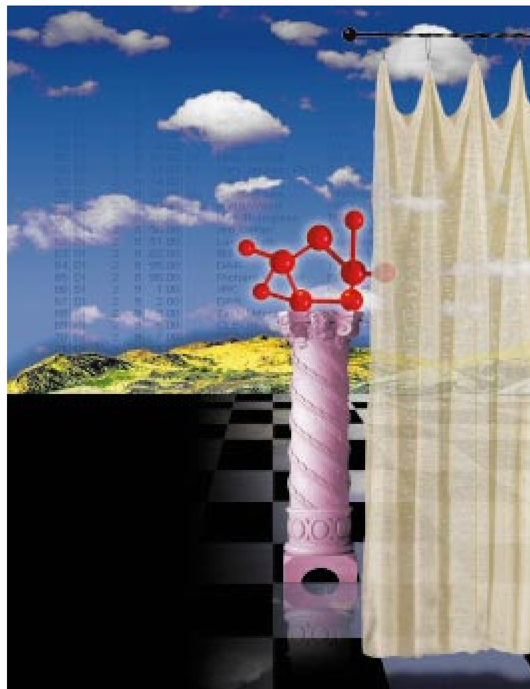
June 1997, Volume 3, Number 6

InterBase

The RDBMS behind the Curtain



Cover Art By: *Tom McKeith*

ON THE COVER

6 InterBase Stored Procedures — Jerry Coffey
Although the stored procedure is a cornerstone of client/server programming, the InterBase variety is woefully underdocumented — especially regarding the Delphi connection.

11 InterBase Triggers and Generators — Bill Todd
You can't write client/server applications without triggers. Okay, you can; but triggers may be the most powerful tool in a database server. Learn how they can ease your burden, even in applications of modest complexity.

15 InterBase Event Alerters — Alexander Le and Donna Burbank
You know the drill: The Delphi client sends commands, and the database mutely accepts them. But InterBase event alerters reverse this relationship. Here's how to create a robust database application with the ability to *talk back*.

FEATURES

19 Informant Spotlight
Interfacing with COM — Jim Scammahorn
The Component Object Model (COM) allows objects to be distributed across multiple programs, or even multiple machines, while still appearing as a single application. Here's how to create such a united front with Delphi 3.

24 First Look
New Visuals — Robert Vivrette
Delphi 3 is a reality, and many are eager to upgrade. For those who haven't had a chance to "test drive," here's a quick look at the Delphi 3 VCL's new controls.

28 In Development
Deployment: Part II — Bill Todd
InstallShield Express Professional offers more power and flexibility — once you iron out the details. This month, Mr Todd describes how to use extensions to resolve the BDE installation challenges described in Part I, and more.

32 Delphi at Work
Automated Excel — Ian Davies
There's incredible potential for manipulating Excel through its OLE automation interface — and much more complexity than in last month's manipulation of Word. Fortunately, Excel and VBA provide the tools for the job.

36 Columns & Rows
The Paradox Files: Part III — Dan Ehrmann
The "paradox" is inescapable: The file format remains in daily use, yet suffers from a documentation gap. This article continues the remedy by examining the role of primary and secondary indexes.

41 DBNavigator
Cached Updates: Part II — Cary Jensen, Ph.D.
Want to offer more user-interface options, such as the ability to restore "deleted" records, or to edit read-only DataSets? Cached updates could be just the ticket, as Dr Jensen demonstrates.

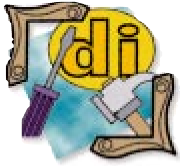
REVIEWS

47 STDynArray 1.0
Product Review by Tim Boyd

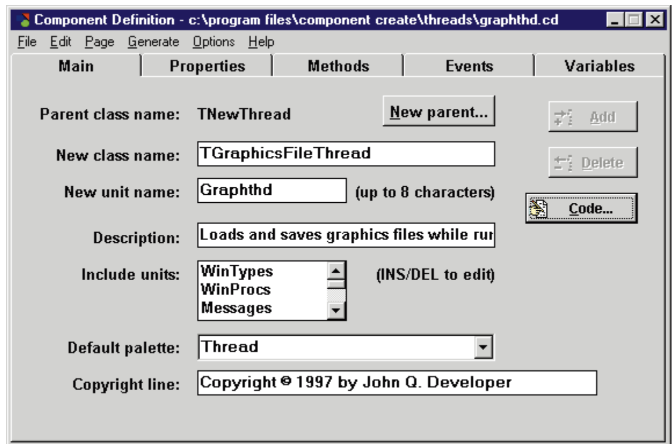
51 Async Professional for Delphi
Product Review by Alan Moore, Ph.D.

DEPARTMENTS

2 Delphi Tools
5 Newsline
56 File | New by Richard Wagner



Potomac Document Software Releases Component Create 2.5



Potomac Document Software of Washington, D.C. has released version 2.5 of ***Component Create for Delphi***, a design tool and code generator for creating Delphi VCL components.

Component Create enables developers to use a point-and-click interface to select the parent class for a new component from an outline view of the VCL, then define the properties, methods, and events of the new component.

Component Create generates a detailed code framework, to which the developer adds only the code needed for the component's functional behavior. An internal code editor allows developers to modify the default code; components can be regenerated without losing the developer's changes.

Component Create allows developers to import container components from .DFM files; build data-aware components with links to multiple fields or data sources; create components that wrap around forms; develop custom property editors; and generate palette bitmaps.

Bundled with version 2.5 is the new version of Thread Component Toolset, a set of base classes for creating thread components.

Developers can use Component Create and Thread Component Toolset to build thread components for performing tasks in the background, such as loading and saving graphics files, backing up files, transferring files, or monitoring real-time processes.

Component Create editions for Windows 95, Windows NT, and Windows 3.1 ship in one box.

Price: US$179
Contact: Potomac Document Software, P.O. Box 33146, Washington, D.C. 20033
Phone: (800) 628-5524
Fax: (202) 244-9065
E-Mail: dprice@compcreate.com
Web Site: http://www.compcreate.com

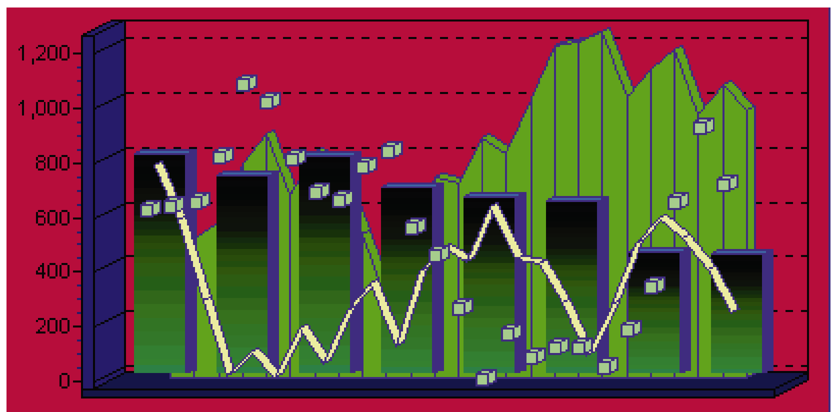
teeMach SL Launches New Charting Tool

teeMach SL has released ***TeeChart-Pro version 3***, a charting tool for Delphi developers.

Available in 16- and 32-bit versions, TeeChart-Pro includes data-aware charts, 16 pre-defined Series types, five extended series types, a developer Custom Series guide, statistical and custom functions, custom printing and drawing, a run-time Chart Editor, and online Help, as well as the source code. It also plugs into *TDataModules* and integrates with QuickReport 2.0.

Borland selected TeeChart's run-time version as the Delphi VCL charting tool in Delphi 3 Developer, Client/Server, and Enterprise versions.

Price: US$99
Contact: teeMach SL, Gran Via, 533, 08011 Barcelona, Catalonia, Spain
Phone: 34-3-453-48-06
Fax: 34-3-454-29-39
E-Mail: teechart@redestb.es
Web Site: http://www.teemach.com



Delphi Training

GenoTechs, Inc., a Borland Training Center, offers hands-on Delphi training courses in Arizona, Florida, Tennessee, and Massachusetts. The courses include: object-oriented and event-driven programming with Delphi; introduction to Pascal; building robust applications using exception handling; detailed study of components; developing an application; adding standard components to an application; Delphi data access architecture; using database components; using system components, dialog components, ReportSmith, and QuickReport; advanced database programming topics; linking data sets; and creating custom components and Windows DLLs. In addition to sample exercises, each student ties the concepts together by developing an application. Each five-day course is US$1,440. On-site training, customized training, project mentoring, and consulting are also available. For details, call (800) GENOTEX or (602) 438-8647, e-mail genotex@genotechs.com, or visit http://www.genotechs.com.

Delphi TOOLS

New Products and Solutions

Nesbitt Software Sells ShareLock for Delphi Online

Nesbitt Software Corp. of Spokane, WA began online sales of Kenn Nesbitt's ***ShareLock***, a component that can turn any application written in Delphi into a trial version.

ShareLock lets Delphi programmers lock their software after a specified number of days, executions, or an absolute date, allowing users to evaluate the software for a limited period before purchasing. ShareLock also provides an optional "grace period" and special "extension key codes," that allow the program to run for a specific number of days or executions beyond the trial period.

Programmers can choose to use ShareLock's built-in data encryption, key generation, and dialog boxes, or may override any of these features to provide greater security or customized messages. ShareLock also watches for users who may try to defeat the locking mechanism by using extension codes more than once, changing the date on their system, etc.

Price: US$39.95
Contact: Nesbitt Software Corp., 2251 San Diego Ave., Ste. A141, San Diego, CA 92110
Phone: (619) 220-8601
Fax: (619) 220-8324
E-Mail: support@nesbitt.com
Web Site: http://www.nesbitt.com

Delphi Training

Para/Matrix Solutions, Inc., a company specializing in training, consulting, and developing PC solutions for large and small businesses, is offering hands-on training for Delphi.

Getting Started with Delphi Client/Server Edition is scheduled for June 16-18, and is priced at US$1,050. Extending Your Delphi Applications is scheduled for June 19-20, and is priced at US$700. For a July schedule of classes, or for more information, call Para/Matrix Solutions, Inc. at (206) 246-4211.

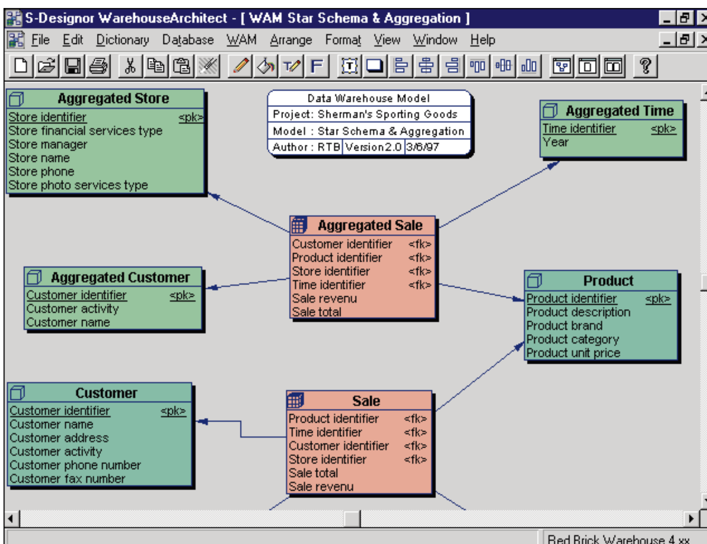
Sybase Unveils Powersoft S-Designor Version 6.0

Sybase, Inc. of Emeryville, CA has announced ***Powersoft S-Designor 6.0***, providing database, data warehouse, and data-aware component design and generation for database designers and developers.

S-Designor 6.0 contains a modular toolset with an integrated set of analysis and design tools. The six modules in the S-Designor family include DataArchitect, WarehouseArchitect, ProcessAnalyst, AppModeler, MetaWorks, and Viewer.

DataArchitect, the database design and generation module, adds entity neighbor selection, tool tips, word wrapping, report preview, and more. It also features Mass change capability and enhanced relationship options.

WarehouseArchitect allows designers to reverse-engineer and import source information and generate schemas for warehouse-optimized databases. It also supports dimensional modeling, including star and snowflake schemas, aggregation, partitioning, summarization, and dimensional hierarchies. Additionally, this new module maintains a map between source information and the warehouse for use in the data cleansing, extraction, and end-user query process.



ProcessAnalyst, S-Designor's data flow discovery and diagramming tool, has added business rules at the process level, along with the ability to link them to any ProcessAnalyst object.

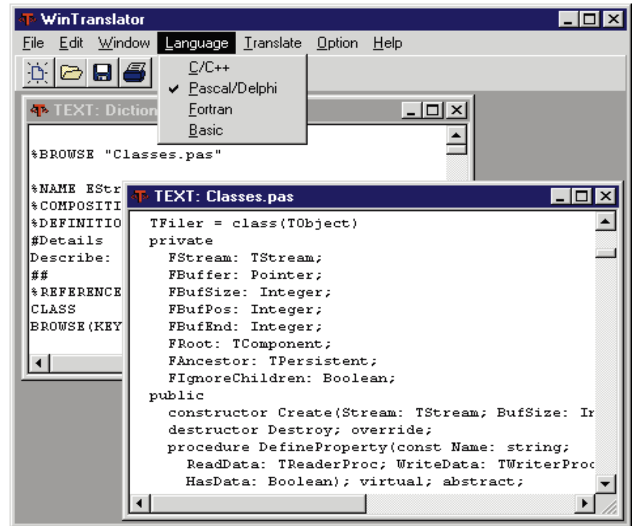
AppModeler adds new generators for Delphi 2, enabling developers to use database models by producing data-aware components and application prototypes for 3GL and 4GL development tools. It also adds a Web generator that enables developers to create data-driven Web sites directly from a data model that provide access to databases and data warehouses.

MetaWorks, the teamwork module, extends its dictionary platform support to include DB2 and Ingres. Models can now be extracted at the submodel level, providing teams of designers greater flexibility for sharing models.

Price: WarehouseArchitect, US$4,995; DataArchitect, US$2,495; Process-Analyst, US$1,495; AppModeler, US$995; AppModeler Desktop, US$295; MetaWorks, US$995; and Viewer, US$395. The S-Designor DataArchitect Suite bundles DataArchitect, ProcessAnalyst, AppModeler, and MetaWorks, and is priced at US$4,995. The S-Designor Warehouse Suite bundles WarehouseArchitect, DataArchitect, ProcessAnalyst, AppModeler, and MetaWorks, and is priced at US$9,295.
Contact: Sybase, Inc., 6475 Christie Ave., Emeryville, CA 94608
Phone: (800) 8-SYBASE or (510) 922-3500
Fax: (510) 922-3210
Web Site: http://www.sybase.com

Excel Software Introduces WinTranslator 1.0

Excel Software of Marshalltown, IA has released *WinTranslator 1.0*. Used with Excel's WinA&D software engineering tool, WinTranslator 1.0 generates class diagrams from Object Pascal, C++, or structure charts, as well as dictionary information from C, Pascal, Basic, or Fortran code.



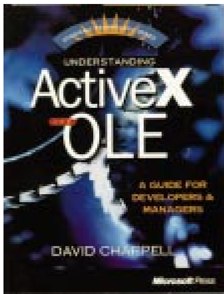
WinTranslator documents legacy code for human understanding, enabling forward engineering of existing projects using modern software engineering methods, and supports an iterative development process between design and code. It can process source code written for any Pascal/Object Pascal, C/C++, Basic, or Fortran compiler for DOS, Windows, Macintosh, UNIX, mainframe, or an embedded computer system.

The reengineering process involves a command to add comment-delimited code keys to the source files and a command to generate a text list of referenced modules. WinTranslator's output files are then imported into WinA&D. Users can customize WinTranslator and its output with language and default options. Several types of library files are supported for optimizing WinTranslator when processing large projects with multiple folders of code files.

Developers can review the code structure by clicking through graphic diagrams in WinA&D or by clicking the code. WinTranslator can also extract information from source code, including class attribute and operation data types, function parameter lists, and programming comments. Design changes and enhancements can be made with WinA&D, with new code generated.

Price: US$495
Contact: Excel Software, 212 Waconda Rd., Marshalltown, IA 50158
Phone: (515) 752-5359
Fax: (515) 752-2435
E-Mail: info@excelsoftware.com
Web Site: http://www.excelsoftware.com

Understanding ActiveX and OLE
David Chappell
Microsoft Press

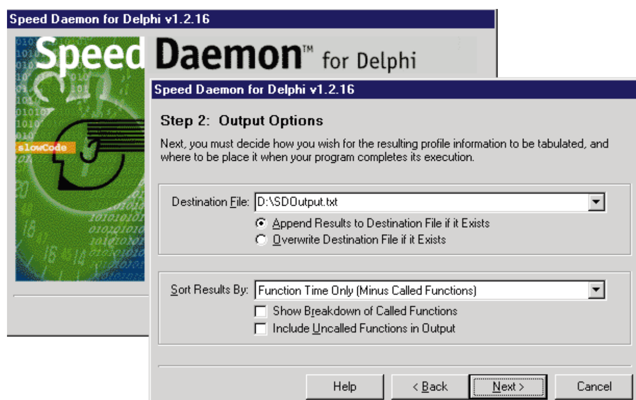


ISBN: 1-57231-216-5
Price: US$22.95
(328 pages)
Phone: (800) MSPRESS

IntegrationWare Releases Speed Daemon Version 1.2

IntegrationWare Inc. of Deerfield, IL has released *Speed Daemon Version 1.2*, a source code profiler for Delphi. It provides an analysis engine for optimizing and tuning applications.

Speed Daemon enables developers to monitor the efficiency of key sections of code and realize improvements in productivity and code efficiency. It also decreases the time required for tracking bugs and performance issues.



Speed Daemon parses the source code and adds any constructs it requires to generate function timings. This modified version of the application is then recompiled, and executed automatically or saved. The entire process is guided by the wizard interface.

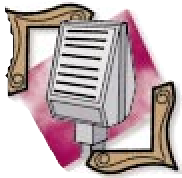
Given a Delphi 1 or 2 project, the utility produces statistics for functions, including the number of times a function is called, the total time spent executing the function relative to other pieces, average time per call, and more. This type of analysis allows developers to isolate potential problems early in the development process.

Speed Daemon works with Delphi 1 and 2 applications, including DLLs and OLE automation servers developed in Delphi. It can also be used on single- and multi-threaded applications.

Price: US$149
Contact: IntegrationWare Inc., Deerfield Tech Center, 111 Deer Lake Rd., Ste. 109, Deerfield, IL 60015
Phone: (888) 773-1133
Fax: (847) 940-1132
E-Mail: sd_feedback@integrationware.com
Web Site: http://www.integrationware.com

News LINE

June 1997



8th Annual Borland Developers Conference Scheduled

Scotts Valley, CA — Featuring over 200 sessions, this year's Borland Developers Conference is slated for July 12-16 at the Opryland Hotel in Nashville, TN. The conference will cover Borland tools, including Delphi, C++Builder, JBuilder, IntraBuilder, and Entera.

Beginning, intermediate, and advanced developers can choose between sessions such as: components; design and methodology; enterprise computing; InterBase; Internet and intranet; management issues; programming, tools, and techniques; and more.

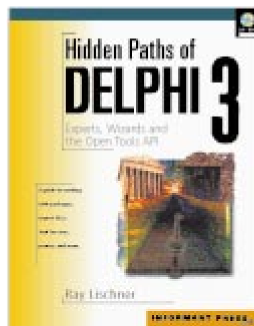
In addition, a free conference CD-ROM is offered to attendees, featuring technical papers, source code, examples from presentations, and more.

For the latest conference information, visit Borland Online at http://www.borland.com.

Informant Press to Release *Hidden Paths of Delphi 3*

Elk Grove, CA — Informant Press announced the publication of *Hidden Paths of Delphi 3* by Ray Lischner. Available in July 1997, this title builds on Lischner's first book, *Secrets of Delphi 2* [Waite Group Press, 1996], revealing more of Delphi's undocumented interfaces. *Hidden Paths of Delphi 3*, and its accompanying CD-ROM, present a thorough explanation with useful examples of every Open Tools API feature.

Introduced in Delphi 2, the Open Tools API allows users to create experts (or wizards) that automate the creation of new forms and projects, and store them in Delphi's Object Repository. It also allows programmers to create new items in Delphi's menu, create new design windows for editing projects, or define new file systems for storing and retrieving source, form, and resource files.



This advanced programming book covers the Open Tools API, including when the programmer should release interface objects, how to create a working stream class, and a Code Summary Expert.

Hidden Paths of Delphi 3 is priced at US$39.95. To order, call (800) 884-6367 or (916) 686-6610, or fax (916) 686-8497. Orders can also be placed via the Informant Web site at http://www.informant.com/-bookclub/index.htm.

Borland Appoints Zack Urlocker Vice President of Product Management

Scotts Valley, CA — Borland recently announced the promotion of Zack Urlocker, 34, to vice president of product management. Urlocker, who is responsible for Borland's product management and marketing activities, reports to Chairman and CEO Delbert W. Yocam.

Previously, Urlocker was director of product management for all Borland products after serving as director of Delphi product management. Urlocker was involved in the development of Delphi and is credited with developing Golden Gate, Borland's client/server and Internet/intranet product strategy.

ZAC Catalogs Delphi Training Tour

Clearwater, FL — ZAC Catalogs conducts interactive lecture-style Delphi training courses each year. The following is a listing of upcoming courses. For a complete schedule and pricing information, contact ZAC Catalogs at (813) 298-1181, or visit their Web site at http://www.zaccatalog.com.

City	Delphi Fundamentals 3 Days - US$995	Advanced Delphi 2 Days - US$695	Creating Custom Components 2 Days - US$695
Los Angeles	June 2-4	June 5-6	June 9-10
New York	June 9-11	June 12-13	June 16-17
Atlanta	June 16-18	June 19-20	June 23-24
Portland	June 16-18	June 19-20	June 23-24
San Francisco	June 23-25	June 26-27	June 30-July 1
Detroit	June 30-July 2		
Chicago	July 7-9	July 10-11	July 7-8
Orlando	July 14-16	July 17-18	

[calendar 1997]

June

1-5 *BFMA Symposium*, Pointe Hilton at Tapatio Cliffs, Phoenix, AZ. Contact Business Forms Management Association at (800) 876-4683 or (503) 227-3393, or visit http://www.bfma.-org/~bfma/symp.htm.

2-4 *Internet World Middle East*, Holiday Inn Crown Plaza, Dubai, United Arab Emirates. Contact Mecklermedia at (800) 632-5537 or (203) 226-6967, or visit http://www.events.iworld.com.

2-6 *Object Expo/Java Expo*, New York Coliseum, New York, NY. Contact SIGS Conferences at (212) 242-7447.

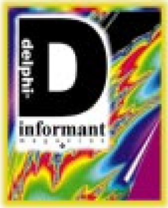
4-6 *Internet World Mexico '97*, World Trade Center, Mexico City, Mexico. Contact Mecklermedia at (800) 632-5537 or (203) 226-6967, or visit http://www.events.iworld.com.

9-12 *Silicon Graphics Developer Forum '97*, Hyatt Regency San Francisco Airport, San Francisco, CA. Contact Silicon Graphics at (800) 770-3033 or (415) 933-3033, visit http://www.sgi.com/support/devprog, or e-mail devprogram@sgi.com.

16-18 *Internet World Israel '97*, Jerusalem Convention Center, Jerusalem, Israel. Contact Mecklermedia at (800) 632-5537 or (203) 226-6967, or visit http://www.events.iworld.com.

17-19 *Web Innovation*, San Jose Convention Center, San Jose, CA. Contact SOFTBANK Expos at (800) 953-4932 or (415) 578-6900, or visit http://www.sbexpos.com.

19-21 *Internet World Portugal '97*, Feira International de Lisbola, Lisbon, Portugal. Contact Mecklermedia at (800) 632-5537 or (203) 226-6967, or visit http://www.events.iworld.com.



By Jerry Coffey

InterBase Stored Procedures

Creating Them and Using Them from Delphi

One of the cornerstones of successful client/server programming is the stored procedure. Unfortunately, InterBase stored procedures are woefully underdocumented, especially regarding the Delphi connection. This article attempts to help fill the documentation gap.

This article has two major goals. The first is to provide a general description of an InterBase stored procedure, describe the benefits of stored procedures, and provide specific examples of the more common procedures you'll need to create. The second is to explain how Delphi uses the stored procedures, i.e. explain how they're called.

What Is a Stored Procedure?

A stored procedure is a routine written in InterBase trigger and procedure language (catchy, hunh?) that can be called by a client (e.g. a Delphi application) or another procedure or trigger. Stored procedures can be used for many things, but this article will focus on their use with the mainstay SQL statements: SELECT, INSERT, UPDATE, and DELETE.

SELECT statements are the most common, so let's tackle them first. A stored procedure that contains a SQL SELECT statement is often referred to as a *select procedure*.

```
CREATE PROCEDURE SPS_Address_ProviderID(ProviderID INTEGER)
RETURNS (Address CHAR(60),
         City    CHAR(30),
         State   CHAR( 2),
         ZipCode CHAR( 5))
AS
BEGIN

  FOR
    SELECT Address, City, State, ZipCode
      FROM Address
     WHERE ProviderID = :ProviderID
      INTO :Address, :City, :State, :ZipCode
  DO SUSPEND;

END
```

Figure 1: An example of a select procedure.

Stored Procedure Basics

An InterBase stored procedure is created using a CREATE PROCEDURE statement. The code in Figure 1, for example, creates a select procedure named `SPS_Address_ProviderID`. This straightforward select procedure returns four columns from any Address table row that has a ProviderID column value equal to the ProviderID provided as an input argument.

As you can see, a select procedure is essentially a SQL SELECT statement in the form of a function call. Input parameters — if there are any — are included in the CREATE statement as a comma-delimited list within parentheses. There's only one in this example; it's named `ProviderID`, and is of type INTEGER. The InterBase data types are shown in Figure 2.

Any output parameters — and there must be at least one for a select procedure — are described in a RETURNS statement, which also takes the form of a comma-delimited list inside parentheses. In this example, there are four output parameters: `Address`, `City`, `State`, and `ZipCode`. All are of type CHAR, which is used for strings.

Header and body. The CREATE and RETURNS statements (if RETURNS is present) comprise the stored procedure's *header*.

Everything following the AS keyword is the procedure's *body*. In this example, the body is contained entirely within the BEGIN and END statements required for every stored procedure.

There can also be statements between the AS and BEGIN keywords that are considered part of the body. These statements declare local variables for the stored procedure; we'll discuss them later, along with the FOR, INTO, and DO SUSPEND statements.

Name	Size	Range/Precision
BLOB	variable	None — BLOB segment size is limited to 64KB
CHAR(*n*)	*n* characters	1 to 32767 bytes
DATE	64 bits	1 Jan 100 to 11 Dec 5941
DECIMAL(*precision*, *scale*)	variable	*precision* = 1 to 15, least number of precision digits *scale* = 1 to 15, number of decimal places
DOUBLE PRECISION	64 bits	1.7×10^{-308} to 1.7×10^{308}
FLOAT	32 bits	3.4×10^{-38} to 3.4×10^{38}
INTEGER	32 bits	-2,147,483,648 to 2,147,483,648
NUMERIC(*precision*, *scale*)	variable	same as DECIMAL
SMALLINT	16 bits	-32768 to 32767
VARCHAR(*n*)	*n* characters	1 to 32765 bytes

Figure 2: InterBase data types (based on a chart from the *InterBase Workgroup Server Data Definition Guide*, pages 46-7).

Why Use Them?

They're fast. A query stored on the server as a procedure executes far more quickly than one built and executed on the client. The speed difference is even more pronounced when your database application is running on a LAN or WAN. The main reason is that when the client application sends the query to the server, the server responds with a large amount of metadata (specific database information about the requested query). The query plan is then built, and the query is re-sent to the server for execution.

In contrast, if a stored procedure is used to perform the SQL statement, the client simply requests that the server execute the procedure, and send back the answer set (if any). The result is that just two trips are made between the client and server, instead of four (one of which contains a large amount of data).

They're reusable. In a database application of significant size, you'll find yourself using the same SQL statements (SELECTs, INSERTs, etc.) repeatedly. Rather than recreate a statement on the client each time, it's better to store the statement in the database, and call it. It's the same idea as maintaining a library of procedures and functions shared between modules. The benefits are the same, as well: readability is enhanced, and redundancy, maintenance, and documentation are greatly reduced.

They're part of the database. Although this has been mentioned, it bears repeating that a stored procedure is part of the database. Not only does this make the procedure readily accessible to the database, it also insures that the procedure is syntactically correct, and that the SQL statements included in the procedure are correct. The database will not accept it until it's valid, i.e. the CREATE PROCEDURE statement will fail.

Creating Select Procedures

We'll examine five types of stored procedures:

- a SELECT statement that may return multiple rows
- a SELECT statement that returns one row
- an INSERT statement
- an UPDATE statement
- a DELETE statement

Creating a Multi-Row SELECT

When a SELECT statement might return multiple rows, the stored procedure must use the FOR..DO looping construct. We've already seen this in Figure 1:

```
FOR

  SELECT Address, City, State, ZipCode
    FROM Address
   WHERE ProviderID = :ProviderID
    INTO :Address, :City, :State, :ZipCode

DO SUSPEND;
```

Here a FOR..DO loop has been placed around the SELECT statement. This will cause the SUSPEND command to be executed for each row returned by the SELECT statement. (SQL programmers will recognize this as a fetch loop on an open cursor.)

Fine — but what does SUSPEND do? It's got a lousy name, but a SUSPEND command is absolutely necessary to make the SELECT stored procedure work. It causes the stored procedure to return a value via the variables associated with the INTO clause. (Note: InterBase will accept a stored procedure without a SUSPEND statement, but the stored procedure will never return a value.)

Loading the output variables. An additional clause on the SELECT statement may be new to you. The INTO clause describes the variables that will be loaded with the result of the SELECT statement, then returned by the stored procedure via the variables described in the RETURNS

```
SET TERM ^ ;
CONNECT "c:\doj\cmis\cmis.gdb"^

CREATE PROCEDURE SPS_Subject_Confidential(
  ProviderID INTEGER)
RETURNS (ConfideCount INTEGER)
AS
BEGIN

   SELECT COUNT(*)
     FROM Party P, CaseStatus CS, Status S
    WHERE P.ProviderID           = :ProviderID
      AND CS.ComplaintID         = P.ComplaintID
      AND CS.Status              = S.Status
      AND S.ConfidentialityFlag  = 'T'
      AND CS.StatusDate =
            ( SELECT MAX(StatusDate)
                FROM CaseStatus Case
               WHERE Case.ComplaintID = P.ComplaintID )
     INTO :ConfideCount;

  SUSPEND;

END^
SET TERM ; ^
```

```
SET TERM ^ ;
CONNECT "c:\doj\cmis\cmis.gdb"^

CREATE PROCEDURE SPI_Payment
        (
        MoneyOwedBMCFID       INTEGER,
        AmountPaid            FLOAT,
        CheckNumber           CHAR(15),
        DateOfCheck           DATE,
        DateMoneyReceived     DATE,
        DateMoneyDistributed  DATE
        )
AS
BEGIN

  INSERT INTO Payments
        (
        MoneyOwedBMCFID,
        AmountPaid,
        CheckNumber,
        DateOfCheck,
        DateMoneyReceived,
        DateMoneyDistributed
        )
  VALUES
        (
        :MoneyOwedBMCFID,
        :AmountPaid,
        :CheckNumber,
        :DateOfCheck,
        :DateMoneyReceived,
        :DateMoneyDistributed
        );
END^

SET TERM ; ^
```

Figure 3 (Top): This ISQL script creates a singleton SELECT. This COUNT statement will always return one row, so there is no need for the FOR..DO loop. **Figure 4 (Bottom):** This stored procedure describes a SQL INSERT statement.

statement. They must agree in number, order, and name, or InterBase will not accept the procedure.

A Singleton SELECT

When a SELECT statement will return only one row, there's no need for a FOR..DO loop (see Figure 3).

```
SET TERM ^ ;
CONNECT "c:\doj\cmis\cmis.gdb"^

CREATE PROCEDURE SPU_Penalty
        (
        PenaltyID       INTEGER,
        PartyID         INTEGER,
        PenaltyType     CHAR(20),
        PenaltyUnitType CHAR(10),
        DateOfPenalty   DATE,
        PenaltyUnits    INTEGER
        )
AS
BEGIN

   UPDATE Penalty
      SET PartyID
    WHERE PenaltyID = :PenaltyID; = :PartyID,
         PenaltyType     = :PenaltyType,
         PenaltyUnitType = :PenaltyUnitType,
         DateOfPenalty   = :DateOfPenalty,
         PenaltyUnits    = :PenaltyUnits
END^

SET TERM ; ^
```

Figure 5: This stored procedure describes a SQL UPDATE statement.

However, it's important to ensure that the SELECT will never attempt to return more than one row, i.e. that the WHERE clause uses a unique row identifier. If InterBase determines that multiple rows are possible, it will not accept the procedure.

The SELECT statement in Figure 3 is returning the result of the aggregate function, COUNT, so it will always return one row. (Incidentally, it also features a sub-SELECT. This type of query is useful in any situation where you need to determine the current status row for something — a "case" in this instance.)

The SELECT statement now requires a terminating semicolon:

```
INTO :ConfideCount;
```

as does the one-word SUSPEND statement that immediately follows it.

This is in contrast to the stored procedure shown in Figure 1. It may seem odd, but in the multiple SELECT shown in Figure 1, there's only one statement in the body of the procedure: It's a FOR..DO statement that's terminated just after the SUSPEND command:

```
DO SUSPEND;
```

Therefore, there is no terminating semicolon for the SELECT itself.

An INSERT

An INSERT statement is used to add a row to an InterBase table. No RETURNS variable is necessary for an INSERT stored procedure (see Figure 4). Not shown is that an InterBase trigger is using a generator to automatically assign

```
SET TERM ^ ;
CONNECT "c:\doj\cmis\cmis.gdb"^

CREATE PROCEDURE SPD_LicenseToBill (ProviderID INTEGER)
AS
BEGIN

  DELETE FROM LicenseToBill
   WHERE ProviderID = :ProviderID;

END^

SET TERM ; ^
```

Figure 6: This stored procedure describes a SQL DELETE statement.

a value to a primary key column — a typical scenario. (These issues are discussed in detail in Bill Todd's article, "InterBase Triggers and Generators," beginning on page 11.)

An UPDATE

An UPDATE statement is used to modify one or multiple columns of an existing row in an InterBase table. No RETURNS variable is necessary for an UPDATE stored procedure (see Figure 5). However, one or more of the input arguments must be used in a WHERE clause to identify the row to update.

A DELETE

A DELETE statement is used to remove an existing row or rows from an InterBase table. No RETURNS variable is necessary for a DELETE stored procedure (see Figure 6). One or more of the input arguments must be used in a WHERE clause to identify the row(s) to delete.

ISQL Scripts

To add a stored procedure to an InterBase database, you must describe the stored procedure in an ISQL script, then run that script using ISQL. The code examples presented so far are ISQL scripts that must be run through InterBase's interactive interface, ISQL (using the menu command **File | Run an ISQL Script**). A couple of tricks are required to make these scripts work.

First, although you may already have connected to an InterBase database using ISQL (**File | Connect to Database**), it is still necessary to explicitly connect each time an ISQL script is executed. This is done with a CONNECT statement; for example:

```
CONNECT "c:\doj\cmis\cmis.gdb"^
```

The trouble with terminators. Second, an ISQL script must satisfy two masters: the ISQL tool itself, and the InterBase database it addresses. Both require statement terminators, and both use the semicolon (;) as their default terminator character. Something's gotta give, so you need to temporarily change the terminator for ISQL. This is done with the SET TERM command. This statement, for example:

```
SET TERM ^ ;
```

tells ISQL to use the carat (^) character as a terminator until further notice. You can use any character you like as the alternate terminator, but I would highly recommend

```
procedure ...
var
  FetchCount   : Word;
  QueryAddress : TQuery;
 ...
  QueryAddress := TQuery.Create(Self);
  with QueryAddress do begin
    DatabaseName := 'CMIS_DB';
    SQL.Add('SELECT AddressType, Address, City, County,  ');
    SQL.Add('       State, ZipCode, ZipPlus4, PhoneNumber');
    SQL.Add('  FROM Address                              ');
    SQL.Add(' WHERE ProviderID = :ProviderID             ');
    ParamByName('ProviderID').AsInteger :=
      SubjectUpdateProviderID;
    Open;

    FetchCount := 0;
    while EOF = False do begin
      with StringGridAddress do begin
        RowCount := FetchCount + 1;
        Cells[0,FetchCount] := Fields[0].Text;
        Cells[1,FetchCount] := Fields[1].Text;
        Cells[2,FetchCount] := Fields[2].Text;
        Cells[3,FetchCount] := Fields[3].Text;
        Cells[4,FetchCount] := Fields[4].Text;
        Cells[5,FetchCount] := Fields[5].Text;
        Cells[6,FetchCount] := Fields[6].Text;
        Cells[7,FetchCount] := Fields[7].Text;
      end;
      Inc(FetchCount);
      Next;
    end;

  Free;

  end;
  ...
end;
```

Figure 7: Describing and executing a SQL SELECT statement with Object Pascal.

that you use something unusual. Typically, the last statement in an ISQL script replaces the semicolon as the terminating character.

Calling Stored Procedures from Delphi

Okay, we know how to build the stored procedures. Now how do we call them from Delphi? There are two ways — one is necessary for SELECT statements (i.e. statements that return a value), the other for INSERT, UPDATE, and DELETE statements.

Stored procedures with SELECT statements are called from Delphi using a Query object (of class *TQuery*). This is despite the fact that we're calling a stored procedure; again, a Delphi Query object is used for any statement that returns the result of a SELECT statement. The other SQL statements — INSERT, UPDATE, and DELETE — are called using a Delphi StoredProc object (of class *TStoredProc*).

Calling a Select Procedure

We'll describe how stored procedures are called, beginning with a SELECT statement. First, however, let's back up a bit and take a look at how we'd describe and call a "conventional" query (i.e. one *not* contained in a select procedure) using Object Pascal (see Figure 7).

First the Query object, *QueryAddress*, is instantiated, and its *Database* and *SQL* properties are assigned values. Then the

single query parameter, *ProviderID*, is assigned a value, and the query is executed using the *Open* method. In this example, a **while** loop is used to take the results of the query and load them into a StringGrid component.

All of this is familiar, but how do we change it to call a stored procedure? For this SELECT statement, the changes are fairly minor (see Figure 8). There are two notable differences:

- First, the FROM clause now refers to the name of the stored procedure, `SPS_Address_ProviderID`, not a specific table. (The difference would be more pronounced if there were a list of tables.)
- Second, there is no WHERE clause; the WHERE clause is described in the stored procedure. The input parameter is simply placed in parentheses following the FROM clause. (Again, the difference would have been more pronounced if there had been an elaborate WHERE clause.)

The rest of the procedure is the same: Multiple rows are being loaded into a StringGrid, with the *Next* method being used to fetch the next record in the answer stream. Note also that a looping structure would be unnecessary if the code were calling a singleton SELECT.

```
var
  FetchCount   : Word;
  QueryAddress : TQuery;
  ...
  QueryAddress := TQuery.Create(Self);
  with QueryAddress do begin
    DatabaseName := 'CMIS_DB';
    SQL.Add('SELECT AddressType, Address, City, County,  ');
    SQL.Add('       State, ZipCode, ZipPlus4, PhoneNumber');
    SQL.Add('  FROM SPS_Address_ProviderID (:ProviderID) ');
    ParamByName('ProviderID').AsInteger :=
      SubjectUpdateProviderID;
    Open;

    FetchCount := 0;
    while EOF = False do begin
      with StringGridAddress do begin
        RowCount := FetchCount + 1;
        Cells[0,FetchCount] := Fields[0].Text;
        Cells[1,FetchCount] := Fields[1].Text;
        Cells[2,FetchCount] := Fields[2].Text;
        Cells[3,FetchCount] := Fields[3].Text;
        Cells[4,FetchCount] := Fields[4].Text;
        Cells[5,FetchCount] := Fields[5].Text;
        Cells[6,FetchCount] := Fields[6].Text;
        Cells[7,FetchCount] := Fields[7].Text;
      end;
      Inc(FetchCount);
      Next;
    end;

    Free;

  end;
```

Figure 8: Executing an InterBase select procedure from Object Pascal.

Calling a Stored Procedure to Perform an INSERT, UPDATE, or DELETE Operation

As mentioned earlier, a Delphi StoredProc object must be used for SQL operations that do not return an answer set, i.e. the result of a SELECT statement. Therefore, they're used to call stored procedures that contain INSERT, UPDATE, and DELETE statements.

From a Delphi standpoint, these three statements are handled the same, so we'll look at just one — an UPDATE. The Object Pascal code in Figure 9 calls a stored procedure that contains the UPDATE statement from Figure 5.

There are some similarities: A StoredProc object is instantiated in the same way as a Query object, and its *Database* property must also be assigned.

After that, however, the similarities disappear. The *StoredProcName* property must be assigned the name of the stored procedure — in this case, `SPU_Penalty`. Also, the *Prepare* method must be used to tell the server to get the stored procedure ready to accept input, and otherwise prepare for execution. Note also that the *ExecProc* method is used instead of *Open* (just as it is when *TQuery* objects return no value).

After *Prepare* has been called, the parameters can be assigned just as they are with Query objects — using the *ParamByName* method. Finally, the *ExecProc* method is used to execute the stored procedure (again, in lieu of the Query *Open* method, because no value is returned).

```
var
  StoredProcPenalty : TStoredProc;
  ...
  StoredProcPenalty := TStoredProc.Create(Self);
  with StoredProcPenalty do begin
    DatabaseName    := 'cmis_db';
    StoredProcName  := 'SPU_Penalty';
    Prepare;
    ParamByName('PenaltyID').AsInteger  := PenaltyPenaltyID;
    ParamByName('PartyID').AsInteger    := PenaltyPartyID;
    ParamByName('PenaltyType').AsString :=
      ComboBoxPenaltyType.Text;
    ParamByName('PenaltyUnitType').AsString :=
      ComboBoxPenaltyUnits.Text;
    ParamByName('DateOfPenalty').AsDate :=
      StrToDate(MaskEditPenaltyDate.Text);
    ParamByName('PenaltyUnits').AsInteger :=
      StrToInt(MaskEditPenalty.Text);
    ExecProc;
    Free;
  end;
```

Figure 9: Executing a stored procedure that contains an INSERT statement.

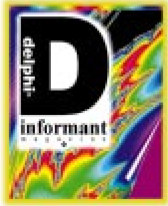
Conclusion

We've examined real-world examples of how to use InterBase stored procedures to develop a client/server application with Delphi. Along the way, we've covered the basics of InterBase trigger and procedure language, and — among other things — learned how to build select procedures, and how to call stored procedures from Delphi.

Another benefit of learning InterBase trigger and procedure language is that it's very much like the procedural languages used by other database vendors (Oracle's PL/SQL, for example), so once you've mastered the InterBase flavor, you'll make short work of the next. ∆

Jerry Coffey is Editor-in-Chief of *Delphi Informant*.

Table of Contents



By Bill Todd

InterBase Triggers and Generators

Inside InterBase: Part I

You cannot write client/server applications without triggers. This may seem a strong statement, and although it's not strictly true, you certainly cannot write even moderately complex client/server applications without triggers. Triggers let you customize the behavior of a database so it responds to changes in data exactly as you specify.

In this article, we'll examine InterBase triggers and generators (a tool for generating unique values), and some specific uses for them in client/server programs. This article is accompanied by the file DITABLES.SQL. This InterBase ISQL script creates the simple, three-table database and triggers used in our examples (see the download message at the end of the article for details).

Similarities and Differences

InterBase triggers are very similar to stored procedures. Like stored procedures, triggers are written in the InterBase procedure and trigger language, and stored as part of the metadata for a database. Triggers differ from stored procedures in the way they're called. Stored procedures are called explicitly in your code; triggers are called automatically in response to a change in data.

When they fire. InterBase triggers can be associated with INSERT, UPDATE, or DELETE events for a table. In addition, you can specify whether a trigger fires before or after the event using the BEFORE and AFTER keywords. Combining these keywords and events provides six triggers: BEFORE INSERT, AFTER INSERT, BEFORE UPDATE, AFTER UPDATE, BEFORE DELETE, and AFTER DELETE.

You can attach as many triggers as you wish to any of these events. Thus you can write short, simple, single-function triggers that are easy to develop, test, and maintain. Additionally, triggers can call stored procedures, which provide another way to divide your code into modules.

Position. When creating a trigger, you also specify a position number for it. An integer between 0 and 32,767, the position number determines the order in which multiple triggers attached to the same event execute.

When attaching multiple triggers to the same event, it's a good idea to increment position numbers by five or 10, so later you can easily add other triggers anywhere in the execution order. InterBase allows you to have multiple triggers with the same position number; however, the order of execution is unpredictable.

Participation in transactions. It's important to understand how triggers participate in transactions. If a trigger makes changes to the database, those changes are part of the transaction that caused the trigger to fire. If the transaction is rolled back, all changes made by the trigger are also rolled back.

Here's an example: You start a transaction and delete a customer record. The Customer table has a BEFORE DELETE trigger that deletes all the customer's orders from the Orders table. The Orders table has a BEFORE DELETE trigger that deletes all the items for each order from the Items table. If you roll back the transaction, all the DELETEs for the Items, Orders, and Customer tables will be rolled back. If any of the triggers calls a stored procedure, the stored procedure's actions will also be part of the transaction.

Logging Changes

Triggers can be used for a variety of tasks, including: logging changes to a history table, enforcing referential integrity, cascading deletes and updates, data validation, and making a multi-table view updatable. Another powerful use for triggers in InterBase databases is to create an event that will use InterBase's unique *event alerter* technology to notify any interested programs that the event has occurred. (Event alerters are the topic of "InterBase Event Alerters," an article by Alexander Le and Donna Burbank, beginning on page 15.)

Creating a log of changes to a database table is a common requirement, and serves as a good introduction to the basics of InterBase triggers. Orders and Order_History are two of the sample tables accompanying this article. Their structures are identical, but Order_History has one additional column, Change_Date, that will store the date the order record was changed.

Figure 1 shows the `log_order_change` trigger. Attached to the Orders table, it will write a record containing the order record's prior image to the Order_History table each time an order record is updated.

```
CREATE TRIGGER log_order_change FOR orders
ACTIVE AFTER UPDATE POSITION 10 AS
BEGIN
  INSERT INTO order_history
    (order_num, change_date, cust_num,
     order_date, ship_date)
  VALUES (OLD.order_num, "now", OLD.cust_num,
          OLD.order_date, OLD.ship_date);
END ^
```

Figure 1: Logging changed records with the `log_order_change` trigger.

The CREATE TRIGGER statement consists of a header and body. Everything up to the AS keyword comprises the header. It must include the trigger's name, followed by the keyword FOR, followed by the name of the table to which the trigger applies.

The optional keyword, ACTIVE or INACTIVE, follows the table name. If you don't want the trigger to fire when its event occurs, create it with INACTIVE. You can activate the trigger later using the SQL statement ALTER TRIGGER. Because ACTIVE is the default state of all triggers, there's no reason to use it.

Next comes the AFTER or BEFORE keyword and the name of the event that will fire the trigger. The `log_order_change` trigger fires after the UPDATE operation because we don't want to log the update until after it happens.

Although `log_order_change` is the first trigger for the Orders table, the CREATE TRIGGER statement includes a position number of 10. The position is optional; however, adding a position number now makes it easy to add another trigger to the same event later, and have it execute before or after this trigger by assigning a higher or lower position number.

The trigger's body follows the AS keyword and consists of one or more InterBase procedure and trigger language statements enclosed in a BEGIN..END block. In Figure 1, the body consists of a single INSERT statement that creates a new record in the Order_History table.

OLD and NEW. Note the use of the context variable OLD in the VALUES clause of the INSERT statement. Within a trigger's body, you can access the old and new values of all columns in the table. For example, the old value of the Order_Num column is accessed as `OLD.order_num`.

Conversely, the column's new value is `NEW.order_num`. Using context variables, you can determine which columns have changed, then perform any operations you want with old and new values. The VALUES clause also uses the keyword NOW to insert the current date and time into the Change_Date column. OLD does not apply in an INSERT trigger because there are no old values, and the context variable, NEW, does not apply in a DELETE trigger because there are no new values when a row is deleted.

Referential Integrity

While InterBase supports declarative referential integrity through FOREIGN KEY declarations, there are two important limitations. InterBase does not support either cascaded updates or deletes via declarative referential integrity. If you need either of these features, you must implement referential integrity through triggers.

Providing referential integrity with cascaded updates and deletes requires three triggers. For the Customer and Orders tables, we need:

1) A BEFORE INSERT trigger on Orders to enforce referential integrity.
2) A BEFORE DELETE trigger on Customer to cascade deletes.
3) A BEFORE UPDATE trigger on Customer to cascade updates.

The code in Figure 2 first creates the exception, `orders_ri`. Then, the `order_prevent_insert` trigger is created. It raises an exception if a user tries to add an order record with a customer number that doesn't match a record in the Customer table. You can create as many exceptions as needed to provide custom error messages within your triggers.

```
CREATE EXCEPTION orders_ri "No customer record.";

CREATE TRIGGER order_prevent_insert FOR orders
BEFORE INSERT POSITION 10 AS

DECLARE VARIABLE RECORD_COUNT INTEGER;
BEGIN
  SELECT COUNT(cust_num)
    FROM customer C
   WHERE C.cust_num = NEW.cust_num
    INTO :RECORD_COUNT;

  IF (RECORD_COUNT = 0) THEN
    EXCEPTION orders_ri;
END ^
```

Figure 2: The referential integrity trigger for the Orders table.

```
CREATE TRIGGER cust_delete_orders FOR customer
BEFORE DELETE POSITION 10 AS
BEGIN
  DELETE FROM orders
   WHERE orders.cust_num = OLD.cust_num;
END ^
```

Figure 3: Cascading deletes.

```
CREATE TRIGGER cust_update_orders FOR customer
BEFORE UPDATE POSITION 10 AS
BEGIN
  IF (OLD.cust_num <> NEW.cust_num) THEN
    UPDATE orders
       SET cust_num = NEW.cust_num
     WHERE cust_num = OLD.cust_num;
END ^
```

Figure 4: Cascading updates to the Orders table.

```
CREATE EXCEPTION customer_delete "Customer has orders.";

CREATE TRIGGER customer_prevent_delete FOR customer
BEFORE INSERT POSITION 10 AS
DECLARE VARIABLE RECORD_COUNT INTEGER;
BEGIN
  SELECT COUNT(order_num)
    FROM orders O
   WHERE O.cust_num = OLD.cust_num
    INTO :RECORD_COUNT;

  IF (RECORD_COUNT <> 0) THEN
    EXCEPTION customer_delete;
END ^
```

Figure 5: Preventing deletion using the `customer_prevent_delete` trigger on the Customer table.

The `cust_delete_orders` trigger in Figure 3 cascades deletes from the Customer table to the Orders table. This BEFORE DELETE trigger fires whenever a customer is deleted, and ensures all order records for this customer are deleted before the customer record is deleted. If the trigger can't delete the order records, an exception will be raised and prevent the customer record from being deleted.

Cascaded updates are handled by the `cust_update_orders` trigger in Figure 4. It checks if the value of the key field in the Customer table has changed by comparing the old and new values. If the value has changed, the UPDATE statement updates the foreign key in all the order records. Using a BEFORE UPDATE trigger will prevent the customer record from being updated unless all the order records are successfully updated. You can use a similar trigger to make a multi-table view updatable. Simply provide an AFTER UPDATE trigger to update the tables from which the view is derived.

Using triggers, you can easily propagate referential integrity with cascading updates and deletes throughout a complex data model. Consider a more realistic database that includes an Items table as a child of the Orders table. In addition to the triggers previously described, you would need three more:

1) A BEFORE INSERT trigger for Items to ensure the order record exists.
2) A BEFORE DELETE trigger for Orders to delete all items for the order.
3) A BEFORE UPDATE trigger for Orders to update the order number in the item records.

Of course, you don't need to cascade both deletes and updates. To allow a customer to be deleted if that customer has orders, you would change the BEFORE DELETE trigger for the Customer table (see Figure 5). The `customer_prevent_delete` trigger counts the order records with the same customer number as the customer record being deleted. If the count is not zero, the trigger raises an exception (`customer_delete`) that blocks the deletion.

On the Server: Validating Data

One of the significant advantages offered by database servers is the ability to handle all your data integrity checks on the server. Validating data on the server means you have one centralized set of business rules that apply to every program that accesses your data. Even the DBA can't bypass your data validation rules without explicitly dropping or disabling them.

Another advantage of centralizing data validation on the server is that it's easy to change the rules if necessary. Changing a constraint or trigger on the server immediately affects every applicable client program; there's no need to recompile or change the application programs in any way.

Constraints and triggers are the two mechanisms for validating data on an InterBase server. InterBase's CHECK constraint is so powerful that you can usually get along without using triggers for data validation. Although they're a little more work, using triggers offers one big advantage: You have complete control over the exception raised and its error message if the data is invalid. Figure 6 is a simple example. The `check_license` trigger won't allow you to insert a new customer record unless the state in which the customer is located is in the Licensed_States table.

```
CREATE EXCEPTION not_licensed
  "We are not licensed in that state.";

CREATE TRIGGER check_license FOR customer
BEFORE INSERT POSITION 10 AS
DECLARE VARIABLE RECORD_COUNT INTEGER;
BEGIN
  SELECT COUNT(cust_num)
    FROM licensed_states L
   WHERE L.state = NEW.state
    INTO :RECORD_COUNT;
  IF (RECORD_COUNT = 0) THEN
    EXCEPTION not_licensed;
END ^
```

Figure 6: A data validation trigger.

Using Generators

A generator is an InterBase object that provides unique numbers in a multi-user environment. If you have a table that has no practical, natural primary key, you can use a generator to provide a unique number to use as a surrogate key. The CREATE GENERATOR statement allows you to create a generator and assign a unique name to it. For example, this statement creates a generator to generate unique customer numbers:

```
CREATE GENERATOR customer_numbers
```

After a generator is defined, you can set its starting value or reset its value using the SET GENERATOR statement. To obtain the next value from a generator, call the GEN_ID function and supply the generator's name and increment value as parameters.

Inside Delphi. There's a trick to using generators in a Delphi program. Normally, to obtain a unique key from a generator, you would use a BEFORE INSERT trigger to get the next value from the generator, and assign it to the key field in the new record.

If you do this in a database used by a Delphi program, however, you'll get an error stating that the record has been deleted each time you try inserting a new record. This happens because the Delphi Visual Component Library automatically rereads each new record after it's posted so you'll retain your position on the new record. The problem is Delphi has no way to know the trigger in the database changed the primary key's value in the new record, so it can't find the record.

The solution is to create an *OnNewRecord* event handler for your table. In *OnNewRecord*, you can call a stored procedure that returns the next value from the generator and assigns that value to the key field before the record is posted.

Figure 7 shows the stored procedure, `next_cust_num`, that returns the next value from a generator. To call the procedure from your Delphi program, drop a *TStoredProc* component on your form. Set its *DatabaseName* property to the alias for your InterBase database and the *StoredProcName* property to `next_cust_num`. In *OnNewRecord*, use this code to call `next_cust_num` and retrieve the next value from the generator:

```
with StoredProc1 do begin
  ExecProc;
  CustTbl.FieldByName('cust_num').AsInteger :=
    ParamByName(new_cust_num).AsInteger;
end;
```

```
CREATE PROCEDURE next_cust_num
RETURNS (new_cust_num INTEGER) AS
BEGIN
  next_cust_num = GEN_ID(customer_numbers, 1);
END ^
```

Figure 7: This stored procedure obtains the next value from a generator.

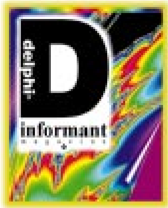
Conclusion

Triggers may be the most powerful tool in a database server. They allow you to create programs that are part of the database and cause the database to respond to changes in the data it contains in any way you wish. Triggers can be used to implement referential integrity with cascaded updates and deletes, validate data, notify programs an event has occurred — or just about anything else you can think of. Using triggers, you can centralize your business rules in the database for consistency and easy maintenance, and reduce the amount of code required in your client programs.

While this article is a good introduction to InterBase triggers, it's not a thorough treatment of the InterBase procedure and trigger language. When you start working with stored procedures and triggers, read the appropriate sections of the InterBase manuals. They present all the language's features, giving you a good overview of the power and flexibility available in triggers. ∆

The ISQL script, DITABLES.SQL, is available on the Delphi Informant Works CD located in INFORM\97\JUN\DI9706BT.

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 is also a nationally known trainer and has been a speaker at every Borland Developers Conference and the Borland Conference in London. He can be reached on CompuServe at 71333,2146, on the Internet at 71333.2146@compu-serve.com, or at (602) 802-0178.



By *Alexander Le* and *Donna Burbank*

InterBase Event Alerters

An Introduction to the IBEventAlerter Component

As Delphi developers, we are experienced in using Delphi as a front-end tool. Commands are sent from our application to update and populate the database; the Delphi client application is in complete control of data exchange, and the database mutely accepts the data. With the use of InterBase event alerters, however, this relationship is reversed. When the Delphi IBEventAlerter component is combined with InterBase events, we can create a robust database application with the ability to *talk back*. This article will focus on InterBase event alerters, and using these events with Delphi's IBEventAlerter component.

Our goal is to understand how to create an event inside InterBase, and trap for that specific event within our database application. Consider, for example, a typical inventory application written in Delphi. It would probably be helpful to alert the user when a certain stock item has been deleted — perhaps by another application, such as an order-entry system. InterBase allows us, as developers, to register events that will notify our front-end application of a certain occurrence.

On the InterBase Side

In InterBase, such an event alerter would be implemented using the POST_EVENT command. First, we must decide where to initiate this particular event. We can accomplish this by creating a stored procedure or trigger to surface the particular event message:

```
CREATE TRIGGER Time_to_Order_More FOR Stock
AFTER ACTIVE INSERT
AS
BEGIN
  POST_EVENT "order_more";
END
```

On the Delphi Side

This trigger will post an `order_more` event to the InterBase Event Manager. Once this event has been implemented in InterBase, the Delphi application must be able to receive this particular message. There are two ways that an application can wait for an InterBase event: synchronously and asynchronously.

Synchronous vs. asynchronous. With a synchronous wait, an application expresses interest in an event, stops all other processing, and continually polls the database to see if this event has occurred. This polling consumes resources on both the server and client. This is clearly a suboptimal solution in most cases. For this method to be used successfully, the application should run as an automated process on the server or client machine, and execute its specified task only when the event occurs.

With asynchronous event trapping, the application informs the database that it's interested in an event, but does not stop other processing to wait for that particular event to occur. The application simply passes the name of a status function to the InterBase Event Manager, and moves on. When the event occurs, the Event Manager calls the function on behalf of the application.

The Sample Application

Now that we've learned what events are and how to use them in InterBase, let's see how to harness this functionality with a sample application.

Keep in mind that any client programming environment can respond to an InterBase event by making the appropriate low-level InterBase calls. Delphi 2, however, makes these low-level calls unnecessary, by providing the IBEventAlerter component found on the Samples page of the Component palette. (Although Delphi 1 doesn't include the IBEventAlerter component, it can be downloaded from Borland's Delphi forum on CompuServe: GO DELPHI.) This component insulates the programmer from the complexity of the underlying InterBase low-level routines.

Phase I

Our sample application will be divided into two phases. In Phase I, we'll create a script to define a trigger in the InterBase database. This trigger will post a `new_record` event each time an insert occurs in the Country table of the IBLOCAL database. This database and associated alias ships with Delphi 2. Simply select *IBLOCAL* from the databases listed in the *Database Name* property of the *TDatabase* component. We will then execute the script using ISQL.

In Phase II, we'll create an application that traps for the `new_record` event. The application will display a message each time this event occurs, informing the user of an insert in the Country table.

Let's create the event trigger. Open any text editor, and enter the following script:

```
/* Connect to IBLOCAL Sample Database */
CONNECT
  "C:\Program Files\Borland\IntrBase\EXAMPLES\Employee.gdb"
USER "sysdba" PASSWORD "masterkey";

/*Create Trigger to post an event */
SET TERM ^ ;

CREATE TRIGGER insert_new_country FOR country
AFTER INSERT
AS
BEGIN
  POST_EVENT "insert_record";
END ^

SET TERM ; ^

/* Commit the changes to the database */
COMMIT;

/* Disconnect from the Database */
DISCONNECT;
```

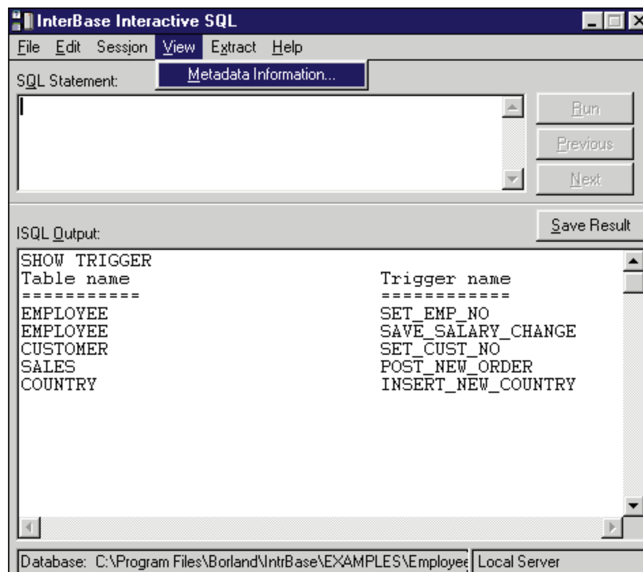


Figure 1: Viewing events in InterBase Interactive SQL.

Our script accomplishes several tasks. Initially, the script connects to the IBLOCAL database using the `sysdba` login and `masterkey` password. Once connected, we create a trigger for the Country table. This trigger is tied to the INSERT event of the table. The database change is then committed, and we disconnect from the database. Make sure you save the script.

To execute this script, start ISQL and select **File | Run an ISQL Script**. In the File Open dialog box, choose the script we created earlier, and select **OK**. Once executed successfully, we can view the event using the **View | Metadata Information** option (see Figure 1). This will display the current list of triggers and their associated tables within the database.

Phase II

Now open Delphi and create a new application (**File | New Application**). Place the following components on the form, as shown in Figure 2:

- Database
- Table
- DataSource
- DBGrid
- DBNavigator
- IBEventAlerter

After you've saved the form and project, attach the DBGrid and DBNavigator components to the DataSource component. Now attach the DataSource component to the Table component. After the DataSource and Table components are connected, double-click on the Database component and, using the Database Component editor, connect it to IBLOCAL under *AliasName* (see Figure 3). The Table component should then be connected to the Database component, and the *TableName* property should point to the Currency table. Then attach the IBEventAlerter component to the Database component.

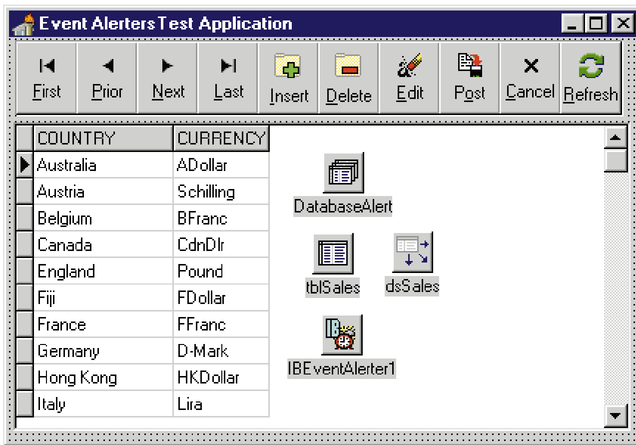


Figure 2: Placing the components on the example form.

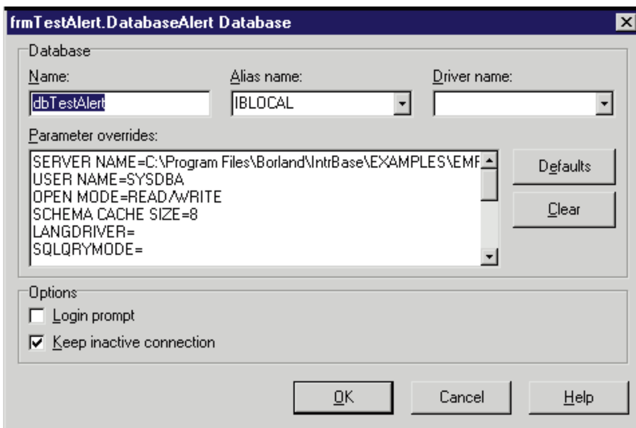


Figure 3: Connecting to the database.

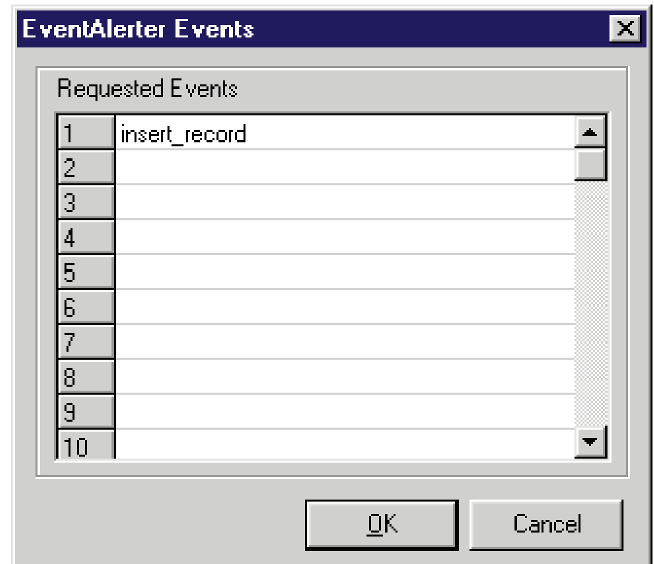


Figure 4: The Events Property editor.

You'll notice the IBEventAlerter component has several key properties. The *Database* property allows us to connect to the Database component that will register a specific event with the InterBase Event Manager. The *Events* property gives us a property editor to specify which events we would like to register with the InterBase Event Manager.

Registration Issues

The Boolean *Registered* property determines whether the events will be registered with the InterBase Event Manager. This property can also be set in code through the use of two methods: *RegisterEvents* will set it to *True*; *UnregisterEvents* will set it to *False*.

Now select the IBEventAlerter component and double-click on the *Events* property in the Object Inspector to invoke the Events property editor (see Figure 4). In this application, we would like the IBEventAlerter component to trap for the `insert_record` event that we created inside the Currency table. Request that the `insert_record` event be registered by entering the event string into the Events property editor.

We would also like our application to register and unregister the `insert_record` event with the InterBase Event Manager. To do this, create the following *OnCreate* and *OnDestroy* events for the form:

```
procedure TfrmTestAlert.FormCreate(Sender: TObject);
begin
  IBventAlerter1.RegisterEvents;
end;

procedure TfrmTestAlert.FormDestroy(Sender: TObject);
begin
  IBEventAlerter1.UnregisterEvents;
end;
```

In addition to the properties, the IBEventAlerter component also defines an event handler called *OnEventAlert*. This event handler provides several key parameters for the developer's use. The *EventName* parameter provides the string of the specific event passed to this event handler; *EventCount* provides the number of posted events; and, *CancelAlerts* allows us to cancel the trapping of events for this particular application.

We want to show a message that will list the name of the event when the occurrence is trapped by the IBEventAlerter component, so we must include the following code in the *OnEventAlert* event handler:

```
procedure TfrmTestAlert.IBEventAlerter1EventAlert(
  Sender: TObject; EventName: string; EventCount: Longint;
  var CancelAlerts: Boolean);
begin
  MessageDlg('A new record has been posted and the ' +
    EventName + ' event was triggered by InterBase',
    mtInformation,[mbOk],0);
end;
```

The sample application is finished. The complete code listing is shown in Figure 5. Compile and test it; each time you insert a record, you should receive the dialog box seen in Figure 6.

Conclusion

As you can see, it's relatively easy to create and register your own events. Learn to take full advantage of this functionality in your next Delphi database application. No longer will you need to code a specified event trapper, or write a slow, polling application.

```
implementation

{$R *.DFM}

procedure TfrmTestAlert.IBEventAlter(Sender: TObject)
  EventName: string; EventCount: Longint;
  var CancelAlerts: Boolean);
begin
  MessageDlg('A new record has been posted and the ' +
    EventName + ' event was triggered by InterBase',
    mtInformation,[mbOk],0);
end;

procedure TfrmTestAlert.FormCreate(Sender: TObject);
begin
  IBEventAlerter1.RegisterEvents;
end;

procedure TfrmTestAlert.FormDestroy(Sender: TObject);
begin
  IBEventAlerter1.UnregisterEvents;
end;

end.
```

Figure 5: The UTestAlert.pas file.

Figure 6: This dialog box is displayed each time a record is inserted.

By combining the IBEventAlerter component with InterBase events, your systems will benefit from not having to poll the database for specified occurrences, leaving the application more processing power. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUN\DI9706AL.

Alexander Le is a Certified Delphi and InterBase Instructor. He has developed systems, lead projects, and provided technical training in stand-alone and Client/Server environments. Alex currently provides Data Warehousing Architecture and Methodology consulting with Platinum Technology, Inc. When not immersed in Client/Server tools, Alex rallies for a spot on the cast of MTV's "The Real World." He may be reached at alle@platinum.com.

Donna Burbank is a consultant with the Information Management Consulting division of Platinum Technology, Inc., specializing in data warehousing solutions. A certified Delphi Client/Server developer, Donna has extensive development experience using Delphi 1 and 2, and InterBase. She can be reached at burbank@platinum.com.

By *Jim Scammahorn*

Interfacing with COM

Delphi 3 Makes Creating COM Objects Easy

The Component Object Model (COM) is a dynamic tool that allows the distribution of task-specific objects, not only inside the same address space of a program running on a single computer, but across address spaces (multiple programs), or even across machine boundaries. These objects can interact with each other, via COM interfaces, in such a way as to appear as a single application, when in reality they are multiple applications running on multiple machines.

DDE (Dynamic Data Exchange), OLE (Object Linking and Embedding), ActiveX, and DCOM are all components of COM, but COM is not any one of these things; it's a much more encompassing concept. COM allows the building of tools that can be added to the Delphi Component palette, brought into a Visual Basic application, or used by MicroFocus COBOL, as well as other programming tools. It was once a Microsoft-specific specification, but was given to the ActiveX Consortium for improvement and cross-platform development. Most other operating systems have made — or are making — COM available on their specific platforms, so that COM objects will run on Microsoft, UNIX, and Macintosh platforms in the not-so-distant future, if not already.

Delphi 3 is, by far, the leader in implementing COM into its available tool sets, allowing developers to create COM objects that revolutionize the way we think about programs and programming. The most basic concept in COM development with Delphi 3 is the COM interface. This article will lead you through the steps to create a simple COM interface with Delphi 3. The concepts provided here can be built upon to create dynamic, distributable application solutions.

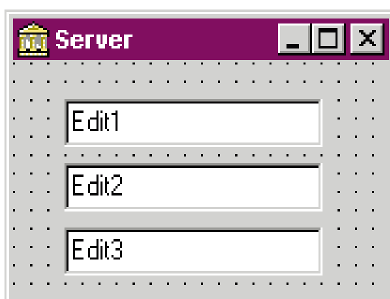


Figure 1: A simple server application.

Interface Construction

The COM interface provides a means to access attributes and methods contained in an in-process (.EXE), or out-of-process (.DLL) server. When a COM interface is accessed from another process or program, the program doing the calling is considered the client, and the process or program being called is considered the server.

In the simple model outlined here, the boundaries of client and server are well laid out, but as reality comes into play, the model becomes more complex. As the complexity increases, the concept of which is a client and which is a server will often depend on the specific functionality being discussed. It's possible, even normal, for the server of one process to be the client of another; two processes can interact with each other as both clients and servers, as needed. There are six basic steps to building a COM interface, and these same steps are repeated for almost all interfaces, regardless of the complexity of the design.

The first step is to build a program or .DLL that will act as the server. This server can be as complex or simple as the overall program requires. In other words, it could be a basic form that will develop into a more complex program as functionality is added, or it could be an entire subsystem of a large pro-

gram written in Delphi. The example in Figure 1 is a simple form with three edit boxes. The interface will manipulate these edit boxes.

The second step is to add the interface to the program. To do this, select **File | New** from Delphi's menu. Then, from the ActiveX page of the New Items dialog box, select the Automation Object icon. After the Automation Object is selected, the Automation Object Wizard appears, as shown in Figure 2.

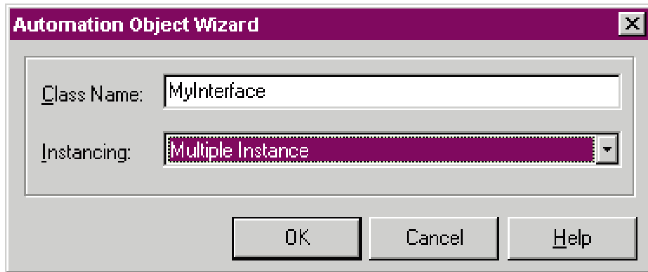


Figure 2: The Automation Object Wizard.

Instancing

In this dialog box, you provide a name for the interface class, and select the type of instancing required. The term "instancing" can be a little confusing at first. It refers to the number of interfaces that can be created for the server. Perhaps a better way of looking at instancing is to determine how many clients can connect to a single instance of the server object. If multiple clients will be attaching to a single instance of the server, then select **Multiple Instances**. If a one-to-one relationship exists between clients and servers (for every call to the interface, a new server is created), select **Single Instance**. This would be used in SDI (Single Document Interface) applications, where the user is allowed to view two instances of the same form with different information on each (two employee forms with different employees on each, for example).

The Internal instancing interface can only be created internally, not by any outside application. This can be useful when an object can be created and used only by the owning object (e.g. an Emergency Contacts object that can be accessed only via the employee object). Once the two options are completed, click **OK** to continue.

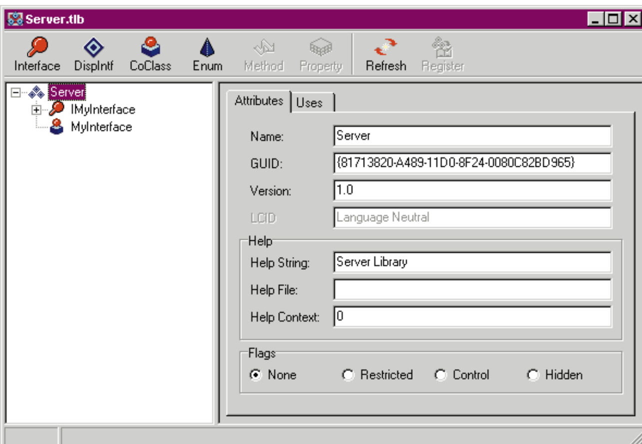


Figure 3: The Type Library Editor.

Type Library Editor

The third step involves adding properties and methods to the type library and the interface object via the Type Library Editor, which opens when the **OK** button is clicked in the Automation Object Wizard. The Type Library Editor can also be opened by selecting **View | Type Library**. The Type Library Editor, shown in Figure 3, is an interface provided by Delphi 3 to the type library of a particular project. The type library is a key component of most COM objects, in which all details of the COM object are defined.

Notice that the name of the type library (shown in the upper-left corner of Figure 3) is the project name (Server) with a .tlb extension. The project name is also used as the base for the tree view. Below the base is the class name that was entered in the Automation Object Wizard dialog box with the "I" prefix added (Delphi uses this prefix for interfaces). A detailed tree view is shown in Figure 4.

The fourth step in building an interface is to add properties and methods to the interface that are exposed to client processes. There are three ways to add these.

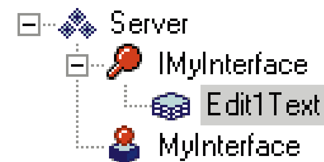


Figure 4: The Type Library Editor tree view.

One way. The first is to click on the interface TreeView Node (**IMyInterface**, in this example), then click on either the **Method** or **Property** toolbar icon. At this point, a name for the method or property can be added in the caption of the new item, just as the `Edit1Text` property was added in Figure 4. Next, click on the **Declaration** edit box of the property or method's **Attributes** section (see Figure 5). In this edit box, you can change the declaration of the property or method from the default (Integer) to the type desired, as well as change a property to a function, and so on.

Another important aspect that is editable here is the **Property Type**. **Read / Write** is the default, but other types are

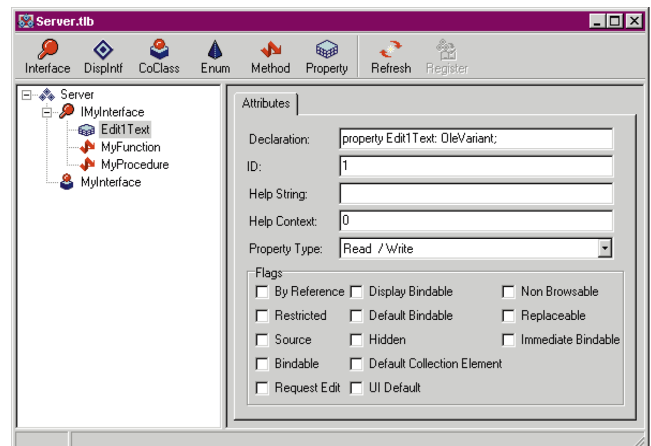


Figure 5: The Attributes page of a method or property in an interface.

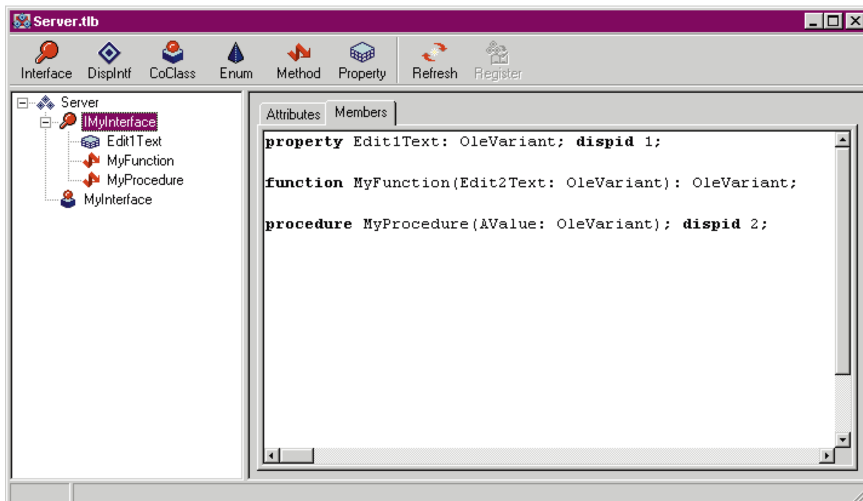


Figure 6: The Members page of the interface tabbed notebook.

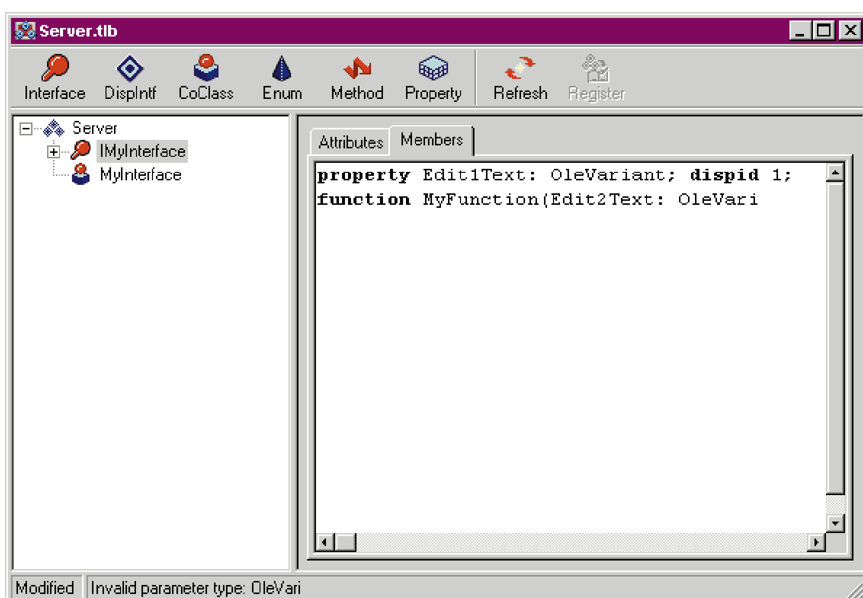


Figure 7: When adding a function to the Type Library Editor, the status bar informs you of problems.

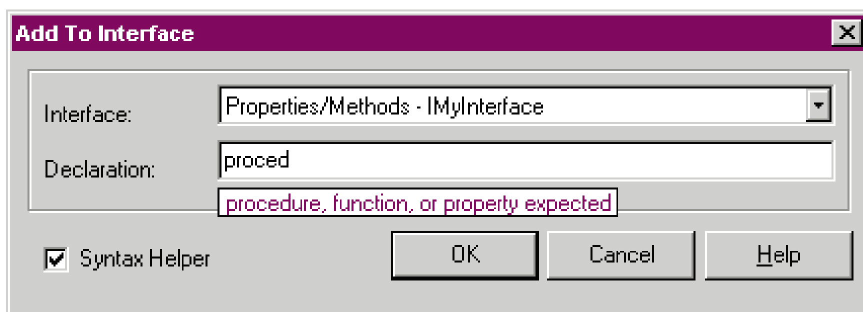


Figure 8: The Add To Interface dialog box.

available via the drop-down list. Be aware that properties and methods can only be of valid COM types. Some of the common types used instead of the standard Delphi types are: WideString instead of string; OleVariant instead of Variant; and WordBool instead of Boolean. I typically use the OleVariant type, because it inherently provides the conversion to the Delphi standard types; however, variant types consume more resources than standard types.

Another way. The second way to add properties and methods is to select the **Interface Node** of the tree view, then select the Members tab, as shown in Figure 6.

In the Members page, you can define properties and methods just as you would in Delphi. The **dispid** doesn't have to be entered; it will be added automatically by the Type Library Editor when the library is saved. While entering information using either of these two methods, the status bar informs you if an invalid type is entered, or other problem occurs (see Figure 7).

Yet another way. The third way of entering properties and methods into the type library is from the Delphi menu. The information entered in Type Library Editor is automatically saved in the .TLB file, but to update the .PAS files, either the Delphi save file icon on the Delphi tool bar must be selected, or the **Refresh** button on the Type Library Editor tool bar must be clicked. To enter a property or method from outside the Type Library Editor, select **Edit | Add To Interface** from the Delphi menu. The Add To Interface dialog box will be displayed (see Figure 8).

Properties and methods can be added to any interface included in this project by simply choosing which interface, and entering the definition. If the **Syntax Helper** check box is selected, hints for the syntax will appear as needed. This functionality is shown in Figure 8. The appropriate declarations and methods will be automatically added to the type library, and to the interface .PAS files.

Saving. After the Type Library Editor is saved, four new files are created:

1) the .TLB type library file (Server.tlb in this example),
2) the *ProjectName*_TLB.dcr Delphi component resource file (Server_TLB.dcr),
3) the *ProjectName*_TLB.pas file (Server_TLB.pas), and
4) the Interface*n*.PAS file (Interface1.pas in this example), which the programmer names when it is saved.

The *ProjectName*_TLB files are generated by Delphi, and contain all the information the COM system requires. The

interface .PAS file is where the interface creator enters code to perform the desired functionality, and is by default, named Unit*n*.pas (*n* being the next available unit number). As new methods or properties are added to the type library, these files are updated automatically.

The fifth step in creating a COM interface is to add code to the methods that have been automatically added to the interface .PAS file. If the methods of the interface are referencing controls or objects contained in another unit, then that unit must be added to the **uses** statement of the interface unit. So in this example, fmServer1 must be added to the **uses** statement of Interface1.pas.

And running. The sixth and final step is to run the application so the COM interface is registered in the Windows registry. The running application is shown in Figure 9.

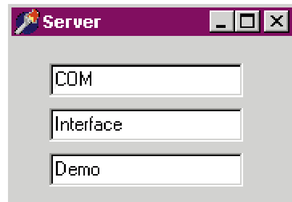


Figure 9: Registering the server application (with the Windows registry) by running it.

The interface can also be registered via the **Register** button on the Type Library Editor toolbar. Each time changes are made to the type library, the application should be reregistered, or the changed functionality may not be available to client objects.

That's basically all there is to creating a COM server interface. The next step is to call that interface from an external program, the client.

Client Construction

After creating the server, there really isn't much to creating the client. Some events will cause the server to run and close, and other events will cause the transfer of the information. All these events are up to the programmer.

To open the server, you must first add the *ProjectName*_TLB.pas unit from the server to the **uses** statement of the unit from which it will be referenced. In this example, Server_TLB is the unit added to the **uses** statement. The *ProjectName*_TLB unit contains all the GUIDs (globally unique identifiers) for the COM interface, as well as the methods and properties that can be accessed. In the example client program (Client.dpr), a separate object does all the communication to the interface (CInterface1.pas). This is by no means mandatory, but it's more object-oriented and therefore, more maintainable because everything for the COM communication link is contained in one location. This client interface has

```
if not Assigned(FCOMServer) then
  FCOMServer :=
    CreateOleObject(ClassIDToProgID(Class_MyInterface))
    as IMyInterface;
```

Figure 10: Creating an instance of the COM Server Interface (found in CInterface1.pas).

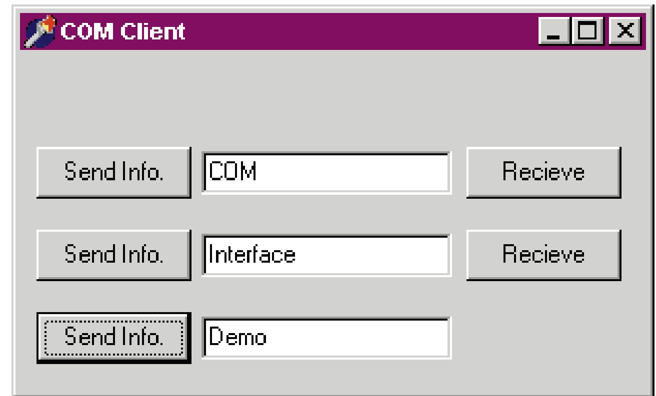


Figure 11: The Client program in operation. The third edit box is write-only, because the interface is a procedure.

a private property that tests if there is an interface already established. If there isn't an interface, one is created using the code in Figure 10.

The *CreateOleObject* and *ClassIDToProgID* methods are provided by Delphi. Other properties in the CInterface1.pas unit transfer information to and from the server. The resulting client program is shown in Figure 11.

Once the events — and reference properties or methods in the client interface object — are created, everything is ready to go. To close the COM interface, you simply assign the interface variable to **nil**, which will close the server. The Windows environment will respond with an error if the COM server is closed by anything other than the application that created it. So, as in good object-oriented programming, the object that created the instance is responsible for freeing the instance. The Delphi environment gives a meaningful warning (see Figure 12) if the user attempts to close the server directly, instead of the Windows standard "Run Time Error 216" dialog box.

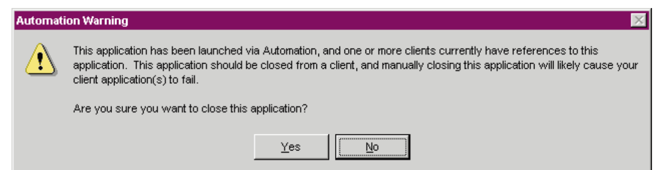


Figure 12: The Automation Warning dialog box.

That's about all there is to creating a simple client program that uses a COM server, but of course, there are other considerations.

More on COM

The best location for information on the details of COM and ways of implementing COM is on Microsoft's Web site at http://www.microsoft.com. A search for COM, DCOM, OLE, or ActiveX on this site will return hundreds of links to documents that cover just about every aspect of the Component Object Model. A more organized listing of COM information is available at Microsoft's OLE Development page (http://www.-microsoft.com/oledev/).

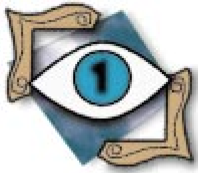
The SiteBuilder Network (http://www.microsoft.com/sitebuilder/default.htm) also has a lot of information concerning ActiveX and COM technologies, as does the Microsoft Developer Network (http://www.microsoft.com/msdn/).

COM objects are not restricted to Delphi, or even a single platform, but can be built using different tools, and distributed across multiple platforms. The opportunities of a truly scalable architecture, independent program subsystems, and dynamic maintenance are now available to all of us. COM is one of the more exciting technologies to come along in some time, but until the advent of Delphi 3, quite difficult and time consuming to implement. Delphi 3 makes COM objects truly simple to create. ∆

Important note: *This article is based on a prerelease version of Delphi 3. Features may differ or be absent in the shipping version.*

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUN\DI9706JS.

Jim Scammahorn has been writing business applications and enterprise solutions in Delphi since shortly after its release, and is currently working for Berkley Information Services. For six years before that, Jim was a research programmer at the University of Arkansas College of Engineering under contract to the U.S. Postal Service, creating image analysis software and hardware simulations using Borland C++, C, and Turbo C. Jim enjoys sightseeing, camping, and hiking with his wife and two children. You can contact Jim at jscammah@ideasign.com.



First Look

Delphi 3

By Robert Vivrette

New Visuals

A Look at Some New Delphi 3 Components

Delphi 3 is now a reality, and many of you are no doubt eager to upgrade to this latest release from Borland. We provided an overview of most of the new features last month. This article takes a quick look at some of the new controls available in the Delphi 3 VCL.

Because this "First Look" was prepared before Delphi 3's official release, I can't say which of these components will be available in the various flavors of Delphi 3. No doubt the Client/Server package will have them all, but some of them — particularly the database controls — might not be available in the less expensive packages, such as the Desktop version.

Splitter

Many of you have probably used various applications, such as Microsoft Excel, that allow you to divide the screen into independent regions, the size of which are adjustable by the user. The technique involved here is the use of what is called a Splitter Bar. Delphi 3 includes this new component, which lets you adjust the size of two (or more) components that have an *Align* property.

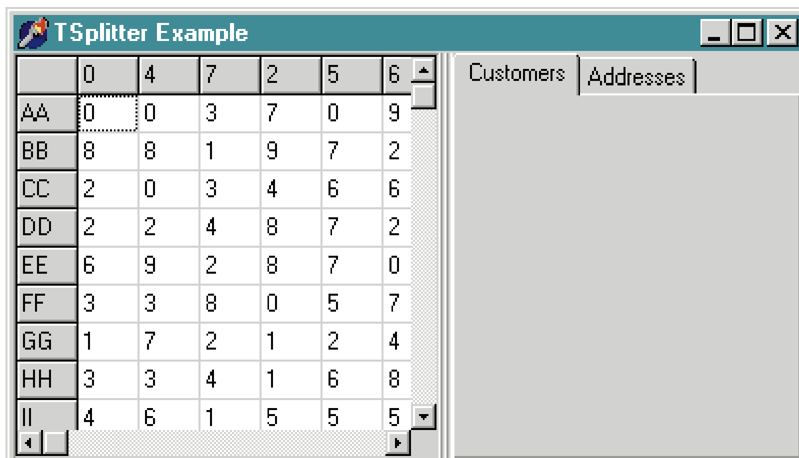


Figure 1: An example of the Splitter component.

In Figure 1 you'll notice a beveled line between the StringGrid component on the left and the TabControl on the right. This is the Splitter component, and it can be grabbed and moved from side to side. Achieving this is simple. First, I placed a StringGrid on the form and set its *Align* property to *alLeft*. Then I dropped the Splitter on the form and it "snugged" itself up against the grid. Last, I dropped the TabControl on the right and set its *Align* property to *alClient*.

Although Splitter only works with components that have an *Align* property, this really isn't a restriction. If you need a "splitter" between two components that don't have *Align*, you can always drop a borderless panel on each side.

The Splitter component also has properties that allow you to control its width and beveled appearance, as well as the minimum distance that the bar will be allowed to go in either direction.

CheckListBox

The CheckListBox component is simply a variation on a standard list box control. The difference is that each item in the list has a check box associated with it. You gain access to the checked states of the items by means of the *Checked* array property. You can also gray out individual items with the *States* array property.

The example in Figure 2 simply fills a normal list box (on the right) with those items that have been checked in the CheckListBox

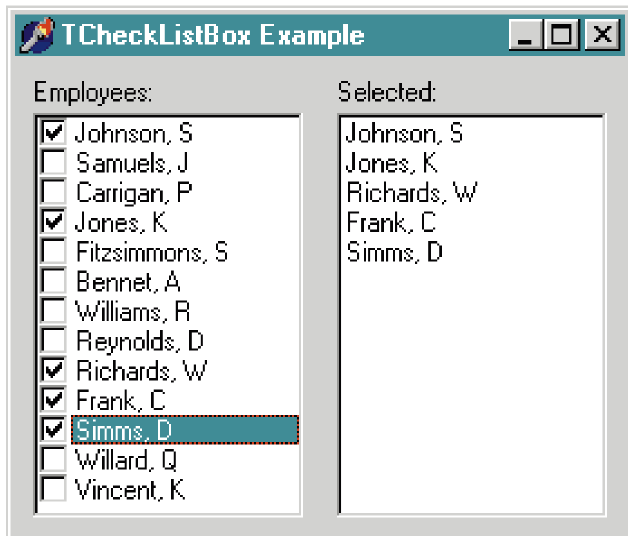


Figure 2: The CheckListBox component in action.

component on the left. Before the availability of this control, developers could simulate this kind of behavior by allowing a list box to have multiple selections. However, forcing a user to hold down Ctrl and/or Shift while selecting items is not the best way to accomplish such a straightforward task.

As an added feature, clicking on the list item itself simply selects the item. To change the state of the check box, you click only on the check box area. This logical separation of functions allows you to move through the list with the mouse, without changing the state of its associated check box. I have worked with many controls that don't do this, and it's a pleasure to work with one that does it right.

Animate

Every time a user of Windows 95 (and the latest version of Windows NT) copies a file, or empties the trash can, they see the new AVI animations built into the operating system. Now Delphi programmers can work these visual effects into their applications by means of the Animate component.

In a nutshell, Animate plays .AVI files. However, I believe its more common use will be to play the operating system animations. It has a property called *CommonAVI* that you can set to any of the values listed in Figure 3. Animate then finds the animation (in a resource in the operating system) and plays it. Pretty handy!

The control has properties allowing you to play only selected frames, as well as to specify the number of loops through the animation. Those currently using the bulky MediaPlayer control to play .AVI files will appreciate the more functionally focused Animate.

DateTimePicker

Modern applications continue to go out of their way to simplify user input. Programs such as Quicken now give you the option of using a small pop-up calendar to enter a date. Not only does choosing a date from a pop-up calendar eliminate typing the date, it eliminates the problem of invalid input.

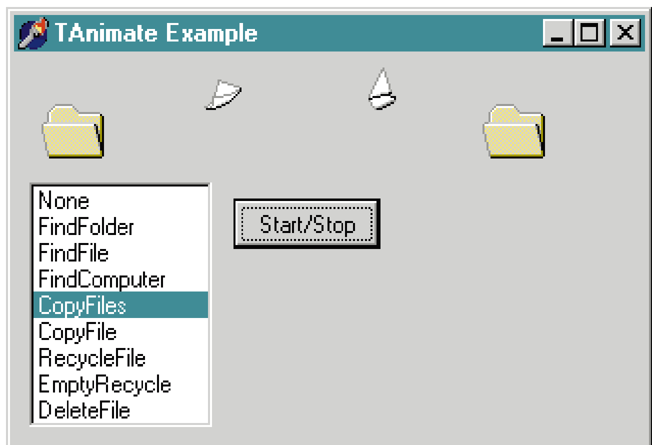


Figure 3: The Animate component can add visual effects to your applications.

When a user can enter any number, the application designer must validate the input. With a component like DateTimePicker, that validation is built in, so the application designer can be assured of valid data.

Note that in Figure 4, the DateTimePicker component looks like a combo box with a date in it. However, when the down arrow is clicked, a pop-up calendar appears. The user can then navigate months with the mouse or keyboard. Today's date is shown circled in red, and the selected day is highlighted with a small, colored oval. After selecting the date, the pop-up disappears. There are also properties that allow you to modify the colors used in the pop-up calendar.

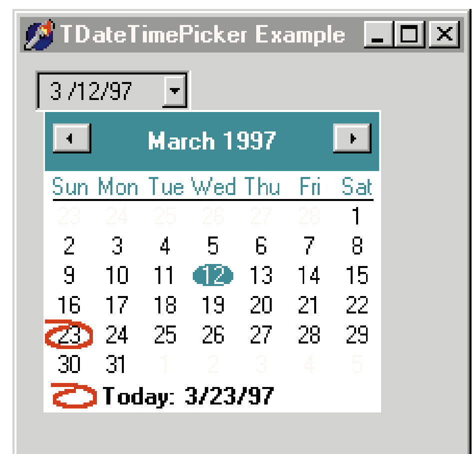


Figure 4: Stay organized with the DateTimePicker component.

The DateTimePicker component can also display and edit time values, but the method of editing is a bit different. The control has up/down arrows on the right, allowing the user to increase or decrease the hour, minute, second, or AM/PM status independently within the field. Keyboard support is also built in, so you can select the minutes portion of the time and hit the + and - keys to increase and decrease it appropriately.

DBRichEdit

A while back, I wanted to save RTF data in a BLOB field; I went through all sorts of fits trying to get it to work. Wouldn't you know that as soon as I came up with a solu-

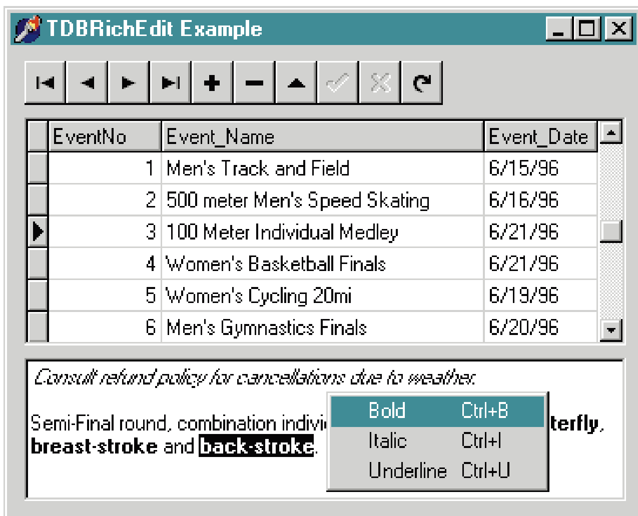


Figure 5: A data-aware version of RichEdit, DBRichEdit lets you save and restore RTF data in a BLOB field.

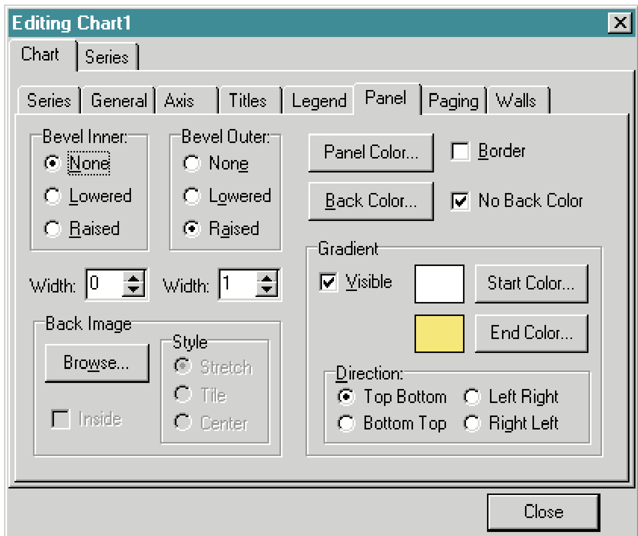


Figure 6: The properties editor for the Chart component.

tion, Borland went and made a data-aware RichEdit control. It's okay, though — I didn't really like my solution anyway!

DBRichEdit does pretty much what you would expect. It's a data-aware version of the RichEdit control. In the example shown in Figure 5, I tied a memo BLOB field to an RTF from one of the DBDEMOS tables supplied with Delphi. Then I added a pop-up menu that set the appropriate *SelAttributes* for the RichEdit. Even though the BLOB memo field in the database was originally plain text, it saved and restored the RTF data as you would expect.

Chart

Where do I start? Clearly, Chart is one of the more configurable controls in the Delphi VCL. Developed by a company in Spain, this control's real name is TeeChart, and it's compatible with all versions of Delphi (available in separate 16- and 32-bit versions). Delphi 3 includes both data-aware and non-data-aware versions of Chart, and information is provided in its associated Help file on how to obtain the Professional versions of both.

Because they're so highly configurable, Chart and DBChart can be a bit daunting at first, but you'll soon be making charts like a pro. Developers can adjust the graph types (there are eleven 2D and 3D styles), numbers, line styles, data points, titles, legends, etc. (see Figure 6). You can even give the chart a graphic to put on the background (or just inside the chart), or even create gradient-filled backgrounds. If you want to see what the chart might look like, but don't have the numbers ready, you can have it populated with random values (see Figure 7).

ToolBar and CoolBar

Perhaps the most interesting of the new Delphi 3 components are ToolBar and CoolBar. ToolBar is pretty much what you would expect it to be. It's a bar that allows you to insert ToolButtons for easy access to common program functions. The CoolBar however, is a spin-off of one of the new Microsoft controls available in the latest version of Internet Explorer.

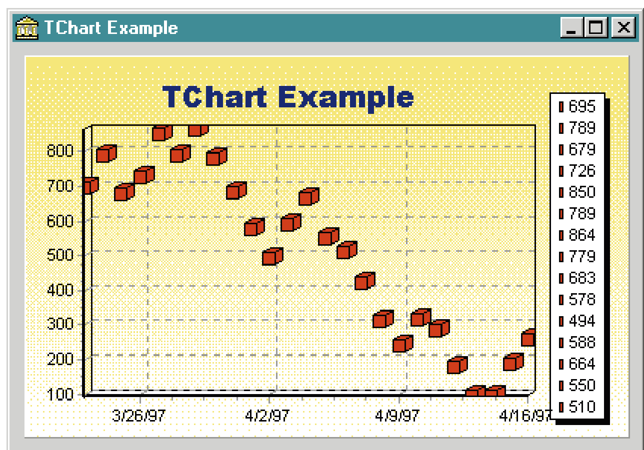


Figure 7: An example of what the Chart component can produce.

Figure 8 shows a sample Web browser application that Borland put together. The top portion of the window shows a CoolBar with four CoolBands. The first band holds a ToolBar with the **Back**, **Forward**, **Stop**, and **Refresh** ToolButtons. The second band holds a combo box defining the URL that the Web browser is showing. The third CoolBar is holding an Animate component (the "Cool" logo). When the browser is connected to the Web, this AVI animation cycles through its frames to show the browser is active. A fourth CoolBar holds another ToolBar/ToolButton combination defining links to common Web sites.

In case you're not familiar with the way a CoolBar works, I'll run through it briefly. Notice a bevel pattern, indicating a grab point, on the left side of each CoolBar. The user can slide these areas back and forth to redefine the amount of space the CoolBar will occupy. You are not limited to simply sliding a CoolBar left or right. For example, you can drag the **Address** CoolBar down to the area occupied by the **Links** CoolBar. The two of them would then share the space in the second row. This makes it very easy for users to modify the look and behavior of an application.

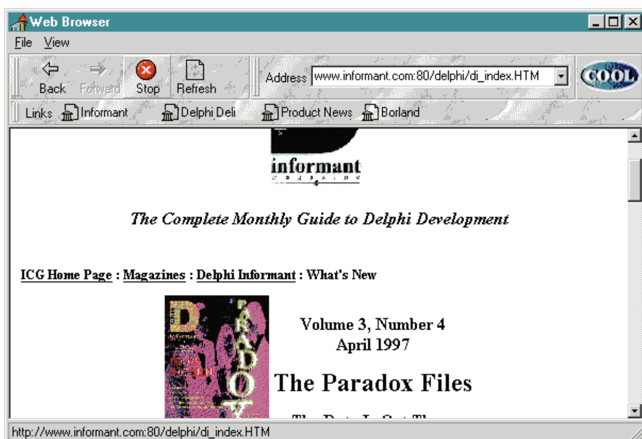


Figure 8: The ToolBar and CoolBar components add flexibility and functionality.

A CoolBar can hold all sorts of controls; think of it as an adjustable panel. There's also a property for setting a background image (as in the marble texture shown in Figure 8). The ToolBar component improves application resource management by allowing you to attach an ImageList component to it with all the graphics you'll be using. Then the individual ToolButtons that sit on the ToolBar simply reference a particular button by means of an index into the ImageList. You can even define buttons that "snap up" when the mouse passes them.

Conclusion

Hopefully this brief look at some of the new controls in Delphi 3 will give you an idea of what to expect as you switch to the new version. Although few of these controls will completely change the way you program, the enhancements will undoubtedly be appreciated by your users.

Important note: *This article is based on a prerelease version of Delphi 3. Features may differ or be absent in the shipping version.*

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUN\DI9706RV.

Robert Vivrette is a contract programmer for Pacific Gas & Electric, and Technical Editor for *Delphi Informant*. He has worked as a game designer and computer consultant, and has experience in a number of programming languages. He can be reached via e-mail at RobertV@compuserve.com.

By Bill Todd

Deployment: Part II

Deploying Delphi 2 Applications with InstallShield Express Professional

If you need more power and flexibility than InstallShield Express (which ships with Delphi 2) provides, one option is to purchase InstallShield Express Professional, its commercial big brother.

You'll find the upgrade easy and intuitive; both use the same user interface and the same steps for building your setup. The user interface and basic steps for building a setup were covered in Part I of this series. This article will focus on the following additional features of InstallShield Express Professional:

- A printed user's manual
- Support for creating 16-bit Windows 3.x setups
- Support for creating an installation as a single self-extracting EXE
- Support for installing ODBC drivers and OCX controls
- Support for installing the local InterBase server
- Support for internationalized setups
- Support for calling functions in DLLs
- Support for launching EXEs in the background
- Support for Visual C++, Visual Basic, Borland C++, Paradox, and Delphi
- Support for changing system files, such as WIN.INI, SYSTEM.INI, PROTOCOL.INI, AUTOEXEC.BAT, CONFIG.SYS, and application-specific .INI files
- Support for selecting file groups from a component during a custom installation
- An unlimited number of billboard graphics
- 60 days of free technical support

The first difference I noticed about InstallShield Express Professional is that it comes with one of the best user's manuals I've seen. The manual is as thick as the Delphi 2 *User's Guide*, and takes you through each phase in detail.

16- and/or 32-bit. While InstallShield Express Professional runs only under Windows 95 or NT, it will generate both 16- and 32-bit installation setups. When you start a new project, the New Project dialog box (see Figure 1) contains a pair of radio buttons that allow you to select whether to create a 16- or 32-bit setup. If you have an application that must be installed under both 16- and 32-bit operating systems, you will probably want to create a 16-bit setup *and* a 32-bit setup. One reason for this is to ensure that under a 32-bit operating system, users will have the option to uninstall the program from the Control Panel's Add/Remove Software applet.

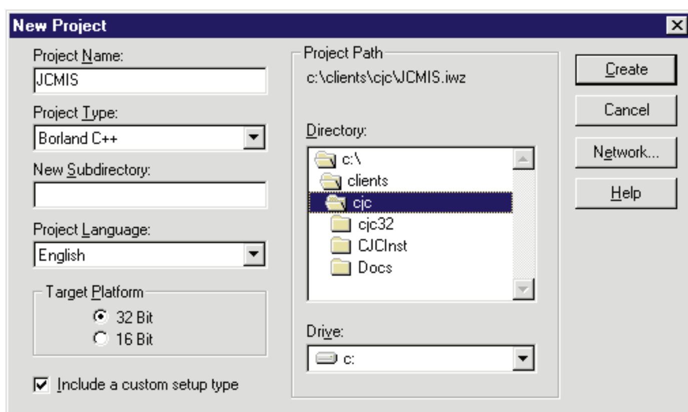


Figure 1: The New Project dialog box.

English, German, or French. The New Project dialog box in InstallShield Express Professional also enables you to select the

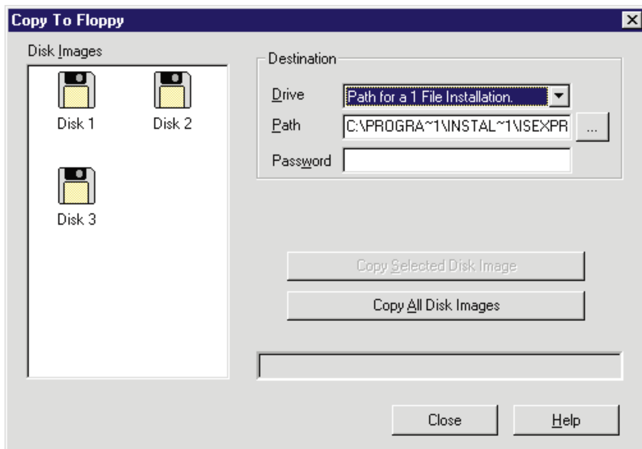


Figure 2: The Copy To Floppy dialog box.

installation language from a drop-down list. Currently English, German, and French are available. This list will be expanded in the next version. All the dialog boxes the user sees during installation will be in the selected language. In addition, InstallShield Express Professional will make any necessary language changes to the registry entries it creates.

Single-file executables. In addition to creating setup diskettes, InstallShield Express Professional allows you to create a single self-installing EXE file. This can be useful if you're distributing on bulletin boards, CD-ROM, or from your Web site. This is also a handy option if you have many network users who need to install your application or update. Simply create an EXE file and place it on the file server. Users can then install directly from the server; no need to distribute diskettes to everyone. The file can also be password protected.

To create a single-file EXE, select Copy to Floppy (as you would to make a set of installation diskettes). Next, select Path for a 1 File Installation from the Drive drop-down list (see Figure 2). Change the Path to the directory where the setup file will be created. Finally, enter a Password, if necessary, and select Copy All Disk Images to create the setup file.

ODBC and Local InterBase installation. In addition to its ability to install the BDE, SQL Links drivers, and ReportSmith Runtime, InstallShield Express Professional has the ability to install ODBC drivers and the Local InterBase Server. Selecting General Options under Select InstallShield Objects for Delphi displays the dialog box shown in Figure 3.

Checking the ODBC check box displays the first screen of the ODBC installation wizard shown in Figure 4. The list of available ODBC drivers is dynamically built from the drivers installed on your system. Simply check drivers you want to install with your application. The ODBC wizard then leads you through setting the attributes for each driver.

When installing the Local InterBase server, you have three options. In addition to installing the server, you can install the Windows and command-line tools for InterBase. If the target system already has Local InterBase installed, the

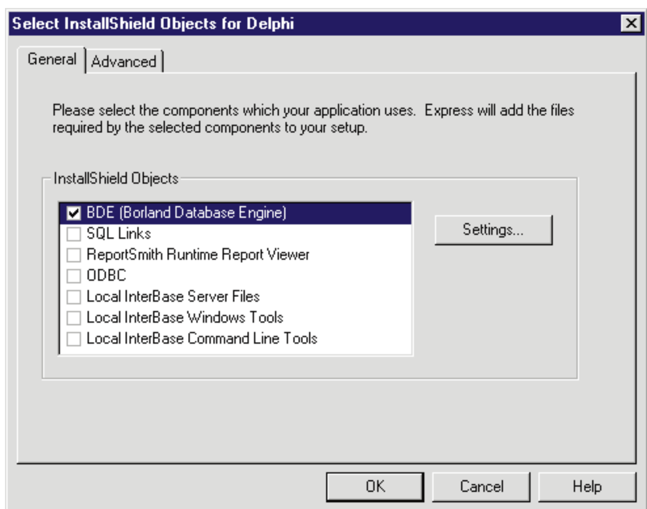


Figure 3: The Select InstallShield Objects for Delphi dialog box.

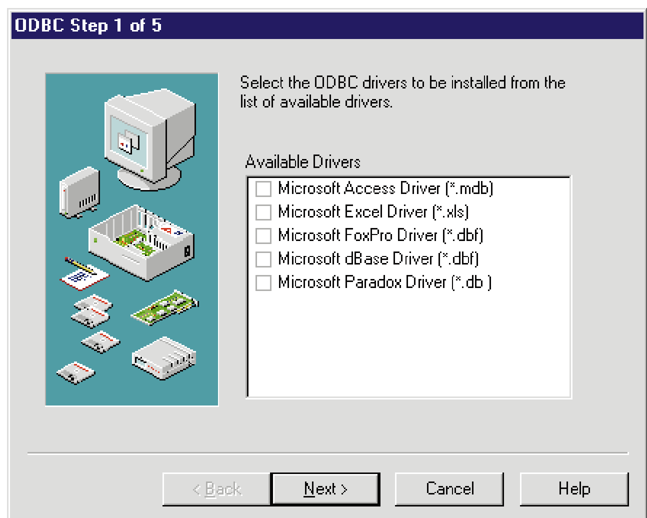


Figure 4: Installing ODBC drivers.

InstallShield installation will not overwrite the existing security database ISC4.GDB, because this would make the existing databases on the target machine inaccessible to users. Your setup can also register any DLL or OCX it installs.

Extensions

As useful as the other features of InstallShield Express Professional are, by far the most valuable and powerful is the ability to call functions in a DLL you write and to run a background EXE as part of the installation process. With these capabilities you can, as part of the installation, do anything that you can program.

Any DLL or EXE you call as part of your setup process is called an *extension*. To add an extension, click the Express Extensions button. The Extensions list on the left side of the Express Extensions dialog box in Figure 5 shows the extensions that have been added to the setup. The list on the right includes all the dialog boxes displayed during the installation process. Clicking the New button to add an extension displays a dialog box that enables you to tell Express whether the extension is a DLL or EXE.

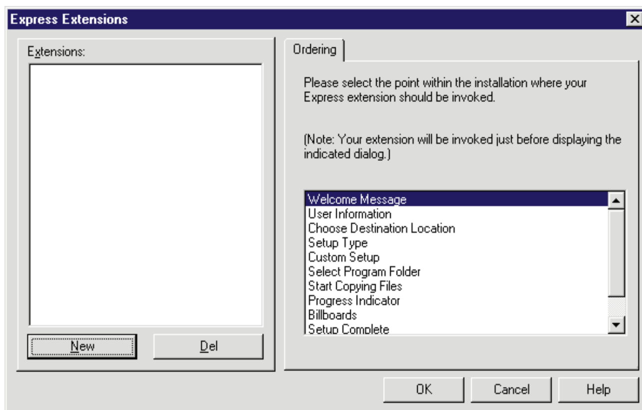


Figure 5: The Express Extensions dialog box.

When? Now that your extension has been added to the Extensions list, you must indicate at what point in the installation you want your extension to run. Do this by selecting the dialog box (in the list on the right) that's displayed before the extension is to be called. Now that you have added an extension, a Settings tab will be displayed next to the Ordering tab.

What you see when you click the Settings tab depends on whether the new extension is a DLL or EXE. For a DLL, you must enter the name of the DLL, and the name of the function to call. Express will pass five parameters to the function:

- the window handle of the Express main window
- the source directory path
- the support directory path (temporary directory for the installation files)
- the installation directory path
- a parameter reserved for future use

These parameters give your DLL function access to all the directories involved in the installation. Your DLL function can return a value of zero to terminate the installation process, or any other value to indicate the installation should continue. Because the manual provides sample 16- and 32-bit DLL functions in C only, Figure 6 shows the shell of a function in a Delphi DLL that can be called by InstallShield Express Professional.

Note that for a 32-bit installation, Express expects your extension DLL routines to use the **stdcall** directive; and for a 16-bit installation, you must use the **cdecl** calling convention.

If your extension is an EXE file, the Settings page prompts you for the name of the program to run, and any command-line parameters to pass to the program. You can also check a check box that instructs InstallShield Express Professional to wait for the program to terminate before continuing the installation. However, your program must have a window handle for this to work.

Where? You can put your extension files in one of three locations. The first is in the file group with your program

```
unit Install;

interface

uses
  BDE, Windows, DB, SysUtils, FileCtrl;

function SetBDEParams(ISWindow: HWND;
                      szSourceDir,
                      szSupportDir,
                      szInstallDir,
                      szReserved: PChar): Byte; stdcall;

implementation

function SetBDEParams(ISWindow: HWND;
                      szSourceDir,
                      szSupportDir,
                      szInstallDir,
                      szReserved: PChar): Byte;
begin
  Result := 1;
end;

end.
```

Figure 6: An Express extension DLL function.

files. This means the extensions will be installed in the same directory as your program files, and at the same time as your program files. This also means you cannot run your extension until your application files are installed, and that the extension files will not be automatically deleted.

The second alternative is to include your extension files in Express' _SETUP.LIB file. If you do this, your extension files will be copied to the temporary support directory as the first step in the installation process, and they will be automatically deleted at the end of the installation.

The third option is to place your extension file in uncompressed form on the first disk of the set. This makes your extension available before any file-copy operation begins, but might force the user to swap diskettes if the extension is called later in the installation.

Conclusion

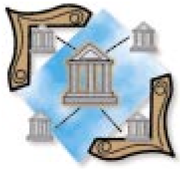
Using extensions, you can resolve the BDE installation problems described in Part I of this series. You can call a DLL function that uses the BDE API function, **DbiOpenCfgInfoList**, to set the Local Share property in the BDE configuration file to True. You can also use it to examine the value of the Paradox driver's Net Dir parameter, and — if the value is null — create a subdirectory below your application's installation directory and set the Net Dir path to that directory.

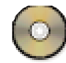
If you find that InstallShield Express for Delphi doesn't provide some of the features you need, InstallShield Express Professional may be the answer. It offers excellent documentation and a good balance between flexibility and ease-of-use. Of particular value is the ability to call DLL functions and run programs in the background as part of the installation process. This lets you do anything you can code, at any point during the installation.

If you have worked with InstallShield Express for Delphi, you will find that you already know how to use InstallShield Express Professional. They have the same interface; the professional version simply offers more options. Because InstallShield Express Professional provides all the Delphi-specific functionality found in InstallShield Express for Delphi, you can remove the latter from your hard drive, and handle all your installations with the former. If InstallShield Express for Delphi doesn't meet your needs, I highly recommend InstallShield Express Professional. ∆

InstallShield Corp.
900 National Parkway, Ste. 125
Schaumburg, IL 60173-5108
Price: US$395
Sales: (800) 374-4353
Phone: (847) 240-9111
E-Mail: info@installshield.com
Web Site: http://www.installshield.com

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 is also a nationally known trainer and has been a speaker at every Borland Developers Conference and the Borland Conference in London. He can be reached on the Internet at 71333.2146@compuserve.com or at (602) 802-0178.



Delphi 2 / Object Pascal / Excel for Windows / Visual Basic for Applications 

By Ian Davies

Automated Excel

Creating OLE Automation Clients: Part II

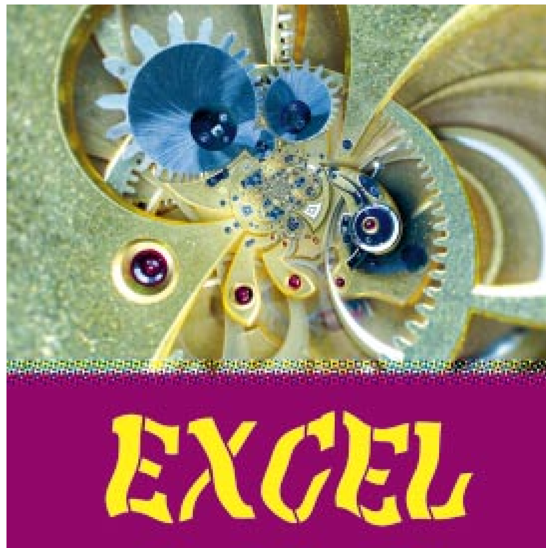
Last month, we looked at using Microsoft Word as an OLE automation server from a Delphi 2 application. This month, it's the turn of Microsoft Excel.

It's a more complex process to use Excel as an OLE Automation server than Word, because it exposes 77 automation objects (in version 5) compared to the *one* Word Basic object exposed by Word 6. However, this provides more power and flexibility to the client application by offering a greater degree of control over the server.

As I mentioned last month, it's important to remember that although I refer to Excel specifically in this article, these principles can be applied to any OLE automation server, and can be easily converted to suit your favorite spreadsheet provided that it supports OLE automation.

Excel's Object Hierarchy

You can think of objects in Excel as being similar to those in Delphi's VCL, in that they can have properties, methods, and events. The top object in the hierarchy is the Application object, which acts as a parent to all other objects, similar to Delphi's *TObject*. Excel provides external applications (i.e. those accessing objects using OLE automation) a direct interface to three of its objects — the Application, Chart, and Worksheet objects — with all other objects being accessed via these. Because the Application object is the ancestor of all other objects, you can access the Chart object, for example, via the Application object rather than directly, if you want.



Excel's object hierarchy also makes extensive use of collections. The concept of a collection is similar to a Pascal record; a group of related objects can be referenced as a whole or individually. For example, the workbooks collection contains a list of all open workbooks, and can be referenced as a whole (e.g. `Workbooks.Close` to close all open workbooks), or individually (e.g. `Workbooks[2].Close` to close only the second workbook in the collection).

Discussing every Excel object is beyond the scope of this article. We'll focus on the objects relevant to implementing the fundamental operations performed when using a spreadsheet.

Fundamental Principles

Here's a brief introduction to manipulating the Excel objects that an OLE automation programmer commonly works with. An application can contain many workbooks,

and each workbook can contain many worksheets. An OLE automation session is initiated with the Application object:

```
xlApplication := CreateOLEObject('Excel.Application');
```

where *xlApplication* is a Variant declared to hold the instance data of the Automation object. No workbooks or worksheets are initially loaded; you have to do that yourself.

To create a new, blank workbook, call the *Add* method of the Workbooks object:

```
xlApplication.WorkBooks.Add;
```

This will create a new workbook with the default number of worksheets (normally 16); alternatively, you could use the *Open* method to open a file that already exists.

Two properties of the Application object are *ActiveWorkbook* and *ActiveSheet*, which return a reference to the active workbook and active worksheet, respectively. Worksheets other than the active one can be referenced, but must be fully qualified:

```
xlApplication.Worksheets['Sheet1'].
  Cells[5,3].Font.Size := 14;
```

As you can see, the worksheets collection has a *Cells* method, which returns an object (of type Range) referenced by its parameter, in this case the fifth row and the third column (cell C5). The *Range* method also returns an object (of type Range) which can be used to specify a contiguous block of cells. For example, this statement sets the font to 14 point for every cell comprising the range A1:F10:

```
xlApplication.Worksheets['Sheet1'].
  Range['A1:F10'].Font.Size := 14;
```

The Range object (which includes the Cells object of type Range) can be used to modify the contents of individual cells or groups of cells through the use of the *Formula* or *Value* properties. This statement, for example:

```
xlApplication.Range['A1:H8'].Formula := '=rand()';
```

fills the cells between A1 and H8 on the current worksheet, with the Excel function that generates random numbers. Since no explicit worksheet is named, the current (active) worksheet is implied. The current worksheet can be changed by calling the *Activate* method of the Sheets collection, indexed either with a name:

```
xlApplication.Sheets['Sheet2'].Activate;
```

or a reference:

```
xlApplication.Sheets[3].Activate;
```

The current workbook can be changed in the same way:

```
xlApplication.Workbooks['MySheet.xls'].Activate;
```

Excel macros written in Visual Basic for Applications (VBA) language can be executed by calling the *Run* method of the Application object:

```
xlApplication.Run['MacroName'];
```

As stated previously, when Excel is initiated as an OLE automation server, it's hidden from the user by default. You can control its visibility with the Application object's *Visible* property:

```
xlApplication.Visible := True;
```

For further information on these and other features provided by VBA, the VBA_XL.HLP file provided with Excel is an excellent place to start. Also, using Excel's macro recorder facility provides a head-start in determining which objects are used for which purpose.

A Functional Example

Let's look at another example. This example creates a new workbook, inserts a formula that accesses an Excel function, and displays the result. The function we'll use calculates the future value of an investment, when given various details about the investment. Excel's **FV** (future value) function takes five parameters:

```
=FV(InterestRatePerPeriod, NumberOfPeriods,
    PeriodicInvestment, LumpSumInvestment, PaymentDate)
```

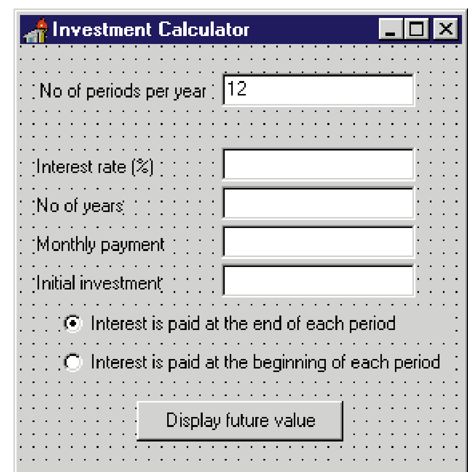


Figure 1: The demonstration form.

Create a new Delphi project, and place five Label components, five Edit components, two RadioButton components, and a Button component on the form, as shown in Figure 1. Next, declare a Variant called *xlApplication* as a public to the form which will be used to store the instance data of the Automation object:

```
public
  { Public declarations }
  xlApplication: Variant;
```

Because we're using OLE automation, we also need to add the OLEAuto unit to the form's **uses** clause.

We'll create the Automation object in the *FormCreate* method. This will result in Excel being loaded into memory, but remaining hidden from view. The object will be unloaded and freed in the *FormDestroy* method. Both methods are shown in Figure 2.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  try
    // Attempt to create an OLE Automation link with
    // Excel and store its instance data in the
    // xlApplication variant.
    xlApplication := CreateOLEObject('Excel.Application');
  except
    MessageDlg('Could not start Excel',mtError,[mbOK],0);
  end;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  try
    xlApplication.Quit;  // Quit Excel.
  except
    MessageDlg('Could not unload the server',
      mtError,[mbOK],0);
  end;

  // Free the variable holding the instance data of Excel.
  xlApplication:=unAssigned;
end;
```

Figure 2: The form's *OnCreate* and *OnDestroy* methods.

To send the function to Excel, it must first be built into a string variable based on the values of the edit boxes and the radio buttons. I've implemented this using Delphi's *Concat* function:

```
// Build the Excel FV function.
FuncString := Concat(
  '=FV(',                           // Formula preceded by '='
  FloatToStr(IntRate), ', ',        // Interest rate
  IntToStr(NoPayments), ', ',       // Number of payments
  Edit4.Text, ', ',                 // Payment amount
  Edit5.Text, ', ',                 // Initial investment
  ')');
```

This simply adds the values of the relevant components to the *FuncString* variable in the format expected by Excel.

Before the function can be passed to Excel, a new workbook must be temporarily created to hold it. The *Add* method of the Workbooks object is called to achieve this:

```
xlApplication.WorkBooks.Add;
```

The string containing the function is then assigned to the *Formula* property of the Cells collection of the active sheet:

```
xlApplication.ActiveSheet.Cells[1,1].Formula := FuncString;
```

where it's immediately evaluated. The result is read back, by examining the relevant cell's *Value* property:

```
InvestValue :=
  StrToFloat(xlApplication.ActiveSheet.Cells[1,1].Value);
```

This is a simple function that would be straightforward to create in Delphi, thereby removing the overhead of using Excel. However, rather than creating a new workbook to hold your temporary functions, you could open a previously prepared workbook that may contain many, considerably more complicated, functions. Values could then be passed to this workbook,

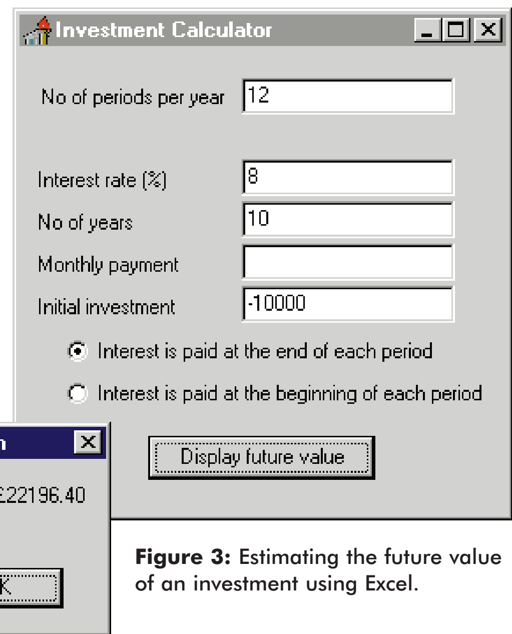


Figure 3: Estimating the future value of an investment using Excel.

and the results, which may now be difficult or impractical to recreate in Delphi, can be retrieved in exactly the same way as in this example. We're simply using Excel to do what it does best.

In Figure 3, the initial investment is shown as a negative figure, because it indicates an outgoing cash transaction. The result, however, is a positive figure, because it's the projected income received from the investment.

Leveraging Excel Pivot Tables from Delphi

Pivot tables, available in Excel 5 and later, provide a dynamic view of a list of data that a user can interact with to produce customized summaries. A pivot table consists of a number of fields that can be classed as either data fields, row fields, column fields, or page fields. Data fields contain the data to be summarized (using all the standard summary operators, such as sum, count, etc.); row fields summarize the data in that field extending down the page; column fields summarize the data in that field extending across the page; and page fields allow the user to apply a filter to the information summarized in the table.

As a demonstration, we'll summarize the MASTER.DBF table (a dBASE table) provided with Delphi 1 and 2, which is available as part of the DBDEMOS examples. This table is used by the Stocks demo in Delphi 1, and contains various details concerning the sale and purchase of shares in various fictitious companies. This table will be opened in Excel, and a pivot table will be created based on it. By manipulating the fields in the pivot table, we will be able to identify the total value of shares per investment rating, the total value for each risk band allocated, and the total value broken down between the two. The operations will be performed in Excel, and the results will be read back into our Delphi application and displayed in a StringGrid component (see Figure 4).

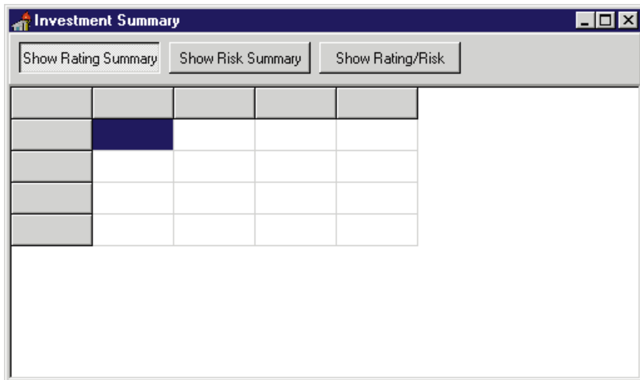


Figure 4: The Pivot Table example in design mode.

Figure 5: Results of an operation on an Excel pivot table.

The link to Excel is established in the *OnCreate* event of the form, followed by a call to the *Open* method of the Excel Worksheet object to open the MASTER.DBF table. The pivot table is created by calling the Pivot Table Wizard, and passing various parameters concerning the layout and position of the source and destination tables:

```
xlApplication.Sheets['Sheet1'].PivotTableWizard[
  SourceType        := xlDatabase,
  SourceData        :='Database',
  TableDestination  :='R3C1',
  TableName         := 'PivotTable1'];
```

Contained on our main form are three SpeedButton components which are used to send the relevant instructions to Excel to manipulate the pivot table. This is achieved by changing the orientation of the relevant fields:

```
xlApplication.Sheets['Sheet1'].PivotTables['PivotTable1'].
  PivotFields['RISK'].Orientation    := xlPageField;
xlApplication.Sheets['Sheet1'].PivotTables['PivotTable1'].
  PivotFields['RATING'].Orientation := xlRowField;
```

The first line removes the RISK field from the pivot table, and redeclares it as a page field, which is being used in this instance as a temporary holding area. The orientation of the RATING field is changed into a row field, thereby generating a summary based on each unique value contained in the RATING field. The results produced are shown in Figure 5. The other buttons simply contain different combinations of page, row, and column fields to display the appropriate summary.

The Possibilities

This is a simple example based on a simple table, the results of which can easily be recreated in SQL. However, the main benefit of pivot tables is that they offer the user a means of interactively manipulating the views, and therefore, the behavior of the data in the table. By removing Excel's toolbars, status bar, and formula bar, and modifying the title in the title bar, a pivot table can be made visible and presented to the user as a seamless part of your application. Excel could be exposed as a rather clever dialog box that appears as an integral part of your application, giving the impression that a great deal of time and effort has gone into its creation when, in reality, all that's involved is a few simple OLE automation calls to a predefined series of objects.

There's incredible potential for manipulating and customizing Excel through its OLE automation interface. You're able to change practically every aspect of Excel's appearance to suit your needs. Whether you choose to display Excel to your users and allow them to interact with it directly (and retrieve the results using OLE automation into Delphi), or whether you communicate entirely through code, Excel and VBA provide you with the tools necessary to complete the job.

That's it for this month. In the next — and final — part of this series, I'll cover how to use Microsoft Access, and the various components of its databases, from a Delphi application. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUN\DI9706ID.

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.

Paradox / BDE / Delphi

By Dan Ehrmann

The Paradox Files: Part III

Understanding Primary and Secondary Indexes

As I said last month, Delphi developers use the Paradox file format every day, yet the Delphi documentation offers little information about it. To help fill that gap, the first two articles in this series explored the internals of Paradox table (.DB) files: table structure, record and block management, field types, and record size calculation. In this article, we'll examine primary and secondary indexes.

The Paradox Index

An index is a sorted list. When you create an index on a Paradox table, the BDE creates a separate file containing the value of that field for every record, including duplicates. These values are sorted in alphabetical or numeric order. (Dates and times are stored internally, as numbers.) The BDE also includes a pointer to that value's actual position in the table.

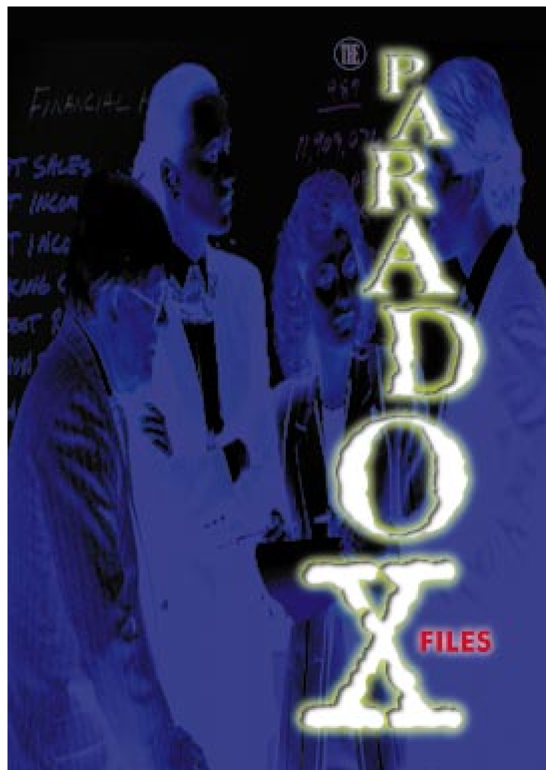


Figure 1 shows entries in a Code field, listed in no particular order. An index maintains a sorted list of these codes, together with a pointer to the record's position in the table.

If your program asks the BDE to find a particular value, and an index exists on the field, the BDE uses the index to locate the value in its sorted position, then reads the pointer to find the actual position in the table.

Unsorted List		Sorted List	
1	S	B	2
2	B	C	6
3	P	D	8
4	N	G	5
5	G	H	9
6	C	J	10
7	T	M	12
8	D	N	4
9	H	P	3
10	J	S	1
11	W	T	7
12	M	W	11

Figure 1: The BDE uses the index to locate the record's position.

The BDE uses a simple *binary search* to locate a value within the sorted list; by successively halving the sorted list — and eliminating the half where the value could not be — it "zeroes in" on the desired value very quickly. This exponential series is governed by the formula:

$2^n - 1$ = size of list

where *n* is the worst-case number of jumps the BDE will need to find any entry. (In half the attempts, a specific entry is found in fewer than *n* jumps.) There are other strategies for indexing data, including domain-based algorithms that attempt to use specifics about the data, but the Paradox file format uses none of these.

The Primary Index

In the relational model, a Primary Key is one or more fields which, when considered together, define the uniqueness of a record. Even if you need multiple fields to guarantee uniqueness, there is only one Primary Key in the table, considered to be a concatenation of all these fields. Primary Keys are often used to locate a record. They also frequently appear as Foreign Key fields in other tables, serving to link the tables.

Because Primary Keys play such an important role, the BDE places an index on the Primary Key of a Paradox table, and calls this the Primary Index. Stored in the .PX file that is part of the table's "family," the Primary Index is a little different from other indexes, in that it doesn't include an entry for every record in the table.

As you saw in the first article of this series, the Paradox file format will store a minimum of three records per block if the table has a Primary Key. Based on the record size, a typical table may have tens of records per block, and within the block, these records are always maintained in Primary Key order.

To keep the Primary Index as small as possible, the BDE stores the value of the complete Primary Key for only the *first record in each block*. When the BDE is searching for a record, it will locate the block where that record is stored — by finding the largest value in the list that doesn't exceed the value for which it's searching. This value defines the first record in the block where the desired record is stored. From there, the BDE has only to check the Primary Key for each record in the block — and this is done sequentially.

Another approach is to think of the Paradox Primary Index as a two-level *B-tree* (see Figure 2). Because this is a tree structure, the leaf nodes at each level must be kept in sorted order. In the Paradox format, the top-level nodes are the values at the beginning of each block. These effectively point to the records, starting with the indexed one, and finishing with the record just before the next indexed one.

The Paradox file format is formally known as a *clustered index* format, where the indexed values are physically stored in sorted order. This is why you can have only one clustered index per table. By way of comparison, in a dBASE file format index, or a Paradox secondary index (described later), nodes of the lowest level are pointers to individual records. The records can be arranged in any order, and many such indexes can occupy a single table. The downside, of course, is that the index is much larger. This is because it's storing every value, together with the pointers to each value's actual position.

The Paradox approach provides two especially important advantages in database structures:

1) The clustered index is smaller than a non-clustered index, because the lowest and most populous level has been eliminated (one index entry per record). Therefore, more of the index can be held in memory, and searching is faster — especially in larger tables.
2) Records with adjacent key values are stored physically close to each other, and often on the same block. This means that retrieving a range of records with adjacent keys is faster, because fewer disk reads are necessary.

Other database formats extend the concept of B-trees to multiple levels, obtaining benefits that can be enormous. A Paradox Primary Index uses only one level.

The relational model allows any field or fields in the table, even non-contiguous ones, to constitute the Primary Key. The Paradox file format places the fields in the Primary Key at the beginning of the table to simplify the searching and positioning algorithms. This is not a significant limitation, however, since you can place these fields in any order on a Delphi form, using the edit, grid, or other control types.

The Paradox file format also does not allow Logical, Memo, Formatted Memo, Graphic, OLE, Binary, or Byte fields to be included in the Primary Key (or in secondary indexes.) These field types are either too small (Logical) or too large (Memo) to be indexed, or contain binary data that cannot be indexed.

If you need to include more than three or four fields in the Primary Key to guarantee uniqueness across all fields, consider adding an ID field to the beginning of the table, and using this as the Primary Key. The other fields are then considered to be *attributes* of the Primary Key, helping to *describe* it.

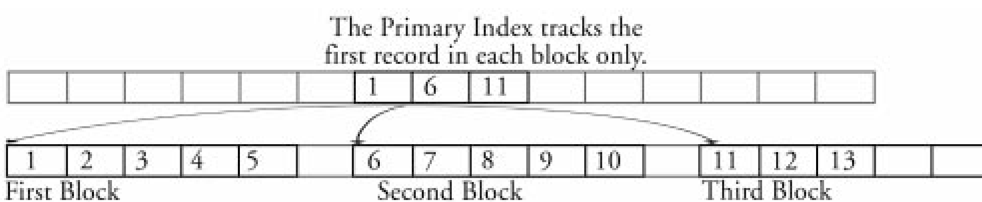


Figure 2: The Paradox Primary Index is a two-level B-tree.

Secondary Indexes

There are many times when you need to perform operations on fields other than the Primary Key of a table. For example, when you use the Primary Key of one table as a linking field in

another table, this is known as a Foreign Key, and is often used for searching and queries. Most tables contain other code or categorization fields, used to slice-and-dice, or organize the table in different ways. These fields also often appear in filters and SQL WHERE, ORDER BY, and GROUP BY clauses.

The Paradox file format provides a way to manually place an index on these fields as well, to speed up these searching and filtering operations. Paradox secondary indexes can be:

- single or multi-field;
- ascending, descending, or mixed;
- case-sensitive or case-insensitive;
- unique across all values, or with duplicates allowed; and
- automatically and incrementally maintained by the BDE, or not.

The type of index you place depends on what you want the index to do, and on the type of data being indexed.

Secondary indexes are stored in two files with the extensions .X*nn* and .Y*nn*. (As with all Paradox family members, you control the filename, and the BDE controls the extension.) The purpose of the .X and .Y files, and the *nn* numbering scheme, will be explained later.

Internally, the .X*nn* file is just another table. This makes sense when you consider that the BDE code to open and navigate Paradox tables is likely to be highly optimized, and very fast. In fact, in early versions of Paradox, you used to be able to copy this file to a new name with a .DB extension, then open it as a table! (Unfortunately, this trick no longer works.)

Maintained vs. Non-Maintained Indexes

With a maintained index, whenever you make a change to the data being indexed — by adding or deleting a row, or by updating the indexed field for an existing row — the BDE automatically updates the index as well, keeping it fully synchronized with the table. When you perform an operation that would benefit from the index, it's ready with no delay.

With a non-maintained index, when you make a change to the data being indexed, the BDE immediately flags the index as being out-of-date. When you perform an operation that would benefit from the index, the BDE must first rebuild the index to make it current. Although they've been part of the Paradox file format for more than 12 years, non-maintained secondary indexes are now regarded as obsolete. They were a response to the limitations of hardware and operating systems of the time. Note that a unique index cannot be defined as non-maintained. To enforce the uniqueness constraint, the index must be incrementally maintained.

Table Levels and Index Naming Schemes

As you may recall from the first article in this series, Paradox tables have a "level" associated with them in the BDE. This level corresponds to features that have been added to the format over the years.

Before Level 4, Paradox secondary indexes could be only single-field, ascending, case-sensitive, and non-unique. (Only the index placed on the Primary Key continues to have a uniqueness constraint.) Because of the close link between a field and an index on that field, the original version of Paradox used a simple naming scheme for the .X*nn* and .Y*nn* files for such indexes: *nn* is a hexadecimal number — from 01 to FF, inclusive — that matches the field number. Because tables can have a maximum of 255 fields, this scheme allows for such an index to be defined on any — or even *every* — field of the table.

For example, if you place a single-field, ascending, case-sensitive, and non-unique index on the third field of your table, the BDE will generate the following two files to hold the index: .X03 and .Y03. If the index is on the fourteenth field, the files are named .X0E and .Y0E.

In the Paradox file format, a secondary index must also have a *name*. For a single-field, ascending, case-sensitive, and non-unique index, the BDE assumes that the index name is the same as the field name. If any of these conditions are not true, you must provide a different name, which must follow the same naming conventions as a field (see Part II of this series), but which cannot be the same as a field name.

With Level 4, the BDE added support for multi-field and case-insensitive indexes. With Level 7, the BDE added support for the uniqueness constraint, and for individual fields to be sorted in an ascending or descending order within the index.

With these new index types, Borland needed a new naming scheme for secondary indexes, to maintain the unique filenames for each index. If an index is placed on multiple fields, or if it's descending, or in a mixed order, or case-insensitive, or unique, or any combination of these options, *nn* will be a sequential value starting from G0, and incrementing using a pseudo-hexadecimal scheme.

For example, the .X*nn* and .Y*nn* files for the first 16 of the newer-style indexes will be numbered G0 through GF, and the one that follows will be numbered H0. Numbers are not reused. If you delete the index numbered G5, the next newer-style index will still be numbered H1. Each index will also have a name that you define.

A Note about Descending Indexes

The Paradox file format allows for a multi-field index to have each field sorted in ascending or descending order. Unfortunately, Delphi doesn't give you a way to create such an index, nor can you test for one using the *TTable.IndexDefs* methods. When you create an index using the Database Desktop, the **Descending** option is a check box that applies to

all fields in the index, not each field individually. The *TIndexOptions* set object contains an *ixDescending* property that also applies to the whole index.

Interestingly, if you use the original Paradox product to create or restructure a table, the index definition dialog box allows you to set the index sort order on a field-by-field basis. Furthermore, Delphi's Database Explorer also shows the sort order for each field. It's unfortunate that the file format's full functionality hasn't surfaced in this instance.

Structure of the Secondary Index Files

Earlier in this article, I mentioned that the .X*nn* file is actually a Paradox table. The structure of this table tells us a lot about how secondary indexes work. Figure 3 shows the structure of the .X*nn* table for a maintained secondary index. The Secondary Key field contains the values being indexed. If more than one field is indexed, Secondary Key is a concatenation of the fields. For a case-insensitive index, these values are converted to upper case. Next, the .X*nn* table contains a field for each field in the table's Primary Key, where the indexed value is to be found.

Secondary Key	Primary Key Columns			Hint

Figure 3: Structure of the .X*nn* "table" for a maintained index.

Finally, the Hint field contains the physical block number in the base table where this Primary Key can be found. This value is not maintained incrementally; it's updated only when the table is restructured and the indexes are rebuilt. If there have been no changes, the BDE can use the Hint field to speed up the process even further, by jumping directly to the block in question. The only process that renders the Hint column obsolete is a block split — and even then, only for those records moved to the new block.

Figure 4 shows the structure of the .X*nn* table for a non-maintained secondary index. Remember that with a non-maintained index, the BDE cannot guarantee that the table will have a Primary Key. As with a maintained index, the Secondary Key field contains the values being indexed. The Tup No field contains the physical record number where this indexed value can be found. Blk Num contains the physical block number where this record can be found, while Tup Offset contains the byte offset from the beginning of the block where the record can be found. The BDE can use this information to jump immediately to the record, as long as it knows the table hasn't been changed since the index was refreshed.

In both cases, values in the Secondary Key field may not be unique. The .X*nn* table in each case is keyed on its initial fields (indicated by the asterisks in each table) because it's already sorted, and because this allows a Primary Index to be used for further optimization. And where is this Primary Index stored? Why, in the .Y*nn* file! Think of the relationship between the .X*nn* and the .Y*nn* files as being the same as between the .DB and .PX files.

Secondary Key	Tup No	Blk Num	Tup Offset

Figure 4: Structure of the .X*nn* "table" for a non-maintained index.

The .X*nn* file is usually much larger than the .PX file, and in some tables, it may even be larger than the .DB file itself. This is because the .X*nn* table contains an entry for every record in the table, whereas the .PX file lists only the first record in each block. The .Y*nn* file, on the other hand, is always small, because it contains only the Primary Key values for the first record in each block of the .X*nn* table, and this is almost always a narrow table with many "records" in each block.

The fact that *each* maintained secondary index also contains the complete Primary Key of the table is one strong reason for keeping the Primary Key as short as possible — ideally, only one field.

How a Secondary Index Works

Suppose you place a Query component on a form, and bind DataSource and DBGrid components to it. To show all customers located in Canada, you would specify the following expression in the Query's SQL property:

```
SELECT * FROM Customer WHERE Country = "Canada"
```

Keep in mind that Canadian customers are likely to be scattered throughout the Customer table. But if you have a maintained secondary index on the Country field, the BDE will use it for a query such as this. The BDE performs the following steps to extract matching records:

1) It opens the .Y*nn* file for this index, and uses it like a .PX to find the starting block in the .X*nn* file containing the "Canada" entries. (This operation is performed using a simple binary search.)
2) It jumps to the indicated block in the .X*nn* file, then sequentially reads through the records in that block until it finds the first "Canada" entry.
3) It then sequentially reads each record in the .X*nn* file until it reaches a record where the Secondary Key field is not "Canada".
4) For each matching entry found, it reads the Primary Key information for the table from the second and subsequent fields of the .X*nn* file. It also reads the value in the Hint column.
5) The BDE then jumps to the block indicated by the Hint column, and reads sequentially through this block for the desired record.
6) If the record isn't found in that block, it reads the .PX file, using a simple binary search to locate the block in the .DB containing that Primary Key. It then jumps to this block in the .DB file, and reads it sequentially until the record is found.
7) Finally, this record is fetched, and displayed in the grid.

This process is harder to describe than to perform. The

BDE can execute this sequence of steps far more quickly than if it had to search sequentially through a large file, and the benefits increase exponentially as the table size increases.

Using Keys and Indexes in Delphi

Indexes are used transparently by the BDE for Paradox tables, when you invoke a method that would benefit from the index. By default, a table is opened in Primary Key order. (This is analogous to saying that the Primary Index is used to order a newly opened table.) You can change the index by using the *IndexName* property. (The *GetIndexNames* method populates a *TStrings* list with the names of available indexes.) Setting this property to null switches the table back to Primary Key order.

There is no *TTable.IsKeyed* property that you can test. Instead, to determine if a table is keyed, use the following code, which counts the number of fields in the index when the Primary Key is the current table order:

```
var
  IsKeyed: Boolean;
...
TTable.IndexName := '';
IsKeyed := TTable.IndexFieldCount > 0;
```

Table bookmarks use the Primary Key automatically. When you set a bookmark on a table, Delphi defines a buffer to hold that record's Primary Key. When you issue a *GotoBookmark* method call, the BDE performs a "Locate" on the Primary Key to position the current record pointer.

Certain *TTable* methods perform significantly faster with the appropriate indexes. These include the *Filter* property, the *ApplyRange* method, various "find" methods (*GoToKey*, *GoToNearest*, *FindKey*, *FindNearest*, *FindFirst*, *FindNext*, *FindPrior*, and *FindLast*), and the *Locate* and *Lookup* methods. You should specify options for these methods to match available indexes on the table.

Indexes can be dynamically added to a table by using *TTable.AddIndex*. This method provides the parameter set by *TIndexOptions* to control whether the index created is primary or secondary — and for secondary indexes, the uniqueness, sorting and case-sensitivity options. (Indexes created using *AddIndex* are always maintained.) The matching *DeleteIndex* method allows you to delete a named secondary index with exclusive access to the table.

You must be careful defining indexes for use with Query components. In Delphi 1, queries were processed internally by a QBE-based parser, whether or not they were formulated as QBE or SQL. This parser, which still exists in the 32-bit BDE, can use only single-field, ascending, and maintained indexes, the original type that used the field name for the index name. (The original version of the QBE parser used only case-sensitive indexes, but newer 32-bit versions of the BDE also recognize the case-insensitive variety.)

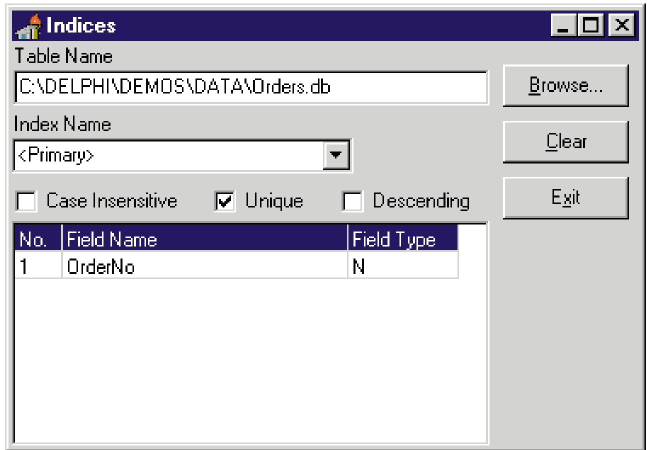


Figure 5: The Paradox index lister.

Starting with Delphi 2, SQL-based queries are now processed by a new SQL-based parser that will use any available index, no matter how it's defined.

The Index Lister

Figure 5 shows a screen shot of a sample application that lists the indexes for a specified Paradox table. It shows the field structure of each index, together with Case Insensitive, Unique, and Descending properties.

As I noted previously, native Delphi VCL methods offer no way to determine if an index is maintained or non-maintained, or if the individual fields have descending sorts. This information is available from the BDE API, but API-level programming is beyond the scope of this series.

The application's source code makes extensive use of the *TIndexDef* and *TIndexOptions* object types, which document most of what we need to know about an index. It also demonstrates one possible technique for breaking up the contents of the *TIndexDef.Fields* property into the individual index fields.

Next Time

The next article in this series will explore validity checks, which are simple business rules enforced at the table level. It will also discuss how the format implements *referential integrity*, i.e. the ability to maintain links between fields that appear in two or more tables. ∆

The sample application referenced in this article is available on the Delphi Informant Works CD located in INFORM\97\JUN\DI9706DE.

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.

By *Cary Jensen, Ph.D.*

Cached Updates: Part II

Improving the UI and Updating Your SQL

Part I of this series introduced the use of cached updates in 32-bit Delphi. This month's installment continues this discussion with a look at how to improve your user interface using cached updates, and how to use the UpdateSQL component and UpdateSQL Editor.

A Recap

Cached updates are a mechanism by which all changes made to a DataSet are stored locally, then applied simultaneously using the *ApplyUpdates* method of either a DataSet or Database component. The primary difference between these two methods is the Database's *ApplyUpdates* applies updates to one or more DataSets within a transaction. This method also encapsulates a call to *CommitUpdates* to empty the cache after the edits have been applied. By comparison, if you use the DataSet's *ApplyUpdates*, you must start, commit, or roll back your own transaction, as well as make an explicit call to *CommitUpdates*. Regard-less of the *ApplyUpdates* method you call, a DataSet remains in Cached Updates mode following the call to *ApplyUpdates*.

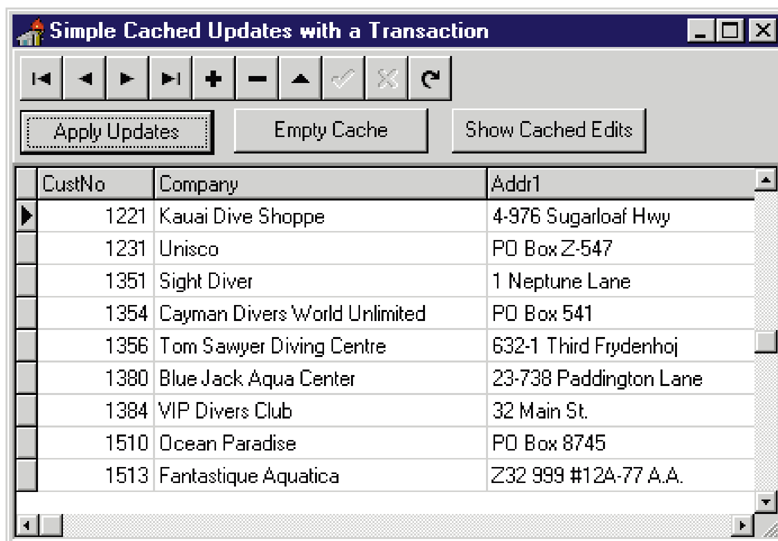


Figure 1: The main form of CACHE3.DPR. This project demonstrates how to enhance the user interface with cached updates.

Cached updates are enabled by setting a DataSet's *CachedUpdates* property to *True*. Cached edits are not applied to the corresponding DataSet unless they are specifically applied. Closing a table, or setting *CachedUpdates* to *False*, without actually applying the updates, cancels all cached edits.

Using cached updates affords four primary advantages:

1) decreased network traffic,
2) increased performance,
3) more user interface options, and
4) greater programmatic control over posting individual records.

Enhanced User Interface Options

As mentioned last month, one advantage of using cached updates is you can offer more options to users for viewing their edits. These options include allowing the user to: see edited records, preview only edited records (even deleted records), revert edited records to their original states, and see the pre-edited values of individual fields. Let's discuss how to provide these features.

Controlling record views. With cached updates enabled, you can control a user's access to specific records using the DataSet's

UpdateRecordTypes property. A set property, *UpdateRecordTypes* can accept zero, one, or more of these flags: *rtModified*, *rtInserted*, *rtDeleted*, and *rtUnModified*. These flags are defined by the *TUpdateRecordTypes* type. For example, to display only modified records to the user, use a statement such as:

```
Table1.UpdateRecordTypes := [rtModified];
```

To display all records in the cache, including those scheduled for deletion, use:

```
Table1.UpdateRecordTypes :=
  [rtModified, rtInserted, rtDeleted];
```

The default set for *UpdateRecordTypes* contains the flags, *rtInserted*, *rtDeleted*, and *rtUnModified*.

(Note: At the time of this writing, *TUpdateRecordTypes* was declared in the DBTables unit of Delphi 3 (pre-release), but in the DB unit of Delphi 2. If you use one of the flags defined by *TUpdateRecordTypes*, and receive a compiler error, make sure you have included the appropriate unit in your **uses** clause.)

The use of *UpdateRecordTypes* is demonstrated in CACHE3.DPR (see Figure 1). This project allows a user to display any edits to records, before committing the updates. These edits are displayed on a second form — Form2. When the user clicks the **Show Cached Edits** button, the button's event handler creates Form2, sets the *UpdateRecordTypes* property of Table1 to display all cached edits, then displays Form2 (see Figure 2). Incidentally, Form2 uses the unit for Form1. This is how the DBGrid on Form2 (see Figure 3) can display records cached by Table1, located on Form1.

The event handler in Figure 2 uses two custom exceptions to simplify the code. If Table1 is not caching edits, an *ENotCaching* exception is raised. If no edits are currently in the cache, an *ENoPendingEdits* exception is raised. Otherwise, Form2 is created, Table1's *UpdateRecordTypes* is set to display all cached edits, then Form2 is displayed modally. After Form2 closes, it's released. Finally, *UpdateRecordTypes* is reset to its default value.

Here's the declaration of the two custom exceptions:

```
type
  ENotCaching      = class(Exception);
  ENoPendingEdits  = class(Exception);
```

Determining a record's update status. Each record's status is displayed in Figure 3. This was done using a calculated field and the DataSet's *UpdateStatus* method. Recall that an active DataSet points to a single record. To determine the record's cached update status, use the DataSet's *UpdateStatus* method; it returns a *TUpdateStatus* value. *TUpdateStatus* is declared in the DB unit:

```
type
  TUpdateStatus = (usUnmodified, usModified,
                   usInserted, usDeleted);
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  if not Table1.CachedUpdates then
    raise ENotCaching.Create('Not caching updates');
  if not Table1.UpdatesPending then
    raise ENoPendingEdits.Create('No edits in the cache');
  Form2 := TForm2.Create(Self);
  Table1.UpdateRecordTypes :=
    [rtModified, rtInserted, rtDeleted];
  try
    Form2.ShowModal;
    Form2.Release;
  finally
    Table1.UpdateRecordTypes :=
      [rtInserted, rtModified, rtUnModified];
  end;
end;
```

Figure 2: The *OnClick* event for the **Show Cached Edits** button.

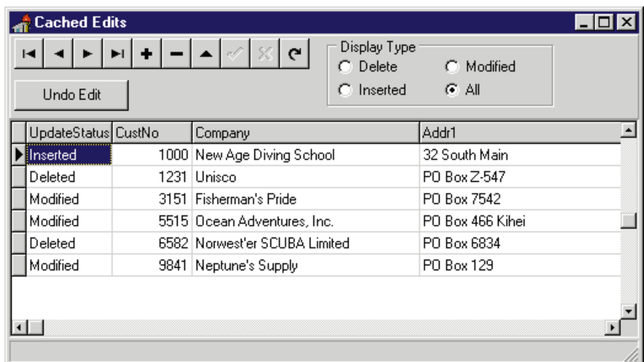


Figure 3: Form2 of CACHE3.DPR can be used to display pending edits in the cache, including those records that have been "deleted."

```
procedure TForm1.Table1CalcFields(DataSet: TDataSet);
begin
if Table1.CachedUpdates then
  case Table1.UpdateStatus of
    usUnmodified : Table1UpdateStatus.Value := 'Unmodified';
    usModified   : Table1UpdateStatus.Value := 'Modified';
    usInserted   : Table1UpdateStatus.Value := 'Inserted';
    usDeleted    : Table1UpdateStatus.Value := 'Deleted';
  end;
end;
```

Figure 4: Calling the *UpdateStatus* method if Table1 caches updates, to avoid raising exceptions.

Delphi raises an exception if you call *UpdateStatus* without being in Cached Updates mode.

Table1's *OnCalcFields* event handler demonstrates *UpdateStatus* in action (see Figure 4). As you can see, Table1 includes a calculated field named UpdateStatus. To avoid generating exceptions, this event handler calls *UpdateStatus* only if Table1 is caching updates.

Displaying old values of modified records. There may be times when you want to know the previous value of an edited field in a cached record. Fortunately, this value is readily available in the *OldValue* property of *TField* components. *OldValue* is a Variant, so it can hold old data of any type.

The CACHE3 project contains a simple demonstration of *OldValue* in the *OnColEnter* event handler of the DBGrid on Form2. Each time the user moves to a new field, *OnColEnter* determines if the current record has been modified. If so, the

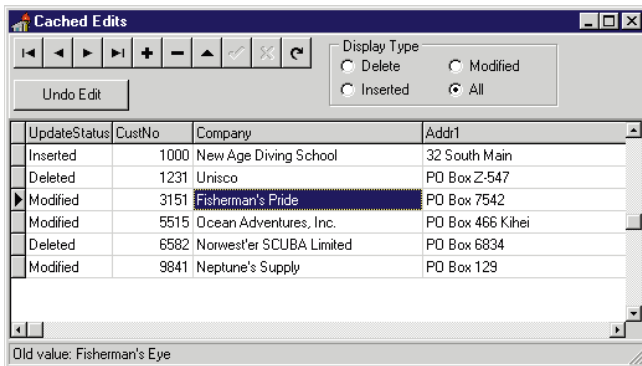


Figure 5: When the user moves to an edited field, the previous value of that field is displayed in the form's status bar.

```
procedure TForm2.DBGrid1ColEnter(Sender: TObject);
begin
  if (Form1.Table1.UpdateStatus in [usModified]) and
     (DBGrid1.SelectedIndex <> 0) and
     (DBGrid1.SelectedField.OldValue <>
      DBGrid1.SelectedField.NewValue) then
    StatusBar1.SimpleText :=
      'Old value: ' + DBGrid1.SelectedField.OldValue;
else
  StatusBar1.SimpleText := '';
end;
```

Figure 6: The *OldValue* and *NewValue* are not compared for a field if its *SelectedIndex* is 0.

OldValue and *NewValue* properties of the field (identified using the DBGrid's *SelectedField* property) are compared. If they do not match, the field has been edited, and the old value displays in Form2's status bar (see Figure 5).

As in Figure 6, *OldValue* and *NewValue* are not compared for a field if its *SelectedIndex* property is 0. This index is associated with the first field displayed in the DBGrid — UpdateStatus, the calculated field. Because it cannot be edited, the UpdateStatus field cannot have an old value.

Reverting individual records. Another powerful capability provided by cached updates is that they allow individual records stored in the cache to be selectively removed from the cache. This feature even permits a "deleted" record to be restored, provided the update hasn't yet been applied. This feature is possible through the DataSet method, *RevertRecord*. When *RevertRecord* is called, the current record to which the DataSet points is restored to its pre-edited value. There's no penalty for calling *RevertRecord* when the current record's *UpdateStatus* is *usUnmodified*. Here's *RevertRecord* in action:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  Form1.Table1.RevertRecord;
end;
```

This event handler is associated with the *OnClick* event property of the **Undo Edit** button on Form2 of the CACHE3 project. When you click **Undo Edit**, the current record displayed in Form2's DBGrid is restored to its pre-edited value. Because the *UpdateRecordTypes* property of Table1 doesn't include *rtUnmodified* when Form2 is open, the reverted record is removed from view.

Advanced Cached Updates

Cached updates are particularly easy to deal with when you're caching the updates to a single table. The same is true when caching updates to an editable query result. In some situations, however, cached updates can become more complex. These include simultaneously caching to two or more interrelated tables, caching updates to a read-only query, or when you want to programmatically control the updates to each record in the cache.

Caching two or more tables simultaneously. Typically, caching updates to two or more tables at the same time is only slightly more complicated than caching a single table. Usually, the primary issue of concern is knowing to which table you'll apply the updates first.

Most multi-table caching involves one-to-many relationships. For example, one table may contain invoice records, and another, the line items for that invoice. Furthermore, these tables will be linked by way of the detail table's *MasterSource* and *MasterFields* properties (in this example, the line items table would be the detail table).

To cache these tables simultaneously, set their *CachedUpdates* properties to *True*. The trick comes when you need to apply these changes. First, you must apply the updates within a transaction. Thus, if the updates to one table cannot be applied, any updates already applied to the other table can be rolled back. Then, you must apply the updates to the master table first.

Because you must control the order in which the updates are applied, and do so from within a transaction, you'll find it easier to apply the updates to these multiple tables using a Database component. Let's assume two Table components, *Master* and *Detail*, employ a Database named Database1. Here's how to apply cached updates to these tables:

```
Database1.ApplyUpdates([Master,Detail]);
```

Although this statement is simple, the tables' order in the array passed to *ApplyUpdates* is critical. It's essential the *Master* table be passed as the first element of the array, the *Detail* table as the second. This same technique can easily be expanded to three or more tables — even to those where a one-to-many-to-many relationship exists. In these situations, as in the two-table example, the order in which the updates are applied is important. This can be controlled by manipulating the order in which the DataSets are passed in the array parameter of *ApplyUpdates*.

In some situations, however, this technique alone cannot work. Specifically, if two or more of the tables being cached are dependent upon each other (either through referential integrity definitions or other constraints defined on the server), it may be necessary to code the application of the updates one record at a time. This is achieved using the *OnUpdateRecord* event handler for each table.

Controlling the update process. Unless you have taken explicit steps to alter the default behavior, calling *ApplyUpdates* causes the cached edits to be applied by code internal to the DataSet object. While this works well in a wide variety of situations, there may be times when you must provide customized update behaviors. This is necessary when you apply updates to read-only DataSets, as well as perform special processing of individual records as they're being updated.

Two techniques are available for this purpose. The first uses the UpdateSQL component, and the second involves the DataSet event properties *OnUpdateRecord* and *OnUpdateError*. Under some circumstances, you must use both. We'll concentrate on the details of the first technique.

The UpdateSQL Component

The UpdateSQL component is an object that can be associated with a DataSet, and is used to store SQL query statements that define how the cached updates are applied. In all, you can associate three query strings with a single UpdateSQL component. Each query is then stored in three *TString* properties named *DeleteSQL*, *InsertSQL*, and *ModifySQL*. They are used to apply deletions, insertions, and modifications, respectively.

The primary use of the UpdateSQL component, when used without update-related event handlers, is to provide updates to read-only query result sets, i.e. update records modified in a non-live query. At first, this sounds ridiculous ¾ that a read-only result set can be modified. It can, however, when the read-only result is placed into Cached Updates mode, and an UpdateSQL component has been assigned to the *UpdateObject* property of the Query.

For the most part, the UpdateSQL component applies to two types of read-only queries:

1) SELECT queries that aggregate data across records using the DISTINCT keyword, and
2) SELECT queries that join records from two or more tables.

The first type can be handled using a single UpdateSQL component. The second type requires more than one UpdateSQL component, as well as the control provided by the *OnUpdateRecord* DataSet event handler (we'll cover this in Part III; see end of article for details).

Consider the following, albeit contrived, scenario: You write a database for an order-entry system. Into this system, a large number of data entry people simultaneously enter customer orders received by phone. Due to circumstances beyond your control, validating the names in the City field at data entry time is not possible. To handle the inevitable misspellings in the City field, you provide a form that allows the database administrator (DBA) to review all the entered city names, and make any necessary corrections.

Because you don't want the DBA to have to view every record, you use a SELECT DISTINCT query to display only unique city names. (This result set, by the way, is read-only.) Using cached updates and the UpdateSQL component, the DBA can then review these names and make any necessary changes. Let's say the DBA finds an entry for "New Yorrk". You want to allow the DBA to change that entry to the correct value, "New York". It doesn't matter how many instances of "New Yorrk" are in the database — you want the DBA to change only a single record. This situation is ideal for the UpdateSQL component.

Let's say you apply a cached update that uses an UpdateSQL component. The DataSet to which the updates are being applied executes the queries associated with the UpdateSQL component, corresponding to the various changes made to the read-only result set. For example, if only insertions were made, only the UpdateSQL *InsertSQL* query is executed. If deletions are also performed, the *InsertSQL* and *DeleteSQL* queries are executed. Note that these queries are executed once per record. For example, if five insertions are made, the *InsertSQL* query is executed five times.

The queries associated with the UpdateSQL component use special parameters to perform their duties. As you may recall, a parameter in a query is defined by preceding the parameter name with a colon (:). Furthermore, the UpdateSQL component automatically defines two parameters for every field involved in the query. Of these two parameters, one has the same name as the field, and the other has a name using the form OLD_*fieldname.* For example, referring to the preceding scenario, the City field would have two corresponding parameters, CITY and OLD_CITY. The parameter with the same name as the field represents the field's current value; the parameter beginning with OLD_ represents the field's original value.

As a result, the *ModifySQL* query associated with the UpdateSQL object used by this scenario would contain this query:

```
UPDATE ORDER
   SET CITY = :CITY
 WHERE CITY = :OLD_CITY
```

When executed by applying updates to the SELECT DISTINCT query, this *ModifySQL* query assigns to the City field of the ORDER table the current value of the City field in the result set, but only for those records where the original value of the City field in the record being updated matches the City field in the ORDER table.

If you were to allow the DBA to delete all records associated with a given city, the *DeleteSQL* query would resemble:

```
DELETE FROM ORDER
 WHERE CITY = :OLD_CITY
```

Likewise, if you permit the DBA to insert records into the result set, the corresponding *InsertSQL* query would look like:

```
INSERT INTO CUSTOMER (CITY)
VALUES (:CITY)
```

Granted, to be consistent with the preceding scenario, it's highly unlikely you would permit either deletions or insertions. Consequently, you would assign only an update query to the *ModifySQL* property of the UpdateSQL component, and leave the *DeleteSQL* and *InsertSQL* properties blank.

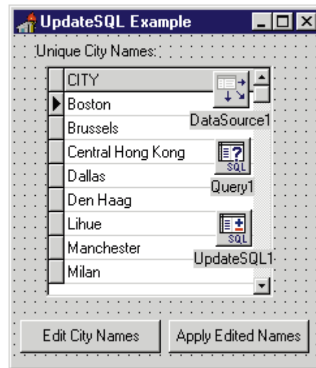


Figure 7: The CACHE4 project demonstrates the use of an UpdateSQL component to update changes made to a read-only query result set.

The CACHE4 project demonstrates the UpdateSQL component used in a task similar to the one described here (see Figure 7). This project contains a Query object that executes this SQL statement:

```
SELECT DISTINCT CUSTOMER.CITY
  FROM CUSTOMER
```

Because this query contains the DISTINCT keyword, the result set is not editable. However, if you click the **Edit City Names** button, this *OnClick* event handler is executed, placing the query into Cached Updates mode:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Query1.CachedUpdates := True;
  Button1.Enabled := False;
  Button2.Enabled := True;
end;
```

The query result then becomes editable. To apply changes made to the query result, it's necessary to apply the cached updates. This is accomplished by clicking the **Apply Edited Names** button (Figure 8 is its *OnClick* event). Note that to refresh the query's view, it had to be re-executed.

In this project, the real work of applying the cached updates is performed by an UpdateSQL component named UpdateSQL1, which is assigned to the query's

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Query1.ApplyUpdates;
  Query1.CommitUpdates;
  Query1.CachedUpdates := False;
  Query1.Close;
  Query1.Open;
  Button2.Enabled := False;
  Button1.Enabled := True;
end;
```

Figure 8: The *OnClick* event for the **Apply Edited Names** button.

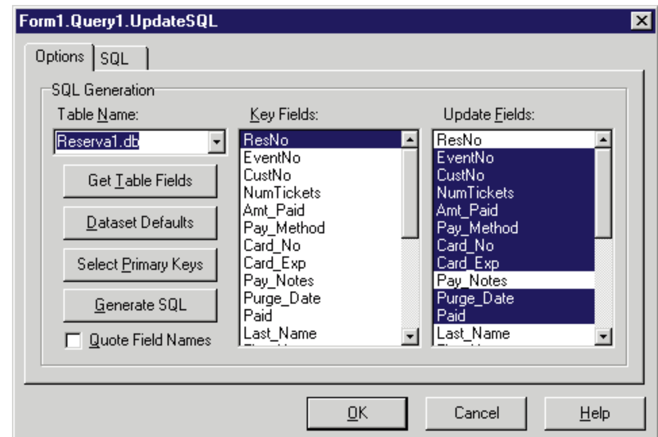


Figure 9: Use the UpdateSQL Editor to simplify the process of creating update queries for your UpdateSQL components.

UpdateObject property. UpdateSQL1 has only a single query associated with it. This query, assigned to the *ModifySQL* property, contains these statements:

```
UPDATE CUSTOMER
   SET CITY = :CITY
 WHERE CITY = :OLD_CITY
```

Using the UpdateSQL Editor

The update queries you assign to the various SQL properties of the UpdateSQL component are not difficult to write. However, they can be cumbersome when a large number of fields are involved. Fortunately, the UpdateSQL component sports a powerful component editor, the UpdateSQL Editor (see Figure 9).

The UpdateSQL Editor is a multi-page dialog box. The first page, Options, permits you to define the query's key fields, as well as fields that must be updated. Using the controls on the left side of this page, you can list the various tables involved in a query, obtain the tables' fields, and even select the primary keys. After highlighting the key and update fields in the **Key Fields** and **Update Fields** list boxes, click the **Generate SQL** button to generate the SQL you can use to update the table being queried.

Figure 10 shows the SQL page of the UpdateSQL Editor. It displays the three SQL statements generated by clicking the **Generate SQL** button on the first page. To switch between the various SQL statements, use the radio buttons at the top of this form. For example, to prevent a particular type of update — deletions for instance — simply select the **Delete** radio button, then delete the displayed query.

When you click **OK**, the generated SQL statements are assigned to the corresponding *TString* properties of the UpdateSQL component for which you invoked this component editor.

Note, however, that you don't need to use the UpdateSQL Editor to generate your update queries. You can write them manually using the String list editor associated with each SQL statement.

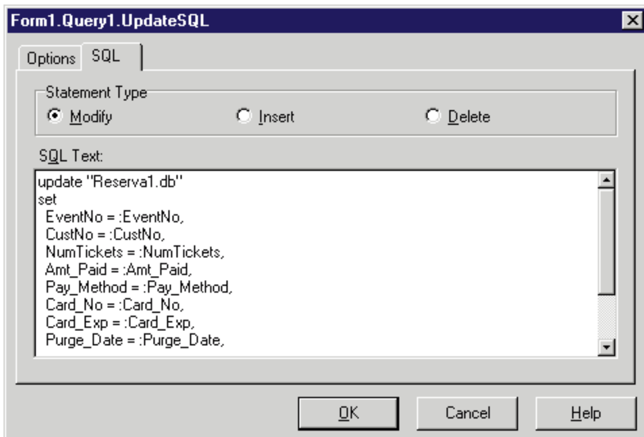


Figure 10: The SQL page of the UpdateSQL Editor permits you to view and edit the SQL generated by clicking the **Generate SQL** button on the Options page. You can also manually enter your SQL statements on this page.

Generally, you'll find the UpdateSQL Editor helpful and easy to use. For additional information on using the UpdateSQL Editor, display it, then click **Help**, or use the index in Delphi's online Help.

Conclusion

We've seen here that cached updates can increase the user interface options you provide. Furthermore, using an UpdateSQL component, you can allow your users to edit read-only DataSets.

Next month's "DBNavigator" column completes this series by considering how to use the cached updates-related event handlers, *OnUpdateRecord* and *OnUpdateError*. We'll also cover using multiple UpdateSQL components with the *OnUpdateRecord* event handler to apply updates to query results that include records from two or more tables. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUN\DI9706CJ.

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.

NEW & USED

By Tim Boyd

STDynArray 1.0

Bringing the Power of Dynamic Arrays to Delphi

As an old Paradox programmer, I've been known to tell anyone who will listen that if I could magically transport only one of its programming constructs into Object Pascal, it would be the DynArray. Now, at last, it's here!

Are you tired of writing lengthy routines to handle relatively simple array functions? Wouldn't it be nice to directly address an array entry using an index consisting of the string value that uniquely identifies it? Have you ever longed for the ability to copy all like-named fields from one table to another with only two lines of code? And wouldn't it be great if those two lines also handled any type-casting conversions needed between different data types? If you answered "Yes" to any of these questions, STDynArray may be for you.

Figure 1: The Order Counts form.

What Is a Dynamic Array?

The term *dynamic array*, as used in this article, refers to an array that can be dynamically resized, and is indexed by a string rather than an ordinal. Thanks to STDynArray, from Software Technology, the flexibility, power, and simplicity of dynamic arrays is now available to Delphi programmers. These features are provided through the clever use of Variant arrays behind the scenes, so STDynArray is strictly a 32-bit tool.

Using Dynamic Arrays

STDynArray can be used as a component or class. The easiest way to demonstrate its capabilities is by example. The source code for a test program that exercises each of its features is available for download — the test program Order Counts is included, along with a trial version of STDynArray that requires the presence of the IDE to run (see the end of the article for details). To install them, simply follow the readme.txt file instructions in each.

The Order Counts form contains two tables and several buttons (see Figure 1). The Orders table is one of the standard Delphi DBDEMOS files, and the Order Copy table is similar in format, but has differences that will be explained later. Both tables specify the dbOrders Database component in their *DatabaseName* property to take advantage of the "local alias" capability of Database; so there's no need to define an alias for the directory in which the tables are installed.

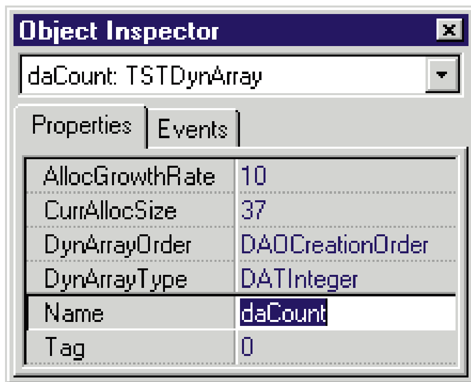


Figure 2: The *daCount* properties.

We'll first experiment with a dynamic array defined as a component. The array, named *daCount*, will be used to accumulate daily sales counts indexed by the string value of the corresponding sale date. Its properties are shown in Figure 2. The *CurrAllocSize* property defines the initial number of array entries (37) for which memory is to be allocated. Because this is a *dynamic* array, however, it's possible for it to grow beyond the initial allocation; so the *AllocGrowthRate* property is used to control the reallocation process. Its value is 10, so a factor of 10 will be applied to allocate more memory when the current size is exceeded.

Therefore, if the initial size of 37 is exceeded, the next memory allocation will be for 370 entries. This happens entirely behind the scenes, so the programmer's main concern is to allocate an initial size that's reasonable for the situation at hand. The *DynArrayOrder* property defines the sequence in which the array indexes are to be maintained. Possible values are *DAOAlphaOrder*, for alphabetical sequence, and *DAOCreationOrder*, to preserve the sequence in which entries are added to the array. The *daCount* array will be in creation order. The *DynArrayType* property identifies the type of data to be stored in the array, and supports many data types, including Variant. The *daCount* array will store Integer values.

When determining the size of a STDynArray, it's critical to ensure the *CurrAllocSize* and *AllocGrowthRate* properties work together to limit the number of times that memory must be reallocated. Memory reallocation involves significant overhead and should be kept to a minimum to prevent unnecessary delays. Because huge Variant arrays also have a detrimental impact on performance, it's important to have controls in place to ensure they don't become too large. When working with a large array, consider using the *DAOAlphaOrder* option to take advantage of its binary index search capability.

The design strategy for the Order Counts program is based on the assumption that the typical request will be for no more than one month, but that it may occasionally be necessary to see counts for up to a year. Therefore, the initial allocation of 37 will handle the typical one-month request. If a request is made for more than 37 days, the growth factor of 10 will be applied to bring the size to 370. Because the program limits user requests to allow no more than 366 days, it will never be necessary for the array to be expanded more than once. This ensures that the array can't become large enough to adversely impact performance; so itchy user fingers won't be tempted to press the computer's reset button.

Once defined, the *daCount* array is ready to be loaded, which is done by pressing the **Select Dates** button. The *btnSelectDatesClick* procedure prompts the user to enter start and end dates for the date range desired, empties the array, resets the *CurrAllocSize* to 37 (because the *Empty* method defaults the size to 10), initializes an array entry for each date within the specified range — using the string value of the date as the index — then spins through the Orders records to count each sale within the specified date range.

The code that does the accumulation saves the sales date as a string variable, then uses that variable as an index to increment that date's count:

```
sDate := tblOrdersSaleDate.AsString;
daCount[sDate] := daCount[sDate] + 1;
```

When the procedure finishes, it prompts the user to press the **View Counts** button to see the results. The *btnViewCountsClick* procedure invokes the array's *View* method, which displays the contents of the array in a resizable dialog box (see Figure 3). In this example, the date range was 7/1/88 to 7/31/88.

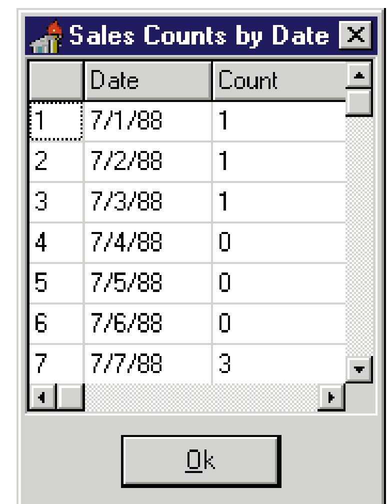


Figure 3: The results of pressing the **View Counts** button.

The **Show One Dt** button (again, see Figure 1) is used to view the count for a specific date. It executes code that prompts the user for the desired date, then displays the count for that date in a small dialog box. The following code demonstrates how the *Contains* method is used to determine whether the string value of the date is present as an index in the array:

```
if daCount.Contains(sDate) then
  ShowMessage('Count for ' + sDate + ' is: ' +
              IntToStr(daCount[sDate]))
else
  ShowMessage('Date ' + sDate + ' is not in the array!');
```

The **Update Array** button invokes an *Update* function that uses a resizable dialog box (see Figure 4) to allow the user to change the contents of one or more array entries. If the user elects not to save the results, pressing the **Cancel** button discards any pending changes. Because the *Update* function doesn't permit Index values to be changed, there is

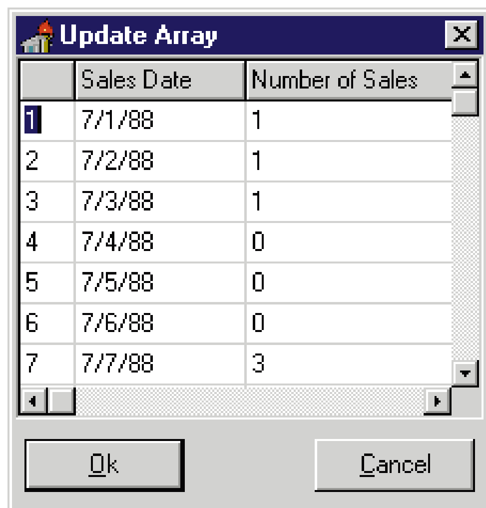


Figure 4: The Update Array dialog box.

no need for extra programming to handle potential error situations such as duplicate entries or indexes outside the range of the original constraints. The syntax of the *Update* function is straightforward. It accepts a dialog box title, and headings for the index and value columns, and returns a Boolean value. The code for the **Update Array** button is:

```
if daCount.Update('Update Array','Sales Date',
                  'Number of Sales') then
  ShowMessage('The array has been updated.')
else
  ShowMessage('The array was not updated.');
```

The purpose of the **Remove Zeros** button is to remove all array entries with zero counts. It demonstrates a technique for controlling a loop through a dynamic array when the number of entries is decreasing:

```
i := 0;
{ Use the array's Size property to limit the loop. }
while i < daCount.Size do
  if daCount.Value[i] = 0 then
    daCount.RemoveItem(daCount.Index[i])
  else
    { Increment counter if latest entry wasn't deleted. }
    inc(i);
```

In the third line, the *Value* property is used to compare the value of the current array entry with zero. The next line uses the *RemoveItem* method to delete the current zero-value array entry that is uniquely identified with the string value of the date contained in the *Index* property.

Any action that causes the number of array entries to change will result in the *daCount.Size* property being adjusted accordingly; therefore the *Size* property is used to limit the number of iterations within the **while** loop. In the last line, the *i* counter used to keep track of the current array item is incremented only if the latest entry was not deleted. As you can see, it only takes a few lines of code to serially navigate a dynamic array.

An Easy Way to Copy Records between Tables

Two important bonus features of STDynArray are its extremely powerful *CopyToDataset* and *CopyFromDataset* methods. These two commands make it possible to copy all like-named fields from one table to another with only two lines of code, handling any necessary typecasting automatically.

In our example, the *btnCopyRecordClick* procedure uses a dynamic array (*daTemp*) to copy the contents of the current Orders record to the OrdCopy table. Because the array is local to the procedure, it isn't defined as a component. Instead, the program uses the STDynArray class to instantiate it, which means it must first be defined as a variable of type STDynArray:

```
daTemp: STDynArray;
```

It's then instantiated as follows:

```
daTemp := STDynArray.Create(varVariant, DAOAlphaOrder);
```

A database table contains fields of different types; so the *daTemp* array's data type is set to Variant. The sequence of the array is set to *DAOAlphaOrder*, to cause the index to be maintained alphabetically by field name. This is done for the purpose of demonstration — in production use of *CopyFromDataSet*, it's normal to use *DAOCreationOrder* to store array entries in field-order sequence. The point is that the array will perform correctly regardless of the sequence of the field values within it. Now let's get to the interesting stuff. The command:

```
daTemp.CopyFromDataSet(tblOrders);
```

empties the *daTemp* array, then copies the contents of every field from the current Orders record to it, using each field name as the index to its entry. There's no need to set the *CurrAllocSize* and *AllocGrowthRate* properties, because *CopyFromDataSet* automatically sets the array size to the number of fields in the record, and the growth rate was initialized with a default value of *2* when the array was created. The statement:

```
daTemp.CopyToDataSet(tblOrdCopy);
```

copies the contents of the array into the OrdCopy table. Figure 5 shows the result of pressing the **Copy Record** button three times in succession. STDynArray accomplishes this by matching array indexes with the field names in the receiving table. For each match, the method leverages Variant behavior to determine if the data types are compatible. If so, the contents are copied into the field, using typecasting when necessary. If matching fields with incompatible types are present, all compatible fields are copied, a single exception is raised for the fields that can't be copied, and the user has the option to manually post the new record. Powerful stuff!

To verify that these methods work as stated, check out two things while experimenting with the form:

1) The receiving OrdCopy table contains a Diff field not present in the sending Orders table. Therefore it will remain untouched after the insertion.
2) The sending SaleDate field is a *TDateTimeField*, while the receiving SaleDate is a *TStringField*. The values are successfully translated into strings.

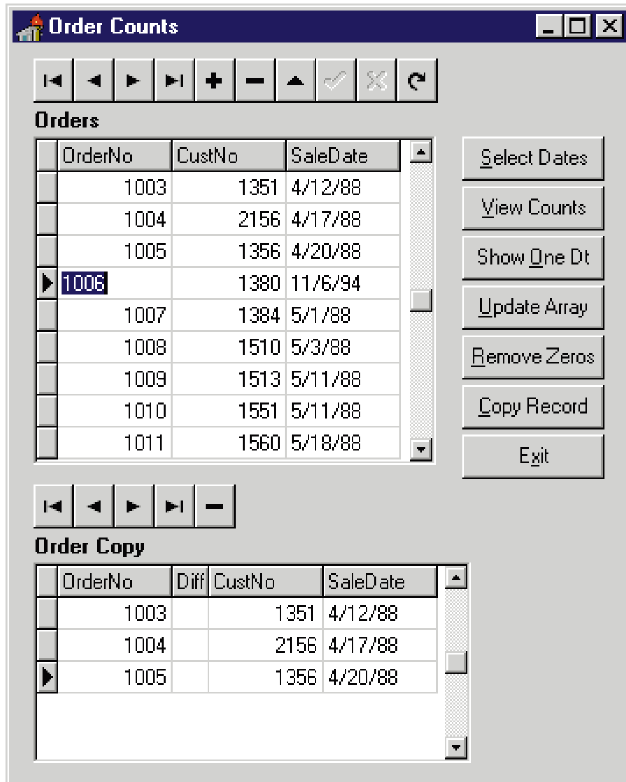


Figure 5: The copy results.

Because the program takes responsibility for creating *daTemp* as an instance of the STDynArray class, it's important to free the array within the **finally** clause of a **try** construct to prevent a memory leak. This is done in the last few lines of the *btnCopyRecordClick* procedure.

Conclusion

STDynArray is a useful product, and its Help file makes it easy to quickly get up to speed. Having been involved in its Beta testing, I am satisfied that it contains no glitches. The only caveats are to use its properties to prevent multiple memory reallocations, limit the maximum allowable size, take advantage of the *DAOAlphaOrder* option's binary search capability when possible, and test large arrays in the environment in which they will be used, to ensure acceptable performance.

I rank STDynArray as a must-have. Your programming tool kit isn't complete without it. Δ

INFORMANT FACT FILE

STDynArray brings Paradox-style DynArray to Delphi. A DynArray is an array that uses strings as indexes and has no declared maximum size as it is limited by available memory. STDynArray provides *CopyFromDataSet* and *CopyToDataSet* methods that permit an entire record in a database to be copied to and from STDynArrays.

Software Technology
1979 Grace Ave., Suite B-8
Los Angeles, CA 90068

Phone: (213) 969-0200
Fax: (213) 969-0555
E-Mail: info@software-tech.com
Web Site: http://www.software-tech.com
Price: Without source code, US$35; with source code, US$70.
Demonstration versions can be downloaded from Software Technology's Web site.

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUN\DI9706TB.

Tim Boyd, an independent consultant, provides Rapid Application Development services to companies in Southern California. He can be reached via the Internet at timboyd@bigfoot.com.

NEW & USED

By Alan C. Moore, Ph.D.

Async Professional for Delphi

A Complete Communications Library

Setting up asynchronous serial communications is not a trivial task, even with all of Windows' support features. A library like TurboPower Software's Async Professional for Delphi (APD) greatly facilitates adding communications capabilities to your programs. Although there are no Internet- or Web-specific components or methods, all the other major areas are covered, including modem components, fax components, terminal-emulation components, and file-transfer protocols. It also allows access to low-level functions and classes. The library includes full source code, many example programs, and an excellent manual, all of which we'll examine.

Working with Modems

Async Professional for Delphi offers two ways of working with modems. For Windows 3.x or Windows NT 3.51, you would likely use the *TApdModem* component and its related dialer components. For Windows 95 or Windows NT 4.0, APD's support for the Telephony Application Programming Interface (TAPI) is quite helpful. Let's take a brief look at each approach.

APD provides several components for setting up and using modems: a modem database component (*TApdModemDBase*), a general-purpose modem component (*TApdModem*), and a phone-dialer component (*TApdModemDialer*). The *TApdModemDBase* component provides an interface to APD's modem database, AWModem.INI. With it, you can manipulate the records in the database (read, write, modify, delete), or load them all into a *TString* class. Each database entry contains a name field (modem name) that serves as an index, five command fields (for initialization, dialing, terminating dial, answering, and hanging up), and eight response strings for various situations (*OkMsg*, *ConnectMsg*, *BusyMsg*, *VoiceMsg*, and so on). Figure 1, taken from the example project ExModDb, shows a partial list of the modems in the database, and part of ConfigCmd, the concatenated string that contains data on all the fields. The entire database entry goes like this:

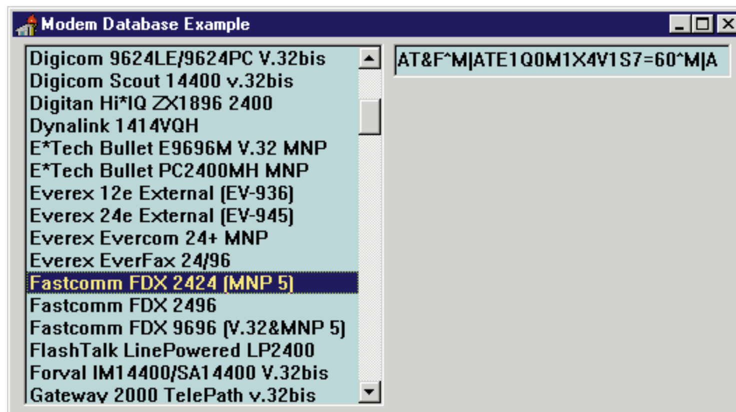


Figure 1: A partial list of the many modems (and corresponding strings) covered in APD's database.

```
[Fastcomm FDX 2424 (MNP 5)]
InitCmd=ATZ^M
DialCmd=ATDT
DialTerm=^M
DialCancel=^M
HangupCmd=DTR
ConfigCmd=AT&F^M|ATE1Q0M1X4V1S7=
  60^M|AT&B1&C1&D2&G0&H1&I1&M6&W^M
AnswerCmd=ATA^M
OkMsg=OK
ConnectMsg=CONNECT
BusyMsg=BUSY
VoiceMsg=NO ANSWER
NoCarrierMsg=NO CARRIER
NoDialToneMsg=NO DIAL
RingMsg=RING
LockDTE=TRUE
DefaultBaud=9600
```

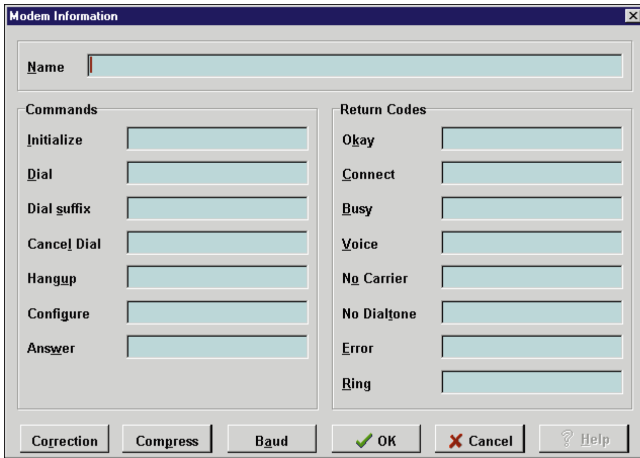


Figure 2: APD's extensive modem database can be modified, by means of this dialog box.

A more extensive demonstration project, ModDemo, provides an interface for adding, modifying, and deleting records in the database. Figure 2 shows the dialog box for adding a new modem to the database.

While the *TApdModemDBase* component provides the basic information about a modem, the *TApdModem* component provides the properties, events, methods, and exceptions needed to actually use the modem. Many of these (such as the property *OkMsg*, the event *OnModemBusy*, and the method *Hangup*) have an obvious relationship to one or more fields in the database. One run-time property in particular, *ModemInfo* of type *TModemInfo* (the same database record discussed previously), provides a convenient means for initializing all of *TApdModem*'s settings with one assignment statement.

TAPI, which ships with Windows 95 and is available for other versions of Windows, provides a link between the Windows environment and telephone hardware. Like plug-and-play, TAPI's approach is to allow the operating system (through various DLLs) to handle many standard software/hardware interface issues. In this instance, TAPI provides two important advantages for communications programmers:

- modems are usually set up automatically (the user may need to select a particular one); and
- applications can share COM ports.

APD provides several components you can use to access TAPI in your applications. *TApdTapiDevice* provides the basic TAPI functions of dialing, answering, and configuring the modem. The *TApdTapiStatus* component implements a display of status information on TAPI's various communications events. Finally, the *TApdTapiLog* class provides a means of automatically logging TAPI events.

TAPI does have a few additional requirements and limitations. It's not automatically available for every version of Windows. While you should consider supporting it in Windows 95-specific applications, you should probably avoid it in Windows 3.x applications. Also, it doesn't give you the same level of control inherent in working with *TApdModem*. Nevertheless, TAPI will no doubt gain importance as it becomes standard in 32-bit Windows implementations (including Windows NT 4.0).

A Terminal Case

The APD library provides several terminal-emulation components. The main one, *TApdTerminal*, can be used with or without emulation. Its various properties make the following capabilities available:

- retrieving characters from the serial port;
- sending keystrokes through the serial port;
- translating escape codes into the colors and formatting they represent;
- storing incoming data in a buffer (to facilitate scrolling); and
- scrolling and resizing.

A non-visual keyboard emulation component, *TApdKeyboardEmulator*, works with the *TApdTerminal* component. Finally, a *TApdBPTerminal* component provides support for showing data from CompuServe B+ file transfers. A *TApdCustomTerminal* class and a *TApdKeyboardEmulator* class (in which none of the properties are published) are provided for those who need to derive terminal or keyboard-emulation classes.

A Matter of Protocol

File transfer protocols are nearly as old as modem communications. No communication library would be complete without support for the major file transfer protocols. APD supports them all: ASCII, B+, Kermit, Zmodem, and various forms of Xmodem and Ymodem. The main component in this group is *TApdProtocol*, which includes properties and methods that apply to all the protocols (such as *ComPort* and *CancelProtocol*), along with those that are protocol-specific (such as *KermitRepeatPrefix* and *ZmodemRecover*).

Of course, a lot can go wrong during a file transfer. APD provides a number of properties and methods to control the process, as well as information and/or remedies when an error occurs. General error-handling properties, events, and methods include *AbortNoCarrier*, *BlockErrors*, *ProtocolError*, *OnProtocolError*, and *WriteFailAction*. Others are specific to a particular protocol. An associate class, *TApdProtocolLog*, provides a means to automatically log a file transfer. One of *TApdProtocol*'s properties, *ProtocolLog*, creates an instance of *TApdProtocolLog* to keep track of the main component's file transfer activities. Several of *TApdProtocol*'s other events and properties also support this tracking, including *OnProtocolLog*, *BytesRemaining*, and *BytesTransferred*.

Just the Fax, Please

With the introduction of fax modems several years ago, support for sending and receiving faxes has become an important feature of asynchronous serial communications.

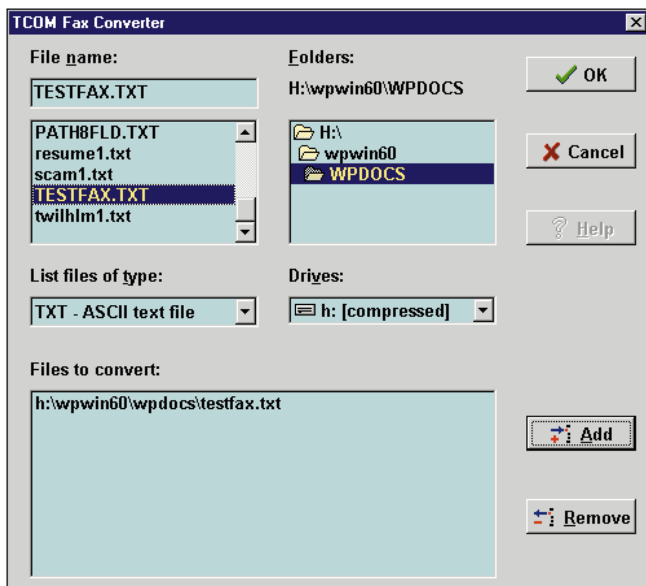


Figure 3: *TApdFaxConverter* converts ASCII, .BMP, .PCX, and other common file formats to compressed (.APF) form.

As with the other major support features of APD, the library supports a host of facsimile sending, receiving, and converting capabilities. Before a facsimile document can be sent via the fax modem, it needs to be converted to a compressed bitmap image. APD provides the *TApdFaxConverter* component (see Figure 3) to convert some of the more common file formats (ASCII, .BMP, .PCX, .DCX, and .TIF) to compressed (.APF) form. It also provides a corresponding component, *TApdFaxUnpacker*, to convert received .APF files to the formats previously mentioned.

What about actually sending or receiving a fax? Through its *TApdSendFax* and *TApdReceiveFax* components, APD supports sending and receiving facsimile documents on Class 1, Class 2, and Class 2.0 fax modems; any of these can communicate with any other Group 3 fax device. Both the sending and receiving components are derived from *TApdAbstractFax*, which defines the many shared properties and methods.

In addition to these basic fax components, APD provides printing, viewing, and status components. The printer component, *TApdFaxPrinter*, provides services for sending a fax document to a Windows printer. With it you can add headers and footers, and scale the document to fit the specified paper size.

The *TApdFaxViewer* component provides the means for you to view a received fax or .APF file. It includes scaling capabilities, white-space compression, drag-and-drop support, and the ability to copy all or part of the fax to the Windows Clipboard.

The *TApdFaxStatus* component (which is included as a property of both *TApdSendFax* and *TApdReceiveFax*) enables you to show progress and other information for faxes being sent or received.

Another small class, *TApdFaxPrinterStatus*, implements a standard printer-status display. Finally, APD provides a Fax Printer Driver. When compiled and installed, this driver allows you to generate files in .APF format as easily as printing to a Windows printer; in fact, it works the same way. The printer can also be used to capture images from a Web browser by selecting **Print** from the **File** menu, and selecting **APF Fax** as the printer.

Low-Level Foundation

APD provides two kinds of low-level support for its main components. Two general (not communications-specific) groups of low-level utilities are used by some of the communications classes. Also, the *TApdComPort* component is vital to the operation of every communications component we've discussed. APD provides full documentation and source code for all these classes.

The first group of low-level utilities consists of timer functions that allow you to wait for a specified period of time, convert between seconds and milliseconds, and monitor elapsed time.

The second group includes three functions for numeric/string conversions. The final low-level utility is a component, *TApdIniDBase* (with a lower-level custom class), that provides support for maintaining a Windows .INI file. The latter is used by the *TApdModemDBase* component discussed previously; however, you can derive your own classes to manage other types of .INI files. The first two groups of utilities are also useful in other contexts.

TApdComPort is the most important and basic component in the entire library. It controls the serial port hardware and all its input/output. It includes some 50 properties (many of which are run-time only), 11 events, nearly 100 new methods, and a host of exceptions. Particularly noteworthy are the built-in monitoring, error-checking, and debugging features.

Power and Flexibility

APD's built-in support for debugging is another powerful and useful feature. The main DLL (if you choose to use DLLs instead of direct linking), which contains the basic communications tools (COM port support, modem support, etc.), comes in two versions: APW.DLL and APWD.DLL. The latter adds special debugging tools for tracing and logging. Tracing saves all sent or received characters in a circular queue of specified size. Logging goes a bit further. As one of the properties of the *TApdComPort* component, it supports most of the other components in this library. When the property is set to *True*, APD logs all the data that goes through the COM port, and can optionally save it to disk. Because it includes a date/time stamp, you can see the exact order in which data is sent or received.

Speaking of power and flexibility, you can link the library's components directly to your executable program,

or you can access them from the DLLs included in APD. Each approach has advantages and disadvantages. With a single .EXE file, all your code is in one place; however, that file is considerably larger. With APD's .DLLs, several programs can access the same APD functions from a single .DLL; however, you need to distribute the APD .DLLs with your application(s).

Just as in Delphi's Visual Component Library (VCL), all the major components have custom classes in which none of the properties are published, and all of APD's major components have custom classes. So we can hide any property we need to in our derived classes. Full source code is provided for all components, supporting classes, and example programs, so we can see exactly how everything works.

Comprehensive Documentation and Support

TurboPower is famous in the software development industry for its first-rate manuals, excellent technical support, and its policy of providing full source code. APD is no exception. The manual does much more than simply describe the included components — it provides an excellent introduction to asynchronous communications programming. Many of the components have code fragments included; all have accompanying example programs. In addition, a massive example program, TCOM.DPR, demonstrates just about every feature of the library through some 30 dialog boxes.

The main screen, with its feature-rich menu and toolbar, indicates the extent of this example program (see Figure 4). Under the main menu bar are 27 submenu items, some of which include further items, or lead to one of the many dialog boxes. Nearly 30 new support files manage the latter; this doesn't even include the APD component units.

When **Fax | Send** is selected, the dialog box in Figure 5 appears, allowing you to select a phone number and a file to send. You can also schedule faxes to be sent later, as shown in Figure 6. And of course, you would expect to be able to configure the COM port. The dialog box shown in Figure 7 accomplishes this task. Keep in mind that these few examples barely scratch the surface in demonstrating the scope of the "example" program.

The Companion Program

To get an idea of what you can do with this library, run this article's downloadable companion program (see the end of this article for details). You'll need either the full version of APD, or the trial version downloadable from TurboPower.

Conclusion

When I began working with this product, I didn't have much experience programming asynchronous communications. Now, having worked through the sample programs, I feel I could tackle a wide range of communications problems. If you're a novice in this area, don't be shy;

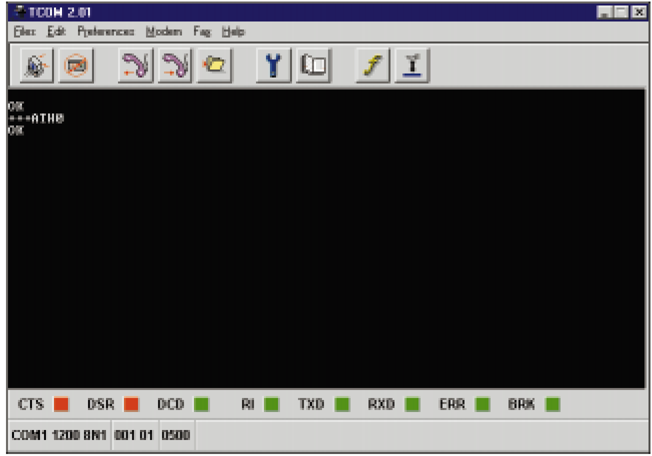


Figure 4: The main screen of the feature-rich example program.

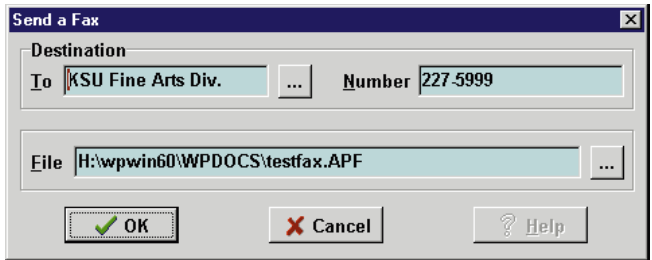


Figure 5: Faxing a file to a selected phone number.

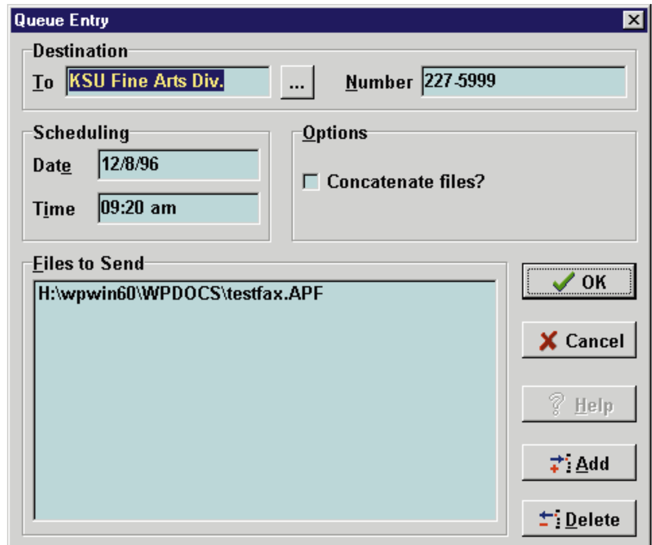


Figure 6: Scheduling faxes to be sent later.

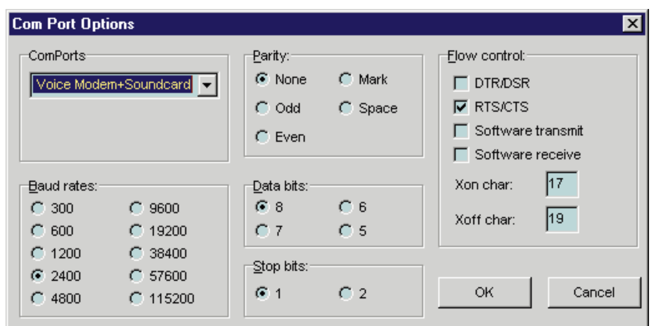


Figure 7: Configuring the COM port.

jump right in. You'll be surprised how easily you'll be able to get up to speed.

All of this is due, of course, to the solid construction of the components in this library, and the excellent example programs and documentation. I recommend Async Professional for Delphi highly to anyone who needs to add communications capabilities to their applications. It's truly a complete communications library. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JUN\DI9706AM.

Async Professional for Delphi is a complete library of components for handling the major aspects of asynchronous communications — including configuring and using modems, terminal and keyboard emulation, file transfer protocols, and sending and receiving faxes. It includes example programs (one of which is a multi-featured communications application), full source code, and extensive online Help. Its user manual is well organized, informative, and well written. APD is perfect for both the experienced and novice communications programmer.

TurboPower Software Company
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
Price: US$199; upgrade from APD 1.x, US$79; from any other TurboPower Async Professional product, US$119; from any other TurboPower product, US$159.

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.

Shakedown 1997

For much of the 1990s, we as software developers got comfortable with the development tool vendors that existed: Microsoft, Borland, PowerSoft, and Symantec were among the major tool providers to whom most programmers swore allegiance. Client/server developers, for example, tended to fall into three distinct camps: Visual Basic, PowerBuilder, or Delphi. Innovation occurred, but mostly among only these influential players. However, as we approach mid-1997, the dynamic nature of the Web has unearthed this status quo and ushered in a new breed of start-up tool companies, which — so far — have been sticking it to the traditional tool powerhouses.

Struggling at the top. Microsoft is perhaps the one major tool provider that, despite a rough start, has largely been able to maintain its momentum into the Web world. In contrast, other dominant players such as Borland and PowerSoft, while owning the desktop and client/server markets, are struggling to redefine their missions and gain momentum in this new arena.

Microsoft is obviously going to continue to be a dominant tool vendor, both for client/server and Web development. Its Visual Studio is convincing, integrating several tools, such as Visual C++, Visual J++, and Visual InterDev, into a single IDE. Yes, other vendors continue to provide more innovative tools, but Microsoft is starting to catch up with them.

Borland could hardly be more compelling than it is now in the client/server world with the Delphi and C++Builder duo. But its chances of strong success in the Web market are much less certain. Ultimately, Borland's best chance for prosperity in this arena rests with JBuilder. Although delayed several months, JBuilder still has a chance of being a market leader in the Java marketplace. Borland has surely been blessed in this regard by the immaturity of Java tools, but if Scotts Valley is not able to deliver JBuilder soon, they could lose out on a large piece of this pie.

Of all the traditional players, PowerSoft seems to be having the most trouble shedding the client/server mentality to produce innovative tools for the Web. While it will maintain a level of marketshare within the PowerBuilder community moving to the Web, it seems unlikely PowerSoft will ever be able to gain the mindshare in 1997 that it was able to in the client/server world in 1994-95.

Upstarts are coming. If the powerhouses are struggling, the same cannot be said of several emerging tool providers. Upstarts — such as Allaire, Marimba, and NetDynamics — are unencumbered by the past, and seem intent on becoming the next market leaders. And although not known as a tool provider, Netscape is also an increasingly important resource for developers. Knowing that going toe-to-toe with Microsoft on "platform" requires accompanying tools, Netscape is starting to develop products such as Palomar. However, the jury is still out on whether Netscape will ever be a legitimate threat (à la Microsoft) to companies such as Borland or Allaire.

Watch and see. If you are a developer, keep a close eye on the marketplace over the next 12 to 18 months for the inevitable shakedown. Some well-known ISVs will recover from their slump and retake a leadership position; others will fall by the wayside and take on that dreaded "legacy" label. Of the vast array of start-ups, most will fail or merge, but a few will work their way through the pack and become the next PowerSoft, Borland, or Symantec. Who do you think these "wunderkinds" will be? ∆

— 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.*