2040 bytes in the block for data, with six bytes reserved at the beginning of the block, and only two bytes of wasted space.

Now let's assume you insert four records into this table. The BDE will fill the first four record slots in the block, and set the record counter to "4", as shown in Figure 5. (The first cell in each of the following figures is the count of active records in that block. Active records are shaded, while available record slots are unshaded.)

If you then delete record "C", the BDE moves record "D" up to occupy slot number 3 — formerly occupied by "C". The space formerly used by "D" is not blanked out, so the original version of "D" still exists. But the record counter is set to "3", to indicate that only this many active records remain in the block. This scenario is shown in Figure 6.

The next record inserted into the table will cause the obsolete copy of "D" in record slot number 4 to be overwritten. For example, if you insert a record after "A", then "B" and "D" are pushed over one record slot to maintain the keyed order shown in Figure 7.

If you insert another six records into the table, the block will be filled, as shown in Figure 8. When you insert the eleventh record into the table, the BDE determines that the current block is full, and that it must allocate another block. When the new block is appended, the BDE also moves the last record from the previous block, to leave an empty slot in that block. This is done so a subsequent insert between "A" and "J" does not require a new block to be allocated, or a split to take place. In the previous example, record "J" is moved to the new block, and the new record "K" is added after this one, as shown in Figure 9.

Note that this is different from the *fill factor* parameter referenced in the BDE configuration program for the Paradox file format. The fill factor applies to index files only; we will discuss it in the third article of this series.

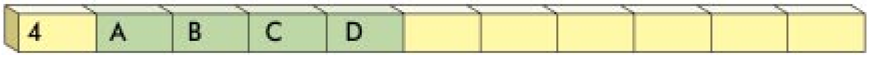
Suppose you next insert two records in this keyed table between records "E" and "F". Assume these are called "E1" and "E2". Since there is only one free slot in this block, "E1"

Data Block Size	Maximum Table Size
1KB	64MB
2KB	128MB
4KB	256MB
8KB	512MB
16KB	1024MB
32KB	2048MB

Figure 4: Maximum table size for different block sizes.

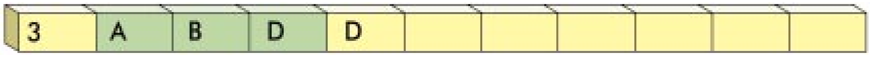
is first placed in the existing block immediately following "E", causing the remaining records to be pushed back one slot each, and the last slot to be filled by record "I". When "E2" is inserted, the block must be split again. But because you are not inserting at the end of the block, the split happens at the insertion point, with all records after this one placed in a new block, as shown in Figure 10.

In the previous example, the new block is added at the end of the table's file. The BDE then renumbers the forward and backward chains for each of the three blocks. The forward chain points to the blocks in this order: 1-3-2. The backward chain points to the blocks in 2-3-1 order.



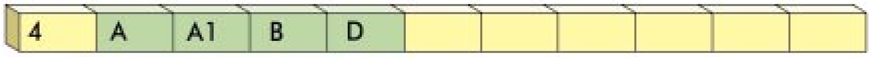
4	A	B	C	D						

Figure 5: Inserting four records into a table.



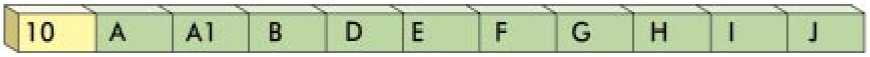
3	A	B	D	D						

Figure 6: Deleting the third record.



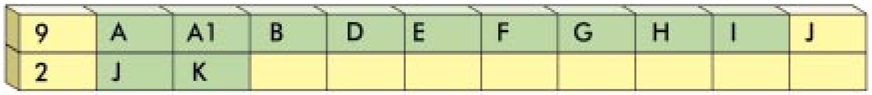
4	A	A1	B	D						

Figure 7: Inserting a new record into the table.



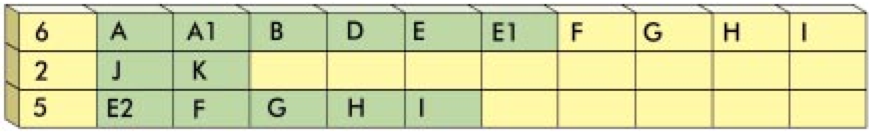
10	A	A1	B	D	E	F	G	H	I	J

Figure 8: Inserting another six records into the table.



9	A	A1	B	D	E	F	G	H	I	J
2	J	K								

Figure 9: The BDE adds a new block and applies the fill factor.



6	A	A1	B	D	E	E1	F	G	H	I
2	J	K								
5	E2	F	G	H	I					

Figure 10: The BDE adds another block.

If you simply restructure the table using the Database Desktop, the BDE does not remove empty record slots. It also does not physically rearrange the blocks. However, if you restructure with the Pack Table option selected, the BDE rebuilds the table and "squeezes out" any unused space. Each block is completely filled, and the free-block chain is emptied. The BDE also re-sequences the blocks to have the same order as the forward chain.

The fill factor applies only to keyed tables. With unkeyed tables, there is less chance that a record will need to be inserted into the table in a specific sequence, so the BDE does not attempt to leave slots available.

Managing Block Use

Some tables grow continuously and almost never have records removed. If records are always added at the end of the table — for example, in an Orders table where the primary key is an incrementing value — the table will grow continuously and all data blocks will be largely filled.

On the other hand, if new records are scattered throughout the table — for example, in a Customer table where the primary key starts with the customer's name — the table will grow more quickly because many blocks will have been split in order to keep records in primary-key sequence. Therefore, many blocks will not yet have been filled.

But consider also a table with a continuous stream of both added and deleted records, such as the Orders table previously described, where paid orders are moved to an archive table. It will tend to grow to an equilibrium size and stay there. As records are deleted near the beginning of the table, blocks will be emptied, and will move to the free-block pool, to be used when new blocks are required at the table's end.

If you frequently insert or delete large numbers of records, and if the table is often queried, it's a good idea to restructure and pack the table after such operations. Compressing and reordering the data blocks in the .DB file shrinks the table and places its blocks in natural order. It also compresses the primary index file (.PX), which indexes only the first record in each block (as we'll see in a subsequent article).

If your table has many partially filled blocks, compressing it can reduce the size of the primary index substantially. In addition, because the complete primary index is contained within every maintained secondary index, compressing the table will shrink secondary index files, and improve their performance as well.

Table Levels

The BDE defines a *level* for each Paradox table, based on the features it uses. Over the years, Borland has added new field and index types to the Paradox file format, and each change necessitated a bump in the level number to ensure that the BDE would correctly handle the newer features.

Note: When the BDE opens a table, it first checks the level, and if it finds a higher number than it knows how to handle, raises an exception.

Level 4 corresponds to Paradox 4.0 for DOS and the first version of the BDE. With this level, Borland substantially modified Paradox's network locking model to improve multi-user performance. They also added new field types, including memo fields, and new index types, including multi-field and case-insensitive indices. Tables that were originally created under earlier versions of Paradox for DOS are treated by the BDE as Level 4 tables.

Tables at Levels 4, 5, and 7 are fully compatible with each other, and can be used concurrently; you don't need to use Level 7 to achieve full 32-bit compatibility. (There is no Level 6.) When you define a new table, the BDE picks the lowest level compatible with the features you specified, but never lower than the level specified in the BDE configuration file. For example, if you don't specify descending secondary indices, the BDE will not use a Level 7 table. If you restructure a table to add a feature that requires a higher level, the BDE changes the level accordingly.

The default table level for new tables is defined in the BDE Administrator program. You can use this parameter to force Level 7 for all tables, although this isn't necessary. Aside from field changes described in next month's article, and index changes described in a subsequent article, Level 5 added support for the 8KB, 16KB, and 32KB block sizes.

Conclusion

The next article in this series will explore the different types of fields in the Paradox file format. It will explain the characteristics of each field, and show you how much space each uses in the .DB file.

With this information in hand, you will learn how to calculate record size and the minimum possible table size. The next article also includes a simple Delphi application to calculate this information for any specified table. ∆

Dan Ehrmann is the founder and President of Kallista, Inc., a database and Internet consulting firm based in Chicago. He is the author of two books on Paradox, and a member of Team Borland and Corel's CTech. Dan was the Chairman of the Advisory Board for Borland's first Paradox conference, which evolved into the current BDC. He has worked with the Paradox file format for more than 10 years. He can be reached via e-mail at dan@kallista.com.

DELPHI AT WORK

Delphi 2

By *Ian Davies*

Rich Text Control

Building a Poor Man's Word Processor with the RichEdit Component

Sometimes the Memo component just doesn't do the trick. Sure, it allows you to enter multiple lines of text (255KB in Delphi 1, unlimited in Delphi 2), but there's not much you can do with the text to add interest or emphasis. With the RichEdit component, however, you can — and a lot more.

The RichEdit component has many features not supported by the Memo component. With it, you can:

- change the typeface, style, size, and color of individual characters in the text;
- alter individual paragraph alignment;
- facilitate the use of paragraph indents, numbering, and tabs;
- open, save, print, and search the contents of the control; and
- drag and drop selected text within the component, or to other components or word processors (such as Microsoft Word) while retaining the text's font attributes.

This article will describe how to use the RichEdit component to create a basic word processor. (The RichEdit component is based on a Windows 95 common control, it is not available in Delphi 1.)

Property	Description
DefAttributes	Default attributes — Sets or returns the default styles to be applied to text being entered in the component.
SelAttributes	Selected attributes — Sets or returns formatting to apply to the selected contents of the component.
Paragraph	Paragraph attributes — Controls the paragraph formatting applied to the current or selected paragraphs.

Figure 1: The fundamental properties of the RichEdit component.

Basic Principles

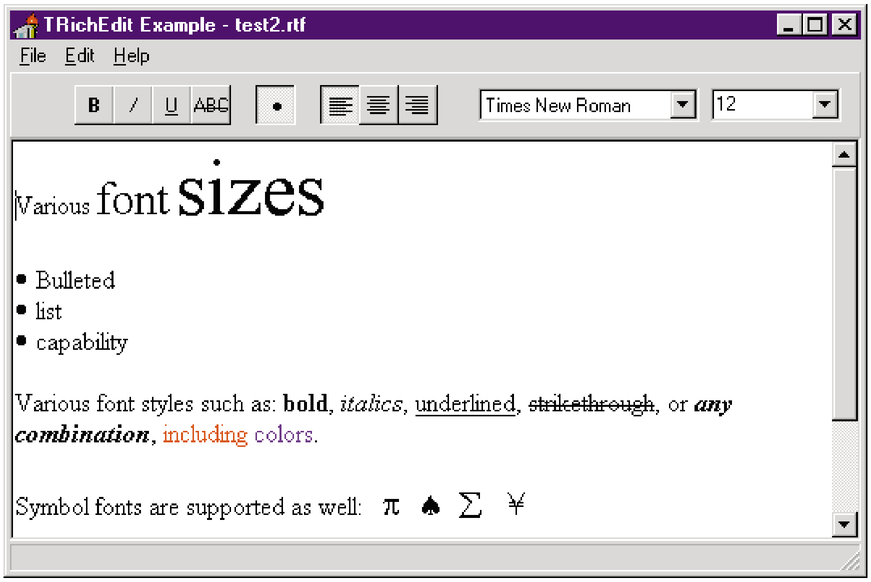
The fundamental properties for manipulating the text style in the RichEdit control are *DefAttributes*, *SelAttributes*, and *Paragraph* (see Figure 1). *DefAttributes* (which is of type *TTextAttributes*) sets or returns the default styles to be applied to text being entered into the component (i.e. the style that will be applied to text if no other style is applied). *SelAttributes* (also of type *TTextAttributes*) sets or returns the formatting to be applied to a particular selection in the component. As its name implies, the *Paragraph* property (of type *TParaAttributes*) controls the paragraph formatting applied to the current or selected paragraph(s). The *TTextAttributes* object implements the functionality surfaced through the *DefAttributes* or *SelAttributes* properties.

The following code sets the *Style* property of *SelAttributes* to *fsBold*, which means the font style of the currently selected area in the RichEdit component will appear in bold:

```
RichEdit1.SelAttributes.Style := [fsBold];
```

Other properties of *TTextAttributes* enable you to set the font, its size, and its color.

RichEdit also has two methods for handling printing and searching: *Print* and *FindText*. These methods, combined with standard dialog boxes, enable us to quickly build a functional example.



Various font sizes

- Bulleted
- list
- capability

Various font styles such as: **bold**, *italics*, underlined, ~~strikethrough~~, or ***any combination***, including colors.

Symbol fonts are supported as well: π ♠ ∑ ¥

Figure 2: This sample application shows how the RichEdit component can be used to create a simple word processor.

A Working Example

Our sample application uses the power of RichEdit to implement much functionality in a few lines of code (see Listing One beginning on page 13). Figure 2 shows the sample application.

In the toolbar, the Left, Center, and Right paragraph alignment speedbuttons set the *Paragraph* property of RichEdit to *taLeftJustify*, *taCenter*, and *taRightJustify*, respectively.

Similarly, the Bullet speedbutton sets the *Numbering* property of the *TParaAttributes* object to *nsBullet* or *nsNone*, depending on whether the speedbutton's *Down* property is set to *True* or *False*. Finally, the font name and size combo boxes set the *Name* and *Size* properties of *SelAttributes* to their respective values.

The style property of the *TTextAttributes* (which is of type *TFontStyles*) is a set of *TFontStyle*. This allows the structure to contain more than one value, so, for example, a font can be bold *and* italic. You can implement this by adding or subtracting the requested font style to the current style:

```
with RichEdit1.SelAttributes do
  Style := Style + [fsBold];
```

This will make the current selection bold, without affecting the other attributes that may have been set. For example, if the font were italic, it will now be bold *and* italic.

RichEdit contains an event, *OnSelectionChange*, which is called whenever the user changes the selected text with the keyboard or the mouse. This event is useful for implementing changes in the state of, for example, the font style speedbuttons. In our sample program, the *Style* attributes are retrieved whenever the selection changes; the style button's *Down* property and the font name and size are set accordingly.

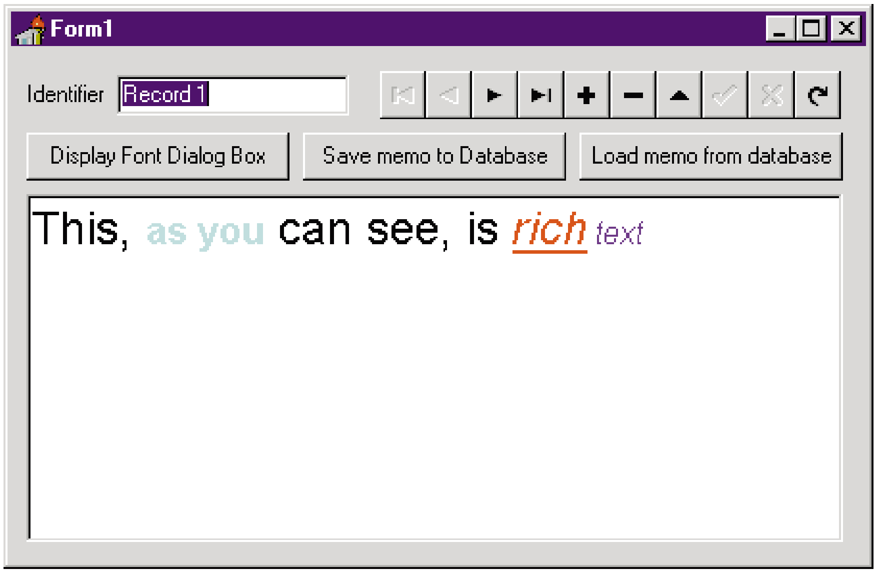
Saving documents uses the standard Windows *TSaveDialog* and a call to the *SaveToFile* method of the *Lines* property of RichEdit. Similarly, existing documents are opened using the *TOpenDialog* component and a call to the *LoadFromFile* method.

RichEdit saves files in Rich Text Format (.RTF), which is compatible with virtually all leading word processors. You also have the option to save the contents as unformatted text, by setting the *PlainText* property to *True*, whereby RichEdit behaves similar to a standard Memo component.

The Print, Print Setup, and Font dialog boxes are standard Windows dialog boxes displayed using their *Execute* methods. Following the display of the Print dialog box, *TRichEdit*'s *Print* method is called to carry out the printing.

A parameter can be supplied that will be used as the document's title when displayed in the print queue. Similarly, provided the OK button of the *Font* dialog box was clicked, the *Font* property of the Font dialog box component is assigned directly to the *SelAttributes* property.

Cut, Copy, and Paste are implemented by calling the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods of RichEdit. Undo, however, does not have a cor-



Identifier Record 1

This, as you can see, is rich text

Figure 3: This sample application demonstrates how RichEdit can be made data aware.

responding method, and, therefore, cannot be called in this way. Instead, we must send the Windows message *WM_UNDO* to the control using the *SendMessage* function.

The Find facility, which locates specific items of text, is achieved differently. The dialog box is displayed using the *TFindDialog Execute* method, but the code that handles the searching of items in the RichEdit control is placed in the Find dialog box's *OnFind* method. This method is called whenever the user clicks the **Find Next** button in the Find dialog box.

The code in the sample application then attempts to locate the text (that the user entered) using the *POS* function. If the text is found, it's highlighted in the RichEdit control; otherwise, an appropriate message is displayed to the user.

This is a basic attempt at creating a useful application using the RichEdit control. Possible enhancements include adding a search and replace facility, updating the Find code to find the next occurrence of a particular string, and implementing a recently used file list.

If you have the confidence to implement the RichEdit component in your application or expand upon the simple application provided here, then I have achieved my primary goal.

Rich Text in Databases

Since version 1, Paradox has provided a way of storing rich text data in a database, and using it in its forms. This facility is notably lacking from versions 1 and 2 of Delphi; despite a Formatted Memo field being available through the Database Desktop, the Database Desktop doesn't provide a means of entering or viewing data contained in it. (Borland has filled the hole, providing a *TDBRichEdit* control in the forthcoming Delphi 3. This is used in the same way as RichEdit, but now has *DataSource* and *DataField* properties.)

Creating a data-aware RichEdit in Delphi 2 is possible by reading and writing the contents of a RichEdit in a Formatted Memo field using a *TBlobField* object:

```
Table1.Edit;
BlobStream:=
  TBlobStream.Create(Table1FormattedMemo,bmWrite);
RichEdit1.Lines.SaveToStream(BlobStream);
BlobStream.Free;
Table1.Post;
```

When the BlobField is created (here it's named `BlobStream`), the field object that will store the rich text is passed as a parameter to the *Create* method. The `BlobStream` object is then passed as a parameter to the RichEdit *LoadToStream* and *SaveToStream* methods, to load and save the rich text in the appropriate field of the current record (see Figure 3). The entire program is shown in Listing Two on page 16.

Conclusion

Back in the days of MS-DOS, plain text was acceptable, even expected. Word processors offered the ability to apply different fonts and font styles to the printed output, but only the most expensive provided the ability to see the effects on the screen. With Windows, rich text is the norm, and anything less just isn't good enough.

In this article, I have demonstrated how the powerful RichEdit component can be used as the basis of a simple word processor, and, further, how the contents of a RichEdit component can be easily stored in a database. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\APR\DI9704ID.

Ian Davies is a developer of 16- and 32-bit applications for the Inland Revenue in the UK. He began Windows programming using Visual Basic about four years ago, but has seen the light and is now a devout Delphi addict. Current interests include Internet and intranet development, inter-application communication, and sometimes a combination of the two. Ian can be contacted via e-mail at 106003.3317@compuserve.com.

Begin Listing One — A RichEdit Word Processor

```
unit CEditForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, Buttons, ExtCtrls, StdCtrls,
  ComCtrls, Menus;

type
  TEditForm = class(TForm)
    RichEdit1: TRichEdit;
    Panel1: TPanel;
    BoldButton: TSpeedButton;
    ItalicButton: TSpeedButton;
    UnderlineButton: TSpeedButton;
    BulletsButton: TSpeedButton;
    LeftAlign: TSpeedButton;
    RightAlign: TSpeedButton;
    CenterAlign: TSpeedButton;
    FontName: TComboBox;
    FontSize: TComboBox;
    StatusBar1: TStatusBar;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    N1: TMenuItem;
    PrintSetup1: TMenuItem;
    Print1: TMenuItem;
    N2: TMenuItem;
    SaveAs1: TMenuItem;
    Save1: TMenuItem;
    N3: TMenuItem;
    Open1: TMenuItem;
    New1: TMenuItem;
    PrinterSetupDialog1: TPrinterSetupDialog;
    PrintDialog1: TPrintDialog;
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    Edit1: TMenuItem;
    Find1: TMenuItem;
    N5: TMenuItem;
    Paste1: TMenuItem;
    Copy1: TMenuItem;
    Cut1: TMenuItem;
    N6: TMenuItem;
    Undo1: TMenuItem;
    FindDialog1: TFindDialog;
    Help1: TMenuItem;
    About1: TMenuItem;
    HowtoUseHelp1: TMenuItem;
    SearchforHelpOn1: TMenuItem;
    Contents1: TMenuItem;
    ColorDialog1: TColorDialog;
    Font1: TMenuItem;
    N4: TMenuItem;
    FontDialog1: TFontDialog;
    StrikeoutButton: TSpeedButton;
    function ContinueEvenIfDirty: Word;
    procedure FormCreate(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure New1Click(Sender: TObject);
    procedure Open1Click(Sender: TObject);
    procedure Save1Click(Sender: TObject);
    procedure SaveAs1Click(Sender: TObject);
    procedure Print1Click(Sender: TObject);
    procedure PrintSetup1Click(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure FormClose(Sender: TObject;
      var Action: TCloseAction);
    procedure Undo1Click(Sender: TObject);
    procedure Cut1Click(Sender: TObject);
    procedure Copy1Click(Sender: TObject);
    procedure Paste1Click(Sender: TObject);
    procedure Font1Click(Sender: TObject);
    procedure Find1Click(Sender: TObject);
    procedure FindDialog1Find(Sender: TObject);
    procedure RichEdit1SelectionChange(Sender: TObject);
    procedure BoldButtonClick(Sender: TObject);
    procedure ItalicButtonClick(Sender: TObject);
    procedure UnderlineButtonClick(Sender: TObject);
    procedure StrikeoutButtonClick(Sender: TObject);
    procedure BulletsButtonClick(Sender: TObject);
    procedure LeftAlignClick(Sender: TObject);
    procedure FontNameChange(Sender: TObject);
    procedure FontSizeClick(Sender: TObject);
    procedure FontSizeKeyPress(Sender: TObject;
      var Key: Char);
    procedure About1Click(Sender: TObject);
  private
    { Private declarations }
    FName: ShortString;
  public
    { Public declarations }
  end;

var
  EditForm: TEditForm;

implementation

{$R *.DFM}

function TEditForm.ContinueEvenIfDirty: Word;
begin
  Result := mrNo;
  if RichEdit1.Modified then
    Result := MessageDlg(
     'Changes were made. Do you wish to save them?',
      mtConfirmation, mbYesNoCancel, 0);
  if Result = mrYes then Save1Click(nil);
end;

procedure TEditForm.FormCreate(Sender: TObject);
begin
  //Initialise the variables.
  FontName.Clear;
  FontName.Sorted := True;
  FontName.Items := Screen.Fonts;
  OpenDialog1.InitialDir := ExtractFilePath(
    Application.ExeName);
  SaveDialog1.InitialDir := ExtractFilePath(
    Application.ExeName);
  FName := '';
end;

procedure TEditForm.FormActivate(Sender: TObject);
begin
  //Update the state of the indicators on the toolbar.
  RichEdit1SelectionChange(Sender);
end;

procedure TEditForm.New1Click(Sender: TObject);
begin
  //Check if changes have been made and,
  //if OK to continue, start from scratch.
  if ContinueEvenIfDirty = mrNo then
  begin
    RichEdit1.Lines.Clear;
    FName := '';
    EditForm.Caption := 'TRichEdit Example';
    RichEdit1.Modified := False;
  end;
end;

procedure TEditForm.Open1Click(Sender: TObject);
begin
  //Check if changes have been made and,
  //if OK to continue, load the RTF file.
  if ContinueEvenIfDirty = mrNo then
  begin
    if OpenDialog1.Execute then
    begin
      try
        Screen.Cursor := crHourGlass;
        Application.ProcessMessages;
        RichEdit1.Lines.LoadFromFile(OpenDialog1.FileName);
        FName := ExtractFileName(OpenDialog1.FileName);
        EditForm.Caption := 'TRichEdit Example - ' + FName;
        RichEdit1.Modified := False;
      finally
        Screen.Cursor := crDefault;
      end;
    end;
  end;
end;

procedure TEditForm.Save1Click(Sender: TObject);
begin
  if FName = '' then
  begin
    //Saving for the first time.
    SaveAs1Click(Sender);
  end
  else
  begin
    //Saved before.
    RichEdit1.Lines.SaveToFile(FName);
    RichEdit1.Modified := False;
  end;
end;

procedure TEditForm.SaveAs1Click(Sender: TObject);
begin
  //Check if changes have been made and,
  //if OK to continue, save the file.
  if SaveDialog1.Execute then
 begin
    RichEdit1.Lines.SaveToFile(SaveDialog1.FileName);
    EditForm.Caption := 'TRichEdit Example - ' +
      ExtractFileName(SaveDialog1.FileName);
    RichEdit1.Modified := False;
  end;
end;

procedure TEditForm.Print1Click(Sender: TObject);
var
  loop: Integer;
begin
  //Display the print dialog box.
  if PrintDialog1.Execute then
  begin
   //Print the required number of copies.
```

```
    for loop := 1 to PrintDialog1.Copies do
      RichEdit1.Print('TRichEdit Example : ' + FName);
  end;
end;

procedure TEditForm.PrintSetup1Click(Sender: TObject);
begin
  //Display the print setup dialog box.
  PrinterSetupDialog1.Execute;
end;

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

procedure TEditForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  //Check if changes have been made and,
  //if OK to continue, free the form.
  if ContinueEvenIfDirty = mrNo then
    Action := caFree
  else
    Action := caNone;
end;

procedure TEditForm.Undo1Click(Sender: TObject);
begin
  if ActiveControl is TRichEdit then
  begin
    //Send the WM_UNDO message to the TRichEdit.
    SendMessage(ActiveControl.Handle, WM_UNDO, 0, 0);
  end;
end;

procedure TEditForm.Cut1Click(Sender: TObject);
begin
  RichEdit1.CutToClipBoard;
end;

procedure TEditForm.Copy1Click(Sender: TObject);
begin
  RichEdit1.CopyToClipBoard;
end;

procedure TEditForm.Paste1Click(Sender: TObject);
begin
  RichEdit1.PasteFromClipBoard;
end;

procedure TEditForm.Font1Click(Sender: TObject);
begin
  //Set the initial values in the Font
  //dialog box to the current style.
  FontDialog1.Font.Assign(RichEdit1.SelAttributes);
  if FontDialog1.Execute then
    //Set the current style to the values
    //in the Font dialog box.
    RichEdit1.SelAttributes.Assign(FontDialog1.Font);
  RichEdit1.SetFocus;
end;

procedure TEditForm.Find1Click(Sender: TObject);
begin
  FindDialog1.Execute;
end;

procedure TEditForm.FindDialog1Find(Sender: TObject);
var
  TextPos: integer;
begin
  //Find the first occurrence of the selected text.
  TextPos := Pos(FindDialog1.FindText, RichEdit1.Text);
  //If the text is found ..
  if TextPos > 0 then
  begin
    // .. highlight it in the RichEdit control.
    EditForm.BringToFront;
    RichEdit1.SelStart := TextPos - 1;
    RichEdit1.SelLength := Length(FindDialog1.FindText);
  end
  else
    MessageDlg('Text not found', mtInformation, [mbOK], 0);
end;

procedure TEditForm.RichEdit1SelectionChange(
  Sender: TObject);
begin
  //Set the state of the items in the toolbar.
  BoldButton.Down := fsBold in
    RichEdit1.SelAttributes.Style;
  ItalicButton.Down := fsItalic in
    RichEdit1.SelAttributes.Style;
  UnderlineButton.Down := fsUnderline in
    RichEdit1.SelAttributes.Style;
  StrikeoutButton.Down := fsStrikeout in
    RichEdit1.SelAttributes.Style;

    FontSize.Text := IntToStr(RichEdit1.SelAttributes.Size);
    FontName.ItemIndex:= FontName.Items.IndexOf(
      RichEdit1.SelAttributes.Name);

  with RichEdit1.Paragraph do
  begin
    if Numbering = nsBullet then
      BulletsButton.Down := True
    else
      BulletsButton.Down := False;

    case Alignment of
      taLeftJustify: LeftAlign.Down := True;
      taRightJustify: RightAlign.Down := True;
      taCenter: CenterAlign.Down := True;
    end;
  end;
end;

procedure TEditForm.BoldButtonClick(Sender: TObject);
begin
  with RichEdit1.SelAttributes do
    if BoldButton.Down then
      Style := Style + [fsBold]
    else
      Style := Style - [fsBold];
end;

procedure TEditForm.ItalicButtonClick(Sender: TObject);
begin
  with RichEdit1.SelAttributes do
    if ItalicButton.Down then
      Style := Style + [fsItalic]
    else
      Style := Style - [fsItalic];
end;

procedure TEditForm.UnderlineButtonClick(Sender: TObject);
begin
  with RichEdit1.SelAttributes do

 if UnderlineButton.Down then
      Style := Style + [fsUnderline]
    else
      Style := Style - [fsUnderline];
end;

procedure TEditForm.StrikeoutButtonClick(Sender: TObject);
begin
  with RichEdit1.SelAttributes do
    if StrikeoutButton.Down then
      Style := Style + [fsStrikeout]
    else
      Style := Style - [fsStrikeout];
end;

procedure TEditForm.BulletsButtonClick(Sender: TObject);
begin
  if BulletsButton.Down then
    RichEdit1.Paragraph.Numbering := nsBullet
  else
    RichEdit1.Paragraph.Numbering := nsNone;
end;
```

```
procedure TEditForm.LeftAlignClick(Sender: TObject);
begin
  with RichEdit1.Paragraph do
  begin
    if Sender = LeftAlign then Alignment := taLeftJustify;
    if Sender = CenterAlign then Alignment := taCenter;
    if Sender = RightAlign then
      Alignment := taRightJustify;
  end;
end;

procedure TEditForm.FontNameChange(Sender: TObject);
begin
  //Set the font to the name contained
  //in the FontName combo box.
  RichEdit1.SelAttributes.Name :=
    FontName.Items[FontName.ItemIndex];
end;

procedure TEditForm.FontSizeClick(Sender: TObject);
begin
  //Set the font size to that contained
  //in the FontSize combo box.
  RichEdit1.SelAttributes.Size := StrToInt(FontSize.Text);
end;

procedure TEditForm.FontSizeKeyPress(Sender: TObject;
  var Key: Char);
begin
  { If the user presses the CR key when the cursor is in
    the font size combobox call the routine to set the
    font size of the current selection in the RichEdit. }
  if Key = #13 then
  begin
    FontSizeClick(Sender);
    Key := #0;
  end;
end;

procedure TEditForm.About1Click(Sender: TObject);
begin
  MessageDlg('This is where the about box goes',
             mtInformation, [mbOK], 0);
end;

end.
```

End Listing One

Begin Listing Two — Data Aware RichEdit

```
unit CMainForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, Mask, DBCtrls, StdCtrls, ComCtrls,
  ExtCtrls, DB, DBTables, Buttons;

type
  TMainForm = class(TForm)
    SaveButton: TButton;
    LoadButton: TButton;
    DataSource1: TDataSource;
    Table1: TTable;
    DBNavigator1: TDBNavigator;
    RichEdit1: TRichEdit;
    DBEdit1: TDBEdit;
    FontButton: TButton;
    FontDialog1: TFontDialog;
    Label1: TLabel;
    Table1Header: TStringField;
    Table1FormattedMemo: TBlobField;
    procedure FormCreate(Sender: TObject);
    procedure FontButtonClick(Sender: TObject);
    procedure SaveButtonClick(Sender: TObject);
    procedure LoadButtonClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Table1.DatabaseName :=
    ExtractFilePath(Application.ExeName);
  Table1.Active := True;
end;

procedure TMainForm.FontButtonClick(Sender: TObject);
begin
  FontDialog1.Font.Assign(RichEdit1.SelAttributes);
  if FontDialog1.Execute then
    RichEdit1.SelAttributes.Assign(FontDialog1.Font);
  RichEdit1.SetFocus;
end;

procedure TMainForm.SaveButtonClick(Sender: TObject);
var
  BlobStream: TBlobStream;
begin
  Table1.Edit;
  BlobStream:=
    TBlobStream.Create(Table1FormattedMemo,bmWrite);
  RichEdit1.Lines.SaveToStream(BlobStream);
  BlobStream.Free;
  Table1.Post;
end;

procedure TMainForm.LoadButtonClick(Sender: TObject);
var
  BlobStream: TBlobStream;
begin
  BlobStream :=
    TBlobStream.Create(Table1FormattedMemo,bmRead);
  RichEdit1.Lines.LoadFromStream(BlobStream);
  BlobStream.Free;
end;

end.
```

End Listing Two

OP TECH

Delphi / Object Pascal

By Bill Todd

Array of Tasks

Using the Components Array for a Variety of Jobs

In the October 1996 *Delphi Informant*, Robert Vivrette introduced the Controls and Components arrays (see Vivrette's article "Parentage and Ownership"). This article expands on that discussion, and offers three practical examples of using it to develop generic solutions to common problems.

Managing Security

A common necessity in Delphi programs is a system that manages security by requiring users to log into the program with a valid user name and password. The user name and password are typically stored in a security table — a table that also contains a security level to control which features of the program the user can access.

One way to control a user's access to program features is to disable all the menu choices and buttons a user isn't allowed to use. This isn't difficult, but does require a substantial amount of code in each form to check the user's security level, then explicitly disable each menu choice and button the user isn't allowed to use.

This process can be simplified by making the user's security level an integer. In this system, larger numbers mean greater access, and lower numbers represent less access. The next step is to assign a security level to each menu choice and button that must be disabled. Fortunately, the Delphi development team gave every component an unused long integer property, named *Tag*. The *Tag* property is not used by Delphi. It exists only for our use. This means that by setting the component's *Tag* property, you can assign the security level to each component that must be disabled. Assign the levels so a user can't use any menu item or button whose security level is greater than the user's security level. Note that because the default value of the *Tag* property is 0 (zero), you don't need to change the value of *Tag* for the components available to all users.

The last step in implementing this system is to write a routine that disables all menu

```
procedure DisableControls(Form: TForm; TagValue: LongInt);
{ Disables any TMenuItem, TButton, or TSpeedButton on the
  form whose Tag property is greater than the TagValue
  parameter. The uses clause of the unit that contains this
  routine must include StdCtrls and Buttons.

  Parameters:
    Form:     The form whose controls are checked.
    TagValue: The maximum value of the Tag property that
              will allow the control to remain enabled. }
var
  I : Integer;
begin
  with Form do begin
    for I := 0 to ComponentCount -1 do
      if (Components[I].Tag > TagValue) then
        begin
          if Components[I] is TMenuItem then
            TMenuItem(Components[I]).Enabled := False
          else if Components[I] is TButton then
            TButton(Components[I]).Enabled := False
          else if Components[I] is TSpeedButton then
            TSpeedButton(Components[I]).Enabled := False;
        end;  { if }
   end;  { with }
end;
```

Figure 1: A procedure to disable controls.

```
function AllTablesPosted(Form: TForm): Boolean;
{ Checks that all tables on form are in dsBrowse mode. }
var
  I : Integer;
begin
  Result := True;
  with Form do begin
    for I := 0 to ComponentCount -1 do
      if Components[I] is TTable then
        if (Components[I] as TTable).Active then
          if (Components[I] as TTable).State <> dsBrowse then
            begin
              Result := False;
              Break;
            end;  { if }
    { Display error message. }
    if Result = False then
      MessageDlg('There is an unposted'+
        (Components[I] as TTable).Name +'record in the'+
        Caption +
        'form. Please Post or Cancel your changes.',
        mtError, [mbOK], 0);
  end;  { with }
end;
```

Figure 2: A function to warn users of unposted records.

items and buttons whose *Tag* value is greater than the user's security level. The Components array provides the perfect way to do this, as shown in the code in Figure 1.

You can call this procedure from the *OnCreate* event handler of any form by passing it two parameters. The first is the form whose controls should be checked. The second is the maximum *Tag* value for controls to remain enabled. All *TMenuItem*, *TButton*, and *TSpeedButton* objects whose *Tag* property is greater than the *Tag* parameter will be disabled. Note that this also takes care of *TBitButton* objects, because *TBitButton* is a descendant of *TButton*.

This procedure uses a **for** loop to iterate through the Components array, checking each component to determine if its *Tag* property is greater than the *TagValue* parameter. If it is, the component is checked to determine if it's a *TMenuItem*, *TButton*, or *TSpeedButton*. If the component is one of these types, or a descendant of one of these types, its *Enabled* property is set to *False*.

Ensuring Records Are Posted

Another problem the Components array can easily resolve is how to handle users who forget to post a new or changed record before they close a form or move to another form. To solve this problem you need a generic routine that will check each Table control and warn the user if its *State* property is anything other than *dsBrowse*. Figure 2 shows such a function.

Again, the code uses a **for** loop to traverse the Components array, checking each component to determine if it's a *TTable* or one of its descendants. If it is, and the table is open and the *State* property is not set to *dsBrowse*, the function displays a message to the user and returns *False*. You might want to modify this code to test for the *TDataSet* type instead of *TTable*. This will detect unposted records in *TQuery* and *TStoredProc* components as well as *TTable*.

```
procedure RefreshAllTables(Form: TForm);
{ Refreshes all active tables on a form. }
var
  I : Integer;
begin
  with Form do begin
    for I := 0 to ComponentCount -1 do
      if Components[I] is TTable then
        if (Components[I] as TTable).Active then
          if (Components[I] as TTable).State = dsBrowse
then
            (Components[I] as TTable).Refresh;
  end;  { with }
end;
```

Figure 3: Refreshing all tables on a form.

You can call this function in a form's *OnCloseQuery* event and prevent the user from closing the form until the record is posted. You can also call this method in a form's *OnDeactivate* event handler to warn users of unposted records when they move focus to another form, or in the *OnChange* event handler for a tabbed notebook to warn users before they move to another page.

Refreshing Data

The last example of using the Components array ensures that users in a multi-user environment are viewing up-to-date information. If you or your users are accustomed to using a desktop database — such as Paradox — that periodically (and automatically) refreshes the user's view of a table, you may want to implement that same functionality in your Delphi applications. The procedure shown in Figure 3 takes a form as its only parameter, and if the table is active and in browse mode, calls *Refresh* to re-read any data changed by another user.

```
procedure RefreshLinkedTables(Form: TForm);
{ Refreshes all active tables on a form if all
  tables are in dsBrowse state. }
var
  I : Integer;
  OkToRefresh : Boolean;
begin
  OkToRefresh := True;
  with Form do begin
    { Make sure all tables are in dsBrowse state. This
      prevents refreshing a master when one of its
      details has an unposted record, since refreshing
      the master will post the detail. }
    for I := 0 to ComponentCount -1 do
      if Components[I] is TTable then
        if (Components[I] as TTable).Active then
          if (Components[I] as TTable).State<>dsBrowse then
            OkToRefresh := False;
    { Refresh the tables. }
    if OkToRefresh then
      for I := 0 to ComponentCount -1 do
        if Components[I] is TTable then
          if (Components[I] as TTable).Active then
            (Components[I] as TTable).Refresh;
  end;  { with }
end;
```

Figure 4: Refreshing linked tables.

However, there is a problem with this. If you have a form that uses linked tables to show a one-to-many relationship, refreshing the master table will cause any unposted records in a detail table to post.

To avoid this problem, change the code as shown in Figure 4. The procedure checks that all tables on the form are in browse mode before refreshing any table. This prevents unexpected posts from occurring.

Conclusion

The Components and Controls arrays are extremely useful for writing generic routines that must determine the type and number of controls in a form or any of the container objects in a form. Using these arrays you can write code that will work with any form in any of your programs to manipulate the form's components in any way you need.

Employing the Components array, we have explored three practical solutions to common problems. But it's just the beginning ... Δ

Bill Todd is President of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is co-author of *Delphi: A Developer's Guide* [M & T Books, 1995], *Delphi 2: A Developer's Guide* [M & T Books, 1996], and *Creating Paradox for Windows Applications* [New Riders Publishing, 1994], and is a member of Team Borland providing technical support on CompuServe. He has also been a speaker at every Borland Developers Conference. He can be reached at (602) 802-0178, on CompuServe at 71333,2146, or on the Internet at 71333.2146@compuserve.com.

By *Cary Jensen, Ph.D.*

Controlling Your Sessions

Using the *TSession* Class

Over the past two months, this column has looked at some components that don't appear on the Component palette. Two of them, the *Application* and *Screen* variables, are defined in the Forms unit, and are automatically available to any application that uses this unit.

A third variable, *Session*, is similar in many respects. This variable, defined in the DB unit, is automatically created for any application that includes this unit. *Session* is an instance variable of the *TSession* class, and it provides access to properties, methods, and events that generally relate to the Borland Database Engine (BDE). Consequently, this is a particularly useful variable as far as database developers are concerned.

Property
Active *
ConfigMode *
DatabaseCount
Databases
Handle
KeepConnections
Locale
PrivateDir
Name
NetFileDir
Owner
Tag
TraceFlags *

Figure 1: *TSession* properties (*Delphi 2 only).

This month's "DBNavigator" takes a look at the *TSession* class, concentrating on the *Session* instance variable. Three issues to consider are:

- accessing BDE information,
- controlling Paradox table-related settings, and
- managing aliases in Delphi 2.

While *Session* is similar to *Application* and *Screen*, it is also different in two important ways:

- Non-database applications generally don't include the DB unit, and therefore these types of applications don't have access to the *Session* variable. Because *Application* and *Screen* are defined in the Forms unit, they are available for any application that includes at least one form, and this accounts for nearly all Delphi applications.
- Delphi 2 includes a *TSession* component on the Data Access page of the Component palette. Consequently, unlike *Application* and *Screen*, which don't appear on the Component palette, at least as far as 32-bit versions of Delphi are concerned, you *can* have design-time access to a *TSession* component.

As mentioned earlier, the *TSession* class is useful in database applications, providing general support for aliases and Paradox table-related issues. Figure 1 contains a list of the *TSession* properties. Figures 2 and 3 contain the *TSession* methods and event properties, respectively.

Method
AddAlias *
AddPassword
AddStandardAlias
Close
CloseDatabase
DeleteAlias *
DropConnections
FindDatabase
GetAliasDriverName *
GetAliasNames
GetAliasParams
GetConfigParams *
GetDatabaseNames
GetDriverNames
GetDriverParams
GetPassword
GetStoredProcNames
GetTableNames
IsAlias *
ModifyAlias *
Open *
OpenDatabase
RemoveAllPasswords
RemovePassword
SaveConfigFile *

Figure 2: *TSession* methods (*Delphi 2 only).

Events
OnPassword
OnStartup *

Figure 3: *TSession* event properties (*Delphi 2 only).

Getting Basic BDE Information

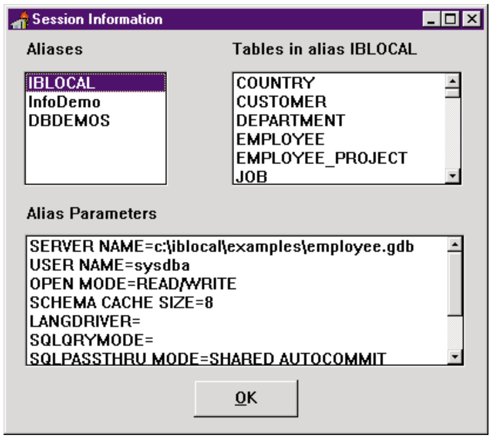
The *Session* variable provides easy access to some of the more basic types of information available from the BDE. This includes the available aliases, tables, stored procedures, and drivers, as well as alias and driver parameters. For example, using the method *GetAliasNames*, you can easily populate a *TStrings* object with the list of available aliases. Likewise, using *GetTableNames*, you can obtain a list of the tables stored in a given alias.

The project BASIC.DPR, shown in Figure 4, demonstrates some of the methods that provide your application with basic BDE information. This project uses three *TSession* methods: *GetAliasNames*, *GetTableNames*, and *GetAliasParams*.

The simplest of these methods is *GetAliasNames*, which has the following syntax:

```
procedure
  GetAliasNames(List:
                TStrings);
```

When you call this method, the *TStrings* object is first emptied, then populated with a list of aliases. In Delphi 1, this list includes only those aliases defined in the BDE configuration file, IDAPI.CFG. If you want to list local aliases, those defined by Database components, you must use the *GetDatabaseNames* method. In Delphi 2, the aliases included in the list are defined by the *TSession* property *ConfigMode*. This property has three possible values: *cmPersistent* (only those defined by IDAPI32.CFG), *cmSession* (only those defined by Database components using the *TSession* component), and *cmAll* (both persistent and local aliases).



IBLOCAL
InfoDemo
DBDEMOS

COUNTRY
CUSTOMER
DEPARTMENT
EMPLOYEE
EMPLOYEE_PROJECT
JOB

SERVER NAME=c:\iblocal\examples\employee.gdb
USER NAME=sysdba
OPEN MODE=READ/WRITE
SCHEMA CACHE SIZE=8
LANGDRIVER=
SQLQRYMODE=
SQLPASSTHRU MODE=SHARED AUTOCOMMIT

Figure 4: The BASIC project displays information gathered using *TSession* methods.

Only slightly more complex is the *GetAliasParams* method. This method, which has the following syntax, empties a specified *TStrings* object and populates it with the parameters of a specified alias:

```
procedure GetAliasParams(const AliasName: string;
                         List: TStrings);
```

The *GetTableNames* method has the following syntax:

```
procedure GetTableNames(const DatabaseName, Pattern: string;
  Extensions, SystemTables: Boolean; List: TStrings);
```

The first parameter, `DatabaseName`, is the name of the alias that points to the table location, while the second parameter allows you to specify a pattern for the inclusion of table names. If you want to include all table names, use the pattern `*.*`. The third and fourth parameters are Boolean values that enable you to choose whether to include extensions in the listed table names, and whether to include the system tables (SQL databases only), respectively. The final parameter is the name of the *TStrings* object that will be emptied, then populated with the table names as specified with the first four parameters.

The code in Figure 5 is associated with the *OnCreate* event handler for the main form of the BASIC project, as well as the *OnChange* event handler for the list box that displays the aliases (*ListBox1*). Note that for this code to work properly, it was necessary to add the DB unit to this project's **uses** clause; this needed to be done manually, because the project didn't include data-aware components. If at least one data-access component (such as a DataSource or a Database) appeared on the form, the DB unit would have automatically been added to the unit's interface **uses** clause.

Other *TSession* methods that provide basic BDE information include *GetAliasDriverName* (Delphi 2 only), *GetConfigParams* (Delphi 2 only), *GetDatabaseNames*, *GetDriverNames*, *GetDriverParams*, and *GetStoredProcNames*.

Controlling Paradox Table-Related Settings

For most local applications (those in which the data are stored in tables on a local hard disk or a local area net-

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Get the alias list.
  Session.GetAliasNames(ListBox1.Items);
  // Select the first alias in the list.
  ListBox1.ItemIndex := 0;
  // Call the OnChange event handler for ListBox1.
  ListBox1Click(Sender);
end;

procedure TForm1.ListBox1Click(Sender: TObject);
begin
  // Display the tables in the selected alias.
  Session.GetTableNames(ListBox1.Items[ListBox1.ItemIndex],
                        '*.*',True,True,ListBox2.Items);
  Label2.Caption := 'Tables in alias ' +
                       ListBox1.Items[ListBox1.ItemIndex];
  // Display the parameters of the selected alias.
  Session.GetAliasParams(ListBox1.Items[ListBox1.ItemIndex],
                         ListBox3.Items);
end;
```

Figure 5: *TSession* methods.

work), the Paradox table format is the preferred format for several reasons.

First, the Paradox table format has the richest set of field types on the desktop. The available field types range from DateTime to Currency, from BLOb (Binary Large Object) to Autoincrement, and from Graphic to BCD (binary coded decimal).

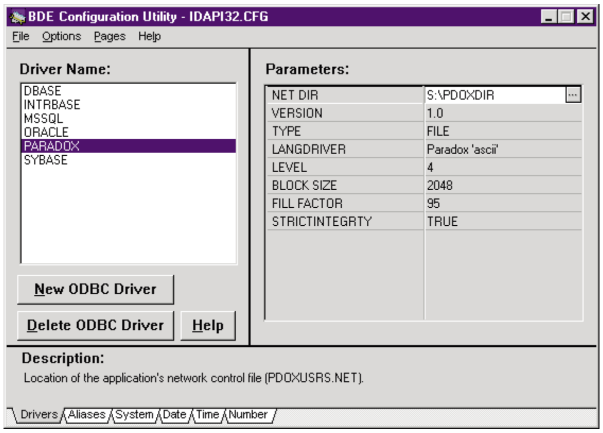
Second, Paradox tables support high-performance indexes that provide quick filtering (using ranges) and sorting, as well as table and record-level locking.

Finally, the Paradox table format is the default format for a number of BDE operations, including temporary files for local SQL queries (SQL queries not run on a remote database server) and error tables for failed BatchMove operations.

A number of *TSession* properties, methods, and events are directly related to the use of Paradox tables — specifically, the *PrivateDir*, and *NetFileDir* properties, the *AddPassword*, *GetPassword*, *RemovePassword*, and *RemoveAllPasswords* methods, and the *OnPassword* event. (The next section will begin by considering the *NetFileDir* and *PrivateDir* properties. The password-related issues are covered in a later section.)

The *NetFileDir* and *PrivateDir* Properties

If you use Paradox tables, or BDE operations that rely on Paradox tables, you need to at least be aware of the *NetFileDir* and *PrivateDir* properties of the *Session* variable. For most applications, two rules apply to these properties. The first rule is that every user who can access shared Paradox tables (tables that can be read or written to by more than one user at a time) must all have their *Session*'s *NetFileDir* property point to the same physical directory. In other words, this directory must be a shared directory (one that is on the network). In addition, each



Driver Name:

DBASE
INTRBASE
MSSQL
ORACLE
PARADOX
SYBASE

Parameters:

NET DIR	S:\PDOXDIR
VERSION	1.0
TYPE	FILE
LANGDRIVER	Paradox 'ascii'
LEVEL	4
BLOCK SIZE	2048
FILL FACTOR	95
STRICTINTEGRTY	TRUE

Description:

Location of the application's network control file (PDOXUSRS.NET).

Figure 6: The BDE Configuration Utility permits you to set the NET DIR parameter for the Paradox driver.

user must have read and write access to this directory. The *NetFileDir* property specifically points to a directory in which the BDE writes a file called the *network control file*. This file, named PDOXUSER.NET, identifies each session that can access shared Paradox tables. (Every BDE-enabled application that has at least one session and some applications, such as those written in Delphi 2 that use additional *TSession* components from the Data Access page of the Component palette, may have more than one.)

If any user attempts to access a Paradox table that is currently being accessed by one or more users, and all users don't use the same network control file, bad things happen. The most common is that one or more of the sessions crash. For this reason, the *NetFileDir* is an important property.

In most applications, and in particular, Delphi 1 applications, the *NetFileDir* property isn't set by the application. Specifically, it's a parameter of the Paradox driver, and it's set when the BDE is installed. Figure 6 displays the Drivers page of the BDE Configuration Utility with the Paradox driver selected. Notice the NET DIR parameter on the right-hand pane of this window. This is where this property is typically set. If you must set this property at run time, it must be set prior to activating any Databases or DataSets. In other words, no form that is auto-created can contain active Tables, Queries, StoredProcs, or Databases. Furthermore, before making any one of these components active, the NET DIR property of the Session component must be set to a common network directory.

The second critical property when Paradox tables are being used is the *PrivateDir* property. In short, every session must use a different private directory. It's into this directory that the BDE will create temporary tables, as needed. For example, if you execute a query against local tables, or execute heterogeneous queries (ones that involve both local and remote tables, or one that involves tables from two or more remote servers), the BDE must write temporary tables that it will ultimately delete. To ensure that such a query being executed by one user is unaffected by queries being executed by another user, the BDE writes these to the private directory of the session.

By default, the private directory of the *Session* variable is set to the directory in which the executable (.EXE) file resides. As long as each user of a multi-user application is running a local version of the .EXE, that is, the .EXE resides on his/her hard drive in an un-shared directory, the default private directory should work fine (assuming that every BDE-aware application on a user's machine is stored in a separate directory, and that only one copy of an application can run at a time).

However, when the .EXE is stored on a shared drive, and run by each user from that drive, your code must assign a unique private directory before any Databases or DataSets are activated. As with the *NetFileDir* property, this means that no auto-created forms can contain activated DataSources or DataSets. Furthermore, prior to activating the first Database or DataSet of the application, your code should set the *TSession* property *PrivateDir*.

The following code demonstrates one example of how this can be accomplished:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if not DirectoryExists('c:\priv') then
    if not CreateDir('c:\priv') then
      raise Exception.Create('Cannot create c:\priv. '+
              'Create this directory and retry');
  Session.PrivateDir := 'c:\priv';
end;
```

This code, which appears in the *OnCreate* event handler for the application's main form, assumes that no Databases or DataSets are active on this form. Furthermore, it's assumed that the main form is the only auto-created form in the application (every other form will be created as needed, and presumably released when no longer needed, but this second part is not critical to the issue of private directories). One final assumption is that each user is running on a machine with a hard drive whose drive letter is C.

Using Encrypted Paradox Tables

If you are using tables from a remote database server, you must supply a valid user name and password prior to accessing those tables. These two pieces of information can either be assigned to the *Params* property of a Database component or automatically be requested by the BDE, by way of the Database Login dialog box, when the application first attempts to access the server.

If your application uses an encrypted Paradox table, a valid password is also required before the table can be accessed for the first time by a user in a session. Unlike remote servers, however, Paradox tables don't use a Database component to supply their passwords.

Instead, passwords are supported through the *TSession* class. (Paradox tables can be encrypted using either Paradox for Windows or the Database Desktop, by creating or restructuring a table, then selecting Password Security from the Table Properties drop-down list.)

Similar to when your application uses tables from a remote database server, the BDE can also automatically display a dialog box asking the user to enter a valid password. In addition, the *TSession* class provides a number of methods for adding and removing passwords at run time. To add a password, your application can call *AddPassword* using the following syntax:

```
procedure AddPassword(const Password: string);
```

You can also invoke the default password dialog box provided by Delphi by calling the *GetPassword* method. When you call this function:

```
function GetPassword: Boolean;
```

Delphi displays the dialog box shown in Figure 7. As you can see, one or more passwords can be removed using the default password dialog box. Alternatively, you can remove a password through code by calling *RemovePassword*. You can remove all passwords from a session in a single call to *RemoveAllPasswords*. The following is the syntax of these methods:

```
procedure RemovePassword(const Password: string);

procedure RemoveAllPasswords;
```

When the BDE is provided a password, either through a call to the appropriate *TSession* method or through the default password dialog box, that password remains in effect until the session is terminated, or the password is explicitly removed. Also, the password is used by the BDE specifically to open a table. Removing a password after a table has been opened has no effect on the access to that table. That is, if you open a table, then remove its password, you can continue to access that table as long as you don't close it. If you close an encrypted table whose password has been removed, you must re-enter the password before being permitted to open it again.

Use of the various password-related techniques is demonstrated by the PASSWORD.DPR project (see Figure 8). This project uses an encrypted Paradox table named COMPS.DB. This table has a master password (required), as well as a secondary password (optional). An encrypted Paradox table has one master password, but can support any number of secondary passwords. Normally, only the owner of the table knows the master password. The secondary passwords are provided to users, and they provide varying levels of access to a table. The master password for COMPS.DB is the string "MASTER", and the secondary password is "SECONDARY". All Paradox table passwords are case-sensitive. (For more information on Paradox passwords, refer to the online Help in the Database Desktop or Paradox for Windows.)

Figure 9 shows the event handlers associated with the *OnClick* events for the buttons on the PASSWORD project's main form. This code demonstrates calls to the various password-related methods of the *TSession* class. Note that if you attempt to use the **Open Table** button without first supplying a password, the default password dialog box is automatically displayed.

Also related to the issue of passwords is the *OnPassword* event property of the *TSession* class. Use this property if you want to replace the default password dialog box with one of your own. Specifically, the event handler you assign to *OnPassword* will be called if you call the *GetPassword* method, or attempt to activate an encrypted Paradox table for which no password has been issued.

If you use this method, however, you should note that Delphi 1 declares *OnPassword* to be a procedure pointer, not a method pointer. Specifically, *OnPassword* is a procedure in Delphi 1. Delphi 2 correctly declares *OnPassword* as a method pointer, meaning that you assign a method to this property. (For more information regarding *OnPassword*, use Delphi's online Help.)

Sessions and Delphi 2

There are two significant differences between the *TSession* class declared in Delphi 1 and that declared in Delphi 2. The first is that there are more properties, methods, and one more event property in Delphi 2. Many of these enhancements provide the ability to control aliases from your Delphi 2 applications using a Session, rather than relying on direct calls to the BDE. The second is that Delphi includes a Session component on the Data Access page of the Component palette. This component permits you to create and manage sessions in addition to the default session created by Delphi.

Using Additional Sessions

There is only one use for a manually-placed Session component. It's provided to support multithreaded access to DataSets. In a multithreaded application, each thread must access DataSets using a different session. Quite simply, doing this permits the BDE to manage each thread's access to an application's DataSets as if the access was by different users. In other words, multithreaded access to data uses the same techniques for managing competition for resources as does a multi-user application.

As you learned earlier, multi-user access to Paradox tables involves two critical properties of the *TSession* class: *NetFileDir* and *PrivateDir*. When using additional session components and Paradox tables, the same rules that apply to these properties must be observed. Specifically, all sessions must use the same network file directory and different private directories.

There is a special event property declared in the *TSession* class in Delphi 2 to manage the assignment of values to these two properties: *OnStartup*. This event handler is triggered immediately before a session becomes active. In this event handler, you should assign the current session's *NetFileDir* property to that associated with the default session, and assign the *PrivateDir* property to one that is different from the default session, as well as different from any other session's *PrivateDir* property.

The following code demonstrates the use of the *OnStartup* event handler:

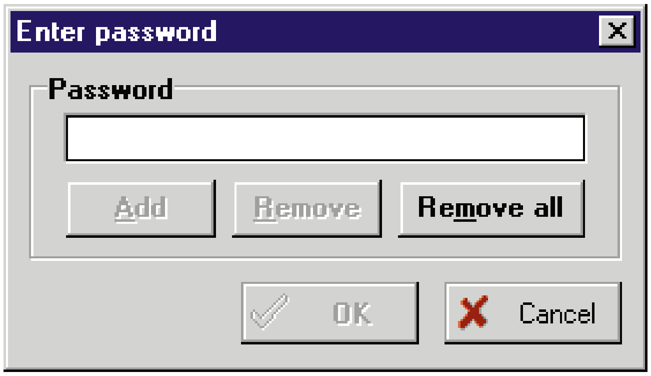


Figure 7: The default password dialog box for encrypted Paradox tables can be displayed automatically when an application attempts to activate an encrypted Paradox table for which no password has yet been provided, or through a call to the *TSession* method *GetPassword*.

```
procedure TDataModule2.Session1Startup(Sender: TObject);
begin
  if not DirectoryExists('c:\priv\thread1') then
    if not FileCtrl.ForceDirectories('c:\priv\thread1') then
      raise Exception.Create('Cannot create new session');
  Session1.PrivateDir := 'c:\priv\thread1';
  Session1.NetFileDir := Session.NetFileDir;
end;
```

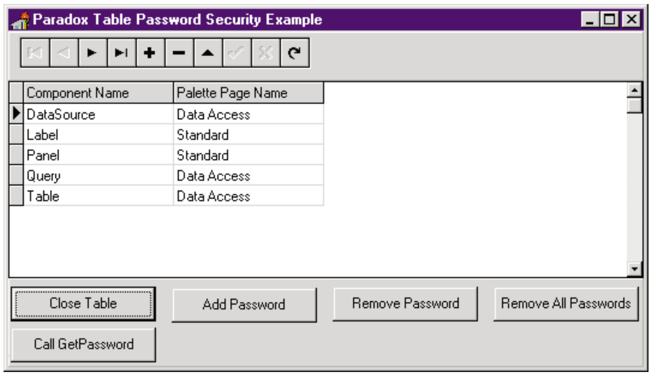
This code assumes that no other thread, application, or user is using the directory C:\PRIV\THREAD1 as its private directory. Notice also that this code uses the *ForceDirectories* procedure (declared in the FileCtrl unit) to create the private directory. This procedure creates nested directories, even when the nested directory's parent directory doesn't exist. Obviously, a unit using this procedure must include the FileCtrl unit in one of its **uses** clauses.

Managing Aliases in Delphi 2

In Delphi 1, the only way to create a new alias and add it permanently to the IDAPI.CFG configuration file is to use BDE calls. Importantly, these calls must be made before initializing the default session. This means the BDE needs to be manually initialized, updated, then un-initialized, from an initialization section of a unit. Otherwise, the attempt to add the alias fails.

Creating new aliases at run time is significantly easier with Delphi 2 due to five new *TSession* methods: *AddAlias*, *DeleteAlias*, *IsAlias*, *ModifyAlias*, and *SaveConfigFile*. In addition, the new property *ConfigMode* plays an important role in alias control.

As you recall from earlier in this article, *ConfigMode* has three possible values: *cmLocal*, *cmPersistent*, and *cmAll*. If you set *ConfigMode* to *cmAll* or *cmPersistent*, any call to *AddAlias* creates an alias that can be written to the BDE configuration file (IDAPI32.CFG). However, simply calling *AddAlias* creates the alias, but doesn't perform the save. To save a new alias that you create, you must call



Component Name	Palette Page Name
DataSource	Data Access
Label	Standard
Panel	Standard
Query	Data Access
Table	Data Access

Figure 8: The PASSWORD project demonstrates the use of the various *TSession* methods for providing and removing passwords for encrypted Paradox tables.

```
procedure TForm1.OpenTableClick(Sender: TObject);
begin
  if Table1.Active then
    begin
      Table1.Close;
      Button1.Caption := 'Open Table';
    end
  else
    begin
      Table1.Open;
      Button1.Caption := 'Close Table';
    end;
end;

procedure TForm1.AddPasswordClick(Sender: TObject);
begin
  Session.AddPassword(InputBox(
    'Enter MASTER or SECONDARY','Password','MASTER'));
end;

procedure TForm1.RemovePasswordClick(Sender: TObject);
begin
  Session.RemovePassword(InputBox(
    'Enter MASTER or SECONDARY','Password','MASTER'));
end;

procedure TForm1.RemoveAllPasswordsClick(Sender:
TObject);
begin
  Session.RemoveAllPasswords;
end;

procedure TForm1.CallGetPasswordClick(Sender: TObject);
begin
  Session.GetPassword;
end;
```

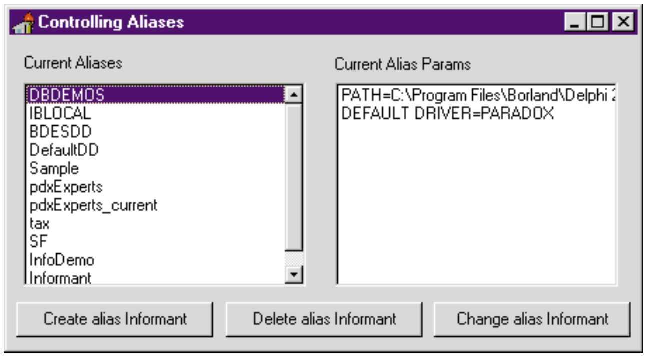
Figure 9: Calls to the various password-related methods of the *TSessions* class.

SaveConfigFile. Following is the syntax of the *AddAlias* and *SaveConfigFile* methods:

```
procedure AddAlias(const Name, Driver: string; List:
                   TStrings);

procedure SaveConfigFile;
```

You can use *IsAlias* to determine if a particular alias already exists, *DeleteAlias* to remove one, and *ModifyAlias* to change one or more of an existing alias' parameters. Here is the syntax of these three methods:

```
function IsAlias(const Name: string): Boolean;

procedure DeleteAlias(const Name: string);

procedure ModifyAlias(Name: string; List: TStrings);
```



DBDEMOS
IBLOCAL
BDESDD
DefaultDD
Sample
pdxExperts
pdxExperts_current
tax
SF
InfoDemo
Informant

PATH=C:\Program Files\Borland\Delphi 2
DEFAULT DRIVER=PARADOX

Figure 10: The ALIAS project demonstrates how to use the Delphi 2 *TSession* class to create, update, and delete aliases at run time.

The project named ALIAS.DPR demonstrates the use of these methods. This project, shown in Figure 10, includes buttons that permit you to add a new alias to your BDE configuration file, remove it, and modify its PATH parameter. The code associated with this project is shown in Listing Three.

Conclusion

The *TSession* class and its instance variable *Session* provide your database applications with access to some of the features of the BDE, without requiring you to resort to low-level BDE calls. Furthermore, while some of the properties and methods of this class are essential in multi-user applications, others are useful in the inspection and management of aliases, tables, and drivers. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\APR\DI9704CJ.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including *Delphi In Depth* (Osborne/McGraw-Hill, 1996). Cary is also a Contributing Editor of *Delphi Informant*, as well as a member of the Delphi Advisory Board for the 1997 Borland Developers Conference. For information concerning Jensen Data Systems' Delphi consulting and training services, visit the Jensen Data Systems Web site at http://gramercy.ios.com/~jdsi. You can also reach Jensen Data Systems at (281) 359-3311, or via e-mail at cjensen@compuserve.com.

Begin Listing Three — ALIAS.DPR

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Set the ConfigMode to display all aliases.
  Session.ConfigMode := cmAll;
  // Populate the Alias list and Alias parameter list boxes.
  Session.GetAliasNames(ListBox1.Items);
  ListBox1.ItemIndex := 0;
  ListBox1Click(Sender);
end;

procedure TForm1.CreateAlias(Sender: TObject);
var
  AParams: TStringList;
begin
```

```
if not Session.IsAlias('Informant') then
    begin
      AParams := TStringList.Create;
      try
        AParams.Add('PATH=' +
                     ExtractFilePath(Application.ExeName));
        Session.AddAlias('Informant','STANDARD',AParams);
        Session.SaveConfigFile;
      finally
        AParams.Free;
     end;

       // Update the alias list.
      Session.GetAliasNames(ListBox1.Items);
    end
  else
    ShowMessage('Alias Informant already defined');
end;

procedure TForm1.DeleteAlias(Sender: TObject);
begin
  if Session.IsAlias('Informant') then
    begin
      Session.DeleteAlias('Informant');
      Session.SaveConfigFile;
      // Update the alias list.
      Session.GetAliasNames(ListBox1.Items);
    end
  else
       ShowMessage('Alias Informant not defined');
   end;

procedure TForm1.ChangeAlias(Sender: TObject);
var
  AParams: TStringList;
  Dir: string;
begin
  if not Session.IsAlias('Informant') then
    begin
      ShowMessage('Alias Informant does not exist');
      Exit;
    end;
  AParams := TStringList.Create;
  try
    // Get the current PATH paramter.
    Session.GetAliasParams('Informant',AParams);
    Dir := copy(AParams.Strings[0],6,255);
    // Get the new PATH parameter.
    if InputQuery('Informant Alias Path','Path:',Dir) then
      begin
        // Update the alias parameters.
        AParams.Clear;
        AParams.Add('PATH=' + Dir);
        Session.ModifyAlias('Informant',AParams);
        Session.SaveConfigFile;
        // Update the alias parameters list box.
        ListBox1Click(Sender);
      end;
  finally
    AParams.Free;
  end;
end;

procedure TForm1.ListBox1Click(Sender: TObject);
begin
  // Update the alias parameters list box.
  Session.GetAliasParams(ListBox1.Items[ListBox1.ItemIndex],
                         ListBox2.Items);
end;
```

End Listing Three

Object Pascal / Delphi 2

By Peter Dove and Don Peer

Getting DIBs on Speed

Delphi Graphics Programming: Part IV

It's time to speed things up! This month, we expand on the texture-mapping theories explained last month, enable our *TGMP* component to perform full shading of texture-mapped objects, and make faster 3D graphics a reality. We'll give *TGMP* a boost in speed by creating a *device-independent bitmap* (DIB) class capable of handling its drawing routines, screen-clearing routines, and related procedures. And we'll discover a little more about pointers and bit manipulation.

On the raw, object side of the component, we'll enhance *TGMP* to process extremely accurate graphic primitives through the use of a custom file reader. To close out the article, we'll explain and implement the world coordinate system. An example of the graphics made possible by the information presented in this jam-packed article is shown in Figure 1.

The DIB Class

To start, we'll create a DIB class derived from *TObject*. We'll take this approach because the class isn't meant to serve as a component that you drop on a form; rather, it's meant to supplement other components such as *TGMP*, and will be declared accordingly in the **uses** clause. However, before we delve too deeply into class development, a small explanation of DIBs might be useful.

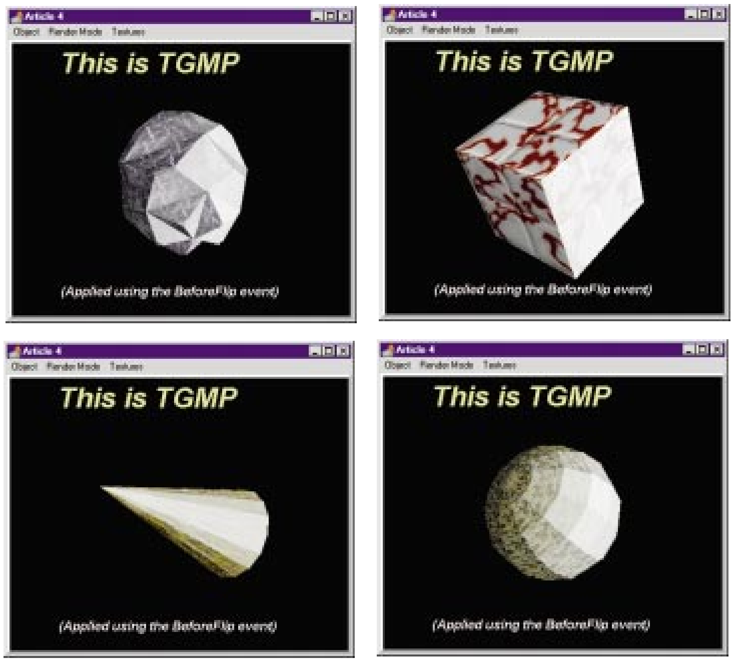

Article 4
Object Render Mode Textures
This is TGMP
(Applied using the BeforeFlip event)
Article 4
Object Render Mode Textures
This is TGMP
(Applied using the BeforeFlip event)
Article 4
Object Render Mode Textures
This is TGMP
(Applied using the BeforeFlip event)
Article 4
Object Render Mode Textures
This is TGMP
(Applied using the BeforeFlip event)

Figure 1: Fully-shaded texture-mapped objects.

A DIB is what a .BMP file becomes on your disk; it's independent of any particular device (such as a printer or monitor) and carries all the information that any device will need to display it. A DIB holds color information, such as palette and bit depth, along with file size information and any compression algorithms used.

For our purposes, we want to create a DIB in memory, rather than load it from disk. To accomplish this, we'll use a Windows API function called **CreateDIBSection** which takes the following arguments:

- a handle of a device context
- a variable of *TBitmapInfo* (described later)
- the constant DIB_RGB_COLORS or DIB_PAL_COLORS indicating the type of color data
- a variable to receive a pointer to the bitmap's bit values
- an optional handle to a file mapping object (which we'll set to **nil**)
- an offset to the bitmap bit values within the file-mappingobject (which we'll set to 0)

To call **CreateDIBSection**, which will create a DIB in memory, we need to understand

```
TDIB16bit = class(TObject)
  private
    { Private declarations }
    FDIBHandle       : HBitmap;
    FBheader         : TBitmapInfo;
    FPointerToBitmap : Pointer;
    FScanWidth       : Integer;
    FDeviceContext   : HDC;
  protected
    { Protected declarations }
  public
    { Public declarations }
    procedure FlipBackPage(DeviceContext : HDC);
    procedure SetPixel(X, Y : Integer; Color : Word);
    procedure ClearBackPage(Color : Word);
    procedure DrawHorizontalLine(Y, X1, X2 : Integer;
                                 Color : Word) ;
    function GetHandle : HBitmap;
    constructor Create(Height, Width : Integer);
    destructor Destroy; override;
  end;
```

Figure 2: The declaration of the *TDIB16bit* class.

the *TBitmapInfo* type. This is best explained by following the creation of our *TDIB16bit* class. The declaration for the class is shown in Figure 2.

FDIBHandle is the DIB handle returned by the **CreateDIBSection** function. *FBHeader* is the *TBitmapInfo* variable mentioned previously. We also must pass *FPointerToBitmap*, another parameter pointing to our DIB pixel data, to **CreateDIBSection**. *FScanWidth* is the width of each line in bytes. *FDeviceContext* is the device context with which the bitmap is associated. We'll follow the *Create* constructor of the *TDIB16bit* class line-by-line. The constructor accepts the *Width* and *Height* of the required DIB:

```
constructor TDIB16bit.Create(Height, Width : Integer);
```

And calls the **inherited** *Create* method:

```
inherited Create;
```

The *biWidth* and *biHeight* of *FBHeader* are obvious; they specify the *Width* and *Height*:

```
FBHeader.bmiHeader.biWidth  := Width;
FBHeader.bmiHeader.biHeight := Height;
```

biPlanes specifies the number of planes for the target device. This is always set to *1*:

```
FBHeader.bmiHeader.biplanes := 1;
```

biBitCount specifies the color depth, and *biCompression* is set to *BI_RGB*, which means "no compression":

```
FBHeader.bmiHeader.biBitCount    := 16;
FBHeader.bmiHeader.biCompression := BI_RGB;
```

The image size is dependent on the width of a line. A bitmap scanline must end on a double-word boundary. For instance, if the 16-bit bitmap were 31 pixels wide, the scanline would hold enough space for 32 pixels, although the last pixel wouldn't be used. The formula to find the right width is:

((bit depth * width) + 31) / 2) * 4

Thus, in Object Pascal:

```
FBHeader.bmiHeader.biSizeImage :=
  ((((16*FBHeader.bmiHeader.biWidth)+31) div 32)*4)*Height;
```

FScanWidth is stored so we won't have to recalculate it:

```
FScanWidth :=
  (((FBHeader.bmiHeader.biWidth * 16)+ 31) div 32) * 4;
```

The following are unimportant for us, and can safely be set to zero:

```
FBHeader.bmiHeader.biXPelsPerMeter := 0;
FBHeader.bmiHeader.biYPelsPerMeter := 0;
FBHeader.bmiHeader.biclrUsed       := 0;
FBHeader.bmiHeader.biclrImportant  := 0;
```

biSize is the size of the *bmiHeader* structure within the *TBitmapInfo* structure:

```
FBheader.bmiHeader.biSize := 40;
```

The *bmiColors* are unimportant for the moment, because they relate only to 256-color bitmaps:

```
FBHeader.bmiColors[0].rgbRed      := 255;
FBHeader.bmiColors[0].rgbBlue     := 255;
FBHeader.bmiColors[0].rgbGreen    := 255;
FBHeader.bmiColors[0].rgbReserved := 255;
```

We must also create a device context that we can supply to **CreateDIBSection**. A device context allows the DIB to be drawn on by other GDI (graphics device interface) objects, such as brushes and fonts. Supplying 0 as an argument to *CreateCompataibleDC* provides a device context compatible with the current screen.

Passing the device context to **CreateDIBSection** with all the other parameters returns a handle to the bitmap, and tells us the location of the memory associated with the bitmap by assigning an address to *FPointerToBitmap*. You can also supply the bitmap handle to a *TBitmap*. This is useful if you want to harness *TBitmap*'s ability to save to disk:

```
FDeviceContext := CreateCompatibleDC(0);
FDIBHandle     := CreateDIBSection(FDeviceContext,FBHeader,
                    DIB_RGB_COLORS,PointerToBitmap,nil,0);
```

A Pointed Discussion

Now let's turn to the procedures and functions for the class. The first procedure, *FlipBackPage*, accepts a device context as a parameter:

```
procedure TDIB16bit.FlipBackPage(DeviceContext : HDC);
begin
  StretchDIBits(DeviceContext, 0, 0,
                FBHeader.bmiHeader.biwidth,
                FBHeader.bmiHeader.biheight, 0, 0,
                FBHeader.bmiHeader.biwidth,
                FBHeader.bmiHeader.biheight,
                FPointerToBitmap, FBheader,
                DIB_RGB_COLORS, SRCCOPY);
end;
```

The device context could be the handle of a *TCanvas* object, for instance. In fact, this is exactly what we'll send from our main *TGMP* class. The API call used in this procedure is **StretchDIBits**. It accepts a device context, the left, top, width, and height of the source bitmap, and the same parameters for the destination area. It also takes *TBitmapInfo*, which indicates the bitmap's type; and the copy mode, in this case SRCCOPY, indicating a straight bit-for-bit copy.

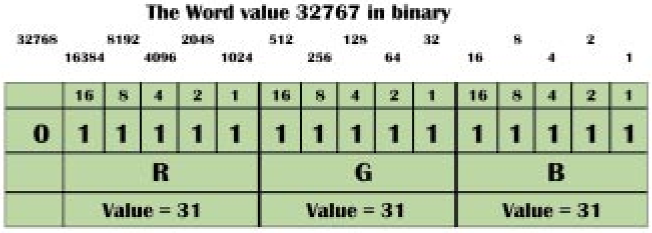
The *ClearBackPage* procedure does exactly that; it accepts a color as a 16-bit word, and clears the back buffer:

```
procedure TDIB16bit.ClearBackPage(Color : Word);
var
  X : Integer;
  BasePointer : ^Word;
begin
  BasePointer := FPointerToBitmap;
  for X := 0 to ((FScanWidth div 2) *
                 (FBHeader.bmiHeader.biHeight))-1 do begin
    BasePointer^ := Color;
    Inc(BasePointer);
  end;
end;
```

We declare a pointer to a Word value, then assign *FPointerToBitmap* to it. The *X* loop counter is worked out by taking the number of bytes in a scanline and dividing it by two, because there are two bytes in a word. The calculated value is then multiplied by the height of the DIB.

To clarify, a pointer is a variable that holds an address to a position in memory. So if we were to look at the contents of *FPointerToBitmap*, we would find a 32-bit number indicating the memory address at which the bitmap begins. *BasePointer* is assigned a color and incremented within the loop body. Notice that the *BasePointer* variable is followed by a caret character (^), the Object Pascal pointer symbol. This tells the compiler to assign the value of *Color* to the address referenced by *BasePointer*, not to the *BasePointer* variable itself. *BasePointer* is incremented on the next line using the *Inc* procedure. A pointer is incremented by the size of the type that it references. In this case, the pointer is incremented by two bytes (a word); in other words, the pointer moves along the DIB memory by two bytes.

If you're still a little stumped with pointers, imagine that you are handed a cinema ticket with your seat number on it. The ticket will point you to the place in the cinema where you are meant to sit. If you think of a pointer as your cinema ticket, it merely references where in memory you want to go. The ticket isn't the seat, but points to it.



The Word value 32767 in binary

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
	16	8	4	2	1	16	8	4	2	1	16	8	4	2	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	R					G					B				
	Value = 31					Value = 31					Value = 31				

Figure 3: Storing a color in a Word variable (16-bit color mode). Red, green, and blue each get five bits; the highest order bit is unused.

The *GetHandle* function is just one line of code that returns a handle to the DIB:

```
function TDIB16bit.GetHandle : HBitmap;
begin
  Result := FDIBHandle;
end;
```

The next procedure, *SetPixel*, accepts an *X* and *Y* position, and color to set the pixel:

```
procedure TDIB16bit.SetPixel(X, Y : Integer; Color : Word);
var
  BasePointer  : ^Word;
begin
  BasePointer  := Pointer(Integer(FPointerToBitmap) +
                          (Y * FScanWidth) + (X * 2));
  BasePointer^ := Color;
end;
```

Again, we declare *BasePointer*, but this time we calculate the memory address of the x,y coordinates. To do this, we take the *FPointerToBitmap* as the start address, then add (*Y * FScanWidth*) to get to the right line, then finally add (*X * 2*). *X* is multiplied by two, because there are two bytes for every pixel, and memory addresses are measured one byte at a time. Then we assign the value of the *Color* variable to the address in memory to which *BasePointer* points.

The *DrawHorizontalLine* procedure works in a similar fashion. It calculates the starting position of the horizontal line in memory, and loops through by a count of the parameters (*X2 - X1*):

```
procedure TDIB16bit.DrawHorizontalLine (Y, X1, X2 : Integer;
                                        Color : Word);
var
  X : Integer;
  BasePointer : ^Word;
begin
  Integer(BasePointer) := Integer(FPointerToBitmap) +
                          (Y*FScanWidth) + (X1*2);
  for X := 0 to (X2 - X1) do begin
    BasePointer^ := Color;
    Inc(BasePointer);
  end;
end;
```

To close out our DIB class, the destructor deletes the DIB and its associated resources:

```
destructor TDIB16bit.Destroy;
begin
  DeleteObject(FDIBHandle);
  DeleteDC(FDeviceContext);
  inherited;
end;
```

It does this by using the Windows API call, **DeleteObject**, which takes the object's handle as a parameter. It then deletes the device context we've held for the DIB using the API function, **DeleteDC**. Finally, the destructor calls the inherited *Destroy* method, ensuring that any code in the class' ancestor destructor will be executed.

Color Crunching

You've probably noticed that many procedures in the DIB class take a Word value (i.e. a two-byte unsigned integer)

```
function TGMP.CalculateRGBWord(Color : TColor) : Word;
var
  Calc : Single;
  R, G, B : Integer;
begin
  { GetRValue, GetGValue, and GetBValue return a value
    based on the color scale of 0 to 255. }
  { Gets the red value and rescale it from a 1/256 number
    to a 0/31 number. }
  Calc := GetRValue(Color) / 2.56;
  Calc := Calc * 0.31;
  R    := Round(Calc);
  { Get the green value and rescale it from a 1/256 number
    to a 0/31 number. }
  Calc := GetGValue(Color) / 2.56;
  Calc := Calc * 0.31;
  G    := Round(Calc);
  { Get the blue value and rescale it from a 1/256 number
    to a 0/31 number. }
  Calc := GetBValue(Color) / 2.56;
  Calc := Calc * 0.31;
  B    := Round(Calc);
  { B Value is last five bits. Leave it alone. }
  { R is shifted left 10 bits to sit at position 15-11.}
  R := R shl 10;
  { G is shifted left 5 bits to sit at position 10..6.}
  G := G shl 5;
  { Add them together. }
  Result := R + G + B;
end;
```

```
TPolygon = record
  { Allow only 4-point polygons. }
  Point : array [0..3] of TPoint3D;
  NumberPoints : Integer; { Number of points in polygon. }
  Visible : Boolean;  { Visibility of polygon; determined
                        by RemoveBackfacesAndShade. }
  AverageZ  : Single; { For Z Sorting. }
  PolyColor : TColor; { Color of the polygon. }
  DibColor  : Word;   { 16-bit value of PolyColor. }
  Intensity : Byte;   { Light intensity. }
end;

TObject3D = record
  { Stores local coordinates of the object's polygons. }
  PolyStore : array [0..MAXPOLYS] of TPolygon;
  { Stores world coordinates of the object's polygons. }
  PolyWorld : array [0..MAXPOLYS] of TPolygon;
  NumberPolys : Integer;  { Number of polygons in object. }
  Color : TColor;  { Color for solid shading & wireframe. }
  DibColor : Word;   { 16-bit value of color. }
  World : TPoint3D;  { Position of object in "the world". }
end;
```

Figure 4 (Top): *CalculateRGBWord* takes a *TColor* as an argument and returns the appropriate 16-bit value. **Figure 5 (Bottom):** The *TPolygon* and *TObject3D* records.

as the color. In 16-bit color mode, a pixel in a DIB takes up two bytes (see Figure 3). The RGB values are embedded in that value; the "first" bit of the 16 — the one with the highest value — isn't used. The next five bits are the red value, the following five are the green value, and the "last" five bits are the blue value. Each color has a value of 0 to 31.

We've written a support method, *CalculateRGBWord*, that converts *TColor* into a Word value (see Figure 4), along with various other methods tied into the shaded texturing model, which we'll address later.

```
procedure TGMP.CalcIntensityLUT;
var
  X, Y: Integer;
  UpIncrement, DownIncrement : Single;
begin
  { Find possible R, G, or B Values — 0 to 31. }
  for X := 0 to 31 do begin
    { The up increment is from the initial
      color value to its brightest. }
    UpIncrement := (31 - X) / 16;
    { The down increment is from the initial
      color value to its darkest. }
    DownIncrement := X / 15;
    { Loops through from color 0 to color 15,
      using DownIncrement. }
    for Y := 0 to 15 do
      IntensityLUT[X, Y] := Round(DownIncrement * Y);
    { Loops from color 16 to color 31, using UpIncrement.
}
    for Y := 1 to 16 do
      IntensityLUT[X, Y + 15] :=
        Round((DownIncrement * 15) + (UpIncrement * Y));
  end;
end;
```

Figure 6: The *CalcIntensityLUT* procedure works out the lookup table.

Shades of Change

In our last article, we explained the methodology behind the texture mapping of polygons. To accommodate texture shading and the new DIB class, we'll need to expand the current *TPolygon* and *TObject3D* records (see Figure 5). In *TPolygon*, we store an *Intensity* value, which is the light intensity against that polygon. We use this with a *texel* (textured pixel) to determine the texel's correct shade. An extra line has been inserted into the *RemoveBackfacesAndShade* procedure. It's shown here above an existing line:

```
AnObject.PolyStore[CurrentPoly].Intensity :=
  Round(Intensity);  { Line to add }
AnObject.PolyStore[CurrentPoly].PolyColor :=
  RGB(R, G, B);      { Existing line }
```

Next, for reasons of speed, we need to set up a two-dimensional lookup table that will return a shade for any R, G, or B value. The *CalcIntensityLUT* procedure works out our lookup table (see Figure 6). You'll also need to place this statement in the *Create* constructor of *TGMP*:

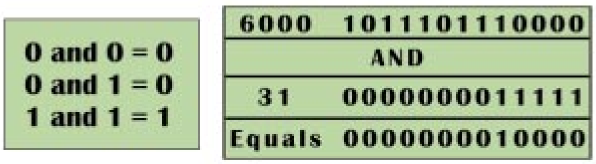
```
CalcIntensityLUT;
```

Now that we have our lookup table, we need a function that takes a 16-bit color and an *Intensity* value as arguments, and returns a new 16-bit color with each correctly shaded RGB element. This is complicated, because the method has to extract the separate RGB values from the Word value, then get the shades for each RGB, and recombine them into a Word value.

The method, named *GetShadedWord* (see Figure 7), is quite interesting, because it covers some new programming ground related to bit-wise manipulation. It shows how to use bit masking, and works on the principle that **and** can conjoin values to extract a new value. Figure 8 shows the logical results of "and-ing" different binary values, then shows how to extract the five-bit *G* value from a 16-bit value.

```
function TGMP.GetShadedWord(Texture : Word;
                            Intensity : Integer) : Word;
var
  intRed, intGreen, intBlue, intBitMask : Integer;
begin
  { Bitmask for 0000000000011111 is 31. This gives us
    the last 5 bits for Blue. }
  intBitMask := 31;
  intBlue    := Texture and intBitMask;
  { Bitmask for 0000001111100000 is 992. This gives us
    the the middle 5 bits for Green. }
  intBitMask := 992;
  intGreen   := Texture and intBitMask;
  intGreen   := intGreen shr 5;
  { Bitmask for 0111110000000000 is 31744. This gives us
    the bits for the Red element — 15-11. }
  intBitMask := 31744;
  intRed     := Texture and intBitMask;
  intRed     := intRed shr 10;
  { Get the new shades using the lookup table
    we worked out before. }
  intRed     := IntensityLUT[intRed, Intensity];
  intGreen   := IntensityLUT[intGreen, Intensity];
  intBlue    := IntensityLUT[intBlue, Intensity];
  { Shift Red and Green into their correct places,
    and add all the elements together. }
  intRed     := intRed shl 10;
  intGreen   := intGreen shl 5;

  Result   := intRed + intBlue + intGreen;
end;
```

Figure 7: The *GetShadedWord* method.



0 and 0 = 0
0 and 1 = 0
1 and 1 = 1

6000	1011101110000
AND	
31	0000000011111
Equals	0000000010000

Figure 8: Bit masking: "and-ing", then extracting the last five bits, for a value of 16.

After all that preparation, we can finally add the *rmShadedTexture* element to *TRenderMode,* then place a new section of code to implement the shaded texturing in the *RenderNow* procedure. The new section of code is shown in Figure 9. Please notice, and tolerate for the moment, that all the points are taken from *PolyWorld* rather than *PolyStore*. This will all be explained shortly, along with the *LocalToWorld* procedure.

The next step in implementing shaded texturing is adding a new statement to the *DrawLine3DTexture* procedure. We have given you a little more of the procedure listing than just the statement, so you can see where the line is to be added:

```
case RenderMode of
  rmSolidTexture :  { Existing line }
    DrawTextureLine2D(NewStartPoint, NewEndPoint,
                      TextStart, TextEnd);
  rmShadedTexture : { New line }
    DrawTextureLine2D(NewStartPoint, NewEndPoint,
                      TextStart, TextEnd);
end;
```

At long last, we show you the final change needed before we can move onto our object file reader. The *RenderYBuckets* procedure has been modified; the new version is shown in Figure 10.

```
//********** Shaded  Texture ****************************
rmShadedTexture :
  begin
    RemoveBackfacesAndShade(Object3D);
    OrderZ(Object3D);
    LocalToWorld(Object3D);
    for X := 0 to Object3D.NumberPolys - 1 do
      with Object3D.PolyWorld[x] do begin
        if (Object3D.PolyWorld[x].Visible = False) then
          Continue;
        ClearYBuckets;
        FIntensity := Object3D.PolyWorld[x].Intensity;
        if (NumberPoints = 3) then
          begin
            TextureStart.X := 63; TextureStart.Y:= 0;
            TextureEnd.X   := 0;  TextureEnd.Y  := 127;
            DrawTextureLine3D(Point[0], Point[1],
                              TextureStart, TextureEnd);
            TextureStart.X := 0;   TextureStart.Y:= 127;
            TextureEnd.X   := 127; TextureEnd.Y  := 127;
            DrawTextureLine3D(Point[1], Point[2],
                              TextureStart, TextureEnd);
            TextureStart.X := 127; TextureStart.Y:= 127;
            TextureEnd.X   := 63;  TextureEnd.Y  := 0;
            DrawTextureLine3D(Point[2], Point[0],
                              TextureStart, TextureEnd);
          end
        else
          begin
            TextureStart.X := 127; TextureStart.Y:= 0;
            TextureEnd.X   := 127; TextureEnd.Y  := 127;
            DrawTextureLine3D(Point[0], Point[1],
                              TextureStart, TextureEnd);
            TextureStart.X := 127; TextureStart.Y:= 127;
            TextureEnd.X   := 0;   TextureEnd.Y  := 127;
            DrawTextureLine3D(Point[1], Point[2],
                              TextureStart, TextureEnd);
            TextureStart.X := 0; TextureStart.Y   := 127;
            TextureEnd.X   := 0; TextureEnd.Y     := 0;
            DrawTextureLine3D(Point[2], Point[3],
                              TextureStart, TextureEnd);
            TextureStart.X := 0;   TextureStart.Y:= 0;
            TextureEnd.X   := 127; TextureEnd.Y  := 0;
            DrawTextureLine3D(Point[3], Point[0],
                              TextureStart, TextureEnd);
          end;

        RenderYBuckets;

      end;   { End of with statement. }
  end;  { End of rmSolidTexture statement. }
```

Figure 9: Implementing shaded texturing in the *RenderNow* procedure.

GEO Files

What's been annoying us most about the *TGMP* class is the fact that we must type all the vertices of an object into the application code. To eliminate any further vertices typing, we'll write a method that will read objects that have been saved to disk. The first object file format that *TGMP* will read will be for .GEO objects. The .GEO object file format provides a useful, generic way of storing data in a text file. The file format is shown below, with comments in braces:

```
3DG1      { This identifies this file as a .GEO file }
3         { This is the number of vertices }
1.000000 -1.000000 0.000000  { Vertice 0 - x,y,z }
0.923880 -1.000000 0.382683  { Vertice 1 - x,y,z }
0.707107 -1.000000 0.707107  { Vertice 2 - x,y,z }
3 0 1 2 25
```

In the last line, the first number indicates the number of points in the polygon. The following three numbers tell you which three vertices — from the list — you must join. The

```
if (RenderMode = rmShadedTexture) then
 begin
  { Loop through all buckets. }
  for Y := 0 to 479 do begin
   if (YBuckets[Y].StartX = -16000) then
    Continue;
   { Calculate all the Texture X,Y increments. }
   Length := (YBuckets[Y].EndX - YBuckets[Y].StartX) + 1;
   TextXIncr := ((TextureBuckets[Y].EndPosition.X -
       TextureBuckets[Y].StartPosition.X)) / length ;
   TextYIncr := ((TextureBuckets[Y].EndPosition.Y -
       TextureBuckets[Y].StartPosition.Y)) / length ;
   TextX := TextureBuckets[Y].StartPosition.X;
   TextY := TextureBuckets[Y].StartPosition.Y;
   { Loop through all pixels on the Y line. }
   for I:=YBuckets[Y].StartX to YBuckets[Y].EndX do
begin
    { Perform clipping if pixel's X value < 0. }
    if (I < 0) then
     begin
      TextX := TextX + TextXIncr;
      TextY := TextY + TextYIncr;
      Continue;
     end;
    { Perform clipping if pixel's X value > width. }
    if (I > Width) then
     begin
      TextX := TextX + TextXIncr;
      TextY := TextY + TextYIncr;
      Continue;
     end;
    { Use DIBClass to set pixel. Get texel from
     FCurrentBitmap, and use GetShadedWord to return
     correctly-shaded texel to pass into SetPixel. }
    FDib.SetPixel(I,Y,GetShadedWord(FCurrentBitmap[
         Round(TextX),Round(TextY)],FIntensity));
    TextX := TextX + TextXIncr;
    TextY := TextY + TextYIncr;
   end;
  end;
 end;
```

```
3DG1 { Standard GEO file header }
8    { The number of vertices. }
-1.000000 -1.000000 -1.000000   { Vertice 0 }
-1.000000  1.000000  1.000000   { Vertice 1 }
-1.000000  1.000000 -1.000000   { Vertice 2 }
-1.000000 -1.000000  1.000000   { Vertice 3 }
 1.000000 -1.000000 -1.000000   { Vertice 4 }
 1.000000  1.000000  1.000000   { Vertice 5 }
 1.000000  1.000000 -1.000000   { Vertice 6 }
 1.000000 -1.000000  1.000000   { Vertice 7 }

4 0 3 1 2 25 { Polygon 1 }
{ The first number in Polygon 1 indicates number the of
  points. Join Vertice 0 to Vertice 3 to Vertice 1 to
  Vertice 2. }
4 4 0 2 6 25 { Polygon 2 }
{ Join Vertice 4 to Vertice 0 to Vertice 2 to Vertice 6. }
4 6 2 1 5 25 { Polygon 3 }
4 5 1 3 7 25 { Polygon 4 }
4 7 3 0 4 25 { Polygon 5 }
4 4 6 5 7 25 { Polygon 6 }
```

Figure 10 (Top): The modified *RenderYBuckets* procedure.
Figure 11 (Bottom): The CUBE.GEO file.

final number (25 in this example) is a number you can use to store color information, or anything else about the polygon (e.g. a number in an array of textures).

We used the .GEO file format because it's easy, and because we had a lot of Lightwave-generated objects that came with a converter to take the binary Lightwave file and output it to an ASCII .GEO file. There are several major modelers, such as 3D Studio, SoftImage, and ElectricImage, that can create similar objects. The .GEO file format is not widely used, but is similar in many respects to the .PLG file format (.PLG files can easily be edited into a .GEO format).

(The .PLG format was invented by the writers of the real-time renderer REND386, and has become quite popular. You can find a large number of converters on the Internet that will take things like 3D Studio binary files and convert them to .PLG. You can find a lot of information on 3D modelers and converters at: http://www.hit1.washington.edu/people/poup/internet/3D.html.)

Figure 11 shows the CUBE.GEO file used by the sample application for this article. It's completely commented, with full explanations for all lines contained in the file, and includes the full implementation of the .GEO file reader function.

The World Coordinate System

The last subject we'll cover in this article is the *world coordinate system* (or WCS). So far we've been using the *local coordinate system*, and have simply added a *Z* value to the local coordinates to move the object back and forth. Now we'll introduce the ability to move the object around.

As you saw earlier, when we listed the *TObject3D* structure, there was a *PolyWorld* array and a world *TPoint3D* structure. The world structure holds the x, y, z position of the object in 3D space. In addition, if you look back at the *rmShadedTexture* section of the *RenderNow* procedure, you'll see another procedure being called, namely *LocalToWorld*. This procedure copies all the polygon information from *PolyStore* to *PolyWorld*, and adds the world x, y, z coordinates to the local coordinates as it copies them. This means that we've translated the object from its local coordinates to its world coordinates.

We won't list the *LocalToWorld* procedure here, because it's fairly simple and is available with the source code. However, we have provided a visual comparison of local to world coordinates (see Figure 12).

A World of Changes

Many minor changes have been made to the code in *TGMP*. In the *RenderNow* procedure, for instance, all references to *PolyStore* have been changed to *PolyWorld* to incorporate the world coordinate system. Also, a call to the *LocalToWorld* procedure has been inserted into each clause of the *TRenderMode* **case** statement. The *LightStrength* property has been added to allow you to control the brightness of the light (a value of 1 represents 100 percent).

Another property is an event called *BeforeFlip*, which is sent to *TCanvas* as a parameter. This allows you to add any text/drawing on top of what has been rendered, before it's drawn onto the *TGMP* Canvas. The necessary code for

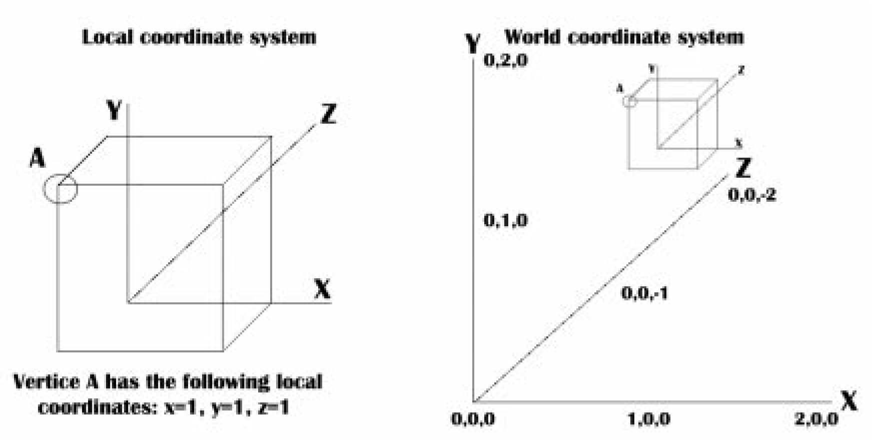


Figure 12: A comparison of local and world coordinate systems. Vertice A retains its original coordinates through the translation.

BeforeFlip is shown below, along with the code to declare our three new mouse movement properties: the familiar *OnMouseMove*, *OnMouseUp*, and *OnMouseDown*:

```
type
  TBeforeFlip = procedure (Canvas : TCanvas) of object;

  { Place in the private section. }
  FBeforeFlip : TBeforeFlip;
  FLightStrength : Single;

  { Place as first line in TGMP.FlipBackPage procedure. }
  if Assigned(FBeforeFlip) then
    FBeforeFlip(FBackBuffer.Canvas);

  { Place in published section. }
  property LightStrength : Single read LightStrength
                                  write LightStrength;
  property BeforeFlip : TBeforeFlip read FBeforeFlip
                                    write FBeforeFlip;
  property OnMouseMove;
  property OnMouseDown;
  property OnMouseUp;
```

These mouse properties are defined similarly to the *Align* property. They've already been declared in the class that *TGMP* was inherited from, so a simple declaration will cause them to appear in the object inspector of *TGMP*.

Other changes include two new constants, MAXPOLYS and MAXPOINTS, which control how many points and polygons we want to allow for *TObject3D*. Another minor change is the call to *ClearBackPage* which now uses the DIB class to do the work:

```
FDib.ClearBackPage(CalculateRGBWord(FColor));
```

Notice how we use the *CalculateRGBWord* function to return the 16-bit value that *ClearBackPage* requires. More small changes throughout the code are similar to those already mentioned.

A look through the *TGMP* source code will reveal the other modifications. (All source associated with this article is available on disk or online; see end of article for details.)

The Fourth Edition

In our fourth application, we've added a new menu item to allow for the selection of various textures, and introduced some limited movement control of the objects through the use of the mouse and keyboard.

You'll also notice that the arrays for the objects have been removed. These arrays are no longer necessary, since the objects are now read into memory via the file reader we've created. Again, you must remember to set your display driver to 16-bit color, rather than 256 colors. Otherwise you'll think we've placed a "mud" texture onto the rendered objects!

Conclusion — and a Look Ahead

This is a long article; still, we have to leave a number of things out until next time. In the next installment we'll cover topics that will make *TGMP* a truly useful game tool. First we'll add the final coordinate system: the *camera coordinate system*, that will allow you to move the camera through your virtual world.

We'll optimize the structure of the component to easily allow multiple objects to be seen at once. We'll also cover many optimization techniques that will let *TGMP* process polygons much faster. These techniques include lookup tables, pre-calculation, fast multiplication, clipping, method parameter reduction, loop optimization, fixed-point math, and some common sense. We'll also add a few design-time features such as positioning of objects, setting of background bitmaps, and light-source positioning. Soon we'll be ready to make a game using *TGMP*. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\APR\DI9704DP.

Peter Dove is a partner in Graphical Magick Productions, specialists in graphics, training, and component development. He can be reached via the Internet at peterd@graphicalmagick.com.

Don Peer is a Technical Associate with Greenway Group Holdings Incorporated (GGHI). He can be reached via the Internet at dpeer@mgl.ca.

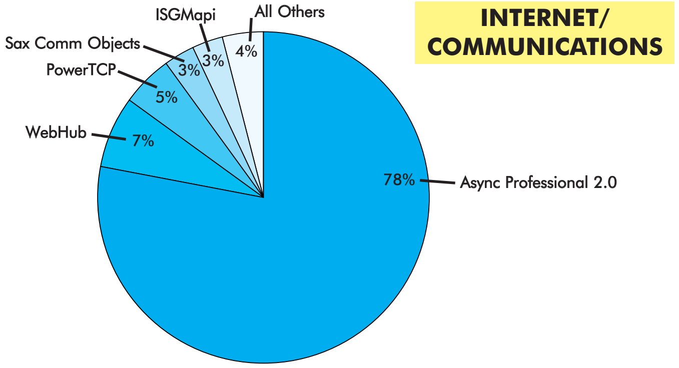
By *James Hofmann and Cathi Pickavet*

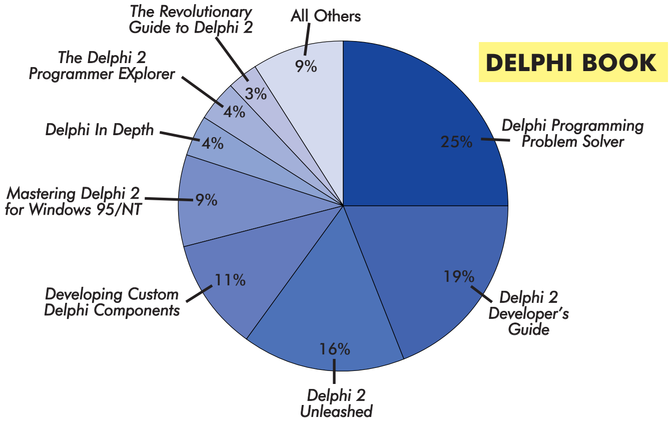
RAD Results

The 1997 *Delphi Informant* Readers Choice Awards

When the votes were tallied last year, we thought the precedents had been set — the Delphi market's most powerful players had emerged. Little did we know how quickly things would change. To start, Borland released Delphi 2, then added Internet and intranet functionality. What followed could only be described as a plethora of new tools developed at RADical speeds.

To reflect Delphi's ever-changing third-party market, we altered this year's ballot. You'll still see some of last year's winners in repeat performances, dominating their respective categories; but don't let your eyes stray, you might miss some of the new categories and winners — several we're sure you'll find quite surprising.





The voting was fast and fierce, so without further ado ...

Best Internet/Communications

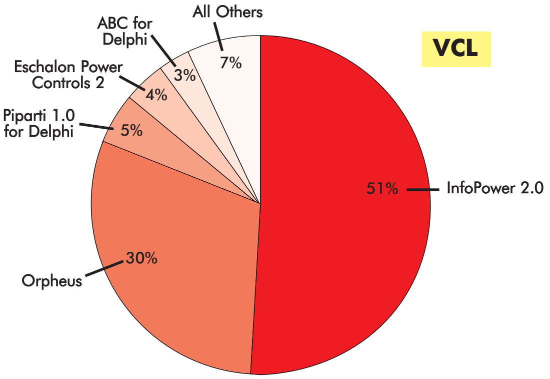
This year brought Internet and intranet functionality into the Delphi developer's hands. To satisfy your end-users' every need, the great majority of you agree that TurboPower's Async Professional 2.0 is the way to go. This communication toolkit for Delphi 1 and 2 took 78 percent of the votes, establishing itself as the clear winner. And don't forget TurboPower, because you'll see the name again. This company won two categories in this year's awards, the first to ever do so.

Async Professional's event-driven architecture allows users to read, view, print, and send faxes. It features complete serial port control, terminal emulation, telephony API, debugging tools, and more. According to TurboPower, an upgraded version is planned for release after Delphi 3 begins shipping.

Best Delphi Book

From a field of over 25, Neil Rubenking hit the tape first with his *Delphi Programming Problem Solver*, from IDG Books Worldwide. His "answer book" locked up 25 percent of the votes to win the title of Best Delphi Book. Covering Delphi 1 and 2, *Delphi Programming Problem Solver* tackles many of the vexing Delphi questions that can hinder first time — or even seasoned — Delphi programmers.

Xavier Pacheco and Steve Teixeira came in second with *Delphi 2 Developer's Guide*, from SAMS Publishing. The sequel to last year's Best Delphi Book, this edition accounted for 19 percent of the votes.

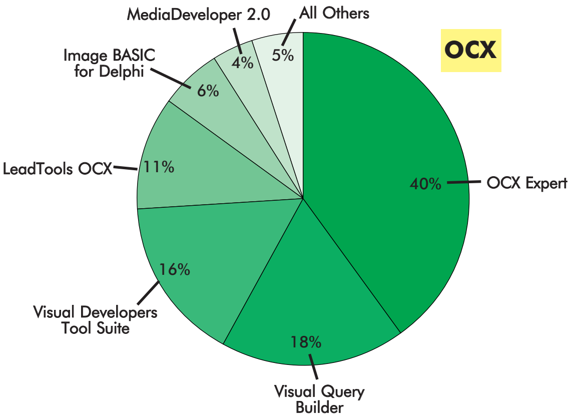


Best VCL

The first of the repeat winners, Woll2Woll's InfoPower 2.0 prevailed as this year's Best VCL. Garnering more than half the votes in the category, InfoPower made an impressive showing. In his January '97 *Delphi Informant* review, Bill Todd touted it as the "must-have addition to Delphi for anyone developing database applications." It's fair to say you agree.

Here are just a few of InfoPower's features: a Table component that fully supports Borland Database Engine (BDE) filters, a QBE component that fully supports QBE queries, an enhanced data-aware grid component, high-performance search controls, a customizable pop-up memo field editor, and DBComboBox, DBComboDialog, DBLookupCombo, and DBLookupComboDialog components. And there are plans for a new and improved version of InfoPower for Delphi 3. Stay tuned to *DI*'s "Delphi Tools" section for details.

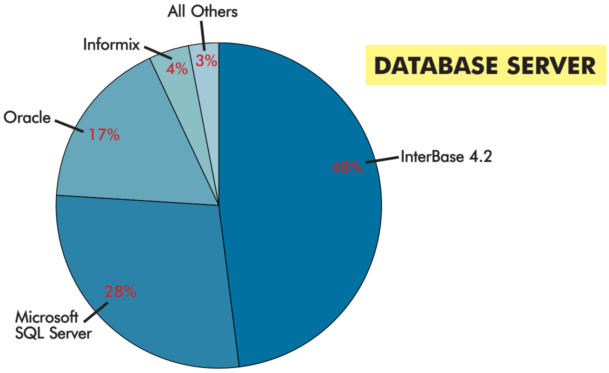
Although still second, TurboPower's Orpheus is closing the gap in this category. Last year InfoPower secured 47 percent and Orpheus followed with 16. This year the percentages have changed to 51 and 30, respectively. And next year's competition has already begun ...



Best OCX

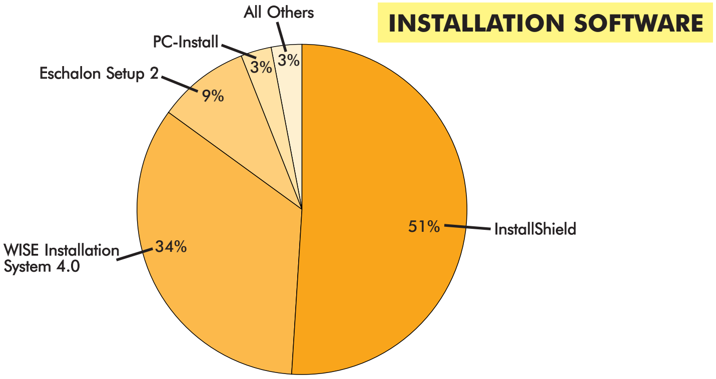
New this year, the Best OCX category was very competitive. Apiary Inc. took the top prize with its OCX Expert, receiving 40 percent of the votes. OCX Expert enables developers to create 32-bit ActiveX controls using Delphi 2. It also converts existing Delphi 2 VCL components to ActiveX controls with few or no changes by the developer.

The run for second place was close, with Integra's Visual Query Builder finishing just two percent higher than Visual Tools' Visual Developers Tool Suite.



Best Database Server

For the second consecutive year, Borland's InterBase has been named Best Database Server, acquiring 48 percent of the votes. The database server field included several industry favorites, such as Microsoft SQL Server (28 percent), Oracle (17 percent), and Informix (four percent).

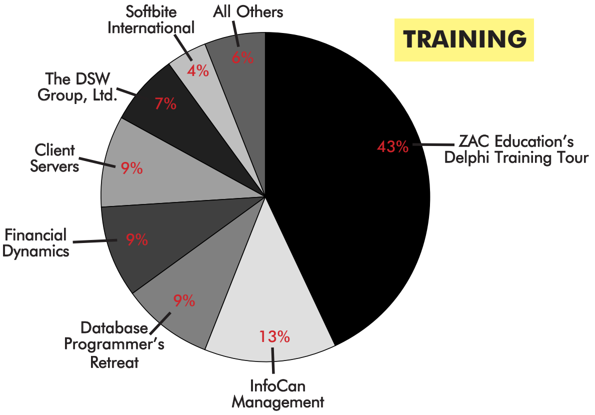


Best Installation Software

In a repeat performance, InstallShield Corp. aced the Best Installation Software category. Great Lakes Business Solutions, Inc. took second with WISE Installation System 4.0. Moving from 30 and 26 percent to 51 and 34 percent respectively, it looks as if InstallShield and WISE are moving away from the pack.

Best Training

There's a new face in the winner's circle for Best Training — ZAC Education's Delphi Training Tour. After a whirlwind tour of the US, ZAC garnered 43 percent of the votes.

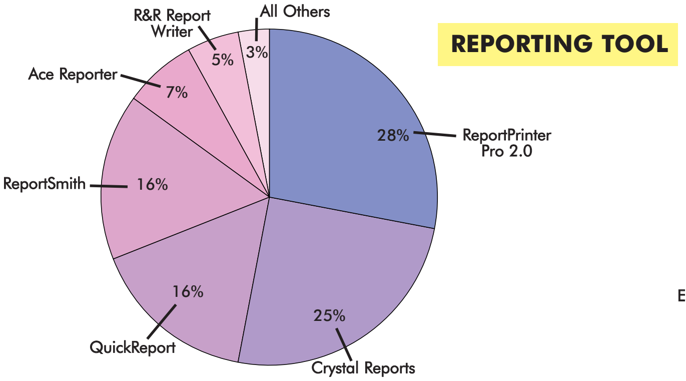


The wide variety of training companies competing this year are yet another reminder of Delphi's popularity. InfoCan Management finished second, followed by a three-way tie between Database Programmer's Retreat, Financial Dynamics, and Client Servers. With Delphi 3 shipping soon, this category is sure to be another tight race next year.

Best Reporting Tool

In a race to the finish, Nevrona Designs' ReportPrinter Pro 2.0 inched to victory as Best Reporting Tool. Finishing third last year, ReportPrinter Pro narrowly bested the '96 champ, Seagate Software's Crystal Reports, by a margin of three percent.

In the September 1996 *Delphi Informant*, Bill Todd gave ReportPrinter Pro a thumbs-up, citing such enhancements as the *TDBTablePrinter* component's Table Editor, the *TReportSystem* component, and some "major" memo printing capabilities.

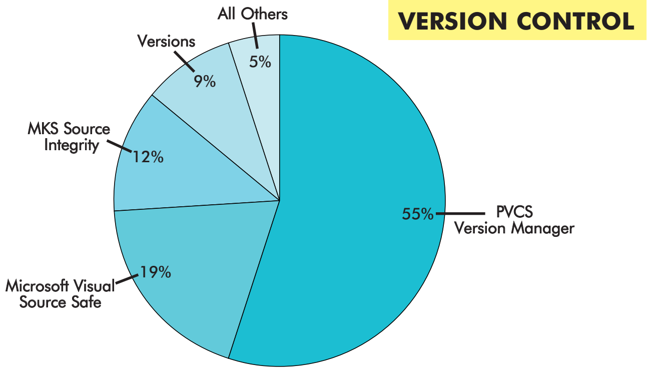


Best Version Control

Proving again to be the Delphi developer's versioning tool of choice, INTERSOLV's PVCS Version Manager dominated the field, acquiring 55 percent of the votes to win the Best Version Control category.

PVCS version 5.2 features an enhanced user interface, allowing developers to drag a file from the project window into a directory to check out the file. Check-in occurs when you drag a file from a directory to the project window. The check-in and check-out process is designed to eliminate redundancy and unnecessary data, improving performance. Apparently it's a worthy enhancement, as the PVCS Version Manager continues to be your favorite.

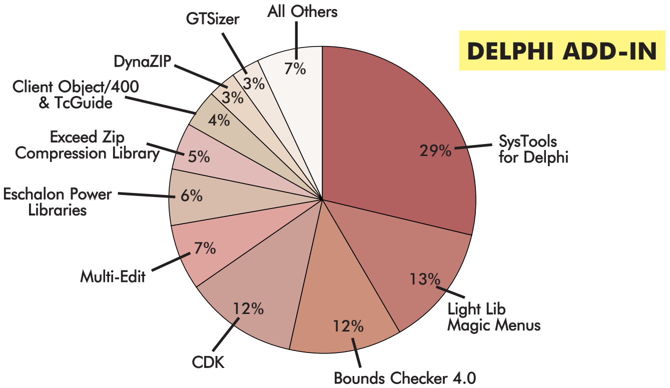
Microsoft's Visual Source Safe — a write-in! — finished second with 19 percent of the votes.



Best Delphi Add-In

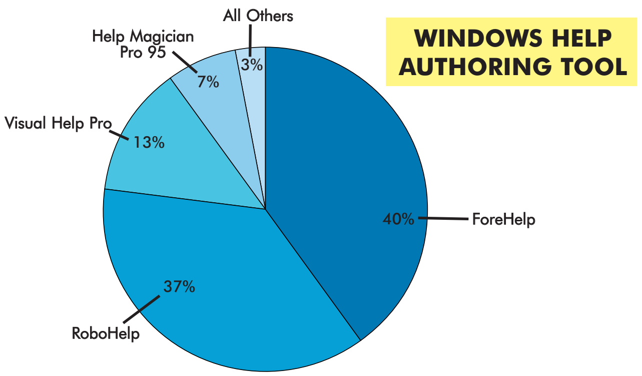
In the category with the most competitors, TurboPower's SysTools for Delphi outscored the others, grabbing 29 percent of the votes for the Best Delphi Add-In. SysTools is a collection of system-level classes that parallel Delphi's menu classes. SysTools includes units for string manipulation, date and time handling, container classes, high-precision math, data sorts, registry and .INI file handling, and system access routines. (See Alan Moore's review of SysTools in the January 1997 *DI*.)

Finishing with 13 percent of the vote, DFL Software took second with Light Lib Magic Menus, a product that adds images to Delphi menus. (See Douglas Horn's review of Magic Menus in the February 1997 *DI*.)



Best Windows Help Authoring Tool

The closest race overall was in the Windows Help Authoring category. The lead changed daily, if not hourly, as votes poured in. In the end, ForeFront Inc.'s ForeHelp grabbed the victory from last year's winner — RoboHelp, from Blue Sky

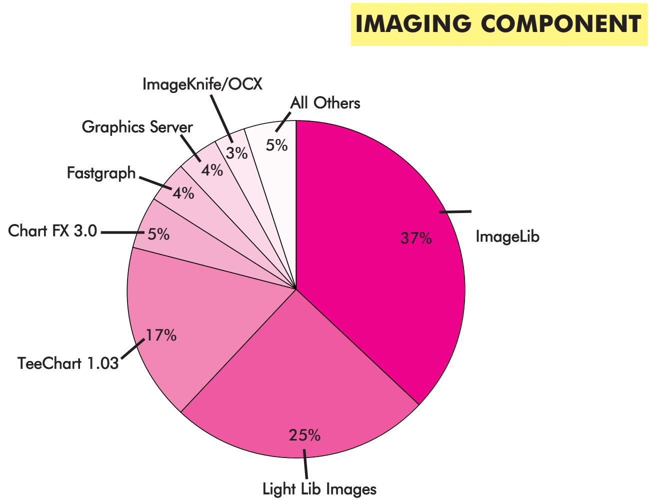


Software. In this donnybrook, ForeHelp finished with 40 percent of the vote, and Blue Sky Software had 37 percent.

Best Imaging Component

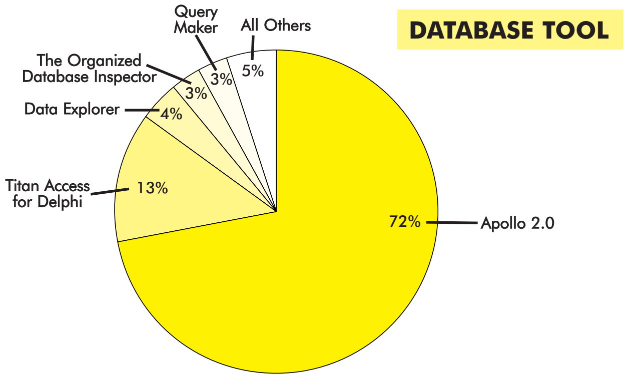
Another first-timer in this year's contest is the Best Imaging Component category. However, the winner with 37 percent of the vote, ImageLib from Skyline Tools, is no stranger to Delphi developers. Skyline offers a special VCL/DLL package for Delphi developers to incorporate multimedia elements into their applications. (See Douglas Horn's review in the July 1996 *DI*.)

DFL Software continued with its competitive edge as Light Lib Images finished second with 25 percent.



Best Database Tool

Is this the year for first timers, or what? Not only did we add the Best Internet/Communications and Best Imaging Component categories to the ballot, we also tossed in Best Database Tool; Delphi is used so heavily in database development, we thought a category recognizing some of the database-centric products was in order.



SuccessWare International, last year's winner in the Best Add-In category, has established its product, Apollo 2.0, as the clear leader in this category. Apollo blasted off to a first-place finish in the first year of competition, earning 72 percent of the votes. Apollo has been recognized in other award circles as well. In December, SharewareJunkies.com, a Shareware evaluation site on the Web, had its visitors vote on the Best Shareware of 1996. Conversion Buddy, a product using Apollo as its database engine, won the award for Best Windows Freeware program.

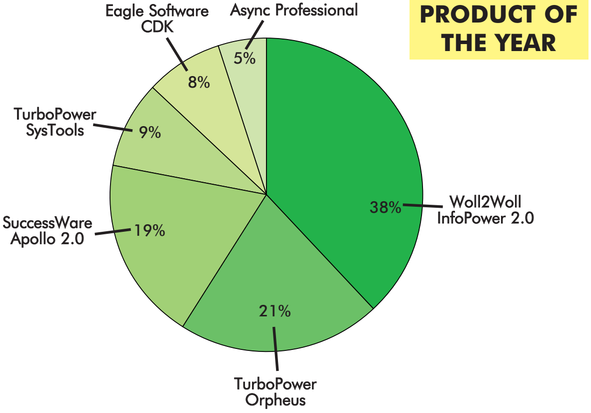
Product of the Year

The Product of the Year category was handled a bit differently this year. (In the inaugural Readers Choice Awards, the product with the most votes overall won the award.)

Because some categories have heavier voting patterns than others, the products in those categories stood a better chance of acquiring more overall votes, thus a better chance of winning the coveted Product of the Year award.

This year, we created a specific category where you either entered the number of your product of choice from the ballot, or wrote in the name of your favorite product. This way, the products in the less mainstream categories had an equal chance.

Having said all that — adding a separate category didn't change a thing! Woll2Woll Software's InfoPower is the hands-down Product of the Year for the second consecutive year. Not only did it receive 38 percent of the votes, besting



TurboPower's Orpheus (21 percent), it also received the most votes overall from the ballot.

Thank You

It takes dedicated, talented people to get such quality Delphi products to the marketplace. We'd like to thank the vendors who dare to dream, plan, create, test, and distribute unique Delphi products. Some began developing third-party products at night and on weekends. We salute your perseverance; without your wares, the Delphi developer's job would be very different.

This year looks to be even more exciting than the last two. The Delphi community is well established and continues to grow, Delphi 3 is set to ship, and developers have a slew of quality products to pick from. As the Delphi projects abound, we hope to see you push the boundaries and set new standards — and bring Delphi development into even greater prominence. Δ

Delphi Informant's Products Editor James Hofmann is available at jhofmann@informant.com. To contact Cathi Pickavet, Editorial Assistant for *DI*, e-mail cpickavet@informant.com.

Contacting the Winners

Best Internet/Communications
Async Professional 2.0
TurboPower Software
Phone: (800) 333-4160 or
(719) 260-9136
Web Site: http://www.tpower.com

Best Delphi Book
Delphi Programming Problem Solver
By Neil Rubenking
IDG Books Worldwide
Phone: (800) 434-3422 or
(415) 655-3000
Web Site: http://www.idgbooks.com

Best VCL
InfoPower 2.0
Woll2Woll Software
Phone: (800) wol2wol or
(510) 371-1663
Web Site: http://www.woll2woll.com

Best OCX
OCX Expert
Apiary Inc.
Phone: (501) 376-3600
Web Site: http://www.apiary.com

Best Database Server
InterBase 4.2
Borland International, Inc.
Phone: (408) 431-1000
Web Site: http://www.borland.com

Best Installation Software
InstallShield
InstallShield Corp.
Phone: (800) 374-4353 or
(847) 240-9111
Web Site: http://www.installshield.com

Best Training
Delphi Training Tour
ZAC Education
Phone: (800) 463-3574
Web Site: http://www.zaccatalog.com

Best Reporting Tool
ReportPrinter Pro 2.0
Nevrona Designs
Phone: (888) PROGSOL or
(602) 899-0794
Web Site: http://www.nevrona.com/designs

Best Version Control
PVCS Version Manager
INTERSOLV
Phone: (800) 547-7827 or
(503) 645-1150
Web Site: http://www.intersolv.com

Best Delphi Add-In
SysTools for Delphi
TurboPower Software
Phone: (800) 333-4160 or
(719) 260-9136
Web Site: http://www.tpower.com

Best Windows Help Authoring Tool
ForeHelp
ForeFront Inc.
Phone: (800) 867-1101 or
(713) 961-1101
Web Site: http://www.ffg.com

Best Imaging Component
ImageLib
Skyline Tools
Phone: (800) 404-3832 or
(818) 766-3900
Web Site: http://www.imagelib.com

Best Database Tool
Apollo 2.0
SuccessWare International
Phone: (800) 683-1657 or
(909) 699-9657
Web Site: http://www.gosware.com

Product of the Year
InfoPower 2.0
Woll2Woll Software
Phone: (800) wol2wol or
(510) 371-1663
Web Site: http://www.woll2woll.com

INSIDE OP

Delphi / Object Pascal

By Gary Warren King

Timely Changes

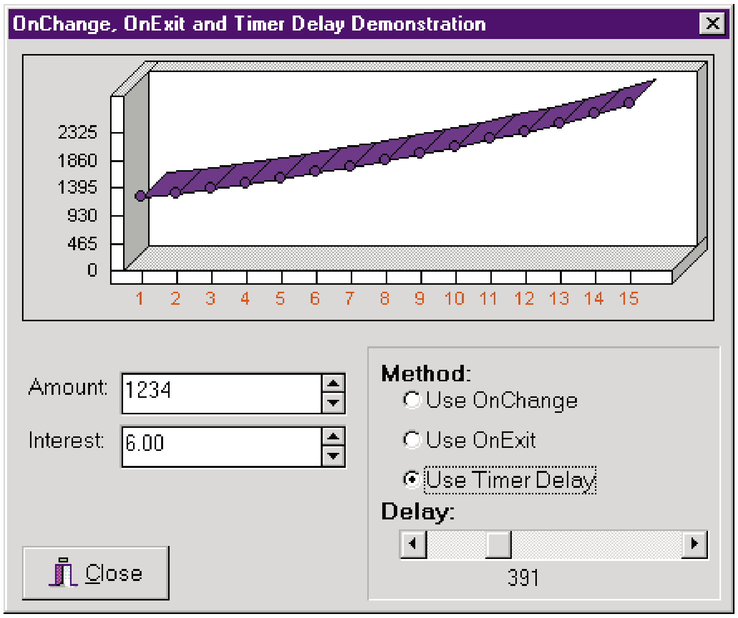
Improve Your UI's Responsiveness with a Nifty Timer Technique

It's common for forms to have interrelated UI controls. For example, we may want a graph to change as we manipulate the values in a form's Edit components. Delphi makes it easy to handle this task with the *OnChange* and *OnExit* events associated with most data entry controls.

There is a hidden problem, however. In short, the *OnChange* event occurs too often, and the *OnExit* event doesn't occur often enough.

Too Much *OnChange*

Because the *OnChange* event fires every time you make a change to a data entry field, it can occur too frequently. [Note: The term *field* is used throughout this article to refer to a data entry control (e.g. an Edit component), not in its strict sense as an object's data member, nor as a database table's column.] For example, entering the number 234.45 will set off six *OnChange* events (more if there are typos and corrections). If the calculations dependent on this field are complex or time consuming (for example, updating a graph), the responsiveness of the user interface will rapidly decay — as will the user's patience and productivity. Even rapid calculations and screen updates may cause a disconcerting flicker.


OnChange, OnExit and Timer Delay Demonstration

Amount: 1234

Interest: 6.00

Method:

○ Use OnChange

○ Use OnExit

● Use Timer Delay

Delay: 391

Figure 1: The *OnChange*, *OnExit*, and *TimerDelay* demonstration form.

To see this in action, take a look at the demonstration project DEMO01.DPR (see end of article for download information). From the **Method** radio button group, select **Use OnChange** to set the *Update* method (see Figure 1). Note that the form's fields do not respond to change, and that an ugly screen flicker occurs.

Getting off on the Last *OnExit*

Another alternative is to ignore *OnChange*, and instead tie the calculations to the field's *OnExit* event. This means that no calculations will occur until the user leaves the field. Here we are guaranteed that the user is (at least temporarily) satisfied with the value they entered, and that it won't be changing anytime soon.

The problem is that the user may make changes and *not* exit the field. In this case, the *OnExit* event will never fire because the field was not exited. The form can therefore be left in an inconsistent state where, for example, the values in the graph don't match the values on the rest of the form. You can see this effect by selecting **Use OnExit** in the demonstration application.

What to Do?

There are many possible solutions to the updating problem. The following solution is

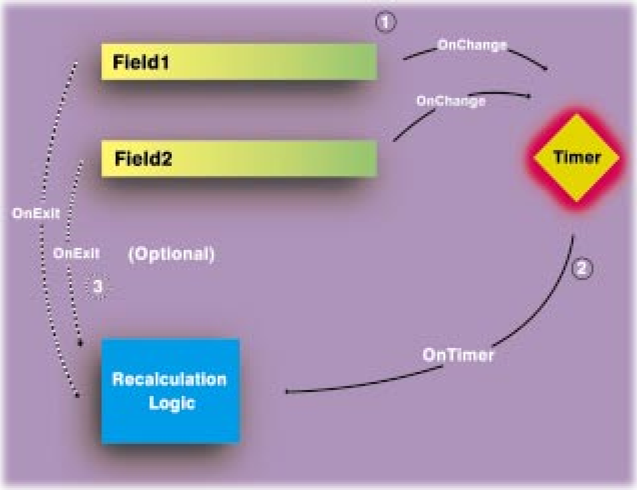


Figure 2: The timer's *OnTimer* event becomes responsible for performing calculations.

Using the Demonstration Program

The demonstration program lets you see how the user interface behaves when using the *OnChange* event, the *OnExit* event, or the "delayed" *OnChange* event discussed in the article. You can use the **Method** radio button to control how the interface behaves. If you are using the *TimerDelay* method, you can use the **Delay** scrollbar to control the length of the delay (in milliseconds) between changes to the **Amount** and **Interest** Edit components and the updating of the graph.

If you look at the code used to update the graph, you'll see I've added a 500-millisecond delay. I added this to the demonstration to make the effects of the *OnChange*, *OnExit*, and *TimerDelay* methods clear. Of course, you wouldn't add a delay to the recalculation routines in a real-world project.

I first used this delayed *OnChange* technique in a project that used VisualTools' FirstImpression charting component. This component provides some wonderful lighting and gradient fill effects that made our graphs more visually appealing. Although these effects took longer to calculate and paint, switching from the standard *OnChange* method to the *TimerDelay* method immediately improved the interface, and made using our product more appealing.

— Gary King

based on the idea that, because we are creating software for people to use, we can make better software by thinking about how people will use it. If someone is interactively entering and modifying data to create some sort of presentation, the flow of their actions will generally be to:

- make some changes
- see what happens
- make some more changes
- see what happens
- etc.

It would be nice if our software could model this sort of activity. And with a little work, it can! To the program, "see what happens" means to run through the calculations and update the presentation of the data. The only tricky part is understanding what the user's view of "make some changes" is. If we were watching someone work, we would see them make modifications, then step back to see how things look. In other words, we need software that can sense a batch of modifications has occurred *and* that the modifications have (temporarily) stopped. What we would like is a sort of delayed *OnChange* event that is armed when a change occurs, but does not go off until the changes have (at least temporarily) stopped.

The Delphi Way

We can do this in Delphi by adding a Timer component to our form and channeling the *OnChange* events of all Edit components through it. The timer's *OnTimer* event then becomes responsible for performing the calculations (see Figure 2).

An example will make this clear. Find a form on which you want to implement the Timer Delayed *OnChange* event. Add a timer (named *Timer1* by default) to the form. Set the Timer component's *Enabled* property to *False* and its *Interval* property to the number of milliseconds you want to elapse between a change to one of the fields and a recalculation. Smaller values will cause the recalculations to occur more quickly (see the sidebar "Using the Demonstration Program" for a discussion of this setting). Set the Timer component's *OnTimer* event to execute a call to the form's recalculation logic. Then attach the following code to the *OnChange* events of each field that's involved in the form's recalculation logic:

```
procedure TMyForm.ResetTimer(Sender: TObject);
begin
  Timer1.Enabled := False;
  Timer1.Enabled := True;
end;
```

This method turns the Timer on if it was off, and resets it to its starting time if it was already running. If a specific field must have more processing in its *OnChange* event, you can use the following code for that field:

```
procedure TMyForm.Field1Change(Sender: TObject);
begin
  ResetTimer(Sender);
  { Other processing for field 1... }
end;
```

Change Happens

To return to our initial example, if a user enters 234.45 into an Edit component, the *OnChange* for that component will still be triggered six times. Each time it is triggered, *Timer1* will be reset to the beginning of its cycle. Only when the user pauses will *Timer1* have a chance to reach the end of its cycle

How Fast Do You Type?

Measuring typing speed sounds relatively straightforward — take the number of keystrokes and divide by the amount of time it took to execute them. However, this approach excludes a few important details:

- A person may type some, stop, then type some more. We don't really want to add the "stopped" time into the total time for our calculation.
- A person may hold a key down and engage the keyboard's automatic repeat action. We don't really want to let someone "cheat" this way.

The TypSpeed sample application solves these problems by only adding keystrokes into the average time when they occur less often than the keyboard repeat rate, and more often than a calculated maximum delay.

Determining the keyboard's repeat speed. Windows provides numerous functions that can be used to query (and sometimes set) the properties of the Windows environment. One familiar API call is the **GetSystemMetrics** function. Among other things, it can tell you how large a standard scrollbar is, or the width of a fixed window frame. A less familiar function is **SystemParametersInfo**:

```
function SystemParametersInfo(A, B: Word; C: Pointer;
                              D: Word): Bool;
```

This API call provides access to a hodgepodge of unrelated system information. As you can tell from the generic types and names of its parameters, the function is not intuitive to use. The use and meaning of each parameter changes, depending on exactly which system property is being queried or set.

We can make the function easier to use, more understandable, and less error-prone by creating a wrapper class that "surrounds" the API call with standard Delphi and Windows types. Because all we need for our current project is the keyboard repeat speed, we'll start with the following simple class (in the DSWinSys unit):

```
TWinSystem = class(TObject)
  private
    function GetKeyboardRepeatSpeed: Word;
  public
    property KeyboardRepeatSpeed: Word
      read GetKeyboardRepeatSpeed;
  end;
```

We can add the remaining **SystemParametersInfo** values as time permits, or as we need them for other projects. To make using this class easy, use the **initialization** section to create a global variable (*WinSystem*). Notice that we use the *AddExitProc* procedure to ensure that what we created is also freed:

```
procedure Cleanup_DSWinSys;
begin
  WinSystem.Free;
end;

initialization

begin
  AddExitProc(Cleanup_DSWinSys);
  WinSystem := TWinSystem.Create;
end;
```

Note: In Delphi 2 and later use the **finalization** section instead of *AddExitProc*. This would look like:

```
initialization
  WinSystem := TWinSystem.Create;

finalization
  WinSystem.Free;
```

Catching the Keystrokes. After we have a method of determining the keyboard repeat speed, the rest of our task is easy. Set the form's *KeyPreview* property to *True*, then attach the following to the *OnKeyDown* event:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key:
Word;
  Shift: TShiftState);
begin
  FThisKey := GetTickCount;
  FElapsed := FThisKey-FLastKey;
  if (FElapsed >= FMinDelay) and
     (FElapsed <= FMaxDelay) then
    begin
      Inc(FTotalStrokes);
      FTotalTime := FTotalTime + FElapsed;
      lblAverageTypingSpeed.Caption :=
        Format('%10.2f', [FTotalTime/FTotalStrokes]);
    end;
  FLastKey := FThisKey;
end;
```

This method records the total number of keystrokes (*FTotalStrokes*), the total time it took to make them (*FTotalTime*), and computes the average. We only count keystrokes whose delay (the difference between *Now* and *FLastKey*) is between *FMinDelay* and *FMaxDelay*. These are set up in the form's *FormCreate* method:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FTotalTime    := 0;
  FLastKey      := GetTickCount;
  FTotalStrokes := 0;
  lblKeyboardRepeatRate.Caption :=
    IntToStr(WinSystem.KeyboardRepeatSpeed);
  lblAverageTypingSpeed.Caption := '';
  Memo1.Lines.Clear;
  FMinDelay := WinSystem.KeyboardRepeatSpeed * 2;
  FMaxDelay := WinSystem.KeyboardRepeatSpeed * 15;
end;
```

After some experimentation, we limited the range to between two- and 15-times the keyboard's own repeat rate. This works well, and seems to provide reasonable answers.

— *Gary King*

and tick, causing an *OnTimer* event. This event will then execute all the recalculation logic for the form.

You can see how this will look (and feel) to your users by selecting the **Use Timer Delay** in the **Method** radio button group in the demonstration application (again, see Figure 1). As long as you keep typing in the **Amount** and **Interest** edit boxes, the graph will not be updated. As soon as you pause, however, the graph will be updated.

Details, Details

It's important to strike a balance between the program constantly recalculating (too small a value) and the program seeming to ignore changes (too large a value). Generally speaking, I have found that a value between 500 and 1000 milliseconds (0.5 to 1 second) works well, and provides a nice responsive feel. You can experiment on the demonstration application by altering the **Delay** scrollbar.

Because different people type at different speeds, you may want to tie the delay to each user's typing speed (see the sidebar "How Fast Do You Type?" on page 41 to learn how to determine your typing speed). This might require adding a setup screen to your application or your program's installation.

Another important point in making programs that seem smart to your users is that no recalculation should occur unless the data has been modified. Delphi makes this easy for an Edit component because the *Modified* property is set to *True* when its value changes. Of course, as King Lear said, "Nothing will come of nothing;" there are three problems with the *Modified* property:

- *Modified* is not reset automatically after your recalculations are complete. You must iterate through each Edit component and reset its *Modified* property to *False*.
- An Edit component doesn't "understand" your application. For example, you and I know that 3.0 is the same as 3.00. However, an Edit component will think the field has been changed and thus set *Modified* to *True* and cause a recalculation.
- Only *TEdit* (and its descendants) has the *Modified* property. If your form is using other user interface controls, you are forced to check for modifications manually.

There are ways around these problems, and they all involve adding code to your application. The demonstration shows one way of resolving the issues by adding properties to the form that mirror the values in the Edit components. Download the code and take a look.

You may also want to provide more feedback to the user to show that the form is in an inconsistent state and that calculations are occurring. The demonstration changes the form's caption during recalculation. If your application has a status bar, you might want to use that instead.

Conclusion

This Timer technique is a great example of how a little extra thinking, and a small amount of code, can produce a better user interface. In this case, we have an interface that will seem more responsive to users: It lets them do their work without getting in the way, and it keeps things updated without the need for explicit recalculation commands.

As the technical environment in which we work becomes more complicated, our users will demand software that helps them *do* their work, not software that *makes* them work. A proper interface is near the heart of this goal. The following references provide a good starting point for learning more about human-computer interaction and user-interface design. Δ

References

Borenstein, Nathaniel. *Programming as if people mattered: friendly programs, software engineering, and other noble delusions* [Princeton University Press, 1991].
Cooper, Alan. *About Face: The Essentials of User Interface Design* [IDG Books, 1995].
Norman, Donald. *The Design of Everyday Things* [Doubleday, 1990].
Norman, Donald. *Things that Make us Smart* [Addison-Wesley, 1993].
Tognazzini, Bruce. *Tog on Interface* [Addison-Wesley, 1992].
Tognazzini, Bruce. *Tog on Software Design* [Addison-Wesley, 1996].

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\APR\DI9704GK.

Gary King is the principal of DesignSystems (http://www.oz.net/dsig), makers of Interface Gizmos and other fine products for Delphi. He can be reached at gking@oz.net.

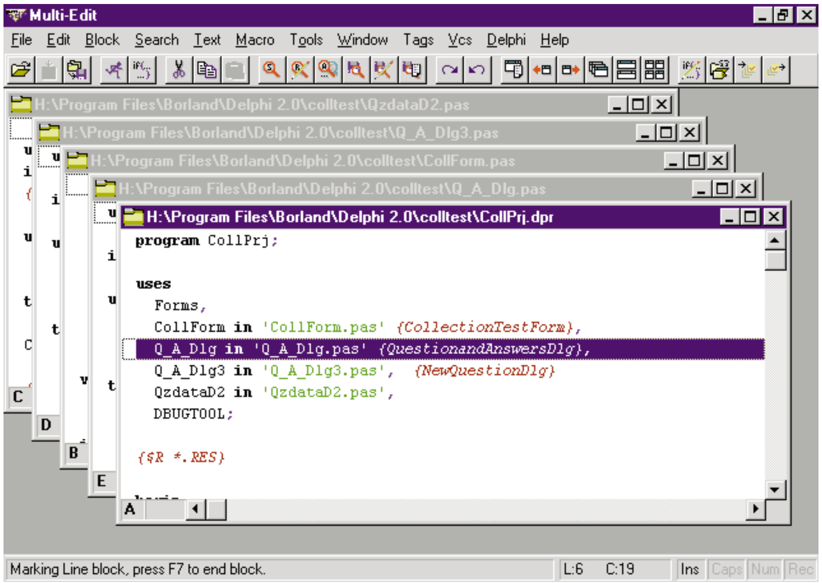
By Alan C. Moore, Ph.D.

Multi-Edit for Windows

An Editor for the Power Programmer

For many of us, Delphi's code editor is more than adequate. It includes all the standard features you would expect: block selection, cut and paste, search and replace, bookmarks, and so on. But what if these capabilities are not enough? What if you need, or want, more control over the code-editing process? Then it's time to consider one of the specialized code editors, of which the best known is Multi-Edit for Windows by American Cybernetics.

This editor is easy to install, and you can implement Delphi support before or after installation. When you enable Delphi support, you inform Multi-Edit of the location of the Delphi BIN directory. This allows you to compile Delphi programs directly from Multi-Edit. A new menu item, **Delphi**, is added to Multi-Edit's already feature-rich menu system, and Multi-Edit is automatically added to Delphi 1's **Tools** menu (I had to manually add it to Delphi 2's **Tools** menu). In Delphi 2, the hotkey you select automatically loads Multi-Edit, and toggles between Multi-Edit and Delphi's IDE. The available menu system and the various hotkeys are completely configurable.


Multi-Edit
File Edit Block Search Text Macro Tools Window Tags Vcs Delphi Help
H:\Program Files\Borland\Delphi 2.0\colltest\QzdataD2.pas
H:\Program Files\Borland\Delphi 2.0\colltest\Q_A_Dlg3.pas
H:\Program Files\Borland\Delphi 2.0\colltest\CollForm.pas
H:\Program Files\Borland\Delphi 2.0\colltest\Q_A_Dlg.pas
H:\Program Files\Borland\Delphi 2.0\colltest\CollPrj.dpr
C
D
B
E
A

```
program CollPrj;

uses
  Forms,
  CollForm in 'CollForm.pas' {CollectionTestForm},
  Q_A_Dlg in 'Q_A_Dlg.pas' {QuestionandAnswersDlg},
  Q_A_Dlg3 in 'Q_A_Dlg3.pas',  {NewQuestionDlg}
  QzdataD2 in 'QzdataD2.pas',
  DBUGTOOL;

{$R *.RES}
```

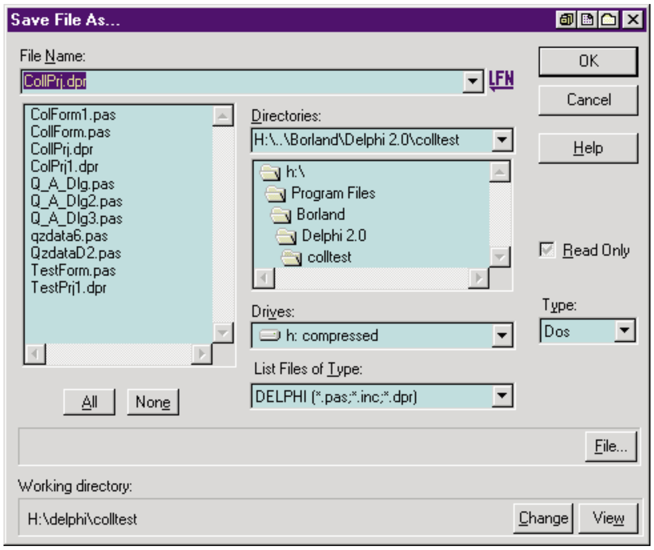
Marking Line block, press F7 to end block. L:6 C:19 Ins Caps Num Rec

Figure 1: Multi-Edit's multiple document interface.

Unlike Delphi, but similar to Borland Pascal, the main window uses a multiple document interface (see Figure 1). While Multi-Edit for Windows is a 16-bit application, it does have a number of Windows 95-compatible features, including support for long filenames, "Explorer-style" file prompts, and background compiling.

Multi-Language Support/Enhanced Capabilities

Multi-Edit supports many other programming languages, including Assembler, BASIC, C, dBASE, HTML, and Java. Enabling support for these languages/-
compilers is less straightforward than for Delphi. Keywords and other language-specific details are built into an internal database you can access and modify. Built-in language-specific templates, which expand keywords into common structures, are also available. For example, in Delphi, you can type `begin`, hit the Expand Template speedbutton, or Alt F9, and a **begin..end** block will be created with the editing cursor placed in the body of the block. Other common Pascal/-Delphi structures include **case**, **repeat**, **record**, **for**, **object**, **if..else**, **function**, **procedure**, **try..finally**, and **try..except**. If the built-in templates are not sufficient, you can define your own. The application includes two special add-on packages to enable seamless integration with either Borland's C/C++ or Delphi's IDE. When you install these, an extra menu item allows you to compile, build, or check syntax for an application in that particular language.



Save File As...

File Name: CollPrj.dpr

ColForm1.pas
CollForm.pas
CollPrj.dpr
ColPrj1.dpr
Q_A_Dlg.pas
Q_A_Dlg2.pas
Q_A_Dlg3.pas
qzdata6.pas
QzdataD2.pas
TestForm.pas
TestPrj1.dpr

Directories: H:\..\Borland\Delphi 2.0\colltest

h:\
Program Files
Borland
Delphi 2.0
colltest

☑ Read Only

Drives: h: compressed

Type: Dos

List Files of Type: DELPHI (*.pas;*.inc;*.dpr)

Working directory: H:\delphi\colltest

Figure 2: The Save File As dialog box.

The main menu bar includes the familiar choices (**File**, **Edit**, etc.), as well as **Block**, **Search**, **Text**, **Macro**, **Tags**, and **Vcs** (Version Control System.) Many of the familiar menu items have enhanced capabilities when compared with other editors (such as Delphi's IDE.) However, there are also some unfortunate limitations. The first half of the manual discusses the menu and sub-menu items in detail. Let's take a look at some of the more unusual features.

Managing Files: Editing, Marking, and Searching for Text

The **File** menu includes all the standard choices, and a few new ones. **Information** provides statistics on the currently active file, including its status in memory. **Merge** activates a dialog box that allows you to integrate any file into the current file, starting at the current cursor position. Finally, the **Session Manager** allows you to organize different projects into sessions, saving the complete editor status for each session. The latter includes all files loaded, history lists, and global variables. The standard File dialog boxes (Open, Save, etc.) contain numerous welcome additions as shown in the Save File As dialog box (see Figure 2). For example, clicking the **File** button opens a list of File Manager options that allow you to delete, move, or rename files, among other choices. The **All** button selects all the files in the current folder (directory), while **None** deselects them all.

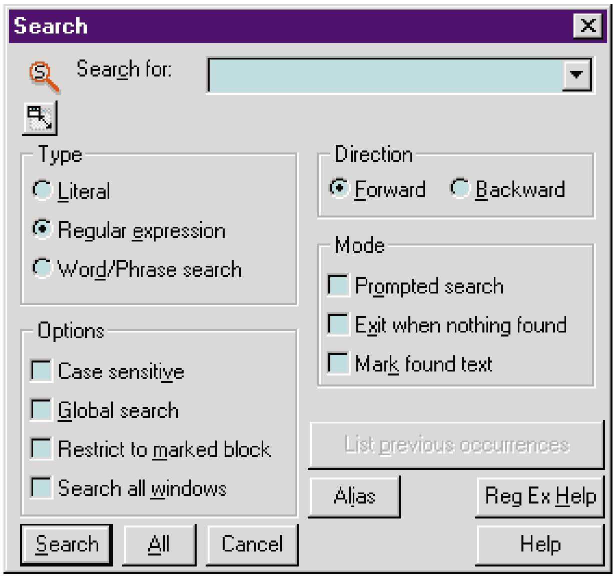
I did encounter a problem with the Open File dialog box. When I browsed different directories beyond the working directory and double-clicked on a file, that file did not open. Instead, a new file with the same name was opened in the working directory. If you want to open a file in a different directory, you must first click the **Change** button. I found this disconcerting.

The **Edit** menu also has some new and enhanced choices. Selecting **Undo** lets you cancel up to 65,000 changes. You can **Redo** as many changes as were undone, provided you don't make other changes to the file. Innovative new features include appending a marked block to text already in the cut-and-paste buffer, showing the contents of that buffer, and repeating a particular keystroke a number of times.

The **Block** menu is even more feature-rich. The obvious choices of indenting, unindenting, copying, moving, and deleting a selected block are present, along with some nice additions, including saving to file, copying a block from another window, and moving it to another window. There are three block-selecting modes, providing tremendous flexibility in editing source files: by lines of text, by columns of text, and by stream (beginning and ending anywhere within particular lines). A very helpful option is the persistent block. With it you can mark a particular block, move the cursor somewhere else, and move or copy the still-selected (persistent) block to that location.

The **Search** menu is also powerful. Using the Search dialog box (see Figure 3) you can search for individual words, phrases, etc. using regular expressions. With other choices in this menu you can set up to 10 bookmarks or random access marks (which, unlike Delphi, remain active the next time you open Multi-Edit), highlight global expressions (the same string in *all the files* open in the current session), or list files from the last file search. Another powerful feature is the Multiple File Search and Replace dialog box (see Figure 4). As you can see, using the same kinds of expressions found in the Search dialog box provides tremendous flexibility to search for strings or complex patterns in multiple files. After a search is completed, you can open any file containing the search string by double-clicking on any of its entries in the resulting window (see Figure 5). After you close this window, you can bring it up again at any time by selecting **List Files from Last Search**, or by hitting Ctrl G. I found this very useful.

The **Text** menu provides a host of text viewing and formatting options. For example, you can view any source or text



Search

Search for:

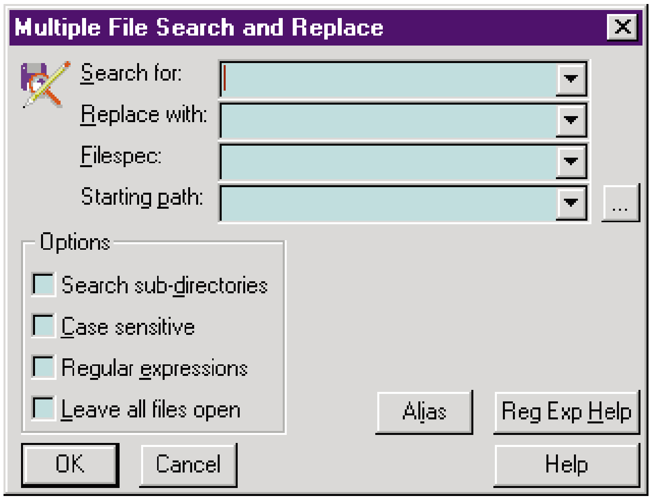
Type
○ Literal
● Regular expression
○ Word/Phrase search

Direction
● Forward ○ Backward

Mode
□ Prompted search
□ Exit when nothing found
□ Mark found text

Options
□ Case sensitive
□ Global search
□ Restrict to marked block
□ Search all windows

Figure 3: The Search dialog box.



Multiple File Search and Replace

Search for:

Replace with:

Filespec:

Starting path:

Options

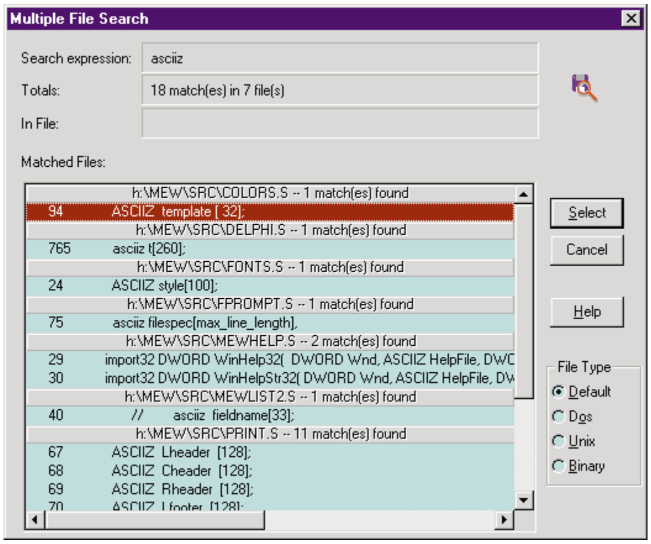
□ Search sub-directories

□ Case sensitive

□ Regular expressions

□ Leave all files open

Figure 4: The Multiple File Search and Replace dialog box.



Multiple File Search

Search expression: asciiz

Totals: 18 match(es) in 7 file(s)

In File:

Matched Files:

```
            h:\MEW\SRC\COLORS.S -- 1 match(es) found
 94      ASCIIZ  template [ 32];
            h:\MEW\SRC\DELPHI.S -- 1 match(es) found
 765     asciiz t[260];
            h:\MEW\SRC\FONTS.S -- 1 match(es) found
 24      ASCIIZ style[100];
            h:\MEW\SRC\FPROMPT.S -- 1 match(es) found
 75      asciiz filespec[max_line_length],
            h:\MEW\SRC\MEWHELP.S -- 2 match(es) found
 29     import32 DWORD WinHelp32(  DWORD Wnd, ASCIIZ HelpFile, DWC
 30     import32 DWORD WinHelpStr32( DWORD Wnd, ASCIIZ HelpFile, DW
            h:\MEW\SRC\MEWLIST2.S -- 1 match(es) found
 40          //       asciiz  fieldname[33];
            h:\MEW\SRC\PRINT.S -- 11 match(es) found
 67      ASCIIZ  Lheader  [128];
 68      ASCIIZ  Cheader  [128];
 69      ASCIIZ  Rheader  [128];
 70      ASCIIZ  Lfooter  [128];
```

File Type

● Default

○ Dos

○ Unix

○ Binary

Figure 5: This window, which shows the results of a multiple file search, allows you to open any of the files by simply double-clicking its entry.

file in hexadecimal mode, set layout options such as word-wrap and indenting, sort text in a file, center a line, perform various operations on a paragraph (reformat, justify, unjustify), set page breaks, and so on. One particularly innovative option allows you to compare two files and create a new file containing their differences. You can also move through files and view the previous or next difference.

Tools, Tags, Version Control, and More

The Tools menu includes a number of useful items, including a spellchecker, a template builder and editor (for creating new templates) as shown in Figure 6, an ASCII reference table, a notebook, and a calculator. With other options you can compile code, manage background tasks, find the next compiler error, comment or uncomment a source file, install or manage add-on packages (authors of add-on packages can download needed documentation from American Cybernetics' Web site), and customize many of Multi-Edit's features.

The Tags menu allows you to create a Multi-Tags database of source file functions, procedures, classes, and types so you can browse the file in hypertext-like manner. After the file has been scanned, you can select **Browse Current File** to access a dialog box listing all the items (procedures, functions, etc.). This allows you to quickly jump from one to another. When you combine this feature with the bookmarks discussed earlier, moving around — even in a large source file — becomes a snap.

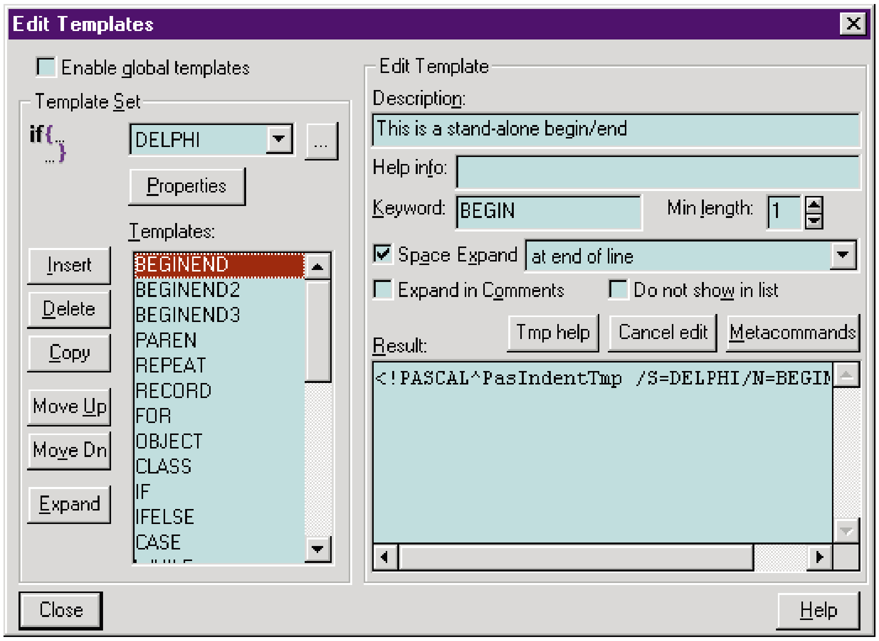
The Vcs menu provides useful editing and archiving extensions to the version control system you have installed. You can lock or unlock files, maintain a log of files stored in the VCS archive, and so on. The Help menu contains many special features, including a **Quick Overview** that provides an introduction to using the product, information about technical support, and FAQs (Frequently Asked Questions). I was particularly pleased with the innovative method for reporting bugs: a dialog box records data about your computer hardware and software configuration, provides a straightforward manner for you to describe the bug, then e-mails the information to American Cybernetics.

Macros also have their own menu. You can run, load, and list macros, globals, and keystroke macros. If the macro you want to load or run doesn't share the exact name as its file, you must place the filename before it as in `filename~macro-name`. The Loaded Macros dialog box (see Figure 7) is disappointing in some respects. You can't double-click on a macro name and execute it, nor can you highlight and save the name to the clipboard. Having addressed the **Macro** menu, let's discuss macros themselves.

Multi-Edit's Macro Language

The Macro Language built into Multi-Edit is powerful and difficult. With a syntax similar to C, Multi-Edit uses its Macro Language to implement many of its internal operations, including language indenting. In fact, the wizard that sets the Delphi options during installation is a macro called DELPHIWIZARD. Each macro exists in a source file with a .s extension and a compiled macro file with a .mac extension. Those macros called by other macros require a prototype, which exist in a text file with a .sh extension. You can edit a .s or .sh file with any text editor.

The Macro Language is sufficiently complex that the entire second half of the Multi-Edit manual — some 100 pages — is devoted to explaining it. It resembles a full-fledged programming language, with labels, expressions, typed variables, branching, functions with parameters, and so forth. There are many pre-defined functions and variables such as BLOCK_BEGIN, COPY_BLOCK, cursor functions, file operation functions, and machine-level interface functions. You can build sophisticated dialog boxes with the language, but there's a catch: You need to study some of the source files such as DLG_Exam to learn the intricate steps involved. There is even low-level access to the Windows API. These features provide tremendous power and flexibility in creating your own macros.



□ Enable global templates

Template Set: DELPHI

Templates:
BEGINEND
BEGINEND2
BEGINEND3
PAREN
REPEAT
RECORD
FOR
OBJECT
CLASS
IF
IFELSE
CASE

Edit Template

Description: This is a stand-alone begin/end

Help info:

Keyword: BEGIN Min length: 1

☑ Space Expand at end of line

□ Expand in Comments □ Do not show in list

Result:

```
<!PASCAL^PasIndentTmp /S=DELPHI/N=BEGIN
```

Figure 6: The Edit Templates dialog box.

As I have already hinted, however, there is a down side: Working with the Macro Language is hardly straightforward or intuitive. For example, from the **Macro** menu you can view all the available macros is a listbox, or run a specific macro. You must, however, type in the name of the macro you want to run. Most macros require parameters. If you don't enter the proper parameters, you'll get an error message. Unfortunately, the only way you can get specific information on parameters required for many macros is to examine the source file where that macro is defined. The process is similar to learning some of the undocumented or poorly documented capabilities of Delphi (e.g. creating experts). Unlike Delphi, however, there is little online Help for pre-defined macros. Such an addition would be a major enhancement, and make this product much more useful. Finally, I found the documentation lacking in several respects. Some features, such as the use of ASCIIZ string arrays in structures, is not adequately explained; many other features of the Macro Language would benefit from expanded examples. However, we *can* create useful and powerful macros if we are willing to put forth the effort required.

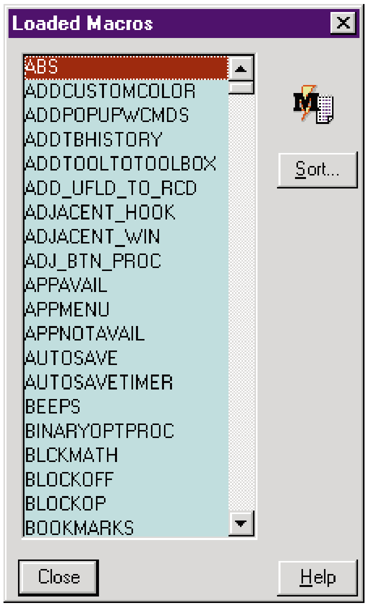
My First Macro

Like many of you (I suspect), I don't like unnecessary work. In Delphi, when I formulate the structure for a new class, I resent having to copy every function and procedure heading, adding the class prefix, and copying everything (including the **begin..end** block) to the **implementation** section. So for my first macro, I created a structure that would automate this process.

Specifically, after scanning the methods (functions and procedures) of the class, the macro would attach the class name before each method and expand each method with **begin..end** blocks in the implementation section. The basic algorithm is as follows:

1) If not on line with class definition, exit.
2) Get class name; save in global variable.
3) Get next procedure or function name.
4) Move to the **implementation** section after **uses** clause (if present).
5) Create method headers with class name prefix.
6) Expand each function or procedure with **begin..end** block.
7) Repeat steps 2-6 until there are no more methods.
8) Exit.

The macro is shown in Figure 8. The test source file is shown in Figure 9, and the resulting source file (after running the macro) is shown in Figure 10. This is not a particularly simple macro, even though it doesn't involve the creation of any dialog boxes. If you *do* want to use dialog boxes in your macros, you should study the files dialog.s and pascal.s. The former is the basic source for dialog box creation; the latter provides excellent examples of various dialog boxes with different controls.



ABS
ADDCUSTOMCOLOR
ADDPOPUPWCMDS
ADDTBHISTORY
ADDTOOLTOTOOLBOX
ADD_UFLD_TO_RCD
ADJACENT_HOOK
ADJACENT_WIN
ADJ_BTN_PROC
APPAVAIL
APPMENU
APPNOTAVAIL
AUTOSAVE
AUTOSAVETIMER
BEEPS
BINARYOPTPROC
BLCKMATH
BLOCKOFF
BLOCKOP
BOOKMARKS

Figure 7: The Loaded Macros dialog box is disappointing: You can't highlight and copy macro names, nor can you double-click and execute one.

Unfortunately, there are no built-in debugging tools for writing macros, so you must roll your own. Here's what I did. During the debugging phase, wherever I assigned a value to a variable, I inserted two lines — one showing the value assigned on Multi-Edit's status line, the other a *KeyPress* function so that execution would be suspended until I had a chance to read the result:

```
ClassName = GET_WORD('');
make_message('ClassName = ' + ClassName);
Read_Key;
WORD_RIGHT;
```

```
macro_file BldClass;
struct MethodArray {str MethodStrs[100];}
void BldClass( ) trans2 {
  str ThisWord;      // Current word read
  str ClassName;     // To append to each function/procedure
  str MethodHeader;  // Current method with all parameters
  WORD_DELIMITS = (' ');
  ClassName = GET_WORD(' '); WORD_RIGHT;
  if (! AT_EOL) {WORD_RIGHT;}
  ThisWord = GET_WORD(' (');
  if (CAPS(ThisWord) != 'CLASS') {
    make_message('This is not a Class Definition');
    Read_Key;
    goto Finished;
  }
  // Find Implementation Section
  Down;
  Set_Mark(1);  // Come here after finding implementation
  FIRST_WORD; ThisWord = GET_WORD(' ;');
  while ( CAPS(ThisWord)!=('IMPLEMENTATION')) {
    Down; FIRST_WORD; ThisWord = GET_WORD(' ;');}
  ThisWord = GET_WORD(' ;');
  if ( CAPS(ThisWord)==('USES')) {
    REPEAT_LOOP:
    FORWARD_TILL(';');
    Down; FIRST_WORD;
    if (AT_EOL) {goto REPEAT_LOOP;}
  }
  Set_Mark(2);  // After implementation, start writing here
  Get_Mark(1);  // Go back to method body
  // Now scan for procedure and function names
  FIRST_WORD; ThisWord = GET_WORD(' ;');
  while (CAPS(ThisWord) != 'END' ) {
    if  ((CAPS(ThisWord)==('PROCEDURE')) ||
        (CAPS(ThisWord)==('FUNCTION'))) {
      GetAnotherWord: WORD_RIGHT;
      MethodHeader = GET_WORD(';');
      MethodHeader = ThisWord + ' ' + ClassName +
                     '.' + MethodHeader + ';';
      Down; First_Word;
      Set_Mark(1); // Next time come back here and continue
      // We have header; expand it in implementation section
      Insert_Mode = 1; Get_Mark(2); CR;
      Text(MethodHeader); CR; Indent;
      Text('Begin'); CR; Indent;
      Text('{Statements}'); CR; Undent;
      Text('End;'); CR; Undent; CR;
      Set_Mark(2); // Next time start writing here
      Insert_Mode = 0;
      Get_Mark(1); // Read Next Line of method
      First_Word; ThisWord = GET_WORD(' ;');
    }
  }
  Finished:
  Insert_Mode = 1;           // Restore to insert mode
  RM('Text^ClearRandomMark'); // Clear random marks
}
```

Figure 8: Listing of BldClass macro.

```
unit Test1;

interface
TestClass = class
  procedure Procedure1;
  procedure Procedure2(param1, param2, param3);
  function  function1: integer;
  procedure Procedure3(param1);
  function function2: Boolean;
end;  { TestClass }

implementation

end.
```

Figure 9: Source file before running the macro.

```
unit Test1a;

interface
TestClass = class
  procedure Procedure1;
  procedure Procedure2(param1, param2, param3);
  function  Function1: integer;
  procedure Procedure3(param1);
  function Function2: Boolean;
end;  { TestClass }

implementation

procedure TestClass.Procedure1;
  begin
    { Statements }
  end;

procedure TestClass.Procedure2(param1, param2, param3);
  begin
    { Statements }
  end;

function TestClass.Function1: integer;
  begin
    { Statements }
  end;

procedure TestClass.Procedure3(param1);
  begin
    { Statements }
  end;

function TestClass.Function2: Boolean;
  begin
    { Statements }
  end;

end.
```

Figure 10: Source file after running the macro.

There are several aspects of the macro source file I would like to address. Most of the macro statements I used are described in the chapter on *System Functions and Variables*, which contains many functions for scanning and writing to source files, file operations, and low-level system functions. First, the macro function itself is declared as *Void*, because it does not return any value. After the declaration, it is customary to list all the variables, with each preceded by its type. As in C++ or Object Pascal 2.0, use a double slash to begin comments. Functions like WORD_DELIMITS, GET_WORD, and WORD_RIGHT set and use strings of word-delimiter characters that give you a great deal of control over selecting words or tokens. Functions like FORWARD_TILL, DOWN, FIRST_WORD, SET_MARK, and GET_MARK allow you to move the cursor around within a source file. And the functions CR and TEXT allow you to write new code in the source file. The line:

```
RM('Text^ClearRandomMark');
```

runs the macro, ClearRandomMark, found in the file Text.Mac. You can call any of the other numerous macros using the same syntax.

Conclusion

Multi-Edit for Windows is clearly a powerful and beneficial tool for Delphi programmers, especially those who spend a good deal of time writing code. The more familiar I became with the product, the less time I spent using Delphi's editor. Now the only time I return to Delphi is when I need to trace through a project and debug it. However, particularly because of the documentation and the scarcity of good examples in the manual, I found working with the Macro Language frustrating. On many occasions, developing a macro caused Windows to freeze. Sometimes I could resolve the problem by simply hitting Ctrl Break; other times I was forced to reboot.

Still, I found the product has grown on me, and as I became familiar with its features, I forgot about the initial difficulties. I was particularly pleased with the new macro I created — in the future, when I create new support classes, it will save me immeasurable keystrokes. I can imagine a number of other useful macros I may write, including one to automate making changes to the **uses** clauses in units to transfer them from Delphi 1 to Delphi 2. I was particularly impressed with the capabilities of many of the feature-rich dialog boxes that provide options I have not seen elsewhere. When I visited American Cybernetics' Web site I found updated files and technical papers for all their products, including Multi-Edit for DOS and Multi-Edit for Windows. You can also download a demonstration version of the product. In summary, I found Multi-Edit for Windows to be a wonderful programmer's editor that should be a welcome addition to many developers' arsenal of tools. ∆

INFORMANT FACT FILE

Multi-Edit for Windows is a powerful programmer's editor that includes built-in Delphi support and support for many other programming languages, including Assembler, BASIC, C, dBASE, HTML, and Java. Its menus include many standard and enhanced options for editing, searching multiple files, and managing projects. Macros and templates help to automate coding and working with files. Tools such as a spellchecker, a notebook, and a calculator are welcome additions.

American Cybernetics
1830 W. University Drive, Ste. 112,
Tempe, AZ 85281

Phone: (602) 968-1945
Fax: (602) 966-1654
E-Mail: tech@amcyber.com
CompuServe: GO CYBERNET
Web Site: http://www.amcyber.com
Price: US$199

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.

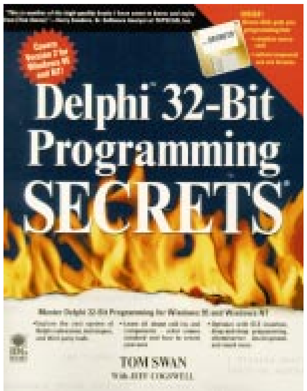
Delphi 32-Bit Programming SECRETS

The first thing that struck me about *Delphi 32-Bit Programming SECRETS* by Tom Swan and Jeff Cogswell is the absence of figures. Normally, a computer book weighing this much has at least a pound of screen shots and diagrams — here, there are two. Chapter 11 includes a good illustration of window-to-viewport mapping, and chapter 7 includes a pointless hierarchy diagram of an OLE storage unit.

The ink is not missed; this book is loaded with information and secrets. Of course you knew that — it's in the title. I had doubts at first; the secrets weren't in the first few chapters. I wondered if I'd graduated to Delphi guru and my diploma was lost in the mail. Fortunately, the hints get better as the book progresses. Many books promise to deliver secrets; it's refreshing when one actually does. [One that definitely delivers is Ray Lischner's *Secrets of Delphi 2* from Waite Group Press, reviewed in the February 1997 *DI*, ed.]

Chapter 4 covers 32-bit programming for Windows, and does so with an unusual topic: a component that encapsulates the creation of a control panel applet. I found the material fresh, and while the topic may be a bit esoteric, the techniques are quite interesting. As with all the code in the book, this component is on the accompanying diskette. The installation of the components is simplified, but I would add the usual caveat about making a copy of CMPLIB32.DCL. Swan explains how to manually create the file in event of an emergency, but I've found it's far simpler to keep a current copy around.

Throughout most of the book, Swan digs deep into the muck of Windows. Covering OLE, however, he treads lightly on the surface. Deciding the subject is just too complex to tackle, he lets Delphi handle the gory details. This is probably a wise course, as OLE is foreboding to most programmers. However, not wanting to disappoint those searching for secrets, Swan includes an excellent tip for debugging OLE, as well as an Object Pascal trick that could be used in any situation.



My favorite chapter is a long, in-depth discussion of printing. Although the topic is dry, the authors make up for it with plenty of good information. As with the other components and code, the print preview component Swan and Cogswell develop in this chapter is given, without restriction, to the readers. In addi-

"Delphi 32-Bit Programming SECRETS" continued on page 50

Hardcore *Delphi Database Development*

With its definitive title, *Delphi Database Development* will be approached by programmers who have "A-to-Z" expectations. Readers can be assured that the A-to-Z promise will be met, but in a different manner than other books. The dense text, lack of illustrations, and absence of the ubiquitous "Tropical Fish" database tell the reader this is not a tutorial guide for neophyte Delphi database programmers.

In contrast to books that offer instruction on using VCL components to build database programs without writing code, *Delphi Database Development* provides in-depth information for experienced Delphi professionals who know many lines of code are often necessary to produce commercial-grade applications. The intended audience is professional programmers with experience in Object Pascal, the Delphi object model, and database-specific program development.

Delphi Database Development begins with a 12-page chapter on database development. Descriptions of the VCL components, the Borland Database Engine (BDE), and the single relation example program are the only offerings to novice programmers. The remainder of the book documents Delphi's database tools in low-level detail.

DELPHI DATABASE DEVELOPMENT
INCLUDES INFORMATION ON DELPHI AND DELPHI 2

The second, third, and fourth chapters cover the Data Access, Visual Data, and *TField* controls, respectively. The reader must have experience with these tools to successfully apply this reference material; the provided examples don't place the function code within a project's context.

"Hardcore *Delphi Database Development*" continued on page 50

Hardcore *Delphi Database Development* (cont.)

Early chapters aren't categorically or functionally indexed. So, the programmer must know which concept to research before delving in. The book offers a consistent, descriptive pattern throughout. For example, a component is presented with a description of its application, followed by tables listing its properties, methods, and events. The programmer focusing on a single control will find this reference faster than navigating the myriad windows in the Help system.

An alphabetical list following the tables then details each scheduled item. The book was disappointing in this respect — the descriptions offer little more than Delphi's online documentation. For instance, the examples often contain only one line of code showing the Object Pascal syntax underlying the component's property, but not the context in which it's used. Without the surrounding code, the developer is left to create example/test code.

The BDE Function Reference is the most useful section, providing complete documentation for the BDE API. The authors provide brief paragraphs to introduce the BDE and its use with Delphi. This section presents the functions in alphabetical sequence (a categorical reference to these functions is presented in Appendix C, which may have improved the workability of this reference chapter). The entries for each API function are much more useful and comprehensible to the developer than the earlier component entries. Additionally, the code snippets used to present the functions are greatly expanded. For example, an API call is shown surrounded by code explaining the function's use.

Experience developing database programs in any language is required to make full use of this reference guide. The programmer must understand the specific task to accomplish before consulting this book.

The last 200 pages are tightly focused appendices, providing explicit information about error return codes, BDE data structures, converting Xbase to Delphi code, database file formats, alternatives to the BDE, and InterBase SQL. This dense collection of information is useful, and after divining the methodology behind the presentation, programmers can quickly locate the required answers. *Delphi Database Development* bridges both releases of Delphi with those functions applicable only to the 32-bit version.

An accompanying CD-ROM in a computer text is almost obligatory in the current sales market; this book follows this trend. However, the programmer must ponder the usefulness of the CD-ROM that accompanies *Delphi Database Development*. It presents few complete programs, and loading the CD-ROM, copying the code, compiling, and running it takes longer than manually entering the code. The CD-ROM has a smattering of demonstration copies of commercial software and publishing ventures. It also includes a set of reasonably valuable programs submitted by a third-party developer. The BDE demonstration programs provide examples of several BDE API calls, as well as several useful tools when compiled.

Delphi Database Development is an important addition to the library of any developer building commercial database applications in Delphi. This dense, highly technical reference provides valuable information not readily available in the general Delphi press.

The book makes no pretensions about teaching the neophyte Delphi developer the basics of database programming — other fine books can fill this need. A working programmer needs quickly accessible information, and this book provides it. If you purchase this book, remember to also buy a good highlighter pen and a package of Post-It notes for marking pages that you'll repeatedly reference.

— *Warren Rachele*

Delphi Database Development by Ted Blue, John Kaster, Greg Lief, and Loren Scott. M&T Books, 115 West St., New York, NY 10011, (212) 886-9222.

ISBN: 1-55851-469-4
Price: US$44.95
(968 pages, CD-ROM)

Delphi 32-Bit Programming SECRETS (cont.)

tion to that excellent component, Swan explains several of Windows' poorly written API calls. I don't know if this knowledge can be classified as secret, but I doubt I'd have figured them out.

Finally, there's chapter 13 — unlucky 13. While the book is not intended to be a comprehensive resource, Swan claims this chapter will pull together all information on files and streams. They do a workmanlike job here, but I want more. I want the secrets of the universe. I want fat-free that *really* tastes good. At least I want to stream my components to and from disk. I've been searching for what seems an eternity for a definitive guide to streams and filer objects. I had high hopes for this chapter; it's fine, but not comprehensive.

While my overall impression of this book is good, I do have a few problems with it. Two of the chapters — a long one on memory management and another on the IDE — are difficult to follow. Perhaps they were rushed to press; that would explain the meandering prose. Also, maddeningly, there are no chapter summaries. A brief terminating paragraph would prevent a tired programmer from trying to peel apart pages when none are stuck. More than once Swan ends a chapter that abruptly.

Despite its problems, I think *Delphi 32-Bit Programming SECRETS* will help most Delphi programmers, and I definitely recommend it.

— *Richard A. Porter*

Delphi 32-Bit Programming SECRETS by Tom Swan with Jeff Cogswell. IDG Books, 919 E. Hillsdale Blvd., Foster City, CA 94404, (800) 434-2086.

ISBN: 1-56884-690-8
Price: US$44.99
(738 pages, Disk)

Support! Now More Than Ever

If you develop software, chances are you or someone in your company is responsible for supporting the applications you build. But how much importance do you give the support itself? Before you take it too lightly, keep this in mind: Unless the software is operable by the targeted user, it quickly becomes "shelfware." I learned this lesson recently trying to install and configure a Web software package on a Microsoft IIS Web server. While I discovered it can be done, the experience was not without a lot of avoidable pain.

Dot your i's, cross your t's. Once I downloaded the software from the vendor's Web site and prepared for set-up, I became confused: I didn't know which installation instructions I should follow. The step-by-step instructions on the Web site were noticeably different than the INSTALL.TXT file included with the download. After I determined the correct set, I still had to wade through a series of errors and inconsistencies in the documentation. To top it off, after I had spent several hours working with a junior technical support engineer to track down a particular bug, I was disheartened to later talk to a senior engineer who had full knowledge of the problem. Such information was neither disseminated internally nor to customers. Thus, gleaning our first applicable principle from this experience: *Ensure documentation is consistent and routinely updated.* Known bugs should be disseminated, particularly if there are workarounds. The Internet — or if applicable, corporate intranets — can serve as an excellent means of making such information accessible to your user base.

Home brew if you can. The software package uses Perl scripts to provide its run-time application environment. Because Perl works with most Web servers, it seems an obvious candidate for vendors not tied to a particular server. The downside is that while Perl is ubiquitous in the UNIX world, getting it to run with NT Web servers (particularly Microsoft IIS) isn't necessarily straightforward. For example, the documentation for the NT version of Perl I downloaded differed from the IIS docs. They did have one thing in common, however: They were both wrong! After considerable trial and error, I was finally able to stumble across a technical note on the Microsoft Web site that discussed how to correctly set the Registry entries. The problem: I was forced to fix the problem myself, because the technical support people I talked with had little experience using Perl on the NT platform. Our second principle therefore is to *avoid relying on third-party software.* In situations where that is not possible, remember this: *Any problem with that software is also your problem.*

Time is money. During much of this ordeal I was able to keep my patience, but I eventually gave up and began exploring alternative options. Soon the thought of starting over with another solution seemed less of a task than completing the one I already spent countless hours on. This leads to our third principle: *Software that is easy to install, configure, and use gives you an advantage over your competitors.* (A corollary is also true: *If your software is difficult to use, a user will often react by switching to the competition, even if it also proves difficult to use.*) Perhaps this is obvious for "end-user" software, but it is also true for developer-oriented tools. I think many software developers and companies miss this point. Remember the principle "time is money" also holds true for your targeted user. Perhaps we can restate this as "the less time required to learn how to operate your software will give you more money."

While I have singled out one software vendor, it is likely each of you have identical stories to tell. Think about these experiences as you and your company support software. Great programming techniques, sound application design, and unmatched performance matter little if your software is poorly supported. ∆

— Richard Wagner

Richard Wagner is Chief Technology Officer of Acadia Software in the Boston, MA area, and is Contributing Editor to Delphi Informant. *He welcomes your comments at rwagner@acadians.com.*