

Cover Art By: Darryl Dennis

## ON THE COVER



### 6 Searching for Records — Cary Jensen, Ph.D.

Dr Jensen takes on a topic nearly every Delphi developer can benefit from: efficient data retrieval. Three general approaches are described and tested for speed: sequential searches, *TDataSet* search methods, and parameterized SQL SELECT queries. The time-trial results may surprise you.

## FEATURES



### 10 Informant Spotlight

#### MTS Development: Part III — Paul M. Fairhurst

Concluding his three-part series on Microsoft Transaction Server, Mr Fairhurst turns to some advanced aspects of MTS, such as security, DCOM, transactions, callbacks, and more.



### 17 Algorithms

#### Hash It Out — Rod Stephens

Mr Stephens provides three Delphi implementations of hash tables, data structures that allow you to quickly store and retrieve items based on a key.



### 23 Columns & Rows

#### Multi-tier Database Applications: Part II —

Thomas J. Theobald

Last month, Mr Theobald described the steps for planning and building multi-tier database applications. This month, he turns to the implementation, i.e. get set for some code.



### 28 The API Calls

#### For Your Eyes Only — Mujahid Beg

Mr Beg demystifies the Microsoft Cryptographic Application Programming Interface (CryptoAPI for short), and shares a *TCryptography* class to make its use from Delphi easier.



### 33 Greater Delphi

#### The BDE Made Easy — Bill Todd

Mr Todd describes an extraordinarily useful — and economical — technique for sharing the Borland Database Engine over a network, while retaining maximum flexibility from application to application.



### 38 At Your Fingertips

#### They Were There All Along — Robert Vivrette

Uncovering some useful gems from the *SysUtils* unit, Mr Vivrette shares tips for handling command-line parameters, searching for multiple files, and replacing multiple instances of text.

## REVIEWS



### 40 LEADTOOLS Imaging 10

Product Review by Warren Rachele



### 44 Delphi 4 Bible

Book Review by Warren Rachele

## DEPARTMENTS

### 2 Delphi Tools

### 4 Newsline

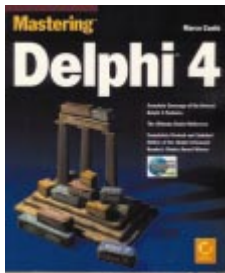
### 45 File | New by Alan C. Moore, Ph.D.



New Products  
and Solutions

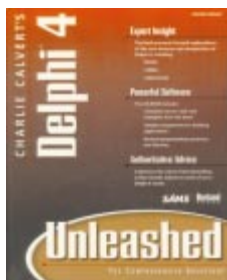


**Mastering Delphi 4**  
Marco Cantù  
SYBEX



ISBN: 0-7821-2350-3  
Price: US\$49.99 (1,247 pages)  
<http://www.sybex.com>

**Charlie Calvert's  
Delphi 4 Unleashed**  
Charlie Calvert  
SAMS Publishing



ISBN: 0-672-31285-9  
Price: US\$49.99  
(1,152 pages, CD-ROM)  
<http://www.samspublishing.com>

## PRICE Systems Announces ForeSight 2.0

PRICE Systems, L.L.C. announced *ForeSight 2.0*, the newest version of its project management software solution for forecasting time, effort, and costs for commercial and non-military government software projects.

The Project Wizard includes a Quick Estimate feature, which enables users to develop a first-glance forecast by answering a few critical questions. This estimate may be used to specify a project's costs as new information becomes available.

The new version includes a Microsoft Project 98 interface, which uses Component Object Model (COM) architecture. Using this interface, users can interface any ForeSight 2.0 project created with Project 98. COM further enables ForeSight 2.0 to operate within an integrated enterprise environment.

Other benefits of ForeSight

2.0 include more implementation tools, including complexity profiles for Delphi, Visual J++, PowerBuilder, JavaScript, VBScript, HTML, and FrontPage; more application types in the Project Wizard selection screen, including Internet, Text Processing, Database, Human Resources, Logistics, Management, Office Productivity, Operating

Systems, and Telecommunications applications; and Auto Lock ability to protect entered or changed values and unlock these values.

**PRICE Systems, L.L.C.**

**Price:** US\$975 per single-user license; bulk rate and site license pricing are available.

**Phone:** (800) 43-PRICE

**Web Site:** <http://www.pricystems.com>

## HyperAct Announces eAuthor Help 3.05

HyperAct, Inc. announced *eAuthor Help 3.05*, the company's template-based RAD authoring tool for HTML Help, hard-copy documents, Web sites, and HTML-based e-mail messages.

eAuthor Help 3.02 includes a WYSIWYG editor, HTML code editor, hierarchical project view, instant preview, object inspector, and property editors. The product comes with royalty-free HTML Help deployment

controls for Delphi, C++Builder, Visual Basic, MFC, and other environments.

New templates are created easily with the included template composer, including wizards. The product also includes a comprehensive SDK for software integration.

**HyperAct, Inc.**

**Price:** US\$250

**Phone:** (402) 891-8827

**Web Site:** <http://www.hyperact.com>

## Pervasive Announces Developers Kit

Pervasive Software Inc. announced the *Pervasive.SQL Software Developers Kit* (SDK), a set of rapid application development resources, including the I\*net Data Server, ActiveX controls, a pure Java API, and support for Windows development environments to speed development of applications based on Pervasive's embedded database engine.

The I\*net Data Server utility lets developers write Pervasive.SQL-based applications and run them as Web- or Internet-based client/server solutions. Developers can create a single-user workstation, as well as Internet-based mobile applications, without cumbersome

scripting languages or multi-layered data access methods.

Applications built on Pervasive.SQL integrate with third-party database controls, such as APEX True DBGRID and Sheridan Data Widgets.

Pervasive.SQL SDK also features the Pervasive.SQL

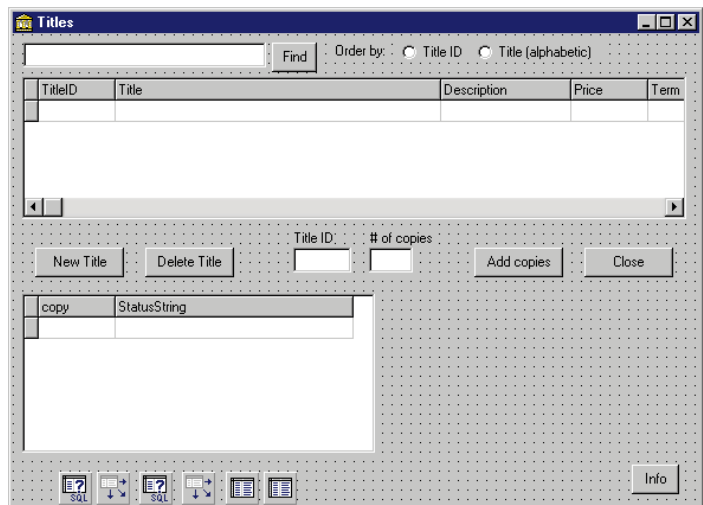
Workstation Engine, Developer's Resource Center, ODBC Driver, and a Java Class Library.

**Pervasive Software Inc.**

**Price:** US\$295

**Phone:** (800) 287-4383

**Web Site:** <http://www.pervasive.com>





### Absolute Solutions Releases ShortcutBar for Delphi 1.65

Absolute Solutions released ShortcutBar for Delphi 1.65, a full implementation of a Microsoft Outlook bar for Delphi. It features unlimited groups, unlimited shortcuts per group, scrolling groups and shortcuts, reactive drag-and-drop, small and large shortcuts, and shortcut re-ordering.

ShortcutBar is available for Delphi 3 and 4, and comes complete with integrated help and a sample application.

For more information, visit the Absolute Solutions Web site at <http://dSPACE.dial.pipex.com/absolutesolutions>.

## Lingscape Announces MultLang Suite 2.11 for Delphi and C++ Builder

Lingscape Ltd. announced *MultLang Suite 2.11*, a new version of the company's globalization tool for Delphi and C++Builder.

MultLang is based on the Unicode standard, and, combined with a conversion engine, provides support for Japanese, Chinese, Arabic, Hebrew, Hungarian, Russian, and all European languages.

The integration with the IDE and compiler helps produce thin, localized EXEs, or link multi-language support into the same EXE.

The Universal Language Modules API can contain context-sensitive dictionaries that are self-describing and have their own interface.

The MultLang Suite 2.11 contains a quality assurance wizard, which keeps projects from showing defect user interfaces, and checks for workable short cuts, correct character set displayed, approval of translated works, etc.

This new version has a migration utility to collect and re-use globalizations

made from other products or implementations, such as Inprise Delphi Translation Suite. You may instantly turn these into multi-language projects without writing additional code.

MultLang Suite 2.11 works with Delphi 2, 3, and 4 and C++Builder 1 and 3.

**Lingscape Ltd.**

**Price:** US\$998

**Web Site:** <http://www.lingscape.com>

## Wise Introduces Installation Suites

Wise Solutions, Inc. announced three new installation suites: *InstallMaker*, *InstallBuilder*, and *InstallMaster*. New features include enhancements for installation scripting, software patching, repackaging, and Web deployment.

InstallMaker provides point-and-click steps that allow developers to create basic Windows installation programs in minutes without writing any code.

InstallMaker includes Wise Installation System 7.0 plus SmartPatch, which creates compact installation patches.

InstallBuilder builds more sophisticated installations, with script-writing capability. InstallBuilder includes Wise Installation System 7.0, SmartPatch, and other

advanced features, including an integrated debugger, built-in Windows API calling, and a script editor.

InstallMaster is for professional developers who want complete control over their installations. InstallMaster includes Wise Installation System 7.0 and advanced features, including custom dialog and graphics editing; SetupCapture for repackaging other installations into Wise scripts; and WebDeploy for efficient installations from a Web site or intranet.

**Wise Solutions, Inc.**

**Price:** InstallMaker, US\$199;

InstallBuilder, US\$399; InstallMaster, US\$799.

**Phone:** (800) 554-8565

**Web Site:** <http://www.wisesolutions.com>

## Automagic Ships Y2K Components

Automagic Software announced its suite of *Y2K Delphi Components*. Using these components, developers can ensure the explicit entry of four digits to represent the year. The suite is comprised of two data-

aware components (*TascY2KDBCCombobox* and *TascY2KDBEdit*) and two non-data-aware components (*TascY2KCombobox* and *TascY2KEdit*).

ActiveX versions of the non-data-aware controls are

also provided on an as-is basis. The Y2K suite of components supports Delphi 1, 3, and 4.

**Automagic Software**

**Price:** US\$99

**E-Mail:** [automagicsoftware@usa.net](mailto:automagicsoftware@usa.net)

## WetStone Announces SMARTCrypt 1.2

WetStone Technologies, Inc. announced *SMARTCrypt 1.2*, an ActiveX security control that allows developers to build applications that employ SmartCards and cryptographic tokens. SMARTCrypt 1.2 abstracts the RSA Public Key

Cryptographic Standard (PKCS) #11.

SMARTCrypt provides abstracted functionality for file signing, file encrypting, and key management. The SMARTCrypt component integrates with Delphi, Microsoft Visual J++ and Visual Basic, and other tools

that support ActiveX or OCX controls.

SMARTCrypt is compatible with Windows 95/98/NT.

**WetStone Technologies, Inc.**

**Price:** US\$995 for single-user license; US\$2,995 for site license.

**Phone:** (607) 539-9981

**Web Site:** <http://www.wetstonetech.com>



February 1999



## DPR Launches New Delphi Courses

Database Programmers Retreat (DPR) announced two new courses aimed at experienced Delphi developers: Web Application Development with Delphi and Delphi 4 Multi-tier Development with MIDAS. The courses will be held at monthly intervals at the company's training center in Gloucestershire.

Web Application Development with Delphi covers the design, coding, and implementation of a Web-enabled application using HTML, client-side Java scripts, and CGI. This is a five-day course and costs US\$1,995 per delegate.

Delphi 4 Multi-tier Development with MIDAS focuses on the issues involved in creating multi-tier database applications. Participants will learn how to create a complete multi-tier application from scratch, and how to convert an existing database application to multi-tier. This is a three-day course and costs US\$1,295 per delegate.

For more information, call (800) 279-9717 or +44 (0) 1452 814 303, or visit the DPR Web site at <http://www.dp-retreat.com>.

## JBuilder 2 Wins Two Awards

*New York, NY* — Inprise Corp. announced that JBuilder 2, its family of visual development tools for creating Java business and database applications for the enterprise, has won two awards at Java Business Expo: the Editors' Choice Award from the *Java Developer's Journal* for Best Java Development Environment, and the Editors' Choice Award from *JavaWorld* magazine for Best Integrated Development Environment. In addition, Inprise's

## Inprise Offers to License JBuilder to Microsoft

*Tokyo, Japan* — Inprise Corp. announced an offer to license its JBuilder Java development tools to Microsoft Corp. This offer is in response to a U.S. district court's injunction against Microsoft requiring the company to conform to Sun Microsystems' Pure Java specification.

Inprise has worked closely as a partner to Sun and Microsoft. Inprise helped Sun develop the JavaBeans specification, and was first to ship a development environment with support for 100% Pure Java and JDK 1.1 with the initial release of JBuilder 1.0.

The JBuilder product family supports Sun's Pure Java specifications, including Java Native Interface, RMI, Java event handling, JDK 1.1.x,

VisiBroker for Java was named a finalist in the best General Class Library category in the *JavaWorld* Editors' Choice Awards.

## Inprise Announces Brazilian Subsidiary

*Sao Paulo, Brazil* — Inprise Corp. announced it had signed a letter of intent to acquire Engine Informatica Ltda., its Sao Paulo, Brazil-based partner and distributor. Pending final contract resolution, Engine Informatica will become a wholly owned Inprise subsidiary, called Inprise do Brasil, Ltda.

JDK 1.2, JFC/Swing components, JavaBeans, Enterprise JavaBeans, and JDBC, many of which may be issues in developing and deploying platform-independent Java solutions with Microsoft's Visual J++.

## Inprise to Acquire Apogee Information Systems

*Scotts Valley, CA* — Inprise Corp. announced it has acquired Apogee Information Systems, Inc., a privately held enterprise systems integration and consulting firm based in Marlboro, MA. As a result, Apogee will become part of Inprise's Professional Services organization.

Using multi-tier information systems, the 22-person company assists global clients with the integration of data and busi-

ness processes. Apogee clients include Beloit Corporation, Dun & Bradstreet, Sheraton Hotels, Bay Networks (Nortel), and the Massachusetts Department of Revenue.

ness processes. Apogee clients include Beloit Corporation, Dun & Bradstreet, Sheraton Hotels, Bay Networks (Nortel), and the Massachusetts Department of Revenue. Designed to assist corporate customers and systems-integration partners develop and implement enterprise solutions, Inprise Professional Services integrates the company's consulting, training, and technical support expertise into a single, world-wide organization.

## InterBase Releases InterBase 5.5

*Scotts Valley, CA* — InterBase Software Corp. announced InterBase 5.5, a new version of the company's embedded database.

InterBase 5.5 includes InterClient 1.5, a high-speed JDBC driver that connects Java applications and applets to InterBase databases, and adds direct international

support for user-specified character sets.

Stability in InterBase 5.5 has been improved by adding such features as protection for online metadata updates of Triggers and Stored Procedures by the InterBase 5.5 versioning engine. User Defined Functions (UDFs) have added safety features in

Windows, and the UDF library has been expanded. Performance enhancements include more efficient memory usage and a multi-threaded ODBC 3.0 driver.

For pricing and other information, call (888) 345-2015 or (831) 431-6500, or visit the InterBase Web site at <http://www.interbase.com>.



By Cary Jensen, Ph.D.



## Searching for Records

### Delphi Database Development: Part VI

In last month's "DBNavigator," you learned how to use DataSets and TField objects to navigate and edit data. The series continues this month with a look at record-searching techniques. Record searching refers to the process of quickly locating a record based on data stored in that record, e.g. searching for a particular record in a customer account table based on an account number, or finding an invoice based on the date of purchase.

There are three general approaches to record searching, which are demonstrated in the following sections:

- sequential searches
- TDataSet search methods
- parameterized SQL SELECT queries

#### Sequential Record Searches

The first, and often least desirable, searching technique is sequential searching. Sequential searches are performed by scanning a DataSet, record by record, testing each record for a value or values. Sequential searches can't make use of indexes to improve the speed with which a given record is found, and average

search times increase in direct proportion to the number of records in your DataSet.

The use of a sequential search on the CUSTNO field of the CUSTOMER.DB table is demonstrated in the example SEQUENCE project (all example projects for this article are available for download; see end of article for details). The main form for this project is shown in Figure 1.

The code shown in Figure 2 is attached to the Find button on this project's main form. There is a technique used in this code worth mentioning. Specifically, a TField variable is declared, and it's assigned to the TField object returned by the FieldByName method. This variable is then used throughout the event handler to reference the LastName field in the table. As you learned in last month's "DBNavigator," the use of FieldByName introduces a slight performance penalty compared with the direct access achieved through the use of the Fields property. However, by using a variable to hold the TField returned by FieldByName, the overhead of FieldByName is incurred only once, while maintaining the code readability afforded by FieldByName.

Normally, sequential searches are only used with local tables, including Paradox and dBASE tables. When a remote database is involved, sequential searches are typically

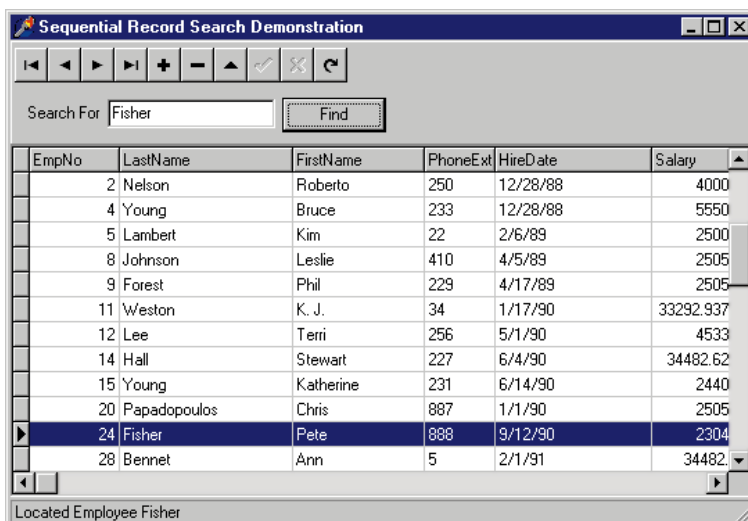


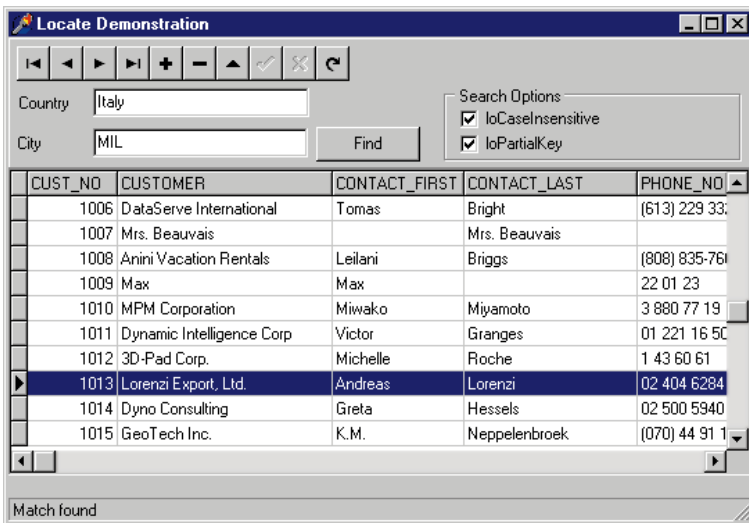
Figure 1: The SEQUENCE project demonstrates a sequential search.

```

procedure TForm1.Button1Click(Sender: TObject);
var
    SearchField: TField;
begin
    SearchField := Table1.FieldByName('LastName');
    Table1.DisableControls;
    try
        Table1.First;
        while not Table1.EOF do begin
            if SearchField.Value = Edit1.Text then begin
                StatusBar1.SimpleText :=
                    'Located Employee ' + Edit1.Text;
                Exit;
            end;
            Table1.Next;
        end;
        StatusBar1.SimpleText := 'Could not find '+Edit1.Text;
    finally
        Table1.EnableControls;
    end;

```

**Figure 2:** Code associated with the Find button on the SEQUENCE project main form.



**Figure 3:** The example LOCATE project.

only performed if the table being searched is very small. Performing sequential searches on remote tables requires that every search record be retrieved from the server. When the remote table is large, a sequential search can have a large negative impact on application performance and network traffic.

### **TDataSet Record-searching Methods**

It's usually better to use a searching method instead of sequential searches, except when your table is very small. There are two primary searching methods: *FindKey* and *Locate*. *FindKey* is available for Table components, and *Locate* is available for all DataSets.

The primary advantage of searching methods is that they can use a table's index to perform nearly instantaneous searches. Unlike sequential searches, where the time it takes to perform a search is directly proportional to the number of records in the table, search methods are largely unaffected by table size.

The most flexible of the searching methods is *Locate*. *Locate* permits you to search on one or more fields, and to choose

whether the search will be case-sensitive, as well as whether to perform a partial match on the last search field. *Locate* has the following syntax:

```

function Locate(const SearchFields: string;
    const SearchValues: Variant; Options: TLocateOptions):
    Boolean;

```

*Locate* requires three parameters. The first is a string that lists the field or fields being searched. If the search is being performed on more than one field, the field names are separated with semicolons.

The second parameter is a variant containing the value or values for which to search. If the first parameter specifies a single field, this second parameter can be either a variant or any valid expression type (variable, constant, literal, etc.). If more than one field is listed in the first parameter, this second parameter must be a variant array.

The third parameter is a set that includes zero, one, or both of the following flags: *loPartialKey* and *loCaseInsensitive*.

When you invoke *Locate*, it first checks for an index that includes the field or fields you listed in the first parameter. If one is found, *Locate* uses that index; otherwise it performs a sequential search. If a single field is being searched, *Locate* attempts to find the first record (the order being based on the identified index order) that contains the data specified in the second parameter within the field named in the first parameter. If more than one field is listed, *Locate* attempts to find the record where the first named field contains the value specified in the first element of the variant array, the second named field contains the value in the second element of the variant array, and so on.

If the third parameter includes the *loCaseInsensitive* flag, the comparison of string fields is case-insensitive. When the flag *loPartialKey* is included in the third parameter, the last field in the field list must be a string field, and is considered a match if the search field contains data that begins with the same characters specified in the corresponding search value.

If a matching record is found, *Locate* repositions the cursor to the located record and returns a value of True. If no match is found, the cursor doesn't move, and *Locate* returns False. The LOCATE project demonstrates the use of the *Locate* method (see Figure 3). The code in Figure 4 is associated with the *OnClick* event handler of the Find button.

While *Locate* is the easiest searching method to use, it's only available in 32-bit versions of Delphi. If you need to create 16-bit applications, you must rely on *FindKey*. *FindKey* is only available for Table components, and it

only operates on the current index. *FindKey* has the following syntax:

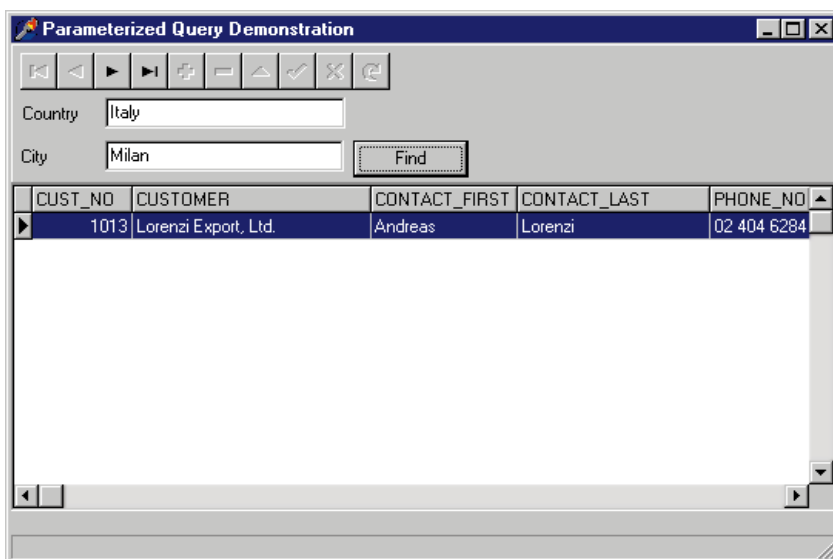
```
function FindKey(const SearchValues: array of const):
  Boolean;
```

The *FindKey* method requires a single parameter consisting of an array of values to search. The first element in the array is compared with data in the first field of the current index, the second element of the array is compared with the second field of the index, and so on. Similar to *Locate*, if a match is found, the cursor is moved to the matching record, and the value True is returned. When no match is found, the cursor doesn't move, and *FindKey* returns False.

Another method similar to *FindKey* is *FindNearest*. This method, which also takes an array of search values as its sole parameter, moves the cursor to the record that best matches the search array. Unlike *FindKey*, *FindNearest* always finds a match, and, therefore, always moves the cursor.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  SearchList: Variant;
  SearchOptions: TLocateOptions;
begin
  SearchList := VarArrayCreate([0,1],VarVariant);
  SearchList[0] := Edit1.Text;
  SearchList[1] := Edit2.Text;
  SearchOptions := [];
  if CaseInsensitiveCheckBox.Checked then
    SearchOptions := SearchOptions + [loCaseInsensitive];
  if PartialKeyCheckBox.Checked then
    SearchOptions := SearchOptions + [loPartialKey];
  if Table1.Locate('COUNTRY;CITY',
    SearchList, SearchOptions) then
    StatusBar1.SimpleText := 'Match found'
  else
    StatusBar1.SimpleText := 'No match found';
end;
```

**Figure 4:** The *OnClick* event handler of the **Find** button on the LOCATE project.



**Figure 5:** The PARAMQRY project demonstrates the use of a parameterized query.

## Searching with Parameterized Queries

The third search technique involves the use of parameterized queries. A parameterized query is one that includes one or more variables, referred to as parameters, in a SQL SELECT statement's WHERE clause. By changing the value of one or more parameters, you change which records are affected by the query. For example, consider the following SELECT statement:

```
SELECT * FROM CUSTOMER
WHERE (COUNTRY = :CTRY AND CITY = :CITYNAME)
```

The two parameters in this query are CTRY and CITYNAME. Note that these parameters are identified in the SQL statement by preceding their parameter names with a colon. Assuming the preceding SQL statement is assigned to the *SQL* property of a query named *Query1*, the following statements assign values to the two parameters, and then execute the query:

```
Query1.Close;
Query1.ParamByName('CTRY').Value := Edit1.Text;
Query1.ParamByName('CITYNAME').Value := Edit2.Text;
Query1.Open;
```

If you read the second installment in this series ("Delphi Database Development Part II: Tables, Queries, and Stored Procedures" in the October, 1998 issue of *Delphi Informant*), you may recall that parameterized queries need to be prepared only once before being executed the first time. Subsequent executions of parameterized queries that have been explicitly prepared are much faster. You may also recall that if you fail to explicitly prepare a query, Delphi will prepare it for you, but will also unprepare the query when it's closed. As a result, whenever you use parameterized queries, it's essential that you explicitly prepare them before their first execution, and unprepare them once they're no longer needed.

The use of a parameterized query is demonstrated in the project named PARAMQRY, shown in Figure 5. The query used in this project is explicitly prepared before its first execution. This task is performed with the following code, which appears on the query's *BeforeOpen* event handler:

```
if not Query1.Prepared then
  Query1.Prepare;
```

and it's explicitly unprepared with the following code from the form's *OnClose* event handler:

```
Query1.Close;
if Query1.Prepared then
  Query1.UnPrepare;
```



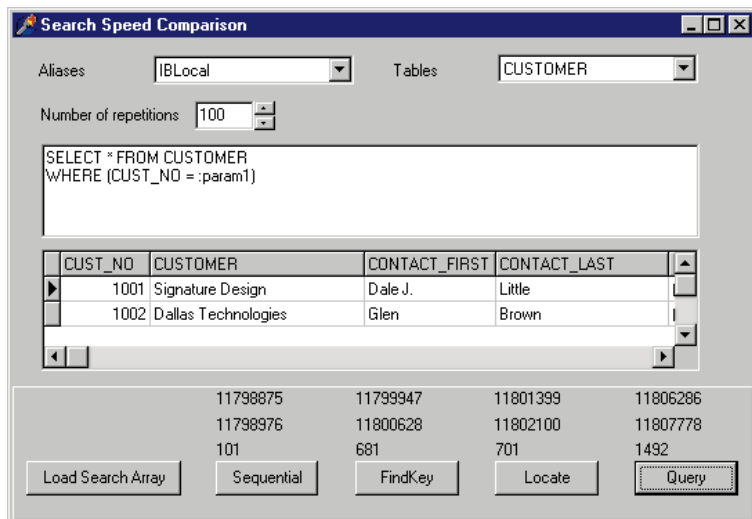


Figure 6: The COMPARE project tests relative performance of search techniques.

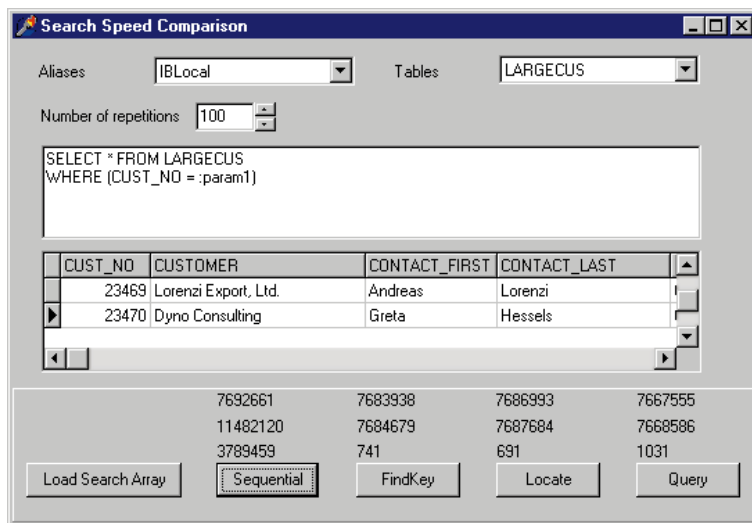


Figure 7: The results from a table of more than 100,000 records.

You'll notice from Figure 5 that the SELECT statement returns only the record or records that satisfy the WHERE clause. This effect is different from what is produced by the search methods *Locate* and *FindKey*, which change the cursor position in the database. It's important to keep this difference in mind when choosing one technique over the other.

## Searching Performance

The three searching techniques described here each use a different mechanism to select a record. They also differ in performance. This begs the question: Which technique results in the best performance?

Logic would suggest that sequential searches will be the slowest, parameterized queries the fastest, and the searching methods somewhere in between. But what is logical isn't always correct. It's been my experience that if you're really concerned about performance, you need to do some direct comparisons and determine empirically which of your options is best. That's just what I did with these search techniques.

The COMPARE project provides you with a means of testing the relative performance of sequential searches, *FindKey*, *Locate*, and parameterized queries (see Figure 6). After selecting an alias and a table, clicking the **Load Search Array** button causes the application to load an array with 10 values selected at random from the first field of the specified table, and to construct a parameterized SQL SELECT statement. Clicking one of the other buttons performs the specified search repeatedly on the first field of the selected table, using the values in the array (again, chosen at random) as the value being searched for. After the search has been performed the specified number of repetitions, the number of milliseconds required for the test is displayed above the corresponding button.

I ran this example many times on a wide range of tables and databases, and the images in Figures 6 and 7 are representative of what I observed. Figure 6 shows the results from the 15-record CUSTOMER table from the InterBase EMPLOYEE.GDB database that ships with Delphi. With this small table, the sequential search was significantly faster than any other technique, taking about one millisecond on average to locate a given record. The query was the slowest.

Figure 7 displays the results from a table of more than 100,000 records (it was generated by duplicating the CUSTOMER table records over 6,700 times, generating unique customer numbers for each record). With the large table, the sequential search was disastrous, taking more than one hour to perform the 100 searches. Both *FindKey* and *Locate* required approximately 700 milliseconds to perform the 100 searches. Interestingly, the parameterized query was nearly 30 percent slower than the search methods, requiring just over one second to perform the 100 searches.

## Conclusion

Sequential searches are the fastest technique when working with a small number of records, but are quickly becoming a poor choice as the size of the table increases. *FindKey* and *Locate*, by comparison, are relatively unaffected by table size, providing for consistently fast performance. Ironically, parameterized queries were always slower than the search methods, but still fast and largely uninfluenced by the number of records in the table. ▲

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\FEB\DI9902CJ.

Cary Jensen is president of Jensen Data Systems, Inc., a Houston-based database development company. He is co-author of 17 books, including *Oracle JDeveloper* [Oracle Press, 1998], *JBuilder Essentials* [Osborne/McGraw-Hill, 1998], and *Delphi in Depth* [Osborne/McGraw-Hill, 1996]. He is a Contributing Editor of *Delphi Informant*, and is an internationally-respected trainer of Delphi and Java. For information about Jensen Data Systems consulting or training services, visit <http://idt.net/~jdsi> or e-mail Cary at [cjensen@compuserve.com](mailto:cjensen@compuserve.com).



By Paul M. Fairhurst

## MTS Development

### Part III: Security, DCOM, and More

If you've been following this series on Microsoft Transaction Server (MTS), you'll know that we've built a simple three-tier application to provide online banking to customers of The Delphi Bank, a fictitious bank. We used a Paradox database for the back-end, a set of middle-tier MTS components supporting our business framework, and a standard Delphi application for the user interface. So far, we've used only a user ID and password to provide user access to the application and business services. MTS offers a much more sophisticated environment for security than this, upon which we can capitalize.

In this third and final part of this series, we're going to look at the more advanced aspects of MTS. The first thing we'll do is implement better security. The other limitation has been that the client application must be run on the same machine as the server components. We'll fix this by allowing the client to run remotely over a DCOM connection. Finally, we'll discuss client-side transactions, callbacks, references, and activities.

#### The Client Key

In [Part II](#), we developed a client application that accessed four objects (*Customers*, *Accounts*, *AccountTypes*, and *Transactions*) in our *DelphiBankServer2* component. Almost every method had a *ClientKey* parameter obtained from the *Logon* method of the *Customers* object. This key is meant to provide timed-out access to the server objects. If the client application doesn't access one of our MTS objects for a certain amount of time after logon, the client key is invalidated. Without a valid client key, methods in the objects won't proceed, and activity is disallowed. This is a simple, but effective, form of security that wasn't fully implemented in [Part II](#), so let's see how it's done.

As I've already mentioned, client keys are created by the *Logon* method of the *Customers* object. Remember that each of the four objects (*Customers*, *Accounts*, *AccountTypes*,

and *Transactions*) has its own copy of a data module descended from a common data module named *TDmDelphiBankCommon*. The methods to create and validate a client key are located here. The *Logon* method calls *CreateClientKey*, which allocates a new key and associates the current system time with it. All the other objects' methods make a call to *IsValidClientKey* — with the client key that the client passed in — before they perform any actions. In this method, the current system time is checked against the known client keys and their times. For the client key to be valid, it has to be known, and the difference of the current system time and the client-key time can't be more than a preset amount (five minutes by default).

This is all straightforward, but where do we store a list of client keys in a bunch of stateless MTS objects? We could store them in the database, but let's think about the problem a little. We could be dealing with hundreds of calls a minute to the *IsValidClientKey* method, because it's called at the start of nearly all methods in all objects. We also need to consider the amount of database access on a potentially very active table that would store these values. A table like this would be a hot spot of activity, and present a bottleneck as the components scale upward. MTS provides the solution to this problem: the MTS Shared Property Manager (SPM).

```

function TdmDelphiBankCommon.CreateClientKey: Integer;
var
  spgClientKeys : ISharedPropertyGroup;
  spClientKey   : ISharedProperty;
  wsPropertyName : WideString;
  bAlreadyExists : WordBool;
begin
  try
    { Create property group. }
    spgClientKeys :=
      CreateSharedPropertyGroup(cDelphiBankClientKeys);

    if not Assigned(spgClientKeys) then
      raise Exception.Create(
        'Could not create client key property group');
    { Create the property. }
    bAlreadyExists := False;
    while not bAlreadyExists do begin
      Result := GetTickCount;
      wsPropertyName := 'CK' + IntToStr(Result);
      spClientKey := spgClientKeys.CreateProperty(
        wsPropertyName, bAlreadyExists);
    end;

    if not Assigned(spClientKey) then
      raise Exception.Create(
        'Could not create client key');
    { Set the value. }
    spClientKey.Value := Result;
  except
    on E: Exception do
      raise Exception.Create(
        'TdmDelphiBankCommon.CreateClientKey() - ' +
        E.Message);
  end;
end;

```

Figure 1: The *CreateClientKey* method in action.

## Sharing State

The SPM is a resource dispenser you can use to share state among multiple objects within a server process. This means that any objects in the same package can share state with each other. This is incredibly useful for stateless objects. The SPM provides shared property groups, which establish unique name spaces for the shared properties they contain. You categorize the properties you want to store into groups, then store properties in the groups as name/value associations. Concurrency is taken care of because the SPM also implements locks and semaphores to protect your properties from simultaneous access (which could result in lost updates, and could leave the properties in an unknown state). Shared properties are held in memory and, therefore, have a very fast access time. The downside is that they disappear when MTS is shut down and restarted, so don't rely on them for persistent data storage; they're geared toward providing run-time shared state only.

The Mtx module of Delphi provides the definition of, and access to, the SPM interfaces. The top level in the hierarchy is the *ISharedPropertyGroupManager* interface, which allows you to create and find property groups. Once you have a group, you access it with the *ISharedPropertyGroup* interface. This allows you to create and find properties in the group. Finally, a property itself is accessed with the *ISharedProperty* interface, which you use to set or get its

```

function TdmDelphiBankCommon.IsClientKeyValid(
  ClientKey: Integer; var strDebug: WideString): Boolean;
var
  spgClientKeys : ISharedPropertyGroup;
  spClientKey   : ISharedProperty;
  nTickCountNow : Integer;
begin
  Result := False;
  strDebug :=
    'Connection has timed out. Please logon again.';
  try
    { Create property group. }
    spgClientKeys :=
      CreateSharedPropertyGroup(cDelphiBankClientKeys);
    if not Assigned(spgClientKeys) then
      raise Exception.Create(
        'Could not create client key property group');
    { Look for the property. }
    spClientKey := spgClientKeys.PropertyByName[
      'CK' + IntToStr(ClientKey)];
    if not Assigned(spClientKey) then
      Exit;
    { Check value. }
    nTickCountNow := GetTickCount;
    if (Abs(nTickCountNow) - Abs(spClientKey.Value)) <=
      (cClientKeyTimeout * 1000) then
      begin
        Result := True;
        strDebug := '';
      end;
  except
    on E: Exception do begin
      strDebug := 'TdmDelphiBankCommon.IsClientKeyValid() -
        Client Key of "' + 'CK' + IntToStr(ClientKey) +
        '" -> ' + E.Message;
      raise Exception.Create(strDebug);
    end;
  end;
end;

```

Figure 2: The *IsClientKeyValid* method.

variant value. Delphi provides two ways into the SPM: the *CreateSharedPropertyGroup* function, which creates a group with a specified name and default features; or, for maximum control, the *CreateSharedPropertyGroupManager* method, which returns an *ISharedPropertyGroupManager* interface you can use to create a group with extra features. We'll use the former to create a group where we can store our client keys.

Figure 1 shows the *CreateClientKey* method in action (all source discussed in this article is available for download; see end of article for details). The first line creates the shared property group with the name of the constant *cDelphiBankClientKeys* (*DelphiBankServer3ClientKeys*). If the group already exists (which it will most of the time), then an interface is returned on the existing group; otherwise, a new group will be created and an interface returned on that. We then generate a unique client key with the help of the Win32 function *GetTickCount*, which returns the number of milliseconds since the system started. Once we have our shared property, we set its value to the current tick count and exit. Not hard at all, is it?

Figure 2 shows the *IsClientKeyValid* method, which is called to check if the client key passed in by a client is still operational. You can see that it starts by creating the same prop-

erty group as before, but then it looks for the shared property in the group by name. If it finds it, it checks the value against the current system tick count and the timeout limit, returning appropriately.

Shared properties can be used for many things: They could help store singleton values; they could be used to hold the current location and status of players in an online multi-player game; and they could even be used to provide the next sequential number for a serial key in a database to save the delay of looking it up. The important thing is to recognize their importance in MTS, and use them when you can. They can literally save you a lot of time.

## Role-based Security

Every organization has specific duties assigned to its employees. For a small company, this is easy to monitor. As the organization grows, it's forced to segregate its workforce into more general categories, such as Payroll, Sales, and Development, to abstract and categorize itself and its staff. The underlying IT technology it uses tends to mimic this structure for similar reasons. With the Windows NT operating system, domains are used to create areas of users, such as Payroll, that share resources. By doing this, the systems administrators can more efficiently perform their jobs, making network security easier to enforce. A set of MTS business objects may have the requirement to be used by one or more domains, or none. MTS security hooks into the Windows NT domain security model, but domains can contain users that have a broad spectrum of duties to perform. We need something more fine-grained with which to pigeon-hole our users and make decisions at run time about what functionality we allow them to perform. The MTS solution to this is to further categorize users with roles.

A *role* is a symbolic name that defines a logical group of users for a package of components. Roles are defined during application development, and Windows NT users or groups are given the necessary roles at deployment time. There are two ways of using role-based security: Declarative and Programmatic. Declarative security is designed to instruct MTS to block access to packages, components, and interfaces. If you design your components well, and split functionality into related groups, then you will be able to take advantage of declarative security more effectively. We'll see how to implement declarative security when we look at the MTS Explorer shortly. On the other hand, programmatic security requires programming to implement, but offers a totally flexible way for you to make run-time decisions inside your objects.

## Programmatic Security

At run time, objects can check if a user is assigned to a specific role. By doing so, actions can be blocked, and are therefore used to enforce business rules. In The Delphi Bank example, we could have roles of Customers, Tellers, and Managers. Customers would represent typical cus-

```
if (TrnAmount > 2000) or (TrnAmount < -2000) then
begin
  if not (IsSecurityEnabled and
    IsCallerInRole('Managers')) then
    raise Exception.Create('Transactions of more than ' +
      '2000 dollars only allowed by Managers');
end;
```

**Figure 3:** An example of enforcing business rules.

tomers. We don't want to allow customers to add new bank accounts or delete transactions, for instance, so we have to write code to prevent this. Tellers work in the bank, but aren't Managers. Adding accounts or account types is acceptable for Tellers, but authorizing a transfer of more than \$2,000 is only allowed by Managers.

Figure 3 shows the sort of code you would use to implement this. In our example, this code would be added to the *TAccounts2.AddToBalance* method, an internal method called by other methods in this object, such as *Transfer*. You can see what it does quite easily. If the amount to add or deduct from the balance is more than \$2,000, and security is not enabled, or the caller isn't a Manager, the call fails. The *IsSecurityEnabled* method normally returns True, but returns False if the MTS package is running in the client's process space, and not in an MTS process space (more on this shortly).

The definition of the caller used in *IsCallerInRole* varies depending on the complexity of your setup. It's defined as the process calling into the current server process in which the object is executing, and is normally the base client. However, if an MTS object calls another MTS object on a separate machine, or in another package (each package runs in a different process), it won't be the base client. Rather, the caller will be the ID of the calling MTS process running the calling object. This can be a tricky subject, and more information is available in the MTS online documentation, but I'll try to give you a brief overview.

Each package runs under the guise of a user account, as we shall see shortly. Any objects in that package run in the same process, and take on the identity of the package's user account. If these objects access resources, such as a database, then they would do so with the rights and privileges of the package's user account, and not of the base client (the user), as you might expect. Also, security credentials are only checked when you cross a process space, so an object calling another object in the same package doesn't have its security checked. MTS security is a complex topic, and you need to read up on this in more detail if you plan on developing a system using MTS.

## The MTS Explorer

The MTS Explorer lets you install and configure — locally or remotely — packages, components, security, and other aspects of MTS running on a computer. We'll start by implementing declarative security on our objects. With

declarative security, you can control access to packages, components, and interfaces by assigning roles to them with the MTS Explorer. Because declarative security uses Windows NT accounts for authentication, you won't be able to use declarative security for a package if MTS is running on Windows 95.

When you first install MTS, there is no security on the System package, because no users have been mapped to the Administrator role. Therefore, security on the System package is disabled, and any user can use the MTS Explorer to modify package configuration on the computer. If you map users to the System package's roles, MTS will check roles

when a user attempts to modify packages in the MTS Explorer. By default, the System package has an Administrator role and a Reader role. Users mapped to the Administrator role of the System package can use any MTS Explorer function. Users that are mapped to the Reader role can view all objects in the MTS Explorer hierarchy, but can't install, create, change, or delete objects, shut down server processes, or export packages.

In the MTS Explorer, you'll see the **System** package under **My Computer | Packages**. If you open this, you'll see **Components and Roles**. Open **Roles** and you'll see the **Administrator** and **Reader** roles as discussed. Add your account and any other accounts that will be allowed to administrate MTS to the Administrator role before enabling security on the System package. Do this now by opening **Administrator**. Right-click on **Users** and select **New | User**. You'll see a dialog box like that shown in **Figure 4**. You can now add your groups and users (which can be from other domains) and select **OK**. On my system, the **System** package now looks like that shown in **Figure 5**.

You can clearly see the groups and users assigned to each role. Once you've done this, right-click on the **System** package and select **Properties**. From the Properties dialog box, select the Security page, and select **Enable authorization checking**. It's important that you only do this after you've added your account to the Administrator role; otherwise, you'll no longer be able to administer MTS, and you will be forced to reinstall. If you now shut down all server processes (by right-clicking on **My Computer** and selecting **Shut Down Server Processes**), System security will be enabled.

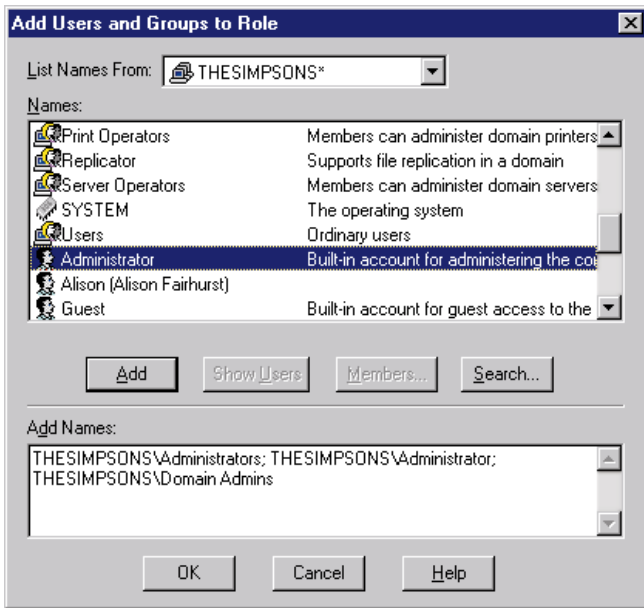


Figure 4: Adding users and groups to a role.

### Declarative Security

So much for the System package, but we also want declarative security on The Delphi Bank package and its objects.

To facilitate this, move administration-related methods from the four main interfaces (*IAccountTypes3*, *IAccounts2*, *ICustomers2*, and *ITransactions2*) into an *IxxxAdmin* interface. For instance, move the *Delete* and *ListAll* from *ITransactions2* to *ITransactionsAdmin*, because these functions should only be used by Tellers and Managers, not Customers. This means we can assign the *IxxxAdmin* interface to the Tellers' and Managers' roles. Any user not in one of these roles trying to access one of the *IxxxAdmin* interfaces will be automatically blocked out by MTS. The original four interfaces will have all roles mapped to them, and thus be accessible by all users.

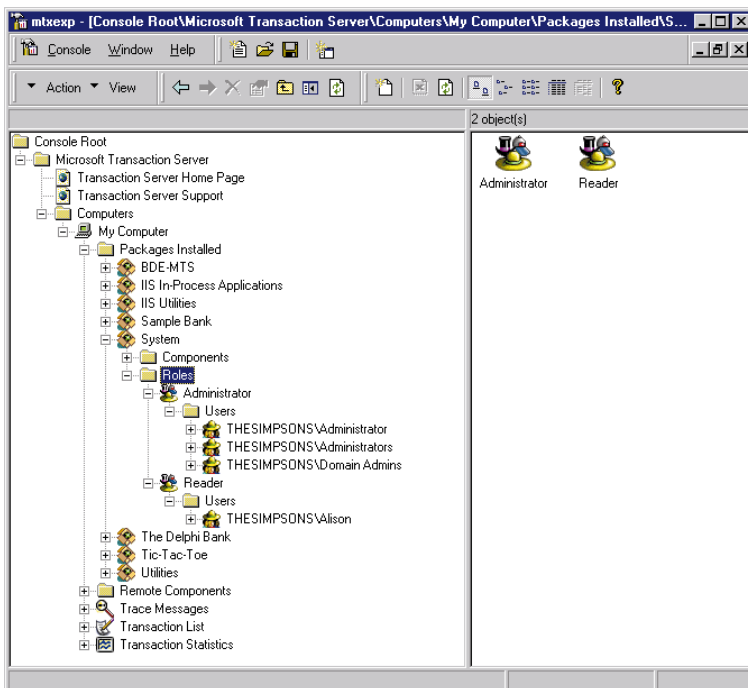


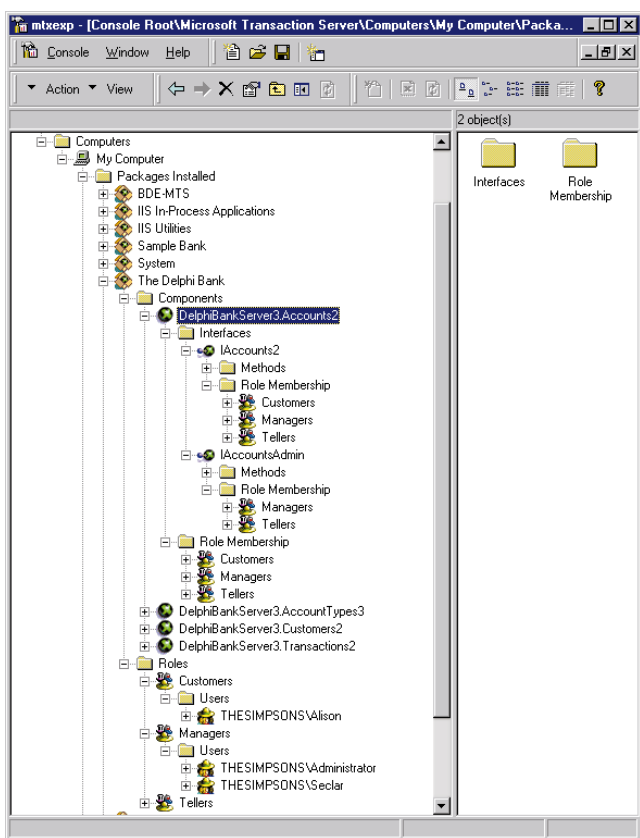
Figure 5: The System package with roles defined.

Open **The Delphi Bank** package, then the **Roles** directory. You should see that no roles are defined. Right-click on **Roles** and select **New**, then **Role**. Type in the name of the role and press **OK**. Do this three times, entering **Managers**, **Tellers**, and **Customers**, respectively. You need to add users and groups to the roles in the same way you did for the System package. It's important to

understand that these are Windows NT users and not the customers we've defined on The Delphi Bank database. You must add the customers you intend to use to Windows NT, as well as The Delphi Bank database Customers table.

Note that whichever user account you use to log on to Windows with when you run the client application will be the one used by MTS to authorize access to the objects. You could log on to the application as three different users, and transfer money and pay bills on their accounts, but MTS will use your Windows logon user name to validate access to the objects and interfaces because that is what is transported by DCOM. To make this work for you then, add your Windows logon user name to the Customers table, and give yourself at least one bank account in the Accounts table. You could use Paradox to accomplish this. Then, for each role you just added, open it, right-click on **Users**, select **New**, then **User**, and add your logon user name.

All that remains is to assign the roles to our components and their interfaces. Open the **Components** folder, and note the four components in our package. Each of them has two interfaces: a main one and an admin one. **Figure 6** shows the setup for the *Accounts2* object. Notice that for the *IAccounts2* interface I've added all three roles because everyone can access this interface. For the *IAccountsAdmin* interface, I added only the **Managers** and **Tellers** roles. For the object itself, *Accounts2*, I added all three roles because we want all users to be able to access the component. You



**Figure 6:** The Delphi Bank package with roles and users defined.

need to do the above for the remaining three objects and each of their two interfaces. When you have done this, enable authorization checking for the package in the same way you did for the System package. Declarative security is now fully enabled for The Delphi Bank.

## Packaging Properties

Before discussing remote access with DCOM, let's quickly go through the property sheets for a package. When you look at this dialog box for The Delphi Bank package, you'll see that there are five information tabs from which to choose.

The General page contains the name and description of the package. The Security page allows you to enable authorization checking, and set the authentication level (usually set to packet). The Advanced page is interesting. Here you can specify what happens once all components in the package have been deactivated. Depending on the expected level of use of the package, you may want to leave it running, or shut it down after a number of minutes to conserve resources. The default is to shut down after three minutes. If you're currently developing the components in the package, it's a good idea to set the package to shut down after zero minutes, because this frees up the lock on the DLL to which you want to compile. Also on this page is the ability to disable deletions and changes to the package. This can be used to avoid accidental changes on a live system. These flags would have to be manually turned off by an administrator before the package could be deleted or changed.

The Identity page allows you to specify the Windows user account that components in the package will use. If the components need to access resources, such as a database or MTS components on another machine, maybe even to read and write files to a server somewhere, then this is the account they will use to do it. Finally, the Activation page specifies how the components in the package are activated. The choices are Library package and Server package. A Library package runs in the process of the client that creates it. This option is only available for clients on the computer on which the package is being installed and configured. A Server package runs in its own process on the computer. Server packages support role-based security. Library packages have no security; all of its components are available to the client. The BDE-MTS package installed by Delphi 4 is an example of such a package.

## Remote Clients via DCOM

You may be wondering by now how clients from across the network call MTS components? How does a client application know where to find them, and what extra work do we have to do to the client to make it run across a network? Thanks to DCOM and an MTS wizard, the answer is "very little."

DCOM is an extension to COM that allows calls to COM objects to take place transparently over a network. Transparent means that the client application is blissfully unaware of whether the COM objects are running on the

local machine, or halfway around the world. By using standard OLE-compatible parameters, a method call is automatically packaged and routed over a network to the server machine, where it's then un-packaged and executed.

DCOM is already installed on Windows NT 4.0, but on a Windows 95 system, you'll have to download and install DCOM for Windows 95 (from the Microsoft Web site) before you can call the MTS components across a network.

Assuming you've installed DCOM, there's a quick way to enable a client computer to run our application and use objects on our server. If you right-click on **The Delphi Bank** package and select **Export**, you'll be presented with a dialog box asking for the name and location of the exported package. Three things are exported:

- 1) the DLL containing your components,
- 2) a .PAK file that can be used to import the package and component(s) into another MTS server installation, and
- 3) a client sub-directory containing an executable that you run on a client computer. This executable performs all the necessary registry entries to register the COM objects for the client application to use, but with additional information about their location on the network.

The client we've been developing can now be run on this computer with no changes. It will instantiate and use the services of the objects without realizing they're running on your server. You can examine what has been registered and adjust your DCOM settings by running the DCOM Configuration Manager (dcomcnfg.exe). See the "References" section at the end of this article for books pertaining to DCOM and other subjects relevant to MTS.

## Client-side Transactions

It's often necessary — or easier — for the client application to control transactions. Suppose you have two independent systems running MTS: an Inventory and a Payments system. Neither have access to each other, nor do they need it. For an order to take place, though, inventory has to be deducted and an entry in the Payments server made. The order has to be completely transactional, otherwise money or inventory could be lost. The order-processing application is being used by a user that has access to both systems, so it's best suited to control the transaction.

Client transactions are easily accomplished with the *ITransactionContextEx* interface. **Figure 7** shows an example piece of code that could be used for such a scenario. We simply create an *ITransactionContextEx* interface, use it to create a *Payments* and *Inventory* object (which would be registered COM objects running on different servers), call the appropriate methods, and call either *Commit* or *Abort*.

Calling *Commit* doesn't guarantee a transaction will be committed. If any MTS object that was part of the transaction has returned from a method after calling *SetAbort*, the transaction will be aborted. If any object that was part of the transaction has called *DisableCommit*, and hasn't yet called

*EnableCommit* or *SetComplete*, the transaction will also be aborted. Any error that causes Microsoft Distributed Transaction Coordinator (MSDTC) to abort a transaction will also abort an MTS transaction.

## Callbacks and References

It's a common programming technique to give an object a reference to another object for it to work with. For example, you call a search routine and pass in a reference to a search object that implements *ISearch*. This would allow you to use a different search method depending on the data being searched. Similarly, you may want to pass a reference to an MTS object to another MTS object, or back to the client. If you're an MTS object, the client may even want to give you a reference to itself for you to call back when you've finished doing your work. MTS doesn't prevent you from doing any of the above, but there are issues you must be aware of, or you may come "unstuck."

You can pass object references, for example, to use as a callback, in only three ways:

- 1) through return from an object creation interface, such as *CoCreateInstance*, *ITransactionContext.CreateInstance*, or *IObjectContext.CreateInstance*,
- 2) through a call to *QueryInterface*, or
- 3) through a method that has called *SafeRef* to obtain the object reference.

*SafeRef* is an MTS method an object can use to obtain a reference to itself that is safe to pass outside of its context. An object reference obtained in one of the above ways is called a safe reference. MTS ensures that methods invoked using safe references execute within the correct context. Never pass a self pointer or a self reference obtained through an internal call to *QueryInterface* to a client or another object.

Objects can make callbacks to clients and other MTS objects, but there are problems in doing so. First, calling

```

procedure MakeOrder(CustomerId, StockId: Integer;
  StockAmount: Currency);
var
  TransactionContextEx : ITransactionContextEx;
  InventoryIntf        : IInventory;
  PaymentIntf         : IPayment;
begin
  TransactionContextEx := CreateTransactionContextEx;

  try
    OleCheck(TransactionContextEx.CreateInstance(
      CLASS_Inventory, IInventory, InventoryIntf));
    OleCheck(TransactionContextEx.CreateInstance(
      CLASS_Payment, IPayment, PaymentIntf));
    InventoryIntf.RemoveStock(StockId, StockAmount);
    PaymentIntf.CreateStockPayment(
      CustomerId, StockId, StockAmount);
  except
    TransactionContextEx.Abort;
  raise;
end;

  TransactionContextEx.Commit;
end;

```

**Figure 7:** Client transactions are accomplished with the *ITransactionContextEx* interface.

back to a base client or another package requires access-level security on the client. The client would also have to be a DCOM server. Second, there may be firewalls blocking the path back to the client. Finally, any work done on the call-back executes in the context of the object being called. It may be part of the same transaction, a different transaction, or none at all. If you're controlling the deployment arena for your application, then you can work around these aspects, and callbacks become a viable operation.

### Activities

There is one final piece of terminology that you may come across when learning about MTS: the *activity*. An activity is a set of objects executing on behalf of a base client application. Every MTS object belongs to one activity, which is recorded in the object's context. For a given object, both the object that instantiated it and all the descendant objects it has instantiated will be part of the same activity. The objects can be distributed across one or more processes, executing on one or more computers.

The reason for an activity is simple: for all the objects running in an activity, MTS tracks the flow of execution and enforces a single logical thread of execution to prevent any inadvertent parallelism that could cause application state corruption or deadlock. Activities aren't something you have to worry about; simply make sure you use the MTS Object Context to create instances of objects you want to use, and all work will belong to the same activity and transaction.

### Tips for Development

To wrap up this series, I'd like to give you some pointers on your road to development. They're not hard and fast, but they may help smooth your progress into multi-tier development with MTS.

Normal classes in Delphi have a set of properties, methods, and events defined for them. Most people I see developing MTS objects head straight down a similar path. An MTS object is a different animal, however, so you have to learn to think differently when you design one. If you write your object with a set of properties on it, then every time a remote client sets a property on the object, there's a DCOM remote procedure call done over the network. This is inefficient and will slow down your application considerably. Also, properties imply statefulness. The object must be retaining state between your calls to set properties so it's not deactivated or scalable. Use methods that encapsulate functionality having parameters that pass in all the information the method needs to perform its work. That way, the object is deactivated on return from the method, and the parameters are efficiently packaged in one bundle for a single network call.

Stateful objects do have their place in MTS, but they're not singletons. The best way to implement a singleton is to have an MTS object that stores its state in the property manager. This way, multiple instances of the same object will have the same properties, but will still be stateless.


Define your components and objects, security requirements, interfaces, and roles before developing any code. Give your interface type libraries to the front-end guys as soon as you've developed them. They can then start work given what is effectively your documentation blueprint. Also, by defining role and security requirements up front, you can better code the functionality of your objects.

Finally, keep your components fine-grained. A lot of objects implementing small, closely related interfaces are much easier to document, debug, reuse, and maintain. They're also easier for MTS to scale.

### Conclusion

Hopefully, you will now be a lot more familiar and confident with the MTS environment. You know what it is, what it's for, what it can do, and how to develop for it. I've tried to give you an overall picture of MTS throughout this series, so you can make judgements about whether to use MTS in parts of your development. As MTS moves into the core of Windows NT, you'll see more system software (particularly enterprise material) use MTS. Microsoft is pushing Windows NT into the enterprise arena. MTS is going to be the foundation that helps make this happen, so you're bound to come across it at some point.

Hopefully, I've proved the point I made when I stated that Delphi makes developing for MTS a snap. It's a pity that at the time of writing this article, no one has yet written a book covering MTS with the Delphi language. Most books on the subject are geared, unsurprisingly, toward using Microsoft development languages. With any luck, this will change.

I'd be pleased to hear any feedback you have about this series, and to hear your success stories with Delphi and MTS. If you have any questions about what you've read, or something you don't understand, please drop me a line. 

### References

- *Inside Distributed COM*, Guy Eddon and Henry Eddon [Microsoft Press, 1998], ISBN: 1-57231-849-X.
- *Inside COM*, Dale Rogerson [Microsoft Press, 1997], ISBN: 1-57231-349-8.
- *Roger Jennings' Database Workshop: Microsoft Transaction Server 2.0*, Steven D. Gray, Rick A. Lievano, and Roger Jennings [SAMS Publishing, 1997], ISBN: 0-672-31130-5.

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\FEB\DI9902PF*

Paul M. Fairhurst is a First Class Computer Science graduate of Sheffield University and freelance consultant/programmer specializing in client/server and multi-tier database development. He is currently developing information systems for BBC Television and Radio in London. You can contact him at paul@c-s-c.demon.co.uk.





# ALGORITHMS

Hash Tables / Probing

By Rod Stephens



## Hash It Out

### Using Hash Tables to Manipulate Key-based Data

Databases typically use trees to store index information. This lets the database locate items very quickly. For databases stored on a hard disk or other slow-storage device, speed is critical. Reading data from a disk takes a relatively large amount of time, so the database must find the data as quickly as possible. Trees make it easy to find ordered data, but, under some circumstances, you can locate items even faster using a hash table. A hash table is a data structure that allows you to store and retrieve items based on a key. You can add items to a hash table specifying a key, and later use the keys to locate particular items.

#### The Basics

Suppose you want to store and quickly locate a few dozen items that have unique integer keys between 0 and 99. You could build an array with index bounds from 0 to 99. Then you could store an item with key K in position K in the array. To locate an item with key K, you would look in position K. The following code fragment shows this simple scheme:

```
var
  values : array [0..99] of Variant;
begin
  // Insert an item with key 13 in the array.
  values[13] := 'This is a really simple
               hashing strategy';

  // See if a value with key 27 is present.
  if (VarIsEmpty(values[27])) then
    ShowMessage('Key 27 is not present')
  else
    ShowMessage(String('values[27] = ') +
                values[27]);
```

This method is very simple. It's also extremely fast. With only a single array access, you can tell if the item is present and find its value.

Unfortunately, in the real world, key values don't always map into such a conveniently small range of values. For example, suppose

you want to locate employees using only their Social Security numbers. There are roughly one billion possible Social Security numbers of the form 123-45-6789. To use the previous hashing strategy, you would need to allocate a one-billion-entry array. If each employee's information occupied 1Kb of storage, the array would take up 1 terabyte (one million megabytes) of memory. Chances are good that you don't have that much memory or disk space available on your computer. Even if you had that much storage, more than 99 percent of it would always be unused, unless you had more than 10 million employees.

In cases like this, where the number of possible key values is huge, you need to map the key values into a relatively small hash table. For example, if you have around 80 employees, you might create an array with 100 entries indexed from 0 to 99. Then you could calculate an employee's Social Security number modulus 100 and map the employee's information to that entry. For example, an employee with Social Security number 1234-56-7890 would map to position 90 in the array.

Using this scheme, you would only need 100 employee entries taking up a total of

100KB of memory. If you have 80 employees, only 20 percent of that space is unused, so you're not wasting a lot of space.

Note that this array contains only 100 entries, but there are one billion possible key values. In that case, 10 million possible key values will map to each array entry. For example, all Social Security numbers that end with 99 will map to position 99 in the array.

That means that sometimes you may try to put a record in the table, and find it's already occupied by another record. This is called a *collision*. To resolve this problem, you need a collision-resolution policy. This is a rule that tells where an entry goes when the place it belongs is already occupied. Usually, the policy is to re-map the record to another position in the table. If that position is also in use, the record is continuously remapped until an empty position is found. The sequence of positions that are examined searching for an empty position is called the item's *probe sequence*.

To summarize, a hashing scheme requires three things:

- 1) A data structure called a hash table that holds the entries.
- 2) A hashing function that maps keys to entries in the hash table.
- 3) A collision-resolution policy that tells where to place an item when its natural position in the table is already occupied.

### Linear Probing

One of the most popular types of hashing is called *open addressing*. Here, the value of the key is used to calculate an offset in memory where the data should be placed. The previous examples used open addressing to map a key value to a position in an array.

There are several varieties of open addressing schemes that differ in their collision-resolution policies. The simplest is called *linear probing*. If an item's position is occupied, the program simply looks at the next position. If that position is also occupied, the program looks at the next position. The program continues looking through the array until it finds an empty position, or it has searched the entire array. If the program finds an empty position, it inserts the item there. Otherwise, the hash table is full, and there is no room for another item.

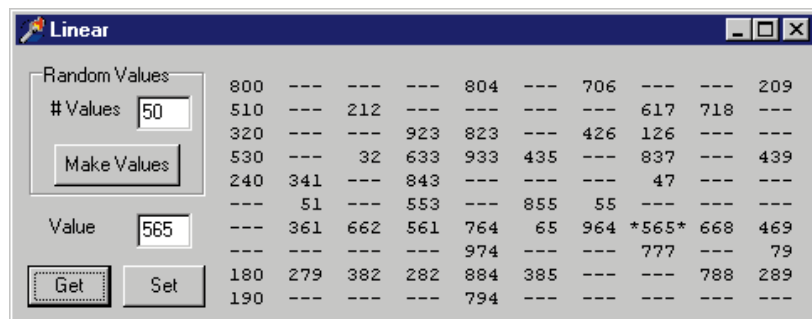
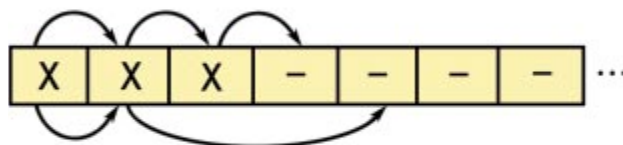


Figure 1: Open addressing with linear probing, as demonstrated by the Linear program.

### Linear Probing



### Quadratic Probing

Figure 2: The quadratic probe sequence will jump through an array, skipping entries and inserting the new item in a position not adjacent to the cluster, thus reducing primary clustering.

The following code shows constants and a record type used to manage simple hash tables. The hash tables described here use a resizable array of *THashType* records to store data items:

```

type
  // Values for hash table operations.
  THashInsertValues = (hashInsertOk, hashInsertTableFull,
                      hashInsertDuplicateKey);

  // Define a record for hashing examples.
  THashType = record
    Value : Variant;
    Key   : Longint;
  end;

  THashTypeArray = array [0..1000000] of THashType;
  PHashTypeArray = ^THashTypeArray;
    
```

Listing One (beginning on page 20) shows the Delphi source code for a hash table class that uses open addressing (available for download; see end of article for details). The main program creates a *TLinearHashTable* object, passing the constructor a parameter that indicates the number of entries the hash table should use. It can then use the *AddItem* function to insert items into the table and the *FindItem* function to locate items in the hash table. This function searches the hash table and returns the value of the item if the key is present in the table. If the key isn't present, the function returns the Variant value *Empty*.

Program Linear, shown in Figure 1, demonstrates open addressing with linear probing (available for download; see end of article for details). Enter a value in the # Values text box and click the Make Values button to make the program insert random values in the hash table. To keep the program simple, each item's value and key are the same and are numbers between 0 and 999.

Enter a value in the Value text box and click the Get button to make the program locate the item in the hash table. In Figure 1, the program has just located the item 565. Enter a value and click the Set button to make the program insert the item in the table.

## Quadratic Probing

Linear probing is fast and simple, but it suffers from an annoying primary clustering effect. When you add many items to the hash table, they tend to cluster together. The hash table shown in [Figure 1](#) is only half full, but many of the items lie next to others. That makes inserting and finding items slower than it would be if the items were more evenly spaced. For example, if you try to insert the value 161 in this table, its probe sequence will collide with the items 361, 662, 561, 764, 65, 964, 565, 668, and 469 before it finds an empty position.

If the values were spaced exactly evenly, there would be an empty spot next to every used position. Then the program could find an empty position for any new item in at most two tries.

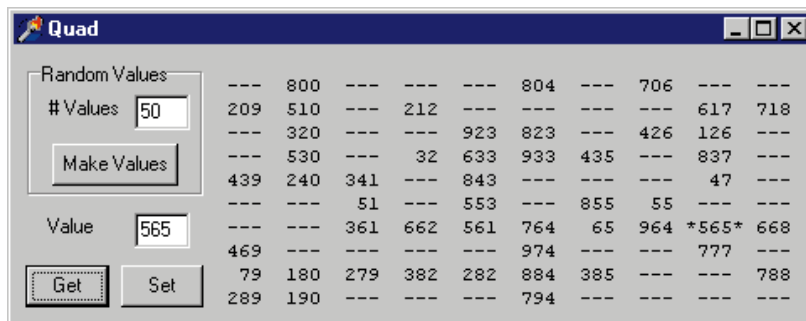
The reason clusters form is that there is a slightly higher probability that a new item will be inserted next to an existing item than it will land somewhere else. For example, suppose a hash table with  $N$  entries contains one item. When you insert a new item into the table, there is a  $1/N$  chance that it will land in any particular position. However, if the item maps to the same position as the item that is already in the table, it will be inserted next to that item, and it will form a small cluster. If the item maps to one of the positions next to the previous item, it will also start a small cluster. There is a  $3/N$  chance that the new item will land next to the previous item. As the table fills, the chances are good that large clusters will form.

The reason clusters form is that any item that maps to part of a cluster gets added to the end of the cluster. You can prevent that behavior using *quadratic probing*. In this method, the indexes in an item's probe sequence are given by the equation  $(K + P^2) \text{ Mod } N$  where  $N$  is the size of the table,  $K$  is the key, and  $P = 0, 1, 2, \dots$ . The program follows this probe sequence until it finds an empty position, or until it has checked  $N$  positions. At that point, the program gives up and refuses to insert the new item into the table.

[Figure 2](#) shows two probe sequences for an item being inserted into the first position in a hash table. The linear probe sequence examines a few adjacent positions, then inserts the new item next to the others, extending the cluster. The quadratic probe sequence, on the other hand, jumps through the array (skipping entries), and inserts the new item in a position that isn't adjacent to the cluster.

The way quadratic probing skips through the hash table makes clusters less likely to form and makes them grow more slowly. That means a program that uses quadratic probing can insert and find items more quickly (on average) than one that uses linear probing.

The code for the *TQuadraticHashTable* class is very similar to the *TLinearHashTable* class. The only differences are in the



**Figure 3:** The Quad program demonstrates quadratic probing.

*AddItem* and *FindItem* functions, so only these are shown in [Listing Two](#) (on page 21).

The Quad program is similar to the previous example, except it uses quadratic probing instead of linear probing (see [Figure 3](#)). Even though the same values were inserted in [Figures 1](#) and [3](#), the clusters in [Figure 3](#) are smaller.

## Pseudo-random Probing

Quadratic probing reduces primary clustering, but it still has some problems. Items that initially map to the same position in the array follow the same probe sequence. For example, if the hash table contains 100 entries, the values 100, 200, 300, and so on all follow the same probe sequence. If the program inserts many items that map to the same position, they will form a secondary cluster spread throughout the array. The effect is not as noticeable as that of primary clustering, but still reduces performance.

One way to eliminate secondary clustering is to use a pseudo-random probe sequence. With this technique, the locations in an item's probe sequence are given by  $(K + \text{Rand}(P)) \text{ Mod } N$ , where  $N$  is the size of the table,  $K$  is the key,  $P = 0, 1, 2, \dots$ , and  $\text{Rand}(P)$  is the  $P$ th number in a sequence of random numbers. The sequence of numbers depends on the key's value, so different values will have different probe sequences, even if they initially map to the same position.

In Delphi, you can use the *Random* function to produce sequences of pseudo-random numbers. To initialize *Random*, set the *RandSeed* system value equal to the new key's value. For example, the program uses the following code to initialize the random number generator for the value *new\_key*:

```
RandSeed := new_key;
```

When the program later needs to locate this item in the hash table, it sets *RandSeed* to the key value again. Then *Random* will produce the same sequence of numbers it produced when the item was added to the table. This is important. If it produced a new sequence of random numbers, the program would not be able to follow the same probe sequence it used to insert the item, so it would be unable to find the item.

The *TRandomHashTable* class is similar to the previous hash table classes. The only differences are in the *AddItem* and *FindItem* routines shown in **Listing Three**, beginning on page 21.

## Conclusion

While quadratic and pseudo-random probing often give better performance than linear probing, they also have some drawbacks. Neither of them is guaranteed to visit every item in the hash table. For example, suppose you have a hash table with only eight entries. To insert the item 4 into the table using quadratic probing, the program would follow this probe sequence:

```
4 + 02 = 4
4 + 12 = 5
4 + 22 = 8 = 0 mod 8
4 + 32 = 13 = 5 mod 8
4 + 42 = 20 = 4 mod 8
4 + 52 = 29 = 5 mod 8
4 + 62 = 40 = 0 mod 8
etc.
```

After eight probes, the sequence has only visited positions 0, 4, and 5. If all the table's entries are full except for entries six and seven, the program cannot insert the item in the table even though there is space available. Similarly, there is no way to know if or when a pseudo-random probe sequence will visit all the entries in the table.

Even so, quadratic and pseudo-random probing usually provide good performance when the table isn't too full. Of course, when the table holds a lot of empty entries, linear probing does quite well, too.

Using these hash-table classes, you can store items and search for keys extremely quickly. If you size your hash table properly, you can get excellent performance — sometimes better than the performance provided by a database doing indexed retrieval.

There are many other types of hash tables that are useful in different circumstances. For example, some work well when the data must be stored on a hard disk. You can learn more about hash tables and other algorithms in Rod's book *Ready-to-Run Delphi 3.0 Algorithms* [John Wiley & Sons, 1998]. The algorithms run in Delphi 3 or later. ▲

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\FEB\DI9902RS.*

Rod's book *Ready-to-Run Delphi 3.0 Algorithms* [John Wiley & Sons, 1998] has lots more to say about hash tables and other algorithms. For more information, visit <http://www.delphi-helper.com/da.htm>. You can contact Rod via e-mail at [RodStephens@delphi-helper.com](mailto:RodStephens@delphi-helper.com).

## Begin Listing One — The *TLinearHashTable* Class

```
type
  // A hash table with open addressing and linear probing.
  TLinearHashTable = class
  private
    HashTable : PHashTypeArray;
    NumItems  : Longint;
    NumUsed   : Longint;
  public
    constructor Create(size : Longint);
    destructor Destroy; override;
    function AddItem(new_value: Variant;
      new_key: Longint): THashInsertValues;
    function FindItem(target_key: Longint): Variant;
    function FindItemAndIndex(target_key: Longint;
      var index: Longint): Variant;
    function ValueByIndex(index: Longint): Variant;
  end;

  // Allocate the hash table.
  constructor TLinearHashTable.Create(size: Longint);
  var
    i : Longint;
  begin
    inherited Create;

    // Allocate the hash table entries.
    NumUsed := 0;
    NumItems := size;
    GetMem(HashTable, NumItems * SizeOf(THashType));

    // Initialize the hash table entries to Unassigned.
    for i := 0 to NumItems - 1 do
      HashTable[i].Value := Unassigned;
    end;

    // Free the hash table.
    destructor TLinearHashTable.Destroy;
  begin
    if (HashTable <> nil) then
      FreeMem(HashTable);
    inherited Destroy;
  end;

  // Add an item into the hash table. Return hashInsertOk,
  // hashInsertTableFull, or hashInsertDuplicateKey to
  // indicate our success or failure.
  function TLinearHashTable.AddItem(new_value: Variant;
    new_key: Longint): THashInsertValues;
  var
    probe : Longint;
  begin
    // Make sure there is room.
    if (NumUsed >= NumItems) then
      begin
        // The table is full.
        AddItem := hashInsertTableFull;
      end
    else
      begin
        // Find the first probe sequence value.
        probe := new_key mod NumItems;

        // While that position is occupied, try the next one.
        while
          (not VarIsEmpty(HashTable[probe].Value)) do begin
            // Make sure this key value is not already in use.
            if (HashTable[probe].Key = new_key) then
              Break;
            // Check the next position in the probe sequence.
            probe := (probe + 1) mod NumItems;
          end;

        // See if we found an empty position.
        if (VarIsEmpty(HashTable[probe].Value)) then
```

```

begin
  // Insert the item.
  HashTable^[probe].Value := new_value;
  HashTable^[probe].Key := new_key;
  Result := hashInsertOk;
  // Keep track of the number of entries used.
  NumUsed := NumUsed + 1;
end
else
begin
  // The key is already used.
  Result := hashInsertDuplicateKey;
end;
end; // End (NumUsed >= NumItems) ... else ...
end; // End function AddItem.

// Find an item in the hash table. Return the item's
// value or Unassigned if it is not here.
function TLinearHashTable.FindItem(target_key: Longint):
  Variant;
var
  trial, probe: Longint;
begin
  // Find the first probe sequence value.
  probe := target_key mod NumItems;

  // Search positions until we find the target, an entry
  // that is unused, or we have checked the entire table.
  for trial := 1 to NumItems do begin
    // See if this entry is unused.
    if (VarIsEmpty(HashTable^[probe].Value)) then
      Break;
    // See if this entry is the target.
    if (HashTable^[probe].Key = target_key) then
      Break;
    // Consider the next item in the probe sequence.
    probe := (probe + 1) mod NumItems;
  end;

  // See if we found the entry.
  if ((not VarIsEmpty(HashTable^[probe].Value)) and
    (HashTable^[probe].Key = target_key)) then
    // We found it. Return the target value.
    Result := HashTable^[probe].Value
  else
    // The value is not here. Return Unassigned.
    Result := Unassigned;
  end;
end; // End of function FindItemAndIndex.

```

## End Listing One

### Begin Listing Two — The *TQuadraticHashTable* Class

```

// Add an item into the hash table. Return hashInsertOk,
// hashInsertTableFull, or hashInsertDuplicateKey to
// indicate our success or failure.
function TQuadraticHashTable.AddItem(new_value: Variant;
  new_key: Longint): THashInsertValues;
var
  trial, probe: Longint;
begin
  // Try up to NumItems times to find an open position
  // or this key.
  for trial := 1 to NumItems do begin
    // Find the next probe sequence value.
    probe := (new_key + trial) mod NumItems;

    // See if the position is unoccupied.
    if (VarIsEmpty(HashTable^[probe].Value)) then
      begin
        // It is. Take this position.
        HashTable^[probe].Value := new_value;
        HashTable^[probe].Key := new_key;
        Result := hashInsertOk;
        Exit;
      end;
    end;
  end;
  // If we get here, we failed to find room.
  Result := hashInsertTableFull;
end; // End function AddItem.

// Find an item in the hash table. Return the item's
// value or Unassigned if it is not here.
function TQuadraticHashTable.FindItem(target_key: Longint):
  Variant;
var
  trial, probe : Longint;
begin
  // Search positions until we find the target, an entry
  // that is unused, or we have tried NumItems times.
  for trial := 1 to NumItems do begin
    // Calculate the next probe sequence value.
    probe := (target_key + trial) mod NumItems;

    // See if this entry is unused.
    if (VarIsEmpty(HashTable^[probe].Value)) then
      begin
        // The value is not here. Return Unassigned.
        Result := Unassigned;
        Exit;
      end;
    // See if this entry is the target.
    if (HashTable^[probe].Key = target_key) then
      begin
        // We found it. Return the value.
        Result := HashTable^[probe].Value;
        Exit;
      end;
    end;
  end;
  // If we get here, we failed to find the item.
  Result := Unassigned;
end; // End of function FindItem.

```

```

end;

// See if this is the target key.
if (HashTable^[probe].Key = new_key) then
  begin
    // It is. The key is already here.
    Result := hashInsertDuplicateKey;
    Exit;
  end;
end;

// If we get here, we failed to find room.
Result := hashInsertTableFull;
end; // End function AddItem.

// Find an item in the hash table. Return the item's
// value or Unassigned if it is not here.
function TQuadraticHashTable.FindItem(target_key: Longint):
  Variant;
var
  trial, probe : Longint;
begin
  // Search positions until we find the target, an entry
  // that is unused, or we have tried NumItems times.
  for trial := 1 to NumItems do begin
    // Calculate the next probe sequence value.
    probe := (target_key + trial) mod NumItems;

    // See if this entry is unused.
    if (VarIsEmpty(HashTable^[probe].Value)) then
      begin
        // The value is not here. Return Unassigned.
        Result := Unassigned;
        Exit;
      end;
    // See if this entry is the target.
    if (HashTable^[probe].Key = target_key) then
      begin
        // We found it. Return the value.
        Result := HashTable^[probe].Value;
        Exit;
      end;
    end;
  end;
  // If we get here, we failed to find the item.
  Result := Unassigned;
end; // End of function FindItem.

```

## End Listing Two

### Begin Listing Three — The *TRandomHashTable* Class

```

// Add an item into the hash table. Return hashInsertOk,
// hashInsertTableFull, or hashInsertDuplicateKey to
// indicate our success or failure.
function TRandomHashTable.AddItem(new_value: Variant;
  new_key: Longint): THashInsertValues;
var
  trial, probe: Longint;
begin
  // Initialize the random number generator for the key.
  RandSeed := new_key;

  // Try up to NumItems times to find an open position
  // or this key.
  for trial := 1 to NumItems do begin
    // Find the next probe sequence value.
    probe := (new_key + Random(NumItems)) mod NumItems;

    // See if the position is unoccupied.
    if (VarIsEmpty(HashTable^[probe].Value)) then
      begin
        // It is. Take this position.
        HashTable^[probe].Value := new_value;
        HashTable^[probe].Key := new_key;
      end;
    end;
  end;
  // If we get here, we failed to find room.
  Result := hashInsertTableFull;
end; // End function AddItem.

// Find an item in the hash table. Return the item's
// value or Unassigned if it is not here.
function TRandomHashTable.FindItem(target_key: Longint):
  Variant;
var
  trial, probe : Longint;
begin
  // Search positions until we find the target, an entry
  // that is unused, or we have tried NumItems times.
  for trial := 1 to NumItems do begin
    // Calculate the next probe sequence value.
    probe := (target_key + trial) mod NumItems;

    // See if this entry is unused.
    if (VarIsEmpty(HashTable^[probe].Value)) then
      begin
        // The value is not here. Return Unassigned.
        Result := Unassigned;
        Exit;
      end;
    // See if this entry is the target.
    if (HashTable^[probe].Key = target_key) then
      begin
        // We found it. Return the value.
        Result := HashTable^[probe].Value;
        Exit;
      end;
    end;
  end;
  // If we get here, we failed to find the item.
  Result := Unassigned;
end; // End of function FindItem.

```

```

    Result := hashInsertOk;
    Exit;
end;

// See if this is the target key.
if (HashTable^[probe].Key = new_key) then
begin
    // It is. The key is already here.
    Result := hashInsertDuplicateKey;
    Exit;
end;
end;

// If we get here, we failed to find room.
Result := hashInsertTableFull;
end; // End function AddItem.

// Find an item in the hash table. Return the item's
// value or Unassigned if it is not here.
function TRandomHashTable.FindItem(target_key: Longint):
Variant;
var
    trial, probe : Longint;
begin
    // Initialize the random number generator for the key.
    RandSeed := target_key;

    // Search positions until we find the target, an entry
    // that is unused, or we have tried NumItems times.
    for trial := 1 to NumItems do begin
        // Calculate the next probe sequence value.
        probe := (target_key + Random(NumItems)) mod NumItems;

        // See if this entry is unused.
        if (VarIsEmpty(HashTable^[probe].Value)) then
            begin
                // The value is not here. Return Unassigned.
                Result := Unassigned;
                Exit;
            end;

        // See if this entry is the target.
        if (HashTable^[probe].Key = target_key) then
            begin
                // We found it. Return the value.
                Result := HashTable^[probe].Value;
                Exit;
            end;
        end;
    end;

    // If we get here, we failed to find the item.
    Result := Unassigned;
end; // End of function FindItem.

```

### End Listing Three





By Thomas J. Theobald



## Multi-tier Database Applications

### Part II: Getting to the Code

**L**ast month we defined an application with four partitions: user interface, business logic, data access, and data storage. We explored the rationale for the various partitions, and described the steps for planning and building such an application. This month we turn to the implementation, i.e. the code. (All source referenced in this article is available for download; see end of article for details).

When I first had the idea for this series, I wanted to demonstrate a couple of ways of building the application. I've tried three, but I'm only going to demonstrate the easiest of them (since we're talking about getting stuff done on time, and on budget, this is probably for the best). I will, however, provide a brief description of the problems I encountered with the other two.

#### Model One: Four-tier "Straight-pipe" Model

This model attempted to take a "normal" three-tier model built in Delphi, and insert an additional physical tier into the stream (see Figure 1). In keeping with the principles described last month, I had hoped this would allow me to manage development as four separate projects, thus allowing me to redeploy whichever one needed changing.

Unfortunately, this meant that the road the information took — database-dataset/-provider-client dataset — needed an additional layer similar to the "client dataset" type to be created. Because *TClientDataSet* doesn't have its own provider, and a *TProvider* wouldn't accept a *TClientDataSet* as its dataset, I assumed that meant I would have to design a whole new component that would be a new variety of *TProvider* to accept the *TClientDataSet*. I don't have that kind of time,

although it sounds like it might be interesting to play around with.

#### Model Two: Inheritance

This model created an ancestor Data Access data module, from which descended the Business Logic remote data module (see Figure 2). With this model, I had hoped to apply a three-tier physical model, while maintaining the DA and BL layers as separate projects, producing a conceptual four-tier model that would still be easy to manage.

I found that, although it was simple to generate the ancestor DA element, I was forced to manually write the type library and all its attendant functions in the descendant class. If I were into writing interfaces, this could be fun, but I'm assuming that most developers have neither the time nor the inclination for this. It's entirely too messy for me to deal with as an application developer — perhaps as a tool developer, but not otherwise.

#### Model Three: Peer-level Middle Partitions

So, instead of a straight-through approach, I decided to make my middle partitions peers instead of hierarchical (see Figure 3). Each one became a remote data module unto itself, with the DA partition drawing rules and validation from the BL partition as needed. This, I hoped, would allow me to maintain physical and conceptual separation. Not only that, but I could also shamelessly pirate the EmpEdit demonstration from Delphi's included MIDAS demonstrations for this exercise.



Figure 1: The four-tier "straight-pipe" model.

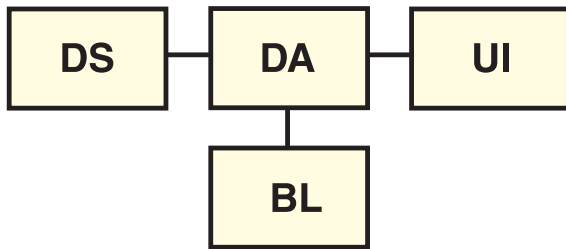


Figure 2: The inheritance model.

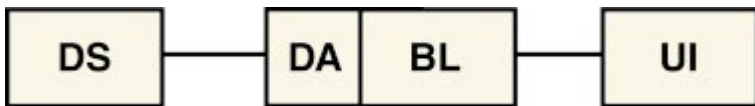


Figure 3: This model has peer-level middle partitions.

I'm going to direct the validation of a few fields at the DA partition to some functions provided from the BL partition. It'll be simple at first. After that, I'll change the rules a little, and actually stream a rules component over to the DA partition from the BL partition to accomplish the same goal (using a technique published in David Body's article "[DCOM Streaming](#)" in the March, 1998 *Delphi Informant*).

### How Do We Do This?

Using Delphi, we have a few components that are specific to this  $n$ -tier stuff. I'll run through the layers and describe the components we use to set it up.

On the DS side, there are no differences between two-tier, three-tier, or  $n$ -tier levels. Our data side doesn't care what gets access, as long as it fits its own criteria of who is allowed.

On the DA side, we build this application as if we were building a standard two-tier model. We use *TDataSet* descendants (*TStoredProc*, *TQuery*, and *TTable* — although probably not *TTable* if we're working with a SQL server) to get the data out of the data server. The difference here is that instead of a standard *TDataModule*, we use a remote data module.

If you select **File | New**, you'll see a **Remote Data Module** available. Choosing that will produce a data module that looks incredibly like the standard one, but includes an interface definition as part of its class, as well as a type library. As you add datasets to the remote, you'll publish them to outside applications by right-clicking on them and choosing **export <dataset name> from data module**. This will add a stub function to the type library, and the data module that returns a provider for that specific dataset. Note that you don't have to do this; there may be some datasets that exist solely for internal use, such as providing lookup fields.

This is where you drop the *TDatabase* control. There might be a temptation to make a single *TDatabase* on a separate data module and have every instance of your remote data module pass through it. Not bad, if you're using implicit transaction control.

However, if you ever want to write a commit or rollback condition, you'll be applying that condition to every attached user. Include a *TDatabase* on your remote data module to give each user separate transaction control. Someone will probably point out that this will do away with the license-cost benefit of having an  $n$ -tier system on servers that license at a per-connection rate; they may have been hoping that they only need to pay for a number of clients corresponding to the number of middle-tier attachments. If it means that much to you, don't use explicit transaction control, or buy a server that charges a per-client rate instead of per-connection, or perhaps one that charges on a per-connected-machine rate. If the savings on client licenses is an issue, development with a smaller model may need to be considered.

Let's look at a hypothetical case. If we have a corporation that can potentially buy licenses for 30,000 users at a ballpark of US\$300 each (don't quote me on that pricing; contact your database vendor), we're looking at roughly US\$9 million for client licenses (I'm sure bulk pricing brings it down a lot). If we put 100 users on each middle tier, we're down to 300 client licenses at a cost of US\$90,000, the cost difference being US\$8.1 million. Considerable. The server(s) software and hardware will probably be another few million. I would like to think that if one could demonstrate a difference of US\$8.1 million to a CEO, CFO, and CIO in the same room, the CEO would probably go along with the CFO and tell the CIO to buy the database server that provides those savings. Of course, each case will be different, but when given hard numbers and argued logically, pitching the case the development team wants shouldn't be that difficult.

It would also probably be of little use to generate anything more than a rudimentary UI for this partition because it won't be viewed by the user community. If your client wants one, great. Budget time for it. Just be aware it isn't necessary.

Now we have DS and DA defined. Next is the client side.

At the client, instead of standard *TDataSets* (I define standard as *TQuery*, *TStoredProc*, and *TTable*), we'll use *TClientDataSet* as our connection. We'll also use the *TRemoteServer* instead of a *TDatabase* here. The *TRemoteServer* will connect to the middle tier, and will give access to all the published *IProviders* on that middle tier. (Neat little factoid: If you don't know the providers coming to you, you can get a full list of them from the *TRemoteServer* once it establishes a connection; it's available through a procedure called *GetProviderNames*.) Step by step, this would be:

- 1) Drop a *TRemoteServer* in your UI application.
- 2) Choose or give it a computer and server name.
- 3) Drop a *TClientDataSet* in your application.
- 4) Wire its *RemoteServer* property to the *TRemoteServer* you just added.
- 5) Choose or enter a provider in the *ProviderName* property corresponding to one provided by the application server.
- 6) Set it active to see if you get data back.



Any of these steps could be done at run time, as well as design time. It might be a bit more difficult, but it's possible.

## Packets

These interactions revolve around “packets,” or “deltas,” which are simply small packages of information passed between the *TClientDataSet* and its application server.

What happens, in short, is as follows: When the client dataset opens, it calls the provider at the middle tier. The provider goes to its dataset, opens it (if it isn't open already), and retrieves a dataset. The provider then packages the dataset into what is called a *Data Packet*, and sends that back downstream to the client dataset. The client dataset then supplies that packet to the system as a representation of the data. The user then makes changes, deletions, or insertions, and tells the UI to *ApplyUpdates* (those of you familiar with cached updates will see many similarities here).

The client dataset acts on that command, assembles a Delta Packet consisting of any differences between the current state of the data at the UI and the original set it received from the provider, and ships this delta back to the provider. The provider then automatically applies this delta to the dataset as a change to stored data, row-by-row. If any updates fail, the provider stores the row with its old value (which it originally supplied to the client dataset), the current value (what the provider just found at the storage side), and the new value (which the user just tried to input) in a resulting data packet to be sent back to the client. The client dataset can then cycle through these erroneous records in its *ReconcileError* event.

## The Elements

As stated earlier, I've pirated a copy of the EmpEdit demonstration included with Delphi and C++Builder C/S versions. I've taken that application and enhanced it to some degree for this example. I suggest you take a copy for yourself and try adding the functionality I describe here, simply to get the hang of *n*-tier thinking.

**DS.** The first partition will be the database side. I'll start with one that everyone has access to and comes supplied for Paradox: DBDEMOS. I've already used DataPump to produce an InterBase version, and the same could be done for an Oracle, Sybase, etc. data side. Fortunately for this example, it's already done.

**DA.** The DA partition will be two partitions; because I'm trying to be independent of database vendors, I'm going to

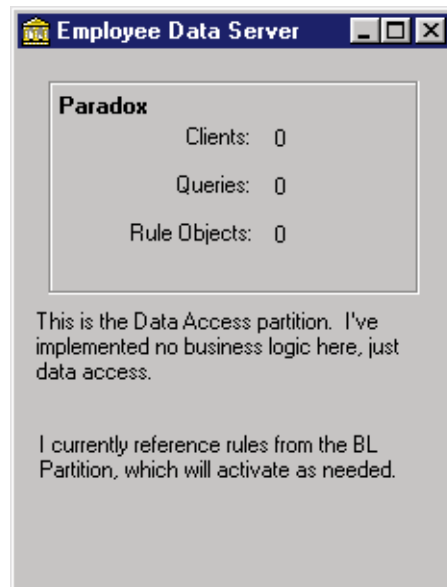


Figure 4: A Paradox-based DA element.

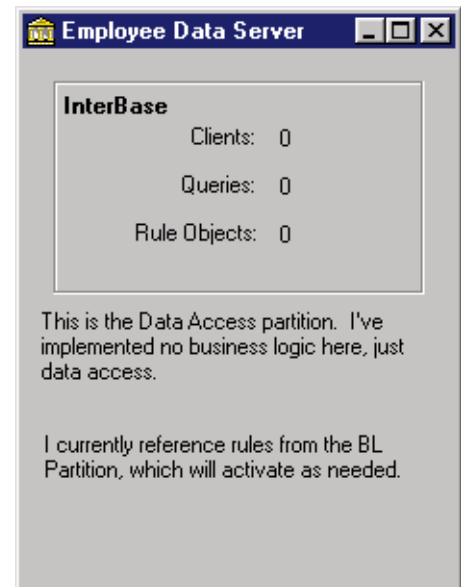


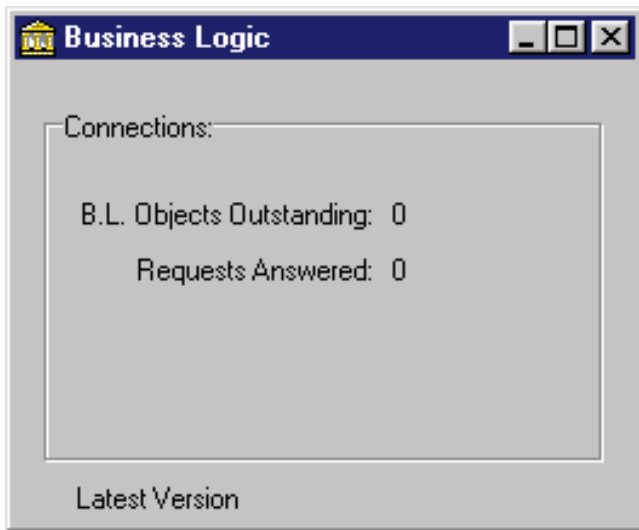
Figure 5: An InterBase version.

end up generating two DA elements that fulfill the same role. I'll start with a Paradox-based one (see Figure 4), and when I've finished the initial application, I'll create an InterBase version (see Figure 5) and install a switch at the UI.

The DA partition will end up linking to the BL partition using a *TRemoteServer*, and will pass validation requests on to the BL partition using that component's *AppServer* property. Operationally, this will be similar to a three-tier application; the difference will be in avoiding the work of recoding the business logic when we create our second DA partition.

This multi-tier stuff does deserve a note on how it seems to know what is going on across multiple machines. The idea of DCOM is that component objects will share a format. This makes it easier for one application to use the objects of another, and so forth. The means by which component objects are identified under COM/DCOM are Globally Unique Identifiers, or GUIDs. So far, GUIDs have proved to be just that — completely unique identification numbers for application components. If you look into a bare-bones type library (like the ones included with my source code, which are available for download; see end of article for details), you'll see three or four GUIDs.

When we tell a *TRemoteServer* about an application we want it to access, we first give it a machine name (or leave that blank if we intend to run off the same box) to identify under Windows where we are going to find the application server. We then specify a *ServerName* to identify the application we want to access; we could specify a bare GUID if we were certain of the one we need. When we make a connection, the *TRemoteServer* goes to Win32 and says: “Please find this object on this server and give me a handle to it.” If the local OS finds the machine in question, it asks it for a handle to the specific object, and returns the result. This is why the application server needs



**Figure 6:** The BL partition supplies a simple rule for salary validation.

to be registered before any other applications try to get to it (if Win32 doesn't know it's there, it can't find it).

Our DA partition will be an enhancement to the server application of the EmpEdit demonstration. The server application will have a few changes made:

- A display of how many rule objects it has invoked will be added, with modifications to the data module's *Create* and *Destroy* event handlers.
- The *EmpQuery* will have its fields instantiated at design time.
- The *EmpQuerySalary* field will have a validation event added.

**BL.** The BL partition of this system (see [Figure 6](#)) will supply a simple rule for salary validation (denying the change of a salary to more than US\$100,000 from this application), which will be referenced by the UI partition passing through DA. It will also surface a property called *SalaryBreak*, which will contain the value of the maximum salary allowed to be assigned by a non-CFO employee. Any changed data that passes through the DA partition will have to qualify as valid under that restriction, or be denied at the UI.

Additionally, I will install a function that will serve up a "live" object for the DA partition to invoke, which will contain much the same information as DA already retrieves from BL. The key point to note in this practice is that, when employing the "briefcase model," most users won't have access to the BL layer, even though they might have local copies of the data. By downloading a "rules object" to the UI, a briefcase-style user could then stream that set of rules to disk at shutdown, and re-load it every time the system runs. When the user finally reconnects, any new version of the rules server can be streamed down to replace the existing local one. I won't demonstrate this full briefcase technique, but I will have an object stream across to the DA side for use at that level. (The briefcase model will be the subject of an article by [Bill Todd in next month's Delphi Informant.](#))

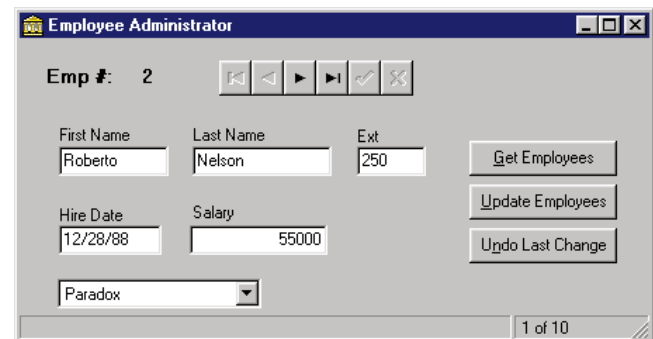
I'd like to point out that the example I've supplied uses a remote data module inside an application for a rules source. Given the nature of the application, it would probably be best run as a single-instance server on the same box as the DA partition, and every instance of the DA server's module could reference the same BL module. Some instances could require a multi-instanced version of the BL partition on a separate machine, but the conditions for that would be rare. I could just as easily have supplied a library file to do the same job as the module I've given here. In fact, this is the difference between an in-process and out-of-process solution. In-process means something tied directly into the process, while out-of-process refers to getting something from an external source. Generally, in-process servers are faster, but use the same machine — and therefore resources — as the application using them, while out-of-process servers take a little more time to operate, but operate in a resource pool that is isolated from the application using them.

I need to mention a few things about the code sample. The BL partition, and the two DA partitions, each have a couple of labels on them referring to "business objects." The BL partition has the label of B.L. Objects Outstanding and the DA partitions refer to "rule objects." These are misleading titles that I had intended to use as representations of the BL partition serving a complete binary object to each DA partition from which to access rules, using DCOM streaming. Unfortunately, I didn't have time to incorporate this before completing this article.

**UI.** This will be a very simple front-end, with the standard form from the original EmpEdit application and the error reconciliation routine from RECERROR.PAS (see [Figure 7](#)). I've added one change: a drop-down listbox with the names of both servers and the GUIDs they use to identify themselves. This allows the user to determine which server they attach to. (Developers would generally use this functionality themselves, but it makes a nice touch for the demonstration.)

## Conclusion

We've seen the components in Delphi/C++Builder that combine to make a functional *n*-tier application: *TClientDataSet*,



**Figure 7:** The UI uses a standard form from the EmpEdit application, and the error reconciliation routine from RECERROR.PAS. I've added a drop-down listbox with the names of both servers and the GUIDs they use to identify themselves.

*TRemoteServer*, *TProvider*, and remote data modules. Each has its role to play, and each fits in well with the DCOM platform this particular set was built to handle.

We also talked about planning. I want to stress again that this stuff isn't easy. Don't let anyone try to convince you this doesn't take a lot of sweat to produce. The application I've used as an example here is trivial (not to make light of the developer who built the demonstration I used as a foundation). A real *n*-tier application will take blood, sweat, and tears and, without proper planning, will never get done. I guarantee that if you don't plan properly, you'll not only have a bad application, you'll also have a high turnover rate from all the disgusted developers who won't be able to believe the architecture they're being asked to pursue.

I'd like to sign off with a note on the potential Internet use of this strategy: Deploying a thin client via the Internet will allow the developer to keep the business logic and storage inside the IS department's "ivory tower," but will allow the users to roam freely around the world. Deploy an ActiveX client application, and they'll be able to browse it from its cache freely. If you can pull that off, your users will love you. ▲

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99FEB\DI9902TT.*

Tom Theobald is a senior software developer with Response Networks, Inc. of Alexandria, VA. He began his career with computers as a NetWare engineer, moving later to include NT and Lotus Notes among his acquired skill set. Currently he is developing network response-time diagnostic tools used in the identification of network brown-outs and the prevention of down time. He can be reached at [ttheobald@responsenetworks.com](mailto:ttheobald@responsenetworks.com) with any questions or comments. Death threats and other matters of a personal nature can be forwarded to [eviltom@worldnet.att.net](mailto:eviltom@worldnet.att.net).





# THE API CALLS

Cryptography / Internet / Communications

By Mujahid Beg



## For Your Eyes Only

### Working with the Microsoft CryptoAPI

With the advent of the Internet as a critical business tool, data security has been pushed into the limelight. "Secure Commerce" and "Secure E-Mail" are becoming mainstream buzzwords. Understandably, in this age of Internet data, application support of security features has become extremely important. Historically, from a developer's perspective, security has always been difficult to implement. Bruce Schneier, noted cryptographer and author of *Applied Cryptography: Protocols, Algorithms, and Source Code in C* [John Wiley & Sons, 1995], says it well: "Building a secure cryptographic system is easy to do badly and very difficult to do well. Unfortunately, most people can't tell the difference."

The average cryptography developer has many obstacles to overcome. Besides the buzzwords and the new concepts, there are also complex algorithms and protocols to master. Then there are those nasty royalty and patent issues for the most popular and proven algorithms. Export restrictions on products using cryptography are also a major headache. On top of all this, just using secure and proven cryptographic algorithms in an application doesn't guarantee it's cryptographically secure. A weak implementation could turn a locked safe into a glass window, and, worse, the security holes may not be discovered until an important client loses a million transactions.

Enter the Microsoft Cryptographic Application Programming Interface, or CryptoAPI for short. Version 1.0 of CryptoAPI was shipped with Internet Explorer 3.0, and Version 2.0 was shipped with Internet Explorer 4.0 and Windows 95 OSR2. This article will explore

some basic CryptoAPI concepts and develop a *tCryptography* class to make its use from Delphi easier. Finally, the developed class will be used to build a utility program to encrypt/decrypt files.

As with any relatively new Win32 API, using CryptoAPI with Delphi is painful because the API header file from Microsoft (WinCrypt.H) is written in C, and a translated Delphi unit from Inprise isn't available yet. For this article's demonstration utility, Crypton, the relevant parts of the WinCrypt.H header file have been translated (this and other files referenced in this article are available for download; see end of article for details). This will allow for the encryption/decryption of data in any application, but all the signature and certificate-related stuff is left out. There are at least two freeware header file translators for Delphi, but neither of them were quite up to the task of automatically translating the complete header file.

Before delving into the details of CryptoAPI, a few cautions are appropriate. By virtue of its inclusion as part of the Windows operating system, many developers will be encouraged to include security functionality within their applications. This means any security hole in CryptoAPI will potentially affect a huge number of secured systems. At first glance, the chance of this happening seems extremely remote. But, although CryptoAPI was devel-

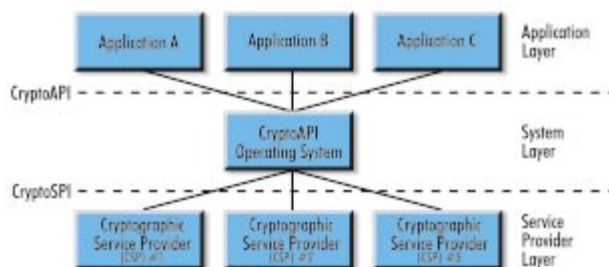


Figure 1: General architecture of the Win32 cryptographic system.

Algorithm	Base Provider Key Length	Enhanced Provider Key Length (Salt)
RSA public-key signature algorithm	512 bits	1,024 bits (No)
RSA public-key exchange algorithm	512 bits	1,024 bits (No)
RC2 block encryption algorithm	40 bits	128 bits (Yes)
RC4 stream encryption algorithm	40 bits	128 bits (Yes)
DES	Not supported	56 bits (No)
Triple DES (2-key)	Not supported	112 bits (No)
Triple DES (3-key)	Not supported	168 bits (No)

**Figure 2:** Base CSP versus Enhanced CSP.

oped by a group of talented designers and programmers at a multibillion-dollar company, it doesn't guarantee the security of the implementation. Consider the fact that most application developers will probably look to Microsoft's CryptoAPI documentation for examples and code snippets. John Boyer of UWI Unisoft Wares, Inc. published an article in the June, 1998 issue of *Dr. Dobbs' Journal*, wherein he points out how one of the sample programs that uses digital signatures with certificates has a severe security weakness.

Another factor that CryptoAPI brings to light has nothing to do with CryptoAPI itself. Because of the ease-of-use of this API, many developers will incorporate security features in their applications using this API. A fair percentage of these developers will have only a vague understanding of the vast and complex topic of cryptography. This creates a potential for a great deal of risk; it's far too easy for a developer to make seemingly clever design decisions that compromise an otherwise secure system.

## CryptoAPI Basics

An analogy might help in understanding the architecture of CryptoAPI. The Win32 Graphics Device Interface (GDI) layer provides an API layer between the application and the graphics card driver, and thus protects the developer from the painful experience of talking to each type of graphics card separately. In a similar fashion, the CryptoAPI provides driver-independent access to cryptography services. In other words, access to all cryptography services is done through a standard API layer that uses the implementation-layer DLL to do the real work. In the world of CryptoAPI, this driver is a Cryptography Service Provider, or CSP.

CSPs from multiple vendors may be installed on a single machine at any time. An application can explicitly choose a CSP to use, or let the system select the default CSP for a particular user. [Figure 1](#) shows the general architecture of the Win32 cryptographic system. Applications talk to the operating system through the CryptoAPI, and the operating system converts the calls to CryptoSPI (Service Provider Interface) calls. For security and other reasons, the application never directly talks to the CSP.

The CSPs are responsible for providing all the cryptographic services, including creation and storage of keys, encryption/decryption of data, storage and management of certificates, and hashing and signing functionality. Different CSPs can implement different sets of algorithms and key storage techniques — some store encrypted keys in the registry, while others may store them in a smart card or use a fingerprint as a persistent pseudo-key. Currently, there are CSPs available from Microsoft and several other vendors (check Microsoft's Web site, <http://www.microsoft.com>, for a listing).

Microsoft Base Cryptographic Provider is a general-purpose CSP that supports digital signatures and data encryption. This provider is included with Internet Explorer 3.0, or later, and will become part of future Windows operating systems. An enhanced provider, which is backward-compatible to the base provider but offers better security, is also available. Because of export restrictions, this CSP is only available to North American users. The difference between the two CSPs is summarized in [Figure 2](#).

In an application, the first requirement is to initialize the desired CSP (see [Listing Four beginning on page 31](#)). Information about the desired cryptographic provider is passed to the *CryptAcquireContext* function, which returns a handle to the provider in the form of an HCRYPTPROV type value. For security reasons, information between the CSP and the application is passed in the form of opaque handles. The value returned by this function is one such handle; there are others for keys, hash objects, etc.

## Keys to the Safe

Every CSP maintains its own set of keys for each user on a machine. The keys are stored in a persistent secure storage known as the *key container*. The most common place to save a key container is in the system registry, but it's not a requirement. A CSP could choose to store the keys on a smart card, or whatever medium is appropriate.

If there is no previously created key container available, the *CryptAcquireContext* call will fail. For the Crypton program, a key container isn't really necessary, because a user-supplied password will be used to generate the keys. But, none of the other CryptoAPI calls can be made without acquiring a cryptographic context first. So the *CryptAcquireContext* function must be called again with the CRYPT\_NEWKEYSET flag set to create a default-key container for the current user.

The *SetPassword* procedure takes a user-supplied password and generates a key from it (see [Figure 3](#)). To do this, a hash object is needed, which is created with a call to the *CryptCreateHash* function. The password string is hashed within this hash object with the *CryptHashData* function. Finally, a call to *CryptDeriveKey* is used to generate the key from the current hash object, after which the hash object and the old key are freed. The newly created key handle, an HCRYPTKEY type value, is stored within the object,

```

procedure tCryptography.SetPassword(PassCode: string);
const
  // Mix up the password a bit.
  cPassText = 'This (%) is the thing';
var
  hOldPassKey: HCRYPTKEY;
  hHash: HCRYPTHASH;
  msg: string;
begin
  if Length(PassCode) < cMinPassLen then
    RaiseErr(Format(
      'Password must be at least %d characters',
      [cMinPassLen]));
  // Create a hash object.
  if CryptCreateHash(hProv, cHASH_ALGID,
    0, 0, hHash) = False then
    RaiseErr('Unable to create has object');
  try
    msg := Format(cPassText, [PassCode]);
    // Hash the Formatted password string.
    if CryptHashdata(hHash, PChar(msg),
      Length(msg), 0) = False then
      RaiseErr('Unable to hash password data');
    // Create a key using this hash object.
    hOldPassKey := hPassKey; // Save old key handle.
    if CryptDeriveKey(hProv, cCRYPT_ALGID, hHash,
      cKEY_FLAGS, hPassKey) = False then
      RaiseErr('Unable to derive password hash key');
    if hOldPassKey <> 0 then // Free old key handle.
      if CryptDestroyKey(hOldPassKey) = False then
        RaiseErr('Unable to destroy old password key');
  finally
    CryptDestroyHash(hHash); // Destroy the hash object.
  end;
end;

```

Figure 3: The SetPassword procedure.

but the password string is not. Note that this provides for better security; the password string isn't available anymore, and the handle to the key is only meaningful to the CSP module within the current cryptographic context. This is an advantage of the opaque handle approach. An application might have a handle to some data, but the actual data (a key in this case) remains unavailable.

The user password string can be passed in to the *tCryptography* object constructor as a parameter. This way, the object can be created with the password, and then the password string may be destroyed or erased. The object with the saved key handle remains usable throughout the application. Although this scheme provides better security, in some applications you will need to change the password. The write-only password property lets you do just that.

The *tCryptography* object constructor automatically converts passwords to upper case before passing them on to the *SetPassword* function. In any significant application, this is a definite cryptographic no-no, because doing so reduces the key space significantly. For the Crypton utility, however, usability and ease-of-use were given more importance.

## Passing the Data

To encrypt a file, complete file contents are read into a buffer, encrypted, and then written back to the file. Because an encryption operation usually doesn't perform any compression of the data, the buffer needed to store the encrypted data (*cyphertext*) is the same size, or larger

```

// Encrypt or Decrypt a file depending on the passed in
// OpMode parameter. The input file name and the output
// file name may be the same. Since the entire file is read
// into a buffer before processing, there are no conflicts.
procedure tCryptography.DoOperation(OpMode: tOpMode;
  InFileName, OutFileName: string);
var
  Buff: pByte;
  BuffSize, CryptBuffSize, ClearTextSize: dWord;
  ft: tFileTime;
begin
  // Load the input file contents.
  AllocAndLoadBuffer(opMode, InFileName, buff, BuffSize);
  if Buff = nil then
    Exit;
  ClearTextSize := BuffSize;
  CryptBuffSize := BuffSize;
  try
    if opMode = omENCRYPT then
      begin // Encryption.
        // First we need to know the buffer size needed to
        // store the cyphertext.
        if CryptEncrypt(hPassKey, 0, True, 0, nil,
          CryptBuffSize, ClearTextSize) = False then
          if GetLastError <> ERROR_MORE_DATA then
            RaiseErr('Unable to get encrypted buffer size');
          // Allocate the required buffer.
          ReallocMem(buff, CryptBuffSize);
          BuffSize := CryptBuffSize;
          // Now encrypt the data buffer.
          if CryptEncrypt(hPassKey, 0, True, 0, buff,
            ClearTextSize, BuffSize) = False then
            RaiseErr('Unable to encrypt data buffer');
        end
      else // Decryption.
        if opMode = omDECRYPT then
          if CryptDecrypt(hPassKey, 0, True, 0, Buff,
            ClearTextSize, BuffSize) = False then
            RaiseErr('Unable to decrypt data buffer.' +
              #13 + 'Possibly incorrect password');
          // Save buffer to output file.
          SaveAndFreeBuffer(opMode, OutFileName, buff,
            ClearTextSize, BuffSize);
      except
        if buff <> nil then
          FreeMem(buff, BuffSize);
        raise;
      end;
    end;
  end;

```

Figure 4: The DoOperation procedure.

than, the original data (*plaintext*) buffer. On the other hand, in the decryption phase, the *plaintext* buffer required is the same size, or smaller, than the *cyphertext* buffer. Buffer sizes are adjusted accordingly in the *DoOperation* procedure (see Figure 4).

The two functions, *AllocAndLoadBuffer* and *SaveAndFreeBuffer*, handle file input/output for the object (see Listing Five on page 30). These two functions also manage the “Magic” text, a constant file header that's written at the beginning of every encrypted file. This enables the object to detect multiple-encryption attempts on an already-encrypted file, or decryption attempts of a plain-text (non-encrypted) file. Any string of characters unlikely to appear by chance in a file can be used as magic text. The *tCryptography* object uses “elifdet-pyrcnenasisiht”. In encryption mode, if the file header matches the magic text, it assumes the file is already encrypted and exits. After all, how often does a file start with the phrase “this

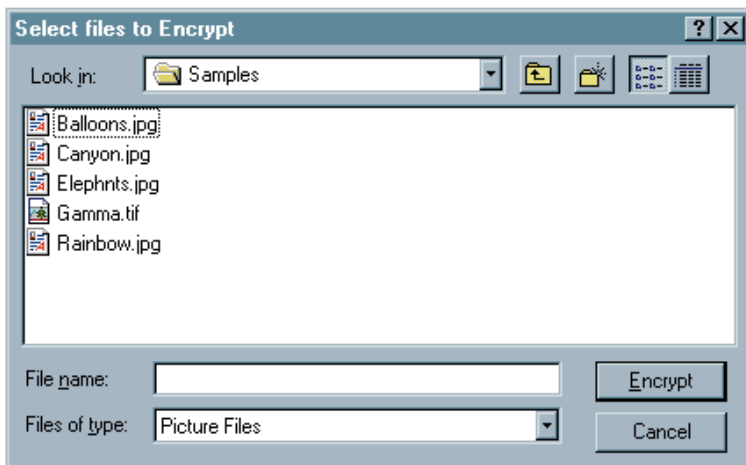


Figure 5: The customized “Open” dialog box.

```

procedure TMainForm.OpenDialogShow(Sender: TObject);
begin
  // Change the window caption.
  SetWindowText(GetParent(OpenDialog.handle),
    PChar('Select files to ' + cOpNameStr[CurrOpMode]));
  // Change the open button caption.
  SendMessage(GetParent(OpenDialog.handle),
    CDM_SETCONTROLTEXT, 1, Integer(PChar(
      '&' + cOpNameStr[CurrOpMode])));
end;

```

Figure 6: The `OpenDialogShow` event handler.

is an encrypted file” written backwards? The opposite happens in decryption mode; if the magic text doesn’t match, decryption is canceled.

## A Non-standard Dialog Box

During development of the utility program, a file selection dialog box, capable of allowing the user to select multiple files, was needed. The standard Windows file open dialog box would work, as long as the multiple file selection property was enabled. However, this standard dialog box is just a little too standard; the caption always reads “Open,” as does the default button caption. It’s preferable to customize this dialog box to reflect the actual operation (see Figure 5).

Changing the caption is easy. In the `TMainForm.OpenDialogShow` event handler, the `SetWindowText` function is called to perform this task (see Figure 6). Changing the `Open` button caption is a little trickier. The dialog resource for this dialog box is stored in a standard Windows system DLL, namely `ComDlg32.DLL`. The control ID of the `Open` button can be looked up by opening this DLL, and loading the dialog resource in a resource editor. In this case, the `Open` button control ID is 1 and the `Cancel` button ID is 2. Now it’s simply a matter of sending a `CDM_SETCONTROLTEXT` message (defined in `CommDlg` unit) to the appropriate dialog control. This is done in the same `OpenDialogShow` event handler with a standard `SendMessage` function call.

## Secure Data

Data security requires a lot of effort. There is no algorithm that’s completely unbreakable, except perhaps the One-Time-Pad

method. All other methods are some form of compromise. The idea is to have a system that takes so much effort to break that it becomes unfeasible in terms of time and/or money required.

The CryptoAPI gives the developer a clean, uniform way to implement proven algorithms and protocols, with minimal effort. But, best of all, because it will be part of the operating system, all kinds of applications will be able to use it seamlessly. At least, that’s the idea.  $\Delta$

*The files referenced in this article are available on the Delphi Informant Works CD located in `INFORM99\FEB\DI9902MB`.*

Mujahid Beg has been programming in various languages for the past 10 years. His primary development environments are Delphi and C/C++. He is the Director of Systems Development for MediNet-EDI Solutions — a medical EDI services company located in Houston, TX. He can be reached at [mujahid@insync.net](mailto:mujahid@insync.net).

## Begin Listing Four — Initializing the desired CSP

```

constructor tCryptography.Create(Container, Provider,
  Password: string; CanCreate: Boolean);
var
  r: bool;
  buff: array [0..1023] of Char;
  size: dWord;
  l, h: Byte;
begin
  inherited Create;
  hProv := 0;
  hPassKey := 0;
  fProviderName := '';
  fContainerName := '';
  fCryptVer := '0.0';

  if CryptAcquireContext(hProv, PChar(Container),
    PChar(Provider), PROV_RSA_FULL, 0) = False then
    // Unable to acquire context, check if failure was due
    // to absence of key container.
    if (CanCreate) and (GetLastError = NTE_BAD_KEYSET) then
      // Create a new key set for the current user.
      if CryptAcquireContext(hProv, PChar(Container),
        PChar(Provider), PROV_RSA_FULL,
        CRYPT_NEWKEYSET) = False then
        RaiseErr('Unable to acquire context');

    // Get some info about the cryptography module.
    // Name of the CSP.
    size := sizeof(buff);
    if CryptGetProvParam(hProv, PP_NAME,
      @Buff, size, 0) = False then
      fProviderName := 'Unknown'
  else
    fProviderName := StrPas(Buff);

    // Name of the key container.
    size := sizeof(buff);
    if CryptGetProvParam(hProv, PP_CONTAINER,
      @Buff, size, 0) = False then
      fContainerName := 'Unknown'
  else
    fContainerName := StrPas(Buff);

```

```

// Version number.
size := sizeof(buff);
if CryptGetProvParam(hProv, PP_VERSION,
    @Buff, size, 0) = False then
    fCryptVer := '*.00'
else
    fCryptVer :=
        Format('%d.%d', [Integer(buff[1]), Integer(buff[0])]);

// Set password key.
if PassWord <> '' then
    SetPassWord(UpperCase(PassWord));
end;

```

## End Listing Four

### Begin Listing Five — AllocAndLoadBuffer and SaveAndFreeBuffer

```

// This function is responsible for opening a file for
// reading, and reading the contents in a buffer.
procedure tCryptography.AllocAndLoadBuffer(OpMode: tOpMode;
    FileName: string; var buff: pByte; var BuffSize: dword);
var
    F: THandle;
    SizeRead: DWORD;
    TempBuff: array[0..cMagicTextLen] of Char;
    pTempBuff: PChar;
begin
    buff := nil;
    // Open and read in the file.
    F := CreateFile(PChar(FileName), GENERIC_READ, 0, nil,
        OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN, 0);
    if F = INVALID_HANDLE_VALUE then
        RaiseErr('Unable to open input file');

    try
        // Assuming filesize < 2 GB.
        BuffSize := GetFileSize(F, nil);
        // Check file header to verify file is encrypted.
        if OpMode = omDECRYPT then
            begin
                pTempBuff := @TempBuff;
                // Read in the header.
                ReadFile(F, pTempBuff^, cMagicTextLen,
                    SizeRead, nil);
                if StrLComp(cMagicText,
                    pTempBuff, cMagicTextLen) <> 0 then
                    RaiseErr('Not an encrypted file: ' + FileName);
                // Data size = File Size - Header Size.
                Dec(BuffSize, cMagicTextLen);
            end;

        // Allocate buffer to hold entire contents of the file.
        GetMem(buff, BuffSize);
        try
            ReadFile(F, buff^, BuffSize, SizeRead, nil);
            if BuffSize <> SizeRead then
                RaiseErr('Unable to read input file');

            // Make sure we are not attempting to encrypt an
            // already encrypted file.
            if OpMode = omENCRYPT then
                if StrLComp(cMagicText, PChar(buff),
                    cMagicTextLen) = 0 then
                    RaiseErr('File is already encrypted: ' + FileName)
        except
            FreeMem(buff, BuffSize);
            Buff := nil;
            raise;
        end;
    finally
        FileClose(F);
    end;
end;

```

```

end;

procedure tCryptography.SaveAndFreeBuffer(OpMode: tOpMode;
    FileName: string; var buff: pByte;
    SaveSize, BuffSize: dword);
// This function is responsible for opening a file for
// writing and saving the contents of the buffer.
var
    F: THandle;
    SizeWritten: DWORD;
begin
    // Open file in write mode.
    F := CreateFile(PChar(FileName), GENERIC_WRITE, 0, nil,
        CREATE_ALWAYS, 0, 0);
    if F = INVALID_HANDLE_VALUE then
        RaiseErr('Unable to open input file');

    // In Encryption mode - write a fixed file header before
    // the actual encrypted data. This protects against
    // multiple encryption attempts against the same file.
    if OpMode = omENCRYPT then
        WriteFile(F, cMagicText, cMagicTextLen,
            SizeWritten, nil);
    WriteFile(F, buff^, SaveSize, SizeWritten, nil);
    FileClose(F);

    FreeMem(buff, BuffSize);
    Buff := nil;

    if SaveSize <> SizeWritten then
        RaiseErr('Unable to write output file.');
```

## End Listing Five







*By Bill Todd*



## The BDE Made Easy

### Sharing the BDE on a Network

Installing and maintaining Delphi applications that use the BDE (Borland Database Engine) in a network environment is too costly if you install the BDE and the application on every PC. It's easy to solve half the problem by putting your application's EXE on the file server so that all users share a single copy. When you need to install an update, you only have to do it once, and the new version is immediately available to everyone. But what about the BDE?

Inprise recommends the BDE be installed on each PC to improve performance. However, the only performance gain is that the BDE DLLs can probably be read into memory faster from a local hard drive than from the file server. Because the load time of the DLLs is generally a minor component of overall application performance — unless the network is very slow — little is gained by installing the BDE on each PC.

The alternative is to install the BDE on the file server. Installing a single copy of the BDE on the file server offers several major benefits:

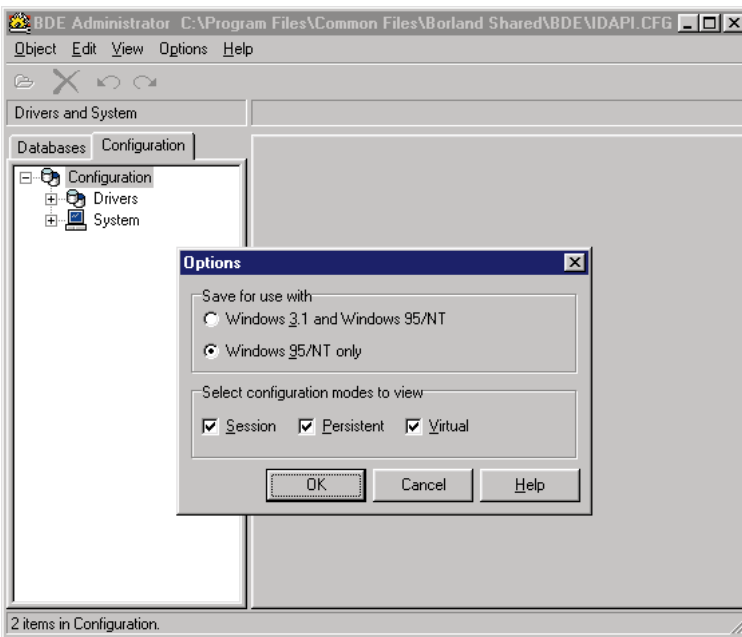
- 1) You only have to install the BDE one time in one location.
- 2) When you need to upgrade to a new version of the BDE, you only need to install the update one time in one location.
- 3) You can be certain that all users are running the same version of the BDE at all times.
- 4) Having all users share a single copy of the BDE configuration file guarantees that all users are using the same BDE settings.
- 5) If you need to add an alias, or change any other BDE configuration setting, you only need to make the change once, and it immediately affects all users.
- 6) A centralized BDE installation makes it easy to have your BDE applications configure the PCs on which they run to use the BDE the first time they run.

Using the techniques described in this article, you can take a new PC out of its box and connect it to the network, and the only thing you have to do to run a BDE application is create a shortcut to the EXE. There are other benefits, as well. You can have different applications use different BDE configuration files, or even different versions of the BDE. The interesting thing is that there are no tricks involved; the BDE was designed from the beginning to allow a single, central copy to be shared by multiple users.

There are two phases to the process of getting applications to automatically configure and use a central copy of the BDE. The first is getting the BDE installed on the file server. The second is adding code to your program so it will automatically create the BDE registry entries when it starts.

#### Phase One

Before you can use a file-server BDE installation, you must install the BDE on the file server. Neither the version of InstallShield Express that comes with Delphi, nor the commercial version of InstallShield Express allow you to install the BDE anywhere but the user's local hard drive. One solution is to install the BDE on one workstation, then copy the BDE directory to the file server. The disadvantage of this technique is that it leaves the BDE registry entries on the workstation pointing to the BDE



**Figure 1:** The BDE Administrator Options dialog box.

on the local hard drive. This can be easily overcome by letting your application reset the BDE registry entries when it runs.

While you can change the registry entries to point to the server, you can also avoid the problem by creating a set of installation diskettes that install the BDE files as though they were an application. This lets you put the BDE where you want it, and doesn't create the undesired registry entries. If your development machine is connected to the network to which you're deploying, you can also copy the BDE directory from your machine to the file server. If you use one of the Wise installation programs, you'll have no problem because Wise lets you install the BDE in any directory you wish.

Once the BDE is installed on the file server, you'll need to set the BDE configuration options in the BDE configuration file on the file server. Choose **Object | Open Configuration** to open the configuration file in the network BDE folder. Next, choose **Object | Options** to display the BDE Administrator Options dialog box shown in [Figure 1](#).

Make sure the **Windows 3.1 and Windows 95/NT** radio button in the **Save for use with** group box is selected. If you don't set this option, some of the configuration parameters will be stored locally in the Windows registry instead of in the configuration file. If settings are stored in the registry, they must be changed on every workstation. However, setting the **Windows 3.1 and Windows 95/NT** option causes all settings to be stored in the configuration file, and causes the configuration file settings to be used even if conflicting settings exist in the registry. You can also create aliases, or change any other settings at this time.

## Phase Two

The next step is to get your application to automatically create the BDE registry entries each time it runs. This system must meet the following requirements:

- All your applications must be able to determine the location of the BDE and its configuration file, and
- you must be able to change the location of the BDE easily.

This requires two .INI files. The first has the same name as your application's EXE — with an .INI extension — and is located in the same directory as the EXE. This file can contain any of the following entries in its [BDE] section:

```
[BDE]
ConfigPath=f:\foo\bde\idapi.cfg
BDEPath=f:\foo\bde
Overwrite=True
IniPath=f:\foo\bde.ini
```

The `ConfigPath` entry contains the full path to, and name of, the BDE configuration file. The `BDEPath` entry supplies the path to the BDE directory. By default, the code discussed later in this article will only create the BDE registry entries

if they don't exist. This prevents your program from changing existing entries. But what if you need to move the BDE to a new directory when a new file server is installed, or for some other reason? The `Overwrite` entry solves this problem by telling your program to create the BDE registry entries using the information in the .INI file, even if they already exist. By setting `Overwrite` to `True`, you could have several versions of the BDE installed on your server, and have different applications use different versions. You could also have all the applications share the same BDE, but have different applications use different BDE configuration files.

There is one disadvantage to this system: If you have 20 different Delphi applications, and you need to make a change, you'll have to change 20 .INI files. The `IniPath` entry addresses this problem. If the `IniPath` entry is present, all other entries in the BDE section of the application .INI file are ignored, and the BDE settings are read from the .INI file in the `IniPath` entry. This allows you to have a single .INI file that controls the BDE settings for many applications. This shared .INI file also contains a [BDE] section, and can contain all the previously shown values, except `IniPath`.

[Listing Six \(beginning on page 35\)](#) shows the code for the `TApplicationIniFile` component that automatically configures the BDE each time your application runs (this is available for download; see end of article for details). Although this is a component, and can be installed on the Component palette, you probably won't want to do so. Instead, copy the unit file into your project directory, and add it to your project. Because it's likely you'll have other information unique to each application to store in the application's .INI file, adding this unit to each project lets you easily customize the component to handle application-specific settings.

The overridden constructor of *TApplicationIniFile* calls its *OpenAppIniFile* method. *OpenAppIniFile* gets the path to the application's .INI file by using the *Application.ExeName* property and changing the file extension to .INI. It then checks for an IniPath entry in the [RemoteIniFile] section. If it finds this entry, it opens the file to which it's pointing. This allows you to have a family of related applications share a common .INI file.

The heart of *TApplicationIniFile* is the *CreateBDEKeys* method. This method begins by setting the Boolean variable *BDEInstalled* to True, and calling *OpenBDEIniFile*, a method almost identical to *OpenAppIniFile*, except that it looks for the IniPath entry in the [BDE] section of the application .INI file. The method then retrieves and saves the value of the Overwrite entry. The registry is accessed and updated using an instance of *TRegistry*. After the *TRegistry* object is created, its *LazyWrite* property is set to False to ensure that changes to the registry are saved immediately.

The next step is to determine if the BDE is already installed on this machine. The BDE is properly installed if the ConfigFile01 and DLLPath entries exist under KEY\_LOCAL\_MACHINE\Software\Borland\Database Engine, and the BLAPIPath entry exists under KEY\_LOCAL\_MACHINE\Software\BLW32. If any of these keys are missing, the *BDEInstalled* variable is set to False. First, *Reg.OpenKey* is called and passed to the Software\Borland\Database Engine key. Next, *Reg.ReadString* is used to read the ConfigFile01 and DLLPath keys. *Reg.OpenKey* is called a second time to move to the Software\BLW32 key, and *Reg.ReadString* is used to read the BLAPIPath value.

The final step is to create the BDE registry entries if the BDE isn't installed, or if the .INI file contains the Overwrite=True entry. If you've looked at the Software\BLW32 key (using Regedit, for example), you've seen that, in addition to the BLAPIPath value, there are entries for each of the BDE Language drivers. You don't have to create these keys when you install the BDE. However, if they're present, you need to change the path in each entry. The code creates a *TStringList* object named *LanguageKeys*, and loads all the value names under the Software\BLW32 key into the *StringList* by calling the *TRegistry* object's *GetValueNames* method. It then searches the *StringList* by calling its *IndexOf* method to determine if the BLAPIPath entry is present. If not, it's added to the list. Finally, a **for** loop iterates through the *StringList*, and changes the path for each key to the BDE path that was read from the .INI file earlier. Another call to *Reg.OpenKey* opens the Software\Borland\Database Engine key. Two calls to *Reg.WriteString* create the ConfigFile01 and DLLPath entries.

For *CreateBDEKeys* to work, you must call it before the program's default BDE session is created. Therefore, the only place you can put this call is in the **initialization** section of one of your program's unit files. The most natural location is the **initialization** section of the main form's unit. Here's an example:

```
initialization
  AppIni := TApplicationIniFile.Create(Application);
  try
    AppIni.CreateBDEKeys;
  finally
    AppIni.Free;
  end;
end.
```

## Conclusion

The advantages of installing the BDE on the file server and sharing it across all users are so great, it's surprising that Inprise doesn't recommend or, at least, document this option. It reduces the cost of installing the BDE and every application that uses the BDE initially, as well as the cost of installing BDE updates in the future. Having your applications automatically create the BDE registry entries offers similar benefits; you'll never get another call from a user who just got a new PC and none of the BDE applications run, even though the support techs copied all the files from the old hard drive. With self-configuring applications and a central BDE installation, anyone who can create a shortcut can install a BDE application on a new workstation.

The only trick to making this system work is to make sure the **Windows 3.1 and Windows 95/NT compatibility** option is set in the BDE Administrator Options dialog box; this is unfortunate. The **Windows 95/NT only** option offers no benefits and, in my opinion, should be removed so that all BDE configuration information is always stored in the BDE configuration file. This would ensure that any BDE configuration file could be safely shared. ▲

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\FEB\DI9902BT.*

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

## Begin Listing Six — ApplniFl.pas

```
unit AppIniFl;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls,
  Forms, Dialogs, IniFiles;

type
  TApplicationIniFile = class(TComponent)
  private
    AppIni: TIniFile;
```

```

BDEIni: TIniFile;
AppIniPath: string;
protected
  procedure OpenAppIniFile;
  procedure OpenBDEIniFile;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  procedure CreateBDEKeys;
end;

procedure Register;

implementation

uses Registry;

const
  cRemoteIniPathTag = 'IniPath';
  cRemoteIniSection = 'RemoteIniFile';
  cBDESection       = 'BDE';

constructor TApplicationIniFile.Create(AOwner: TComponent);
begin
  inherited;
  OpenAppIniFile;
end;

destructor TApplicationIniFile.Destroy;
begin
  AppIni.Free;
  inherited;
end;

procedure TApplicationIniFile.OpenAppIniFile;
begin
  AppIniPath :=
    ChangeFileExt(Application.ExeName, '') + '.ini';
  AppIni := TIniFile.Create(AppIniPath);
  { If there is a remote INI file path in the INI file,
    open the remote INI file. }
  AppIniPath := AppIni.ReadString(cRemoteIniSection,
    cRemoteIniPathTag, '');

  if AppIniPath <> '' then
    begin
      AppIni.Free;
      AppIni := TIniFile.Create(AppIniPath);
    end;
end;

procedure TApplicationIniFile.OpenBDEIniFile;
var
  BDEIniPath: string;
begin
  { If there is a remote INI file path in the BDE section,
    open that file, else set BDEIni to point to AppIni. }
  BDEIniPath := AppIni.ReadString(cBDESection,
    cRemoteIniPathTag, '');

  if BDEIniPath <> '' then
    BDEIni := TIniFile.Create(BDEIniPath)
  else
    BDEIni := AppIni;
end;

procedure TApplicationIniFile.CreateBDEKeys;
{ Creates the BDE Registry entries using values found in an
  INI file with the same name as the application's EXE file
  and located in the same directory. The INI file can
  contain the following sections and entries.

  [BDE]
  ConfigPath=f:\foo\bde\idapi.cfg
  BDEPath=f:\foo\bde
  Overwrite=True
  IniPath=f:\foo

```

```

[IniFile]
IniPath=f:\foo

If the Overwrite=True entry is present any existing BDE
entries will be overwritten with the entries from the INI
file. If the IniPath entry is present the BDE entries
will be read from the file at that location with the same
name as the EXE and the extension .INI. This allows a
central INI file to be used even if the app is installed
on each workstation. }

const
  cIniTrue           = 'TRUE';
  cOverwriteBDETag  = 'Overwrite';
  cIniConfigPathTag = 'ConfigPath';
  cIniBDEPathTag    = 'BDEPath';
  cDatabaseEngineKey = '\Software\Borland\Database Engine';
  cLanguageDriverKey = '\Software\Borland\BLW32';
  cLanguageDriverTag = 'BLAPIPATH';
  cConfigPathTag    = 'configfile01';
  cDllPathTag       = 'dllpath';

var
  Reg:           TRegistry;
  LanguageKeys: TStringList;
  ConfigPath:   string;
  BDEPath:      string;
  OverwriteBDEStr: string;
  S:            string;
  OverwriteBDE: Boolean;
  BDEInstalled: Boolean;
  I:            Integer;

begin
  BDEInstalled := True;
  { Open the INI file that contains the BDE information. }
  OpenBDEIniFile;
  { Read the BDE DLL and Config file paths from the INI
    file. }
  ConfigPath := BDEIni.ReadString(cBDESection,
    cIniConfigPathTag, '');
  BDEPath := BDEIni.ReadString(cBDESection,
    cIniBDEPathTag, '');
  if Copy(BDEPath, Length(BDEPath), 1) = '\' then
    BDEPath := Copy(BDEPath, 1, Length(BDEPath) - 1);
  { Determine if BDE section contains an Overwrite
    parameter. If so, and if the value is True, the BDE
    registry settings in the INI file will overwrite any
    existing settings on the user's computer. }
  OverwriteBDEStr := BDEIni.ReadString(
    cBDESection, cOverwriteBDETag, '');
  if UpperCase(OverwriteBDEStr) = cIniTrue then
    OverwriteBDE := True;
  Reg := TRegistry.Create;
  Reg.LazyWrite := False;
  try
    Reg.RootKey := HKEY_LOCAL_MACHINE;
    Reg.OpenKey(cDatabaseEngineKey, True);
    { See if any of the BDE registry keys are missing. If
      so, all the keys will be created from the values in
      the INI file. }
    if Reg.ReadString(cConfigPathTag) = '' then
      BDEInstalled := False;
    if Reg.ReadString(cDllPathTag) = '' then
      BDEInstalled := False;
    Reg.OpenKey(cLanguageDriverKey, True);
    if Reg.ReadString(cLanguageDriverTag) = '' then
      BDEInstalled := False;
    { If the BDE is not installed, or if the INI file's BDE
      section contains the Overwrite=True entry create the
      BDE keys. }
    if (not BDEInstalled) or (OverwriteBDE) then begin
      { Write the BLW32 language driver subkeys. }
      if BDEPath <> '' then begin
        LanguageKeys := TStringList.Create;
        { If there are no language driver keys, add the
          BLAPIPATH key to the list so it will be created. }
        try
          Reg.GetValueNames(LanguageKeys);

```

```

    if LanguageKeys.IndexOf(
        cLanguageDriverTag) = 0 then
        LanguageKeys.Add(cLanguageDriverTag);
    for I := 0 to Pred(LanguageKeys.Count) do begin
        { If this entry is the BLAPIPATH entry, just
          write the BDEPath. If it is the entry for one
          of the language driver files, extract the
          file name from the existing entry and add it
          to the end of the BDE path. }
        if UpperCase(LanguageKeys[I]) =
            cLanguageDriverTag then
            begin
                S := BDEPath;
            end
        else
            begin
                S := Reg.ReadString(LanguageKeys[I]);
                S := BDEPath + '\' + ExtractFileName(S);
            end;
        Reg.WriteString(LanguageKeys[I], S);
    end; // for
    finally
        LanguageKeys.Free;
    end; // try
end; // if
{ Write the DatabaseEngine subkeys. }
Reg.OpenKey(cDatabaseEngineKey, True);
if ConfigPath <> '' then
    Reg.WriteString(cConfigPathTag, ConfigPath);
if BDEPath <> '' then
    Reg.WriteString(cDllPathTag, BDEPath);
end; // if
finally
    Reg.Free;
end; // try
end;

procedure Register;
begin
    RegisterComponents('DGI', [TApplicationIniFile]);
end;

end.

```

## End Listing Six





## AT YOUR FINGERTIPS

Delphi / Tips

By Robert Vivrette



# They Were There All Along

## Fun with *SysUtils*

Those handy functions we need are often right under our noses — and we aren't even aware of it. Occasionally, I dig through the VCL source to see what interesting functions are there that I hadn't seen before. This month's "At Your Fingertips" is devoted to often-overlooked functions inside Delphi's *SysUtils* unit.

### Finding the Switch When the Lights Go Out

Many applications support command-line parameters. They're often used to control the initial state of the application, or to provide information about a configuration or document file to open.

One of the more awkward things a programmer must do is determine whether command-line switches have been passed into an application. This is often complicated by the fact that the switches can be in any order, might be a different case than expected, or even worse, might be a string value with embedded spaces. In the past, we had to write our own routines to determine this information. Generally, it involved scanning the command line, examining every character to see if it fit what we expected.

Help has arrived! *FindCmdLineSwitch* is an extremely handy routine that does all this for you. It's surprisingly powerful and flexible. All you need to do is pass in the switch for which you're looking, followed by the

characters that define a switch (usually a hyphen or a slash) and a Boolean value to indicate if you want the test to be performed with case sensitivity.

Figure 1 shows a simple example of how this function works. The demonstration shows the command line present when the application was launched (available through the global variable *CmdLine*). It then allows you to type in a switch value to test for. When you click on the **Test!** button, it tells you if the switch exists on the command line.

There are a wide variety of things for which you can test. For example, the routine correctly finds strings with embedded spaces (as long as you put them in quotes on the command line). It can also spot numbers, or switches with trailing on/off signs. Here's all the code necessary to run this demonstration:

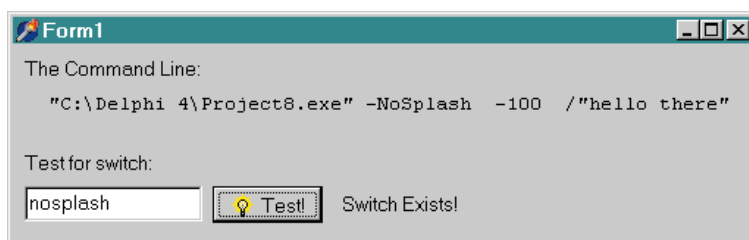


Figure 1: A simple example of how *FindCmdLineSwitch* works.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Label2.Caption := CmdLine;
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  if FindCmdLineSwitch(Edit1.Text, [
    '-', '/' ], True) then
    Label4.Caption := 'Switch Exists!'
  else
    Label4.Caption := 'Switch Not Present';
end;
```

## The Hunt Is On

Finding a file on a disk drive is a fairly common programming chore. Often, you'll use the *FileExists* function to determine if a particular file exists. The problem with *FileExists* is that it only looks at a single location on a disk drive. If the file name passed into *FileExists* includes path information, it will look there. If it has no path information, it will look at the current directory for the file.

But what if the file could potentially be in one of a number of locations? You could do something like this:

```
if not FileExists('C:\MyApp\string') then
  if not FileExists('C:\MyApp\Dir1\string') then
    if not FileExists('C:\MyApp\Dir2\string') then
      ...
```

But that starts getting to be a real maintenance nightmare!

Enter *FileSearch*. This function accepts two parameters: The first is the name of the file for which you are looking, and the second is a list of directories (separated by semicolons) in which to perform the search. For example, the code:

```
TheFile := FileSearch(
  'string', 'C:\MyApp;C:\MyApp\Dir1;C:\MyApp\Dir2');
```

would perform a search similar to the example shown previously.

If *FileSearch* finds the file in any of the directories, it will return a fully qualified path and file name to that file. If it doesn't find it, it returns an empty string. This comes in particularly handy when you're looking for a string that might be in more than one location.

*FileSearch* simulates the behavior present in the *Path* command used by DOS and Windows. In fact, if you want to search the path currently defined on the machine, you can also use the Windows API function *SearchPath*. The Win32 API Help discusses this function in more depth.

## A String in Need of Replacing

Delphi includes a number of interesting string-handling routines for inserting and deleting strings from within another string, two of which are *Insert* and *Delete*.

However, these functions will only perform a single insertion or deletion at a time. What happens if you want to replace all occurrences of a string with another string? In the past, programmers often had to build a **repeat** or **while** loop, where the string would be examined for a string using the *Pos* command. When the string was found, you could delete it with *Delete*, and insert the new string using *Insert*. The loop would continue until no more occurrences of the search string existed.

This problem is solved by means of one of the more overlooked functions in *SysUtils*: *StringReplace*. This function

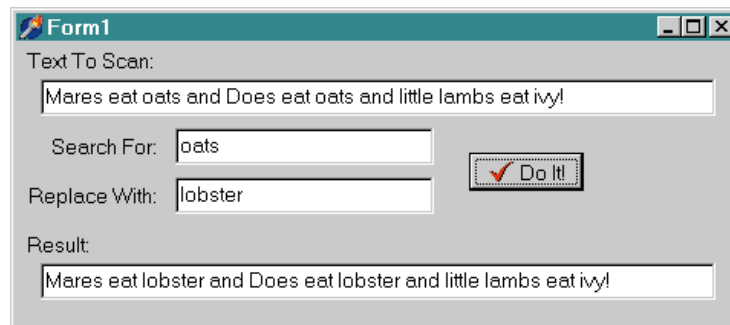


Figure 2: A demonstration of *StringReplace*.

takes a string and replaces in it all occurrences of one substring with another substring. It then returns the new string as its result.

The first parameter used is the string through which you'll be scanning. Next comes the string to search for and delete, followed by the string to put in its place. Last comes a parameter that allows you to specify a set of flags. Currently, these flags can be *rfReplaceAll*, which tells the function to replace all occurrences of the string rather than just the first one, and *rfIgnoreCase*, which tells the function to be case insensitive.

Figure 2 shows a demonstration program on the use of *StringReplace*. Clicking the **Do It!** button replaces all occurrences of text in the **Search For** edit box with the text in the **Replace With** edit box. The results are shown in the **Result** edit box. Here is the code that accomplishes this:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  Edit4.Text:= StringReplace(
    Edit1.Text,Edit2.Text,Edit3.Text,
    [rfReplaceAll,rfIgnoreCase]);
end;
```

There is one added capability worth noting: *StringReplace* allows for the possibility of multi-byte characters.

## Conclusion

One of the best training tools an aspiring Delphi developer has at his or her disposal is to look through the functions available in the source code provided with Delphi (Professional and Client/Server packages). Sometimes you find little gems, such as those outlined here, that can save many hours of programming effort.  $\Delta$

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\FEB\DI9902RV.*

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@mail.com.



## NEW & USED

By Warren Rachele

# LEADTOOLS Imaging 10

## A Sharp and Focused Imaging Toolkit

Users have come to expect the ability to use graphic and document images in their work. Such workaday applications as product databases now routinely offer an image of the product in addition to the text description. Paperless document systems require the ability to acquire and annotate their collected documents. All this is complex enough, but the number of formats available for storing graphic images makes the task of including image processing even more difficult. LEAD Technologies, Inc. has taken all this into consideration, and provides a one-stop solution.

LEADTOOLS Imaging is an imaging toolkit of considerable depth that should be considered when you need to integrate full-featured image processing into an application. The toolkit, which you can purchase off the shelf, is the same LEADTOOLS technology that's been licensed by such heavy hitters as Corel,

Microsoft, and Hewlett Packard. Now in version 10, the LEADTOOLS Imaging package is a collection of more than 600 functions, properties, and methods that provide high- and low-level control for image processing through a single ActiveX control.

The toolkit, implemented through LEADTOOLS, offers imaging technology in 15 broad categories: scanning, color conversion, display, multimedia effects, annotation, image processing, compression, image format import/export filters, imaging common dialog boxes, database imaging, Internet imaging, optical character recognition (OCR), screen capture, multimedia, and the FlashPix extension. It works with any development tool that supports ActiveX components and, as expected, it worked flawlessly with Delphi 4.

### Installation

Plan to devote some time for installation. An installation program creates the appropriate directory structures and registers the ActiveX controls with Windows; the rest is up to you. The documentation for the installation process is brief, so you'll need to be familiar with Delphi's Import ActiveX Control process to successfully integrate the LEADTOOLS controls into your IDE. Each component must be individually imported from the list of registered controls.

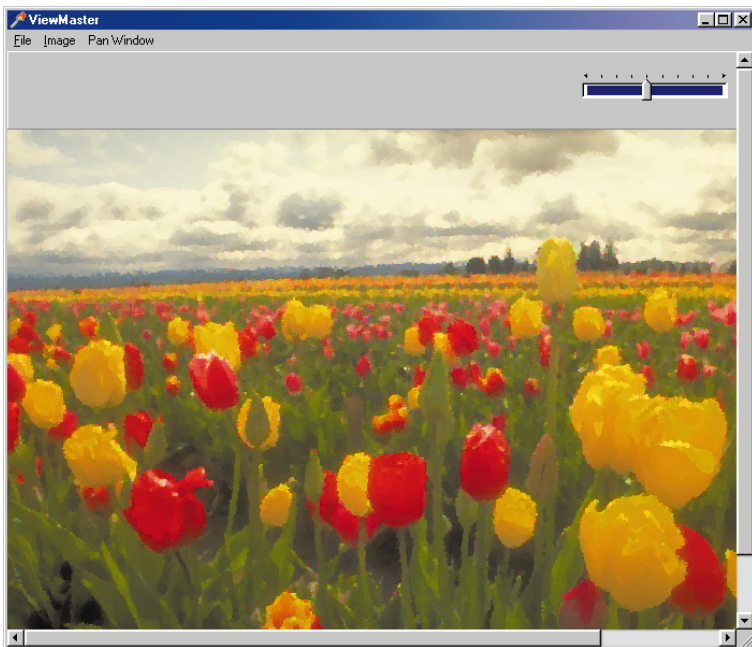
Color and Grayscale
JPEG and LEAD compressed (.JPG and .CMP)
GIF and TIFF with LZW compression
TIFF formats
BMP formats
Icons and cursors(.ICO and .CUR)
PCX formats (.PCX and .DCX)
Kodak formats (.PCD and .FPX)
DICOM format (.DIC)
Exif formats (.TIFF and .JPG)
Windows metafile format (.WMF)
Photoshop 3.0 format (.PSD)
Portable network graphics format (.PNG)
TrueVision TARGA format (.TGA)
Encapsulate PostScript (.EPS)
SUN raster format (.RAS)
WordPerfect format (.WPG)
Macintosh picture format (.PCT)
Windows AVI (.AVI)
Bi-tonal (1-Bit)
TIFF CCITT and other FAX formats
LEAD 1-bit format (.CMP)
Miscellaneous 1-bit formats (.MAC, .IMG, and .MSP)

Figure 1: Some of the over 60 file formats supported by LEADTOOLS.





**Figure 2:** This image is stored in LEAD's CMP format.



**Figure 3:** The Oilify effect renders the image as an oil painting.

Once the proper paths are set, each of the visual and non-visual components takes its place on the ActiveX page of the Delphi Component palette. LEAD includes all the controls they make on the CD, and all of them are registered with Windows; thus, they appear during the import process. Depending on the package you purchase, you may not be licensed to use all of them, and the controls will not function until a license number unlocks them. Don't clutter up your palette with non-functional controls.

### Using the LEAD Control

Putting the toolkit to work is as simple as placing the LEAD Main control onto a form in your project. This control encapsulates the majority of the imaging functionality of the toolkit.

Displaying an image in the sizable control is simply a matter of passing the image's file name as a parameter to the LEAD control's *Load* function. Once the image is defined, a wide array of manipulation is possible. The Main control is also fully database-aware, and can load images directly from a BLOB field.

The LEAD Main image control can be left at its default size and automatically sized to the client window during the display process. If the size of the image exceeds the client size, scroll bars can be automatically added. It isn't necessary to predefine the image type before loading it; image files carry information in their headers that define the storage format, and LEADTOOLS reads this to decide which filter is appropriate to use. LEADTOOLS supports more than 60 file formats (see [Figure 1](#)). The image displayed in [Figure 2](#) is a sample stored in LEAD's proprietary CMP format.

The greatest strength of LEADTOOLS Imaging is its image-processing capabilities; much more can be done with images beyond simply displaying them. Image manipulation is the core of the LEADTOOLS Imaging product, and version 10 expands its stable to support over 2,000 effects, more than 80 shapes, and over 30 gradients. The ViewMaster project has implemented a few of these features through menu options. The image in [Figure 3](#) has had the Oilify effect applied to it, rendering the image in the form of an oil painting. Nearly all the artistic effects available to the user can be interactively adjusted to meet the users needs. For example, the Slider control is used to determine the depth of the Neighborhood setting, modifying the sharpness of the brush strokes used in the "painting."

LEAD Technologies, through all its iterations, has learned from its customers and adapted to their common uses for the controls. To this end, many of the commonly used methods and properties have been encapsulated in dialog boxes for the convenience of the developer. The Imaging Common Dialog libraries provide a set of common dialog boxes that combine Windows dialog-box functionality with imaging features from LEADTOOLS. Placing the non-visual LEADDlg control on a form incorporates the libraries into your project, and gives you access to extended dialog boxes for FileOpen, FileSave, ColorResolution, Image Processing/Filtering, and Effects. Based on a programmer-determined set of user interface flags, the FileOpen dialog box adds or removes features such as the preview window and the File Information button.

A feature new to version 10 is the Pan Window. The Pan Window is a scaled view of the bitmap being displayed in the Main control. It becomes an active control that allows the user



**Figure 4:** The ScanMaster demonstration program.

to display different regions of the image without using the scroll bars. For example, if an image is zoomed in the Main control, the Pan Window displays the original image. The user can use the cursor that appears over the Pan Window to shift the displayed area of the main image to a specific point.

### Image Acquisition

In addition to displaying and manipulating image data, LEADTOOLS is adept at acquiring images. The ActiveX control supports TWAIN devices in its native form, and optionally supports ISIS input devices. Implemented directly through the Main control, the TWAIN interface provides all the necessary functionality to add image acquisition: device selection, acquisition, and error management. The Main control's file-save methods and properties are used to determine the final file-type characteristics. The TWAIN interface allows the programmer to choose from two development paths, depending on the target device. One choice that can be used with a scanner, such as the HP ScanJet that was used for testing in this article, is to let the scanner's own software handle the settings for the scan. In the ScanMaster demonstration program, this method was implemented through the **Get It** button. Few settings need to be programmatically managed using this method. A pair of calls to the *TwainSelect* and *TwainAcquire* methods will handle the job of acquiring the image for your program. **Figure 4** shows the demonstration program after acquiring an image and placing it in the Main control.

The programmer can also exercise complete control over nearly all aspects of the image acquisition process. The TWAIN device, and all the applicable settings, can be controlled from within the program, or through the user interface. The second button, **The Hard Way**, uses this development path, then displays some of the settings on the status

panel that were internally manipulated. Internally controlled acquisition bypasses the scanner software, and relies on the application settings to manage the process.

Acquiring data from the screen is another area in which LEADTOOLS shines. Simple, rectangular regions of the screen can be captured directly using the Main control alone. To extend the options and control over the screen capture function, LEADTOOLS includes a separate, non-visual control that encapsulates a greater range of methods and properties. The LEADTOOLS Screen Capture control can capture a wide variety of regions from the screen: FullScreen, ActiveWindow, ActiveClientArea, MenuUnderCursor, WindowUnderCursor, SelectedObject, MouseCursor, and Wallpaper. An area can also be selected using the freehand tool or a number of capture containment shapes, including circle, square, ellipse, and rounded-corner rectangles. In addition to screen capturing, LEADTOOLS can extract icon, bitmap, and cursor resources from 16- or 32-bit Windows EXE and DLL files.

### Internet Imaging

LEADTOOLS offers two alternatives for supporting Internet images. The first is a Netscape plug-in that works with Navigator or Internet Explorer. The plug-in DLL enables the browser to display any of the LEADTOOLS-supported image formats. This control is designed for use by HTML programmers to provide support for image data contained in your desktop application, or received through an Internet connection. Among other server requirements, the control requires that *Height* and *Width* values be embedded into the page. When the image is displayed on the page, a right mouse-click activates a menu offering options to copy to the Clipboard, zoom in and out, return the image to normal size, and save as a supported file format.

### Purchase What You Need

LEAD Technologies has extensively reorganized their product line. In previous incarnations, LEADTOOLS users could select the modules to include in their toolkit. With the exception of FlashPix support, the purchase of individual classes has disappeared; the package includes all formats, and you pick the ones you want.

The software reviewed here is the LEADTOOLS Imaging package. All the other toolkits build upon it. LEADTOOLS Imaging Pro includes the 16- and 32-bit APIs, which allow programmers to choose between a high-level component interface, or working at a lower level through direct interaction with the DLLs.

LEADTOOLS Multimedia opens access to audio and video tools. Programmers have access to three new capabilities: audio and video, including AVI and MPEG; capture from any Window's Video Capture Device, such as VCR or

Camcorder; and Internet streaming video. LEADTOOLS Multimedia Pro includes the APIs needed for low-level access to the DLLs.

LEADTOOLS Document Express includes all the technology included in the Multimedia toolkit, and expands it to meet the specialized demands of the document imaging market. The package includes the tools necessary to incorporate annotation of images through text, graphics, and sticky notes, and numerous other document processing features. These come in the form of new filters that allow the user to perform operations on images, such as despeckle, deskew, and other clean-up functions, as well as ultra-fast rotation of the image. The LEADTOOLS Document Express Suite includes the Xerox TextBridge, a full-featured OCR class.

LEAD also offers LEADTOOLS Medical Express for specialized, medical image processing. Images from MR and CT scanners are in a high-resolution format named DICOM. The Medical Express toolkit includes filters for this format that allow multiple-level gray-scale mapping and separation through a bit-range. It also includes all the capabilities included in the Multimedia packages.

LEAD has also modified their license agreement. Previous versions of the license required royalty payments based on the number of copies of your application that were sold. With the exception of the Document Express, Document Express Suite, and Medical Express packages, and the FlashPix module, the imaging technology is now royalty-free. These packages require a written royalty agreement on file with LEAD before their distribution. Another licensing issue that arises with the Imaging packages is support for GIF and TIFF LZW formats. This technology is copyright- and patent-protected by the Unisys Corporation, so you must license it from them before unlocking support for it in LEADTOOLS.

## Documentation

The quality of the documentation is the biggest negative issue that I identified with the LEADTOOLS products. The documentation for LEADTOOLS, with the exception of a slim, 61-page summary of features, is provided in PDF format and through Windows Help files. With a product this extensive, online documentation makes learning an onerous process; the back-and-forth of locating topics, pages, and references is better suited to the printed page. Printed manuals for all the products are available at an additional cost.

Learning to implement the ActiveX control is a test of your abilities to decipher and extract meaning from the terse statements contained in the 1,364 pages of documentation for the Main control alone. The introductory sections refer to the product in the most general of terms. For example, to load an image into the control, you are instructed to use the following method: LOAD Method.

If the process were that simple, this would be acceptable, but there are other steps involved in using the control. To locate these, you must find the section in the documentation that refers to your chosen development tool. The majority of the documentation for Delphi users focuses on version 2, with a new, very brief chapter for version 4. Code snippets with some comments are provided to teach you how to use the control's features. A property and method reference is included in the manual, but for the most part, it refers back to the tool-specific examples for expansion on the topic.

The Windows Help files are formatted much the same as the manual, but are available at the push of **[F1]**. The code snippets, when provided, can be quickly copied into your program to speed the development process. Plan on a long, steep learning curve to fully use the LEADTOOLS technology.

## Conclusion

If you need any form of image processing in your development efforts, LEADTOOLS Imaging is an outstanding package to include in your toolbox. The depth and quality of the classes are outstanding, and the amount of functionality you can quickly add to an application is simply amazing. As you work with the controls and discover the capabilities available, you'll find yourself implementing functionality in your application that probably wasn't a part of the original specification.

The documentation and installation provide bumps in an otherwise smooth implementation. Although the majority of the documentation is written for Delphi 2, the controls worked flawlessly with Delphi 3 and 4. Once located, the Help files also worked without fail. Current users of LEADTOOLS should consider this a must-have upgrade. The new features and increased speed make it worth the upgrade costs, although the integration of the new control into an existing project will take some work. **▲**

Warren Rachele is Chief Architect of The Hunter Group, an Evergreen, CO software development company specializing in database-management software. The company has served its customers since 1987. Warren also teaches programming, hardware architecture, and database management at the college level. He can be reached by e-mail at [wrachele@earthlink.net](mailto:wrachele@earthlink.net).

**INFORMANT**  
**FACT FILE**

LEADTOOLS Imaging 10 is an outstanding package, providing a one-stop solution for Delphi developers who wish to answer the demand for applications with full-featured image processing capabilities. This off-the-shelf toolkit provides high- and low-level control for image processing through a single ActiveX control. The installation process is a bump in an otherwise smooth implementation.

**LEAD Technologies, Inc.**  
900 Baxter St.  
Charlotte, NC 28204

**Phone:** (800) 637-4699 or  
(704) 332-5532  
**Fax:** (704) 372-8116  
**E-Mail:** [sales@leadtools.com](mailto:sales@leadtools.com)  
**Web Site:** <http://www.leadtools.com>  
**Price:** US\$495

## TEXTFILE



### Delphi 4 Bible

Tom Swan, author of *Delphi 4 Bible*, has had a long and prolific publishing career, documenting numerous programming languages and environments. The respect his books receive lends an additional note of credibility to Delphi as a development tool. Although the reader level is categorized as “beginning to advanced,” this work is light on beginner topics and heavily weighted toward the advanced programmer looking for task-specific information.

Swan begins by reviewing Delphi 1, and reiterates the key features of each version since before introducing the new features of Delphi 4. Features such as code completion, tool-tip expression evaluation, and the Module Explorer are reviewed in greater detail. Chapter 1 concludes with a feature that is often the best in each chapter, the Expert-User Tips. At the end of each chapter, Swan provides a list of quick shots and insights into Delphi programming. This mixture of hints, ideas, and fixes is an extremely valuable resource — one that readers will return to repeatedly.

The remaining chapters in Part I expand on this quick survey of the Delphi environment. Forms and components are briefly discussed, and some basic projects pull the beginning programmer into the RAD world. The programmer learns to place components on forms, and set the appropriate properties and code methods to build the simple projects. To support the beginning programmer, a minimal introduction to the Pascal language is unfortunately missing from this guide.

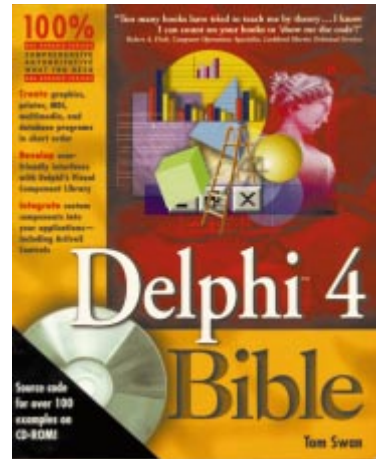
Part II explores the development of the user interface. *Delphi 4 Bible* takes a task-oriented approach to developing the user interface, showing the reader how to complete each step of the process. The first task explored is programming the keyboard and mouse. As with each of the task chapters, Swan introduces the components and methods used to include these services in an application. By the

end of Part II, the reader will have explored areas that include menus, buttons and check boxes, toolbars, lists, text of all types, files and directories, and dialog boxes. Because each of the elements of the user interface was explored in discrete fashion, the reader will not have seen all of these tasks coalesce into a complete application. Don't worry, Swan has a plan.

The application as a whole is the subject of Part III. Having covered the user interface components common to most applications in the previous section, Swan is free to concentrate on the “guts” of an application. These chapters are not in-depth references; rather, there's enough information in each to give the reader a taste of how Delphi supports the development of different types of applications. The experienced programmer will realize that the “applications” discussed are often found together in a single application; still, the single-topic chapters work well.

Swan covers graphics applications, printer applications, and working with the Clipboard, DDE, and OLE. Chapters dealing with database applications and the development of charts and reports are too brief; these topics often fill entire books. However, Swan succeeds in providing the essence of the task. All the information is well presented and, again, the Expert-User Tips that conclude each chapter are worthy of careful examination.

The final chapter is a grab bag of advanced techniques and topics. Ranging from the creation of a console application, to creating a DLL, to creating Internet applications, the chapter is fun to read and explore. However, the depth of some of this material could have been expanded to prevent the reader from wondering how to fully implement some of the ideas. What makes the chapter work are the broad strokes Swan uses to paint the many areas in which Delphi can serve the programmer. Tempted to fire up Turbo



Pascal for a quick DOS utility? Not after you see how easily a CRT application can be created in Delphi.

The book includes a CD-ROM containing all code from the book. It's unlikely you'll use it, however, given the size of the sample programs. I find that concepts are reinforced by creating the sample projects and typing the code.

Tom Swan is an excellent writer; his prose is clear, and is not disrupted by poor attempts at humor. Should you buy *Delphi 4 Bible*? Not if your programming skills are at the beginning stage. There is little here that will help the novice programmer advance. Advanced users may find much of the information in the book repetitive, though the Expert-User Tips are worthy of two or three reads. This book is aimed squarely at intermediate programmers looking to broaden their Delphi programming skills.

— Warren Rachele

*Delphi 4 Bible* by Tom Swan, IDG Books Worldwide, 919 E. Hillsdale Blvd., Suite 400, Foster City, CA 94404, <http://www.idgbooks.com>.

ISBN: 0-7645-3237-5  
Price: US\$49.99  
(953 Pages, CD-ROM)



## Toward a Stronger Delphi Community

In the “**Developer Ethics**” column I wrote last September, I explored the darker side of our community: certain sleazy, shameless “developers” — despicable rip-off artists who undermine the hard work of talented developers by stealing their software. Fortunately, they’re a tiny minority.

In stark contrast, our Delphi community is rich with developers who make positive contributions, freely sharing the fruits of their work with the rest of us. I’ll discuss some notable examples. Interestingly, I’ll also touch on a gray area of situations that can be interpreted positively, or negatively, depending upon your perspective.

**Sharing the wealth.** We all enjoy getting something for free, right? The developer who distributes his or her tools freely also benefits, making a name in the industry. With the Internet, such distribution has become almost trivial in its ease. We can find numerous examples of developers who have become very well known using this strategy. A recent one is Gerald Nunn. Because of his popular freeware library, GExperts (which I hope to write about in an upcoming issue), many of you already know his name. There are many more examples of these innovative writers who have made great tools freely available — Marco Cantù, Ray Lischner, and Bob Swart, to name a few.

The abundance of freely available Delphi tools has greatly enhanced this development environment. If you doubt the popularity of freeware/shareware in the Delphi community, consider the popularity of the major Web sites like the Delphi Super Page. Such sites have enhanced the Delphi community by making freeware/shareware components and tools easily available. And there are some real gems out there. Many come with source code, so we can learn new techniques while acquiring useful tools. This has definitely made our community stronger.

The Internet has strengthened our community in other ways. In the past couple of years, the Delphi Internet venues have expanded exponentially. Once just a handful of CompuServe forums and Usenet newsgroups, Delphi communication venues now include list servers, search engines, and sophisticated newsgroups.

Groups of developers also establish informal relationships with each other, sharing discoveries, tools, and code freely, while critically evaluating each other’s work as beta testers. Project Jedi, its ups and downs notwithstanding, has been a resilient and impressive endeavor. Even during times of little or no activity on the discussion threads, developers have been quietly working on various projects.

Another major strength of the Delphi community is the impressive collection of tool creators and third-party companies. What generally sets them apart from the average shareware developer is the quality of the tools they provide, including support and documentation. While you may pay much less — or nothing — for a shareware or freeware tool, you cannot expect to get the same quality.

**David and Goliath.** The gray area I alluded to is the heated battles that can occur when a freeware/shareware author decides to compete with a commercial product. While the up-and-coming developer may have nothing to lose, this is hardly the case with the established vendor. Commercial companies must continue to make a profit. Otherwise, they cannot provide new and improved tools. If they find themselves undermined by other developers who essentially clone their components or tools, their profits could very well shrink.

One could argue, “If a full-time company can’t compete with some guy working 10 hours a week, then that company shouldn’t be in business.” But consider the counter argument: “A free product that is similar in functionality to a selling product has the effect of decreasing the perceived value of that commercial product.” How so? If the commercial product sells for \$100 and has 20 features, while the competing freeware product duplicates 16 of them, how will the potential customer compare them? That customer may conclude that there

are just four unique features in the commercial product and value it accordingly. You don’t have to be a Wall Street guru to see the business implications inherent in that scenario.

Further, the freeware tool may have more bugs, may not be as well designed as the commercial product, or may come up short on documentation and customer support. But a buyer who’s not familiar with the products may not be aware of these possible problems. Conceivably, he or she might focus solely on the feature set of the two products, and base the perceived value of the commercial product on the sum of its unique feature set. In the previous example, the potential customer would place the value of the commercial product (at \$5 a feature) at just \$20. If the company were to reduce its selling price to a figure close to this, it might risk bankruptcy.

Clearly this is one gray area, and there may be others. However, even those developers who decide to play the role of David against Goliath are seldom satisfied with simply cloning the giant’s product; they usually want to add something new and distinctive. Do you agree with this assessment? On the whole, with our superior development tool, with our strong lines of communication, and with our many talented developers, we indeed have a strong Delphi community. ▲

— Alan C. Moore, Ph.D.

*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](mailto:acmdoc@aol.com).*

