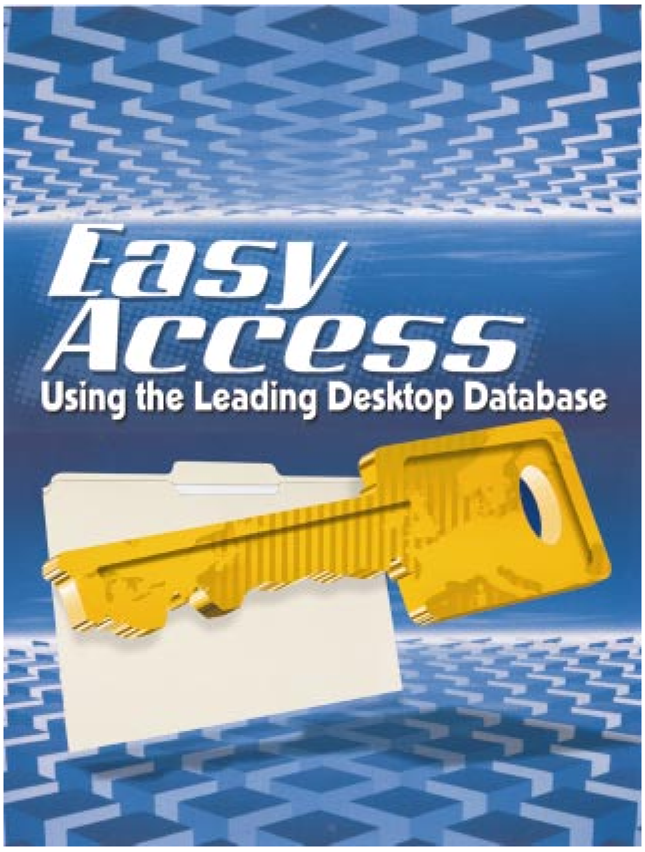
July 1999, Volume 5, Number 7



Cover Art By: *Darryl Dennis*

Delphi TOOLS

New Products and Solutions

SkyLine Tools Announces DICOM Suite for Delphi

SkyLine Tools Imaging announced the release of the *DICOM Suite for Delphi*, which enables developers to create medical imaging applications in Windows 95/98/NT that support the DICOM format. The DICOM Suite ships as an "add-on" to the ImageLib Corporate Suite, and includes an end-user DICOM application with Delphi source code. The suite also includes read and write functionality.

The DICOM Suite offers complete support for Version 3 of the DICOM format, and will read/write/display Version 3 DICOM images and animations.

Some of the features of the DICOM Suite include palette management for highlighting hard-to-find areas; support for study and patient information; real-time Windows leveling; DICOM Write, which can be used for Web publishing; ability to create animated GIFs or Multipage TIFFs from DICOM images; ability to swap, append, add, and delete frames; support for 24-bit grayscale lossless compression; Edge Detection for outlining sections of images; printing functions; and more.

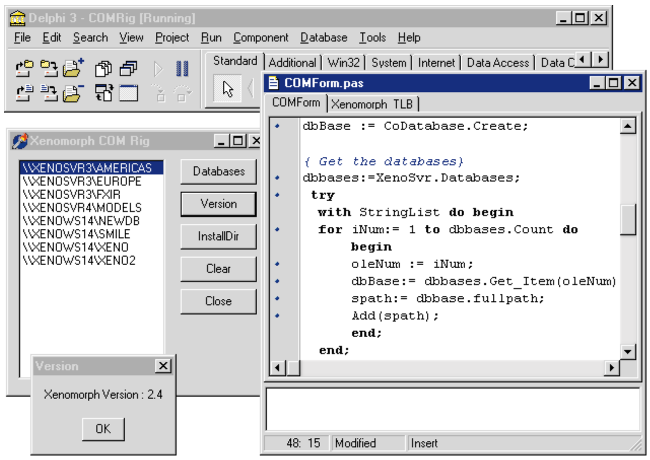
SkyLine Tools Imaging
Price: US$1,499 (read/write); US$899 (read only).
Phone: (800) 404-3832
Web Site: http://www.imagelib.com

Xenomorph Announces COM Interface for Xenomorph System

Xenomorph Software Ltd. announced the release of a new *COM interface for Xenomorph System*, the company's data management and analysis system for risk management, trading, and software application development.

The Xenomorph COM interface contains a new object implementation of the Xenomorph System that covers all aspects of multiple data feed access right through to volatility calculations and database structure manipulation.

The new object implementation is complementary to the Xenomorph System functional interfaces to C and Microsoft Excel. The existing functional interface to Visual Basic is also contained in the Xenomorph System COM library, removing the need for a separate .BAS file for VB, or awkward VB add-in reference paths in Excel.


Delphi 3 - COMRig [Running]
File Edit Search View Project Run Component Database Tools Help
Standard Additional Win32 System Internet Data Access Data C
COMForm.pas
COMForm Xenomorph TLB
Xenomorph COM Rig
Databases
Version
InstallDir
Clear
Close
Version
Xenomorph Version : 2.4
OK
48: 15 Modified Insert

```
\\XENOSVR3\AMERICAS
\\XENOSVR3\EUROPE
\\XENOSVR3\FXIR
\\XENOSVR4\MODELS
\\XENOWS14\NEWDB
\\XENOWS14\SMILE
\\XENOWS14\XENO
\\XENOWS14\XENO2
```

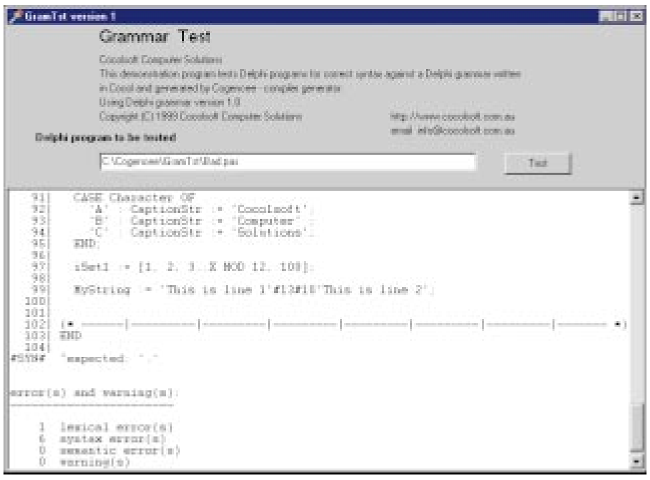
```
dbBase := CoDatabase.Create;

{ Get the databases}
dbbases:=XenoSvr.Databases;
 try
  with StringList do begin
  for iNum:= 1 to dbbases.Count do
      begin
      oleNum := iNum;
      dbBase:= dbbases.Get_Item(oleNum)
      spath:= dbbase.fullpath;
      Add(spath);
      end;
  end;
```

The COM interface allows system designers and implementers access across many different development environments, including Delphi, Visual C++, VB and VBA, and Java. Cross-platform support for the interface is available on many operating systems, such as OpenVMS, 64-bit DIGITAL UNIX, Sun Solaris, and Windows NT.

Xenomorph Software Ltd.
Price: Current users of Xenomorph System automatically receive upgrade.
Phone: +44 (0)181 971 0080
Web Site: http://www.xenomorph.com

Cocolsoft Announces Cocolsoft Delphi Grammar

Cocolsoft Computer Solutions announced *Cocolsoft Delphi Grammar*, grammar designed to work with program code written in Object Pascal for the Delphi compiler sold by Inprise Corp.



Cocolsoft Computer Solutions
This demonstration program tests Delphi programs for correct syntax against a Delphi grammar written in Cocol and generated by Cogencee - compiler generator
Using Delphi grammar version 1.0
Copyright (C) 1999 Cocolsoft Computer Solutions
http://www.cocolsoft.com.au
email info@cocolsoft.com.au

Delphi program to be tested
C:\Cogencee\GramTst\Bad.pas

```
   91|    CASE Character OF
   92|       'A' : CaptionStr := 'Cocolsoft';
   93|       'B' : CaptionStr := 'Computer';
   94|       'C' : CaptionStr := 'Solutions';
   95|    END;
   96|
   97|    iSet1 := [1, 2, 3..X MOD 12, 10];
   98|
   99|    MyString := 'This is line 1'#13#10'This is line 2';
  100|
  101|
  102| (* ------|---------|---------|---------|---------|---------|---------|--------- *)
  103| END
  104|
#SYN#  'expected: '.'

error(s) and warning(s):
----------------------------

    1  lexical error(s)
    6  syntax error(s)
    0  semantic error(s)
    0  warning(s)
```

Cocolsoft Delphi Grammar is one of the grammars included in the latest release of Cogencee, Cocolsoft's compiler generator for Delphi.

Some uses of the grammar are JavaDoc-like documentation for Delphi code, code metrics, pretty printer, or code formatting.

Cocolsoft Computer Solutions
Price: Included in Cogencee; Cogencee is available for Delphi 1 (16-bit) and Delphi 4 (32-bit) for US$300 (Standard), and US$500 (Professional).
E-Mail: info@cocolsoft.com.au
Web Site: http://www.cocolsoft.com.au/Dgram/dgramh.htm

Dart Announces ASP Support in MailBuilder

Dart Communications announced MailBuilder Internet Mail Toolkit now supports the creation of e-mail applications using Active Server Pages (ASP). MailBuilder and ASP make it easier to create server-side e-mail applications that users can access using any Web browser.

MailBuilder now includes two samples in the form of Microsoft Visual InterDev projects: one for sending e-mail, and the other for "popping" e-mail. The Send Mail sample includes facilities to upload and manage attachments using Dart's Mime control. Likewise, the POP3 sample includes facilities to download attachments once they are decoded.

MailBuilder's ready-to-use SMTP, POP3, and IMAP4 drop-in forms allow developers to instantly create useful e-mail applications. Sample applications with Visual Basic source are also included for a list server, an auto response server, a questionnaire editor and vote counter, a mail filter that sorts messages, a sort and forward sample that redirects mail, a bulk mailer, and more.

For more information, call Dart at (315) 431-1024, or visit the Dart Web site at http://www.dart.com.

Active+ Announces ServiceKeeper for Windows NT

Active+ Software announced the release of ***ServiceKeeper***, a Windows NT Server system utility that performs services scheduling and monitoring, and helps prevent crashes.

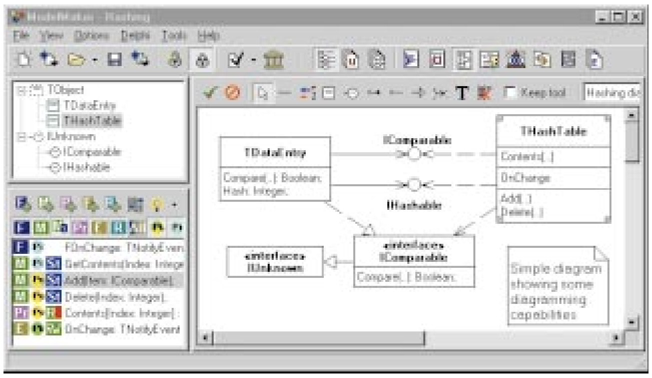
ServiceKeeper provides service failure detection, failed service rescue, error reporting, and integrated administration through Service+ and MMC SnapIn.

Service failures are detected by watching for specific events in the event log, looking up Windows NT counters, and checking TCP/IP protocols — HTTP (including protocol-specific errors 4xx and 5xx), SMTP, FTP, TELNET, PING, etc. ServiceKeeper's service scheduler stops services at night for backups, restricting access at specified time, and restarts services to clean up resources (memory, handles, etc.). Custom error detection is also possible by checking the exit code of a custom program.

ServiceKeeper runs on Intel and Alpha platforms and is Unicode aware.

Active+ Software
Price: US$195 (up to two monitored servers); US$3,995 (site license, unlimited).
Fax: 33 (4) 68054773
Web Site: http://www.activeplus.com

ModelMaker Releases ModelMaker 5 for Delphi



ModelMaker announced the release of ***ModelMaker 5 for Delphi***, a two-way productivity and UML-style CASE tool for generating and reverse-engineering native Delphi code.

ModelMaker supports drawing UML-style class diagrams, and, from that perspective, looks much like a traditional CASE tool. Its active modeling engine stores and maintains all relationships between classes and their members. All changes made (in diagrams or code) are reflected throughout the code and diagrams; code generation is instant. Creating and editing classes and members, overriding methods or properties, and other tasks are handled by selecting and clicking. Multiple-filtered views of the model (class tree, units, diagrams, members, etc.) give overview and restructuring capabilities. Existing code can be imported and visualized using a visualization wizard or simple drag-and-drop.

Documentation features include in-source comment generation, reverse-engineering of source comments, and generation of help files from documentation. Version 5 adds COM interface support, Code Templates, and an Open Tools API that allows you to create your own wizards.

ModelMaker
Price: US$199 for a single-user license.
E-Mail: info@modelmaker.demon.nl
Web Site: http://www.modelmaker.demon.nl

Innoview Announces MULTILIZER Supports Euro

Innoview Data Technologies Ltd. announced its ***MULTILIZER*** product now supports the euro currency, recently adopted by 11 of the countries in the European Economic and Monetary Union, including Austria, Belgium, Finland, France, Germany, Ireland, Italy, Luxembourg, the Netherlands, Portugal, and Spain.

The newest build (26.2.1999) of MULTILIZER adds support for the changes brought by EMU.

MULTILIZER is the RAD way to produce multi-lingual and localized software for the global market place. MULTILIZER can be used in Delphi 1 through 4, C++Builder 1, 3, and 4, and JBuilder 1 and 2.

Innoview Data Technologies Ltd.
Price: US$290 (VCL Edition Standard, without source code).
Phone: +358-9-4762 0550
Web Site: http://www.multilizer.com

Idyle Launches DirectUpdate 1.0

Idyle Software announced the release of ***DirectUpdate 1.0***, a software development kit that allows any computer product developer to integrate professional software update functionality.

Once a product is DirectUpdate-enabled, its user will be able to check for new versions, download, unzip, and install the product. DirectUpdate is designed to work on Windows 95/98 and NT 4. To integrate DirectUpdate into a computer product, a developer needs only to add a small EXE (58KB) and a DUF file (25KB), which are distributed with the product. The DUF file contains company and product names and logos, and the location of the DUVIR file on the internet. The DUVIR file is created by the developer for each new release, and is uploaded to a Web server. It contains the latest version information and download locations.

Idyle Software
Price: Free
Web Site: http://www.directupdate.com

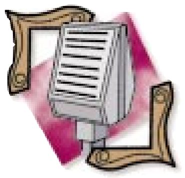
Primoz Gabrijelcic Announces GpProfile 1.2

Primoz Gabrijelcic announced the availability of GpProfile 1.2, his profiler for Delphi. The new version can parse sources that contain conditional compilation directives (*$IFDEF*/*$IFNDEF*, *$ELSE*, *$ENDIF*, *$DEFINE*, *$UNDEF*).

GpProfile 1.2 features include source instrumenting profiling for Delphi 2, 3, and 4; compatibility with Windows 95/98 and NT 4 and 5; multi-threaded program support; a built-in call hierarchy browser; the ability to instrument procedures written in built-in assembler; integrated source preview; an integrated result viewer; an API for profiling control; conditional API execution with meta-comments; and a layout manager.

GpProfile 1.2 also offers several bug fixes, including fixes for *TGpArrowListView*, Instrument&Run, and Run.

GpProfile 1.2 is available, free of charge, at http://members.xoom.com/primozg/gpprofile.



PrimeCare Uses InterBase to Reduce Healthcare Costs

Scotts Valley, CA — PrimeCare Systems, Inc. offers the PrimeCare Patient Management System, PrimeCare on the Web, and CodeComplier, applications that reduce healthcare costs and remove opportunities for error in diagnosis. These applications use the InterBase embedded database, allowing physicians to see up to 66 percent more patients and eliminating dictation and transcription costs by having patients respond to diagnostic computer questions before seeing the doctor.

InterBase provides the database used to store confidential records generated by the questionnaire-based PrimeCare Patient Management System, and is used internally for continual updates to the questionnaire base with the most recent information from the Mount Sinai School of Medicine.

InterBase is also at the heart of PrimeCare on the Web, a secure Internet version of the PrimeCare Patient Management System, enabling patients to answer the questionnaires via the Web before entering the physician's office.

More information on PrimeCare systems can be found at http://www.pcare.com.

Inprise Names Dale Fuller Interim President and CEO

Scotts Valley, CA — The Board of Directors of Inprise Corp. announced that Dale Fuller has been named Interim President, CEO, and a Director.

Fuller succeeds Delbert W. Yocam, the former Chairman and CEO, who resigned on March 31, 1999. To address unfounded rumors, the Board stated that it had requested Yocam's resignation because of philosophical differences regarding the company's growth strategy. The Board reaffirmed its confidence in the accuracy of the company's reported financial statements.

The Board is evaluating the potential benefits of separating Inprise into two independent companies as part of a comprehensive strategic review. Hambrecht & Quist, an investment banking firm, has been retained to advise the Board on the company's strategic alternatives.

Fuller joins Inprise with over 20 years of experience in general management, marketing, and business development in the technology industry. In 1997, he joined WhoWhere? Inc., one of the leading community sites on the Internet as President, CEO, and a Director. At WhoWhere?, Fuller led the expansion of numerous domain sites, including Angelfire.com and MailCity.com, a free e-mail site. Fuller also increased the company's consumer reach by 15 percent.

Previously, Fuller worked at Apple Computer, most recently as Vice President and General Manager of the Powerbook Business Unit. From 1994 to 1996, he was with NEC Technologies, Inc. as Vice President and General Manager, Portable Computer Systems.

The Board is also conducting a search for a new CFO and has retained the executive search firm Rusher Loscavio & LoPresto. The resignation of former CFO Kathleen M. Fisher, which was also requested by the Board, was announced on March 31, 1999.

OFUSA Uses InterBase in Internet-based System

Scotts Valley, CA — Office Furniture USA, Inc. (OFUSA), a network of office furniture manufacturers and dealers, features InterBase Software Corp.'s embedded database in an Internet-based system that improves productivity and financial growth by streamlining communication between the company, its partners, and the general public.

The system uses InterBase as its primary database, responsible for providing access and retrieval of vital day-to-day operations information.

InterBase enhances OFUSA's productivity in several areas, including a new sales order-entry application that prevents order-entry errors. InterBase also enables consumers to access a new online "shopping cart" that enables them to go to the Web site, browse the catalog, and receive a quotation and a reference to the nearest dealer.

The OFUSA solution was developed by United Systems Inc. and has been dubbed "Triple-Net" because it offers information to 1) the general public, 2) furniture dealers, manufacturers, and potential franchisees, and 3) OFUSA's customer service representatives, all from a single home page.

The OFUSA Web site is located at http://www.officefurniture-usa.com.

Borland Announces Borland JBuilder 3

Scotts Valley, CA — Borland announced Borland JBuilder 3, a new version of its visual development tools for Java developers. JBuilder 3 provides comprehensive support for the Java 2 platform and allows individual and corporate developers to create platform-independent business and database applications, distributed enterprise applications, and JavaBean components. At the time of this writing, JBuilder 3 was planned to eventually be available on Microsoft Windows, Solaris, and Linux.

JBuilder 3's open environment supports JDK 1.1.x, JFC/Swing components, JavaBeans, Enterprise JavaBeans, CORBA, RMI, JDBC, and all major corporate database servers.

JBuilder 3 is available in Enterprise, Professional, and Standard versions. JBuilder 3 Enterprise has an estimated street price (ESP) of US$2,499 for new users; JBuilder 3 Professional has an ESP of US$799; and JBuilder 3 Standard has an ESP of US$99.95.

For more information, visit the Borland Web site at http://www.borland.com.

table of CONTENTS

By Bill Todd

Easy Access

Using the Leading Desktop Database

Delphi 4 provides two ways to access data in Microsoft Access databases: the native BDE Access driver and ODBC.

Using the native BDE driver is as simple as creating an alias to your Access database. Begin by starting the BDE Administrator. Click the Databases tab if it's not already selected, then right-click on **Databases** in the tree view and choose **New**, or press Ctrl N to display the New Database Alias dialog box shown in Figure 1.

Choose **MSACCESS** from the drop-down list to display the new Access alias, as shown in Figure 2. Begin by giving the new alias a name. Next, enter the path to, and name of, your Access database in the **DATABASE NAME** field. If you want read-only access to the database, choose **Read Only** from the **OPEN MODE** drop-down list. If your Access database uses user-level security, enter the path to, and name of, the workgroup information file in the **SYSTEM DATABASE** field. You can also enter a default **USER NAME** if you wish.

Next, click the Configuration tab and expand the **Configuration**, **Drivers**, and **Native** nodes in the tree view (see Figure 3). Select the **MSACCESS** driver and check the **DLL32** setting. If you're going to open an Access 97 database, set this value to IDDA3532.DLL. To work with Access 95 databases, the setting should be IDDAO32.DLL.

Although the MSACCESS driver is a native BDE driver, it cannot open an Access database directly, because the Access file format is not published. Instead, the driver uses Microsoft's Data Access Objects (DAO) to interact with the Access database. You must have DAO version 3.5 installed to work with an Access 97 table, and DAO 3.0 for Access 95. Installing any Microsoft Office (e.g. Word) or developer program (e.g. Visual C++) on your system should also install DAO. DAO will remain on your system even if the program that installed it is removed. (You'll need to consult Microsoft for licensing requirements before distributing DAO with an application.)

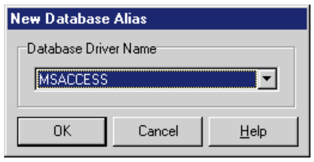


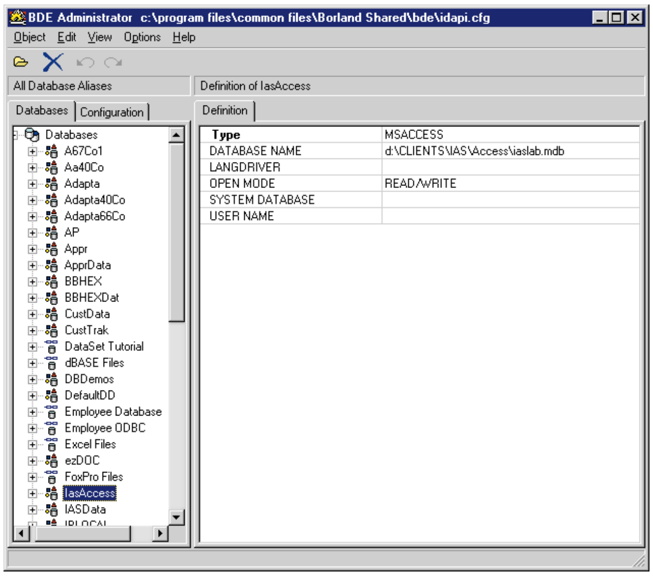
Figure 1: The BDE Administrator's New Database Alias dialog box.

To use the Access database in Delphi, create a new application and add a data module to it using the Data Module wizard in the Object Repository. Add a Database component, a Table, and a DataSource to the data module. Select the Database component and set the *AliasName* property to the alias for your Access database. Next, set the *DatabaseName* property to whatever name you want to use within this application.

If you're not using user-level security, and you're not using a database password on your Access database, set the *LoginPrompt* property to False. Figure 4 shows the property settings for the Database component used to open the sample Northwind database that comes with Access 97. If you don't, Delphi will prompt you to enter a user name and password each time you connect to the database. Finally, set the *Connected* property of the Database component to True to make sure you can connect to the database.

To open a table in the database, select the Table component in the data module and set its *DatabaseName* property to the same value you used for the Database component. Select the *TableName* property in the Object Inspector, and click the drop-down list button. You should see a list of the tables in the Access database. Choose the one you want. Next, set the Table's *Active* property to True. Now set the *DataSet* property of the DataSource component to the Table.

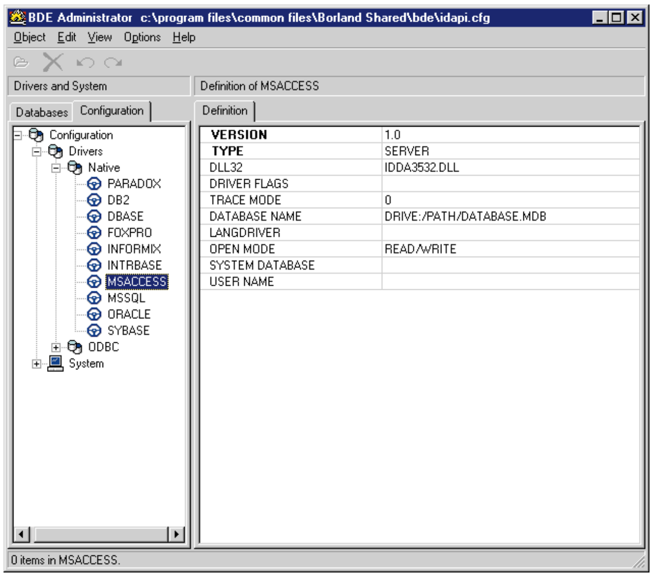
The rest of the process is no different than any other Delphi application. Move to the main form and add a DBNavigator and a DBGrid component. Select **File | Use Unit** from the menu, and add the data module's unit to the form's **uses** clause. The last step is to set the *DataSource* property of the grid and navigator to the DataSource component in the data module. When you set the *DataSource* property of the grid, you should see the data from the Access table.


BDE Administrator c:\program files\common files\Borland Shared\bde\idapi.cfg
Object Edit View Options Help
All Database Aliases
Definition of IasAccess
Databases
Configuration
Definition

Databases
- A67Co1
- Aa40Co
- Adapta
- Adapta40Co
- Adapta66Co
- AP
- Appr
- ApprData
- BBHEX
- BBHEXDat
- CustData
- CustTrak
- DataSet Tutorial
- dBASE Files
- DBDemos
- DefaultDD
- Employee Database
- Employee ODBC
- Excel Files
- ezDOC
- FoxPro Files
- IasAccess
- IASData

Type	MSACCESS
DATABASE NAME	d:\CLIENTS\IAS\Access\iaslab.mdb
LANGDRIVER	
OPEN MODE	READ/WRITE
SYSTEM DATABASE	
USER NAME	

Figure 2: An Access alias in the BDE Administrator.


BDE Administrator c:\program files\common files\Borland Shared\bde\idapi.cfg
Object Edit View Options Help
Drivers and System
Definition of MSACCESS
Databases
Configuration
Definition
0 items in MSACCESS.

Configuration
- Drivers
  - Native
    - PARADOX
    - DB2
    - DBASE
    - FOXPRO
    - INFORMIX
    - INTRBASE
    - MSACCESS
    - MSSQL
    - ORACLE
    - SYBASE
  - ODBC
- System

VERSION	1.0
TYPE	SERVER
DLL32	IDDA3532.DLL
DRIVER FLAGS	
TRACE MODE	0
DATABASE NAME	DRIVE:/PATH/DATABASE.MDB
LANGDRIVER	
OPEN MODE	READ/WRITE
SYSTEM DATABASE	
USER NAME	

Figure 3: MSACCESS driver settings.

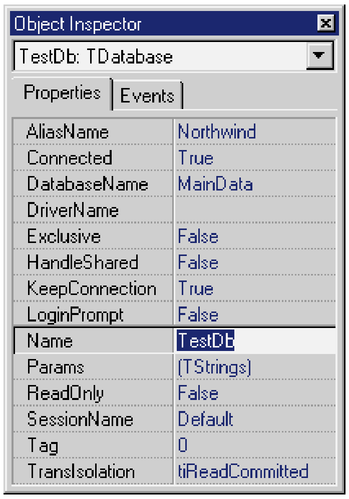
The sample application in the Connect subdirectory in the code for this article (available for download; see end of this article for details) shows the final product of these steps. Of course, you can also use Query components with Access tables.

Changes Made by Another User

While the native Access driver is easy to use, it has some unfortunate problems. The worst is that it does not detect changes made by other users. To see this, do the following:

1) Create a test application in Delphi, as previously described, that displays the employee table in the Northwind database.
2) Start Access 97 and view the same table.
3) In the Delphi application, change the last name of Nancy Davolio to Davoliox, but don't post the record.
4) In Access, change the name to Davolioy and move off the record to ensure that it's posted.
5) Return to your Delphi application and post the changed record. Note that you don't receive any warning that the record has been changed by another user.

If you follow the above steps in reverse, that is, change the record in Access in step 3, Delphi in step 4, and Access in step 5, Access *does* warn you that the record was changed by another user. The only solution to this problem is to add code to the Delphi application that saves the record in the *BeforeEdit* event handler. In the *BeforePost* event handler, the application must reread the record and compare it field-by-field to the values saved in *BeforeEdit* to see if it has been changed by another user. One way to do this is to use the *TdgReadTableBeforeWrite* component shown in Listing One (beginning on page 8).



AliasName	Northwind
Connected	True
DatabaseName	MainData
DriverName	
Exclusive	False
HandleShared	False
KeepConnection	True
LoginPrompt	False
Name	TestDb
Params	(TStrings)
ReadOnly	False
SessionName	Default
Tag	0
TransIsolation	tiReadCommitted

Figure 4: Sample Database component properties.

The *TdgReadTableBeforeWrite* component provides a way to check for changes made by another user. To use it, drop one *TdgReadTableBeforeWrite* component on your data module for each Table component, then perform the following steps:

1) Set the *Table* property of the *TdgReadTableBeforeWrite* component to the Table component it should monitor.
2) Create a *BeforeEdit* event handler for the Table component. In it, call the *TdgReadTableBeforeWrite.SaveRecord* method. This copies the record you're about to edit to a variant array.
3) Create a *BeforePost* event handler for the Table component. In it, call the *TdgReadTableBeforeWrite.RecordWasModified* method. This function returns False if the record hasn't been changed, or if the user chooses to overwrite the changes, and True if the record has been changed and the user chooses not to overwrite the changes.
4) Call *TdgReadTableBeforeWrite.Open* after opening the Table.
5) Call *TdgReadTableBeforeWrite.Close* when you close the Table.

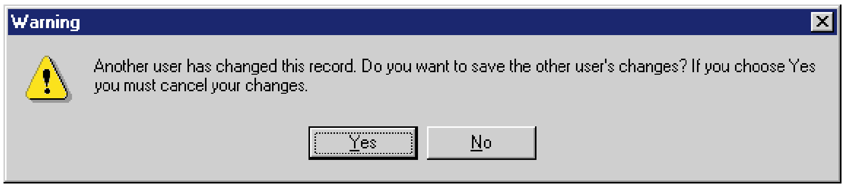
The project in the RecChanged directory is identical to the project in the Connect directory except that it uses this component to detect changes made by other users. Here's the *BeforeEdit* event handler for the EmployeeTable Table, which saves the record:

```
procedure TTestDm.EmployeeTableBeforeEdit(
  DataSet: TDataSet);
begin
  TestDm.EmployeeRw.SaveRecord;
end;
```

The following is the *BeforePost* event handler that aborts the post if the record has been changed:

```
procedure TTestDm.EmployeeTableBeforePost(
  DataSet: TDataSet);
begin
  if TestDm.EmployeeRw.RecordWasModified then Abort;
end;
```

Figure 5 shows the error dialog box the user sees when trying to post a change to a record that has been changed by another user. This dialog box offers two options, to abandon his or her changes, or to overwrite the changes made by the other user.



Warning

Another user has changed this record. Do you want to save the other user's changes? If you choose Yes you must cancel your changes.

Yes No

Figure 5: This dialog box warns that the record has been changed.

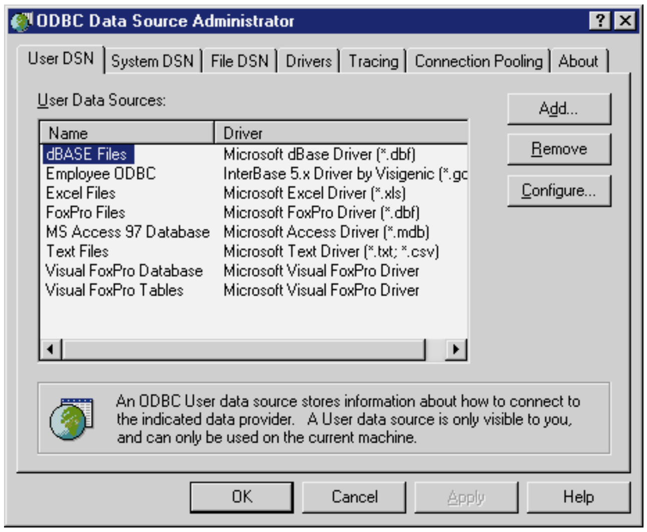
How It Works

The constructor for the *TdgReadTableBeforeWrite* component calls the *CreateShadowTable* method, which creates a Table referenced by the private member variable *FShadowTable* and sets its properties to connect it to the same table as the Table component referenced by the *Table* property. The *Open* method opens the shadow table.

Calling the *SaveRecord* method from the *BeforeEdit* method of the table being monitored calls *CheckValidTables*, which verifies that the *TableName* property is set and that the Table component for the shadow table exists. It then calls the shadow table's *GotoCurrent* method to position the shadow table to the record being edited, and saves the current record in the private variant, *FRec*.

The call to *RecordWasModified* in the *BeforePost* event of the table being monitored calls the shadow table's *Refresh* method so any changes made by other users will be seen. It then compares the shadow table record to the contents of the *FRec* variant array. If the record and the array don't match, another user has changed the record, and an error dialog box is displayed. The error dialog box offers the user the choice of overwriting the other user's changes, or canceling the changes the user has made.

The component's only other method is the overridden, protected *Notification* method. This method is called automatically when the user adds or deletes components, and — in this case — sets the *Table* property to **nil** if the Table it points to is removed.



ODBC Data Source Administrator

User DSN | System DSN | File DSN | Drivers | Tracing | Connection Pooling | About

User Data Sources:

Name	Driver
dBASE Files	Microsoft dBase Driver (*.dbf)
Employee ODBC	InterBase 5.x Driver by Visigenic (*.gc
Excel Files	Microsoft Excel Driver (*.xls)
FoxPro Files	Microsoft FoxPro Driver (*.dbf)
MS Access 97 Database	Microsoft Access Driver (*.mdb)
Text Files	Microsoft Text Driver (*.txt; *.csv)
Visual FoxPro Database	Microsoft Visual FoxPro Driver
Visual FoxPro Tables	Microsoft Visual FoxPro Driver

Add... Remove Configure...

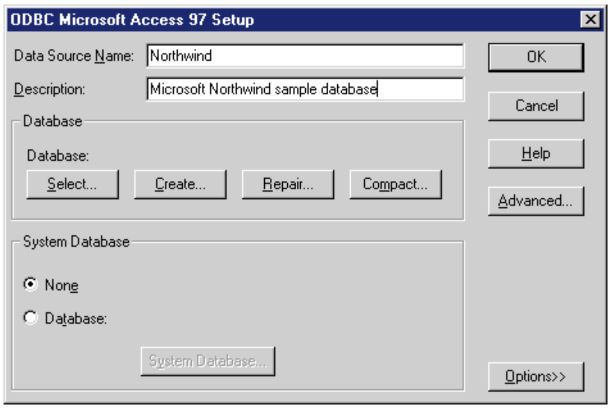
An ODBC User data source stores information about how to connect to the indicated data provider. A User data source is only visible to you, and can only be used on the current machine.

OK Cancel Apply Help

Figure 6: The ODBC Administrator.

AutoNumber Fields

Another problem encountered when using the native Access driver is large jumps in the value assigned to an AutoNumber field when records are inserted into a table. This happens if any of the fields in the table have a default value set. The only work-around for this problem is to assign default values in your Delphi program — not in the table definition.



ODBC Microsoft Access 97 Setup

Data Source Name: Northwind

Description: Microsoft Northwind sample database

Database

Database:

Select... Create... Repair... Compact...

System Database

(● None) ○ Database:

System Database...

OK Cancel Help Advanced... Options>>

Figure 7: The ODBC DSN Setup dialog box.

Using ODBC

The second way to connect to an Access database is with the Microsoft Access ODBC driver. The first step in using the ODBC driver is to create a data source name (DSN) using the 32-bit ODBC Administrator. Begin by opening the ODBC applet in Control Panel (see Figure 6). Select the User DSN page to create a DSN for the current user, or the System DSN page to create a DSN for all users. Click the **Add** button, choose **Microsoft Access Driver (*.mdb)**, then click **OK** to display the Setup dialog box (see Figure 7).

Enter a name and description, then click either the **Select** or **Create** button, depending on whether the DSN will be connected to an existing database, or a new one that must be created. If the database uses Microsoft's user-level security, click the **Database** radio button in the **System Database** group box and use the **System Database** button to select the workgroup information file for the database.

Once the DSN has been created, using it in a Delphi application is no different than using the native driver alias described earlier in this article. The DSN will automatically appear in the drop-down list of alias names for your Database component. It's interesting to note that the ODBC driver has exactly the same problem detecting changes made by another user that the native driver has. If you follow the steps given earlier in this article, you'll see that, again, the user isn't warned if the record he or she is trying to post has been changed by another user.

```
class procedure TAccessUtils.CreateAccessDb(
  DBName: string);
var
  DBEngine:  Variant;
  Workspace: Variant;
const
  Language = ';LANGID=0x0409;CP=1252;COUNTRY=0';
  Version  = 32;
begin
  try
    { Get a handle to the Database Engine. }
    DBEngine := CreateOleObject('DAO.DBEngine.35');
    { Get a handle to the Jet Engine Workspace object. }
    Workspace := DBEngine.Workspaces[0];
    { Create the database. }
    Workspace.CreateDatabase(DBName, Language, Version);
  except on E: EOleException do
    ShowMessage('Could not create database. ' + E.Message);
  end; // try
end;
```

Figure 8: Creating an Access table.

```
class procedure TAccessUtils.CompactAccessDb(
  DBName, TempDBName: string);
var
  DBEngine: Variant;
begin
  try
    Screen.Cursor := crHourGlass;
    try
      { Get a handle to the Database Engine. }
      DBEngine := CreateOleObject('DAO.DBEngine.35');
      { Compact the database into a new file. }
      DBEngine.CompactDatabase(DBName, TempDbName);
      { Delete the original database and
        rename the new one. }
      DeleteFile(DBName);
      RenameFile(TempDbName, DBName);
    finally
      Screen.Cursor := crDefault;
    end; // try
  except on E: EOleException do
    ShowMessage('Could not compact database. '+E.Message);
  end; // try
end;
```

Figure 9: Compacting an Access database.

```
class procedure TAccessUtils.RepairAccessDb(
  DBName: string);
var
  DBEngine: Variant;
begin
  try
    Screen.Cursor := crHourGlass;
    try
      { Get a handle to the Database Engine. }
      DBEngine := CreateOleObject('DAO.DBEngine.35');
      { Repair the database. }
      DBEngine.RepairDatabase(DBName);
    finally
      Screen.Cursor := crDefault;
    end; // try
  except on E: EOleException do
    ShowMessage('Could not repair database. ' + E.Message);
  end; // try
end;
```

Figure 10: Repairing an Access database.

Working with an Access Database

There is no direct way to create an Access database in Delphi; however, it can be done by using Automation to work directly with DAO. The procedure in Figure 8 is implemented as a class method of a component that also includes methods to compact and repair Access databases (see Figures 9 and 10). The procedure starts by calling *CreateOleObject* to create an instance of the DAO 3.5 database engine. Next, it gets a handle to the current *Workspace* object from the *Workspaces* collection. Finally, a call is made to the *Workspace* object's *CreateDatabase* method to create the new .MDB file.

Figure 9 shows the class method *CompactAccessDb*, which compacts the database to defragment it and reduce wasted space. This code is similar to the previous example except that the call to the database engine's *CompactDatabase* method creates a new database. After the compacted database is created, the original is deleted and the new database is renamed to the original name.

Like all desktop databases, Access databases can become corrupt if the PC writing to the database crashes. The Jet engine includes a method that will attempt to fix a damaged database file. Figure 10 shows how to call the database engine's *RepairDatabase* method. All these methods and the *TAccessUtils* object are included in the AccUtil project that accompanies this article (see end of article for download details).

Conclusion

Although the tools available in Delphi 4 for working with Access databases have some limitations, you can write applications that take advantage of all the features that Access databases have to offer. With the Table and Query components to provide data access, and the Database component for transaction control, you can work with Access databases using the native BDE driver or ODBC. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JUL\DI9907BT.

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

Begin Listing One — *TdgReadTableBeforeWrite*

```
unit RBeforeW;

interface

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

type
  EdgRBWException = class(Exception);
  EdgRBWTableMissing = class(EdgRBWException);
  EdgRBWShadowTableMissing = class(EdgRBWException);
  TdgReadTableBeforeWrite = class(TComponent)
  private
    FTable:       TTable;
    FShadowTable: TTable;
    FRec:         Variant;
    FRecordSaved: Boolean;
    procedure CheckValidTables;
  protected
    procedure CreateShadowTable;
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
    procedure SetTable(Value: TTable);
  public
    constructor Create(AOwner: TComponent); override;
    procedure Close;
    procedure Open;
    function  RecordWasModified: Boolean;
    procedure SaveRecord;
  published
    property Table: TTable read FTable write SetTable;
  end;

procedure Register;

implementation

constructor TdgReadTableBeforeWrite.Create(
  AOwner: TComponent);
begin
  inherited;
  CreateShadowTable;
end;
```

```
procedure TdgReadTableBeforeWrite.Open;
{ Opens the shadow table. }
begin
  if FShadowTable <> nil then
    if not FShadowTable.Active then FShadowTable.Open;
end;

procedure TdgReadTableBeforeWrite.Close;
{ Closes the shadow table. }
begin
  if FShadowTable <> nil then
    if FShadowTable.Active then FShadowTable.Close;
end;

procedure TdgReadTableBeforeWrite.SetTable(Value: TTable);
begin
  FTable := Value;
  CreateShadowTable;
end;

procedure TdgReadTableBeforeWrite.CreateShadowTable;
{ Called from the constructor and the SetTable method to
  create the shadow TTable. The shadow table is not created
  at design time and is not created unless the Table
  property is assigned. }
begin
  { Do not create the shadow table at design time. }
  if csDesigning in ComponentState then Exit;
  { If the Table property is not assigned, do not create
    the shadow table. }
  if FTable <> nil then begin
    { If the shadow table component does not exist then
      create it. }
    if FShadowTable = nil then
      FShadowTable := TTable.Create(Self);
    { Connect the shadow table to the same table as the
      Table property. }
    with FShadowTable do begin
      if Active then Close;
      DatabaseName := FTable.DatabaseName;
      ReadOnly := True;
      TableName := FTable.TableName;
    end; // with
  end; // if
end;

procedure TdgReadTableBeforeWrite.Notification(
  AComponent: TComponent; Operation: TOperation);
begin
  if Operation = opRemove then
    if AComponent = Self.Table then
      Self.Table := nil;
end;

procedure TdgReadTableBeforeWrite.CheckValidTables;
begin
  if FTable = nil then
    raise EdgRBWTableMissing.Create(
      'TdgReadBeforeWrite Table is not assigned.');
  if FShadowTable = nil then
    raise EdgRBWShadowTableMissing.Create(
      'TdgReadBeforeWrite ShadowTable is not assigned.');
end;

procedure TdgReadTableBeforeWrite.SaveRecord;
{ This method and RecordWasModified are used to determine
  if another user has changed a record in a table since you
  started editing it. This method is called in the table's
  BeforeEdit event handler to save the record to a variant
  array. RecordWasChanged is called from the table's
  BeforePost event handler. It rereads the record and
  compares the value of each field to the value saved in
  the variant array. If the values are not identical a
  warning message is displayed. }
var
  I: Integer;
begin
  { If either table is not assigned, raise an exception. }
  CheckValidTables;
  { Open the shadow table. }
  if not FShadowTable.Active then FShadowTable.Open;
  { Position the TTable you will use to see if the record
    has been changed by another user to the record you
    are about to edit. }
  FShadowTable.GotoCurrent(FTable);
  { Size the variant array to hold all of the fields
    in the record. }
  if (not VarIsArray(FRec)) or
     (VarArrayHighBound(FRec, 1) <>
     Pred(FShadowTable.FieldCount)) then
    FRec := VarArrayCreate([0,
              Pred(FShadowTable.FieldCount)], varVariant);
  { Save the field values. }
  for I := 0 to FShadowTable.FieldCount - 1 do
    FRec[I] := FShadowTable.Fields[I].Value;
  { Flag the saved record as valid. }
  FRecordSaved := True;
end;

function TdgReadTableBeforeWrite.RecordWasModified:
  Boolean;
{ See the comments for SaveRecord for detailed information
  about using this method. This function compares the field
  values saved when you started editing the record with the
  current values of the table's fields to determine if
  another user has changed the record. If the record has
  been changed the user is warned.

  Returns:
   False if the record has not been changed, or if the user
   chooses to overwrite the changes, and True if the record
   has been changed and the user chooses not to overwrite
   the changes. }
var
  I: Integer;
begin
  Result := False;
  { If this is not a modified record, exit. }
  if not FRecordSaved then Exit;
  { If either table is not assigned, raise an exception. }
  CheckValidTables;
  { If there is no saved record, exit and return False. }
  if VarIsEmpty(FRec) then Exit;
  { Refresh the shadow table so you will see changes made
    by other users. }
  FShadowTable.Refresh;
  { Compare the records. }
  for I := VarArrayLowBound(FRec, 1) to
           VarArrayHighBound(FRec, 1) do begin
    if FRec[I] <> FShadowTable.Fields[I].Value then begin
      Result := True;
      Break;
    end; // if
  end; // for
  { If the record has been changed, notify the user. }
  if Result = True then begin
    if MessageDlg('Another user has changed this record. '+
         'Do you want to save the other user''s changes? '+
         'If you choose Yes you must cancel your changes.',
         mtWarning, [mbYes, mbNo], 0) <> mrYes then
      Result := False;
  end; // if
  { Flag the saved record as invalid. }
  FRecordSaved := False;
end;

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

End Listing One

By Jani Järvinen

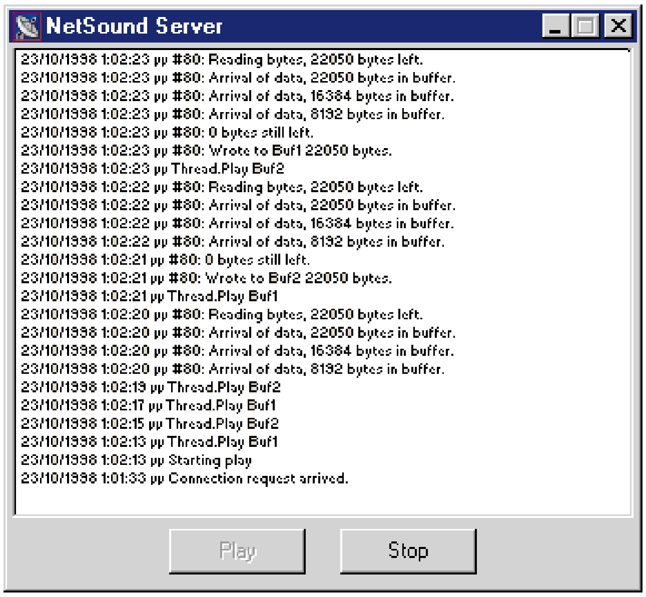
NetSound

Streaming Audio over a TCP/IP Network

Ever used RealAudio? Streaming audio, or even video over a network, are now common, but you might still wonder how they work. This article describes a simple two-program solution for making streaming audio possible using Delphi. You'll also learn how to use the low-level audio functions provided by the Windows API.

By definition, streaming audio means the receiver doesn't have to download the whole audio file before hearing it. Consider a modem connection and a 300KB .WAV file. Without streaming support, the user must first download the whole file. With streaming support, the user might only need to download, say, 50KB, after which the audio clip starts to play.

The program NetSound presented in this article doesn't do any buffering as in the previous .WAV file example. Instead, it assumes the physical transmission channel can transfer a minimum of 11KB of data per second, continuously. Although this is a rather high requirement, audio compression used in commercial products is beyond the scope of this article. Because of the transfer-rate requirement, NetSound simply doesn't work with a modem connection, or even a dual-channel ISDN connection. You should only experiment with it in a local intranet.



```
23/10/1998 1:02:23 pp #80: Reading bytes, 22050 bytes left.
23/10/1998 1:02:23 pp #80: Arrival of data, 22050 bytes in buffer.
23/10/1998 1:02:23 pp #80: Arrival of data, 16384 bytes in buffer.
23/10/1998 1:02:23 pp #80: Arrival of data, 8192 bytes in buffer.
23/10/1998 1:02:23 pp #80: 0 bytes still left.
23/10/1998 1:02:23 pp #80: Wrote to Buf1 22050 bytes.
23/10/1998 1:02:23 pp Thread.Play Buf2
23/10/1998 1:02:22 pp #80: Reading bytes, 22050 bytes left.
23/10/1998 1:02:22 pp #80: Arrival of data, 22050 bytes in buffer.
23/10/1998 1:02:22 pp #80: Arrival of data, 16384 bytes in buffer.
23/10/1998 1:02:22 pp #80: Arrival of data, 8192 bytes in buffer.
23/10/1998 1:02:21 pp #80: 0 bytes still left.
23/10/1998 1:02:21 pp #80: Wrote to Buf2 22050 bytes.
23/10/1998 1:02:21 pp Thread.Play Buf1
23/10/1998 1:02:20 pp #80: Reading bytes, 22050 bytes left.
23/10/1998 1:02:20 pp #80: Arrival of data, 22050 bytes in buffer.
23/10/1998 1:02:20 pp #80: Arrival of data, 16384 bytes in buffer.
23/10/1998 1:02:20 pp #80: Arrival of data, 8192 bytes in buffer.
23/10/1998 1:02:19 pp Thread.Play Buf2
23/10/1998 1:02:17 pp Thread.Play Buf1
23/10/1998 1:02:15 pp Thread.Play Buf2
23/10/1998 1:02:13 pp Thread.Play Buf1
23/10/1998 1:02:13 pp Starting play
23/10/1998 1:01:33 pp Connection request arrived.
```

Figure 1: The Server program.

The Server

NetSound consists of two programs: Client and Server (both are available for download; see end of article for details). The Server plays the audio, and Client supplies the audio data to Server. Although the naming convention might sound strange, they could just as well be called "Sender" and "Receiver."

The workings of Server look simple (see Figure 1). It has to listen only to an incoming TCP (Transmission Control Protocol) connection and, when a connection arrives, accept it. Then it starts to read the data that comes in through the connection, simply feeding it to the audio hardware on the system (remember, no compression is used).

In reality, things aren't as simple as they seem. The Server consists of three distinct parts: the TCP, audio, and multi-threading code. For TCP, the NetManage ActiveX controls are used. These controls ship with Delphi 3 and 4. (They must be installed for Delphi 4. Select **Component | Install Component** to access the Install Component dialog box. Browse to select C:\Program Files\Borland\Delphi4\Ocx\Isp\isp3.pas as the **Unit file name**. The **Package file name** must, of course, be set to C:\Program Files\Borland\Delphi4\Lib\dclusr40.dpk.) Audio code uses the MultiMedia System (MMSystem) API, and the multi-threading code uses Delphi's *TThread* class found in the Classes unit.

Connecting Using TCP

Although the NetManage ActiveX controls have been criticized, the code for Server and Client demonstrates how to use them. Using the *TTCP* control found on the Internet page of the Component palette is easy. The code simply calls the *Listen* method and waits for the *ConnectionRequest* event. In the event handler, a new instance of the *TTCP* class is created; it will handle the newly opened connection. Also, note how an event handler for the *DataArrival* event is set.

Like it or not, the NetManage *TTCP* component requires several tricks to operate correctly. For example, calling the *Listen* method should put the component in the listening state for new connections. However, if the *State* property isn't queried before doing this, connections are simply refused.

The *DataArrival* event is the most important — and the most difficult — event to handle. It's shown in Listing One (beginning on page 13). This event handler reads the data from the TCP connection into an OleVariant, then stores the raw binary data in either of the two output buffers. The threaded *TPlayerThread* class then plays these buffers one at a time, using the default audio output device set up using Windows Control Panel.

The *TPlayerThread* Class

As the class name tells, *TPlayerThread* is a class that efficiently creates a new thread to Server. Though it's not mandatory from a technical standpoint, adding a new thread to Server is logical. The main thread handles the user interface and the TCP communication, and the player thread handles the audio playing.

Upon initialization, the constructor prepares headers and opens a handle to the default audio output device. NetSound is "hard-coded" to use monaural (single channel) eight-bit audio with a sample rate of 11,025 kHz resulting in almost an 11KB data transfer requirement per second. Although the quality of audio with these settings is by no means "high fidelity," it helps keep the transmission rate low. In contrast, sending CD-quality (16-bit stereo, 44,100 kHz) audio would require a throughput of 172KB per second.

A descendant of the *TThread* class does the actual work in the overridden *Execute* method, shown in Listing Two (on page 14). The Player simply makes sure that one audio buffer is always playing, and the other buffer is in the audio driver's queue, waiting to be played.

This two-buffer solution calls for more description. When the buffer has no more audio data to play, it has to be restarted from the beginning if only a single buffer is used. This will only take a few milliseconds, but you will still hear a small "glitch." The solution to this problem is to use two buffers. One buffer is always playing, and the audio driver can immediately proceed to the next buffer as soon as the other one has finished playing. The driver is optimized to do this so quickly that distorting sounds can't be heard.

How does the thread know when it's time to change the buffer? The multimedia functions call our callback procedure, which sets a variable named "Playing" equal to False. The thread then waits in a **repeat** loop until the value of the variable becomes False and then proceeds. In NetSound, the default buffer length is two seconds, but this is easy to change. Note that "synchronization" situations would be best handled using Win32 events; a simple variable will do just fine here.

Accessing Low-level Wave Output Devices

In Windows 95, 98, or NT, multimedia-related services can be found from the WINMM.DLL library. In Delphi, the RTL unit named MMSystem is the key. In the NetSound project, all functions used from this unit happen to be prefixed with the word

```
constructor TPlayerThread.Create;
begin
  BufToUse := 1;
  FillChar(Buf1, SizeOf(Buf1), 0);
  FillChar(Buf2, SizeOf(Buf2), 0);
  with WaveFormat do begin
    wFormatTag := Wave_Format_PCM;
    nChannels := 1;
    nSamplesPerSec := SampleRate;
    nBlockAlign := 1;
    nAvgBytesPerSec := nSamplesPerSec*nBlockAlign;
    wBitsPerSample := 8;
    cbSize := 0;
  end;
  WaveOutOpen(@WaveOut, Wave_Mapper, @WaveFormat,
    Integer(@MyWaveProc), hInstance, Callback_Function);
  FillChar(WaveHeader1, SizeOf(TWaveHdr), 0);
  FillChar(WaveHeader2, SizeOf(TWaveHdr), 0);
  with WaveHeader1 do begin
    lpData := Buf1;
    dwBufferLength := PlayBufferSize;
    dwLoops := 1;
  end;
  with WaveHeader2 do begin
    lpData := Buf2;
    dwBufferLength := PlayBufferSize;
    dwLoops := 1;
  end;
  FreeOnTerminate := True;
  Inherited Create(False);
end;
```

Figure 2: Getting ready for the *WaveOutWrite* call.

"wave." For example, to open an output device, the *WaveOutOpen* function would be called; to open an input device, *WaveInOpen* would be called. This is a logical naming convention.

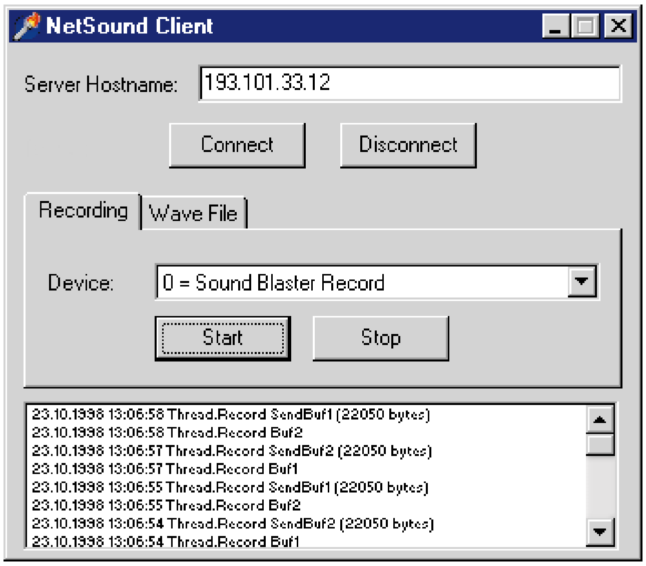
To play audio, the first thing is to open a handle to the output device we want (just as a file must be opened before it can be accessed). For output, it's really not important which device is the best (the user's computer might have multiple sound devices installed). Instead, NetSound lets Windows choose the device it prefers. The only thing needed, then, is the support for 11,025 kHz mono sound.

Playing a buffer of audio is done using the *WaveOutWrite* function. The format of the buffer used in NetSound is plain and simple PCM (Pulse Code Modulated), which is raw, uncompressed data. Of course, NetSound could use A-law compression. However, that would require additional support from the audio hardware.

Before getting a buffer playing with *WaveOutWrite*, a few things need to be done (see Figure 2). First, a header structure needs to be set up. This is easy: The header describes the location of the buffer storing the actual audio data, as well as how many times it should be looped, among other things. After the header has been set up, the audio driver is instructed to stand by for data. This is done with the *WaveOutPrepareHeader* function (see the *Execute* method in Listing Two).

After header preparation, the buffer is ready for playing. After it has finished playing, the header must be unprepared. You guessed it; the function name is *WaveOutUnprepareHeader*. It will free the resources associated with the header.

With the header preparation and un-preparation requirements, audio playing is actually a repetition of the three calls: prepare,



Server Hostname: 193.101.33.12

Device: 0 = Sound Blaster Record

```
23.10.1998 13:06:58 Thread.Record SendBuf1 (22050 bytes)
23.10.1998 13:06:58 Thread.Record Buf2
23.10.1998 13:06:57 Thread.Record SendBuf2 (22050 bytes)
23.10.1998 13:06:57 Thread.Record Buf1
23.10.1998 13:06:55 Thread.Record SendBuf1 (22050 bytes)
23.10.1998 13:06:55 Thread.Record Buf2
23.10.1998 13:06:54 Thread.Record SendBuf2 (22050 bytes)
23.10.1998 13:06:54 Thread.Record Buf1
```

Figure 3: The Client program.

write, and unprepare. Only three other functions are used for output — one for opening the device, one for resetting the device after play, and one for closing the device. The routines are *WaveOutOpen*, *WaveOutReset*, and *WaveOutClose*, respectively.

The Client

The other program in the NetSound project is Client (see Figure 3). This program can either send a .WAV file through the network to Server, or it can record data from any MMSystem-compatible audio recorder, and send that data to Server. Either way, there is no difference as long as Server is considered. However, from Client's point of view, things are very different.

The easiest thing for Client is, of course, to send a .WAV file from disk to Server. However, before sending anything, Client needs a TCP connection to Server. This is again carried out using the NetManage *TTCP* component, just like in Server. There is a difference between using the *TTCP* component in Server and Client, however. In Client's case, the methods used are *Connect*, *SendData*, and *Close* (to disconnect).

When the user chooses to send a file, he or she clicks the **Browse** button to browse for a .WAV file (using the *TOpenDialog* component). Then, the user clicks the **Play** button, and several things will happen (see Figure 4).

First, the file selected will be opened as an untyped file. Next, a variant array of bytes is allocated to match the size of the file. Because the *TTCP* component works with variants, these are used even though they are slow compared to normal *Object Pascal* arrays.

Variant arrays are created with the *VarArrayCreate* call. As the code uses the *BlockRead* procedure to read the .WAV file data in one (big) chunk, the memory associated with the variant needs to be locked. This is done by calling the *VarArrayLock* function. After the *BlockRead* call, the variant array is unlocked using the *VarArrayUnlock* call. After this, the whole file is in the variant array and can be sent through the TCP connection using only a single *SendData* method call.

Note that it's the user's responsibility to select a valid .WAV file. Actually, any file, such as a Word document or an executable, can be sent, but the resulting sound is — well — experimental. The format of the .WAV file is also important. Selecting a stereo, a 16-bit, or a compressed file, or a file with a sample rate different than 11,025 kHz, will result in garbled sound. Also, all .WAV files have headers. This will make the very beginning of the audio sound somewhat distorted, but it's beyond the scope of this article to examine the .WAV file format.

```
procedure TNetSoundClientForm.PlayFileClick(
  Sender: TObject);
var
  F : File;
  I : Integer;
  V : Variant;
  P : Pointer;
begin
  AssignFile(F, FileName.Text);
  Reset(F, 1);
  I := FileSize(F);
  ProgressBar.Max := I;
  ProgressBar.Position := 0;
  V := varArrayCreate([0, I-1], varByte);
  P := varArrayLock(V);
  BlockRead(F, P^, I);
  varArrayUnlock(V);
  TCPConnection.SendData(V);
  CloseFile(F);
end;
```

Figure 4: The Client code for sending a .WAV file through the network.

```
constructor TRecorderThread.Create(DeviceID: Integer);
begin
  BufToUse := 1;
  FillChar(Buf1, SizeOf(Buf1), 0);
  FillChar(Buf2, SizeOf(Buf2), 0);
  with WaveFormat do begin
    wFormatTag := Wave_Format_PCM;
    nChannels := 1;
    nSamplesPerSec := SampleRate;
    nBlockAlign := 1;
    nAvgBytesPerSec := nSamplesPerSec*nBlockAlign;
    wBitsPerSample := 8;
    cbSize := 0;
  end;
  WaveInOpen(@WaveIn, DeviceID, @WaveFormat,
    Integer(@MyWaveProc), hInstance, Callback_Function);
  FillChar(WaveHeader1, SizeOf(TWaveHdr), 0);
  FillChar(WaveHeader2, SizeOf(TWaveHdr), 0);
  with WaveHeader1 do begin
    lpData := Buf1;
    dwBufferLength := PlayBufferSize;
    dwLoops := 1;
  end;
  with WaveHeader2 do begin
    lpData := Buf2;
    dwBufferLength := PlayBufferSize;
    dwLoops := 1;
  end;
  FreeOnTerminate := True;
  Inherited Create(False);
end;
```

Figure 5: Initializing the *TRecorderThread*.

Recording Audio

The most interesting feature in Client is the ability to record waveform audio. The big picture is the same: One thread handles the UI, and the other thread handles the recording. This time, the recording thread is handled by the *TRecorderThread* class. For recording, a repetition of three function calls is needed, just as playing audio in the *TPlayerThread*.

Compared to the player, many things are different. For instance, the user selects the input device. In the UI, a combobox is filled with a list of device names suitable for recording. Such a list is gen-

```
procedure TRecorderThread.SendBuffer;
var
  V : OLEVariant;
  P : Pointer;
begin
  with NetSoundClientForm do begin
    V := varArrayCreate([0, SizeOf(TAudioBuffer)-1],
                        varByte);
    P := varArrayLock(V);
    Move(SyncBuf^, P^, SizeOf(TAudioBuffer));
    varArrayUnlock(V);
    TCPConnection.SendData(V);
    Log(nil, 'Thread.Record SendBuf' +
        IntToStr(Integer(SyncBufNum)) + ' (' +
        IntToStr(SizeOf(TAudioBuffer)) + ' bytes)');
    Application.ProcessMessages;
  end;
end;
```

Figure 6: The *SendBuffer* method.

erated by calling the *WaveInGetDevCaps* function. It expects a device identifier as a parameter and returns a structure full of information about the device in question. The code only checks to see if the device can record eight-bit mono sound with the sample rate used everywhere in NetSound.

When the user selects a device from the list and clicks the **Record** button, an instance of the *TRecorderThread* is created, and, in effect, the recorder thread starts to run. Initialization does roughly the same things as the *TPlayerThread* initialization, i.e. it opens a handle to an input device, and prepares two headers for use by the *Execute* method (see Figure 5).

The *Execute* method starts the recording operation by calling the *WaveInStart* function, shown in Listing Three (on page 14). This makes recording audio possible. As no input buffers have been set up, the audio is simply discarded (recorded to "void"). To add a buffer, the *WaveInAddBuffer* function is called, surrounded by the *WaveInPrepareHeader* and *WaveInUnprepareHeader* calls. Given this, the *WaveInAddBuffer* can be considered equal to the *WaveOutWrite* call. After the user chooses to terminate the recording, the *WaveInStop* function gets called. This will stop the recording. After this, the device handle is freed.

Although these calls are sufficient to record audio, Client still needs to send the recorded data to Server. This is done by calling the *TTCP* component's *SendData* method in a synchronized procedure named *SendBuffer* (see Figure 6). Data is simply sent whenever a buffer is full of audio data, but before the buffer gets re-used.

Note that, because synchronization is used, the *SendBuffer* method is always called in the context of the UI (main) thread, not the recorder thread. Synchronization needs only to be used because, otherwise, the *TTCP* component throws an exception.

A Test Drive

To test the two NetSound applications, you need two computers, connected using a LAN, with TCP/IP as one of the transport protocols. Also, both computers must have a sound card. Note that there aren't many error checks to make sure the *WaveXXXOpen* calls succeed. And, because the NetSound applications aren't end-user programs, but programmers' toys, things need to be done in a certain order. So don't try to record audio without a connection.

To have your personal test drive, connect a microphone to the computer running Client. Place your radio receiver's speaker next to the microphone, and tune in to your favorite station. Make sure the volume is high enough on the radio and the microphone.

On the computer running Server, make sure the speakers are set to medium volume. Fire up Server, and click **Play**. Now return to Client, type in the host name (or IP address) of the Server computer, and click **Connect**. After you see the "Connected" message, click the **Start** button. Recording now starts.

In a few seconds, you should hear your radio station on your Server. Note that the audio is always about four to five seconds late. This is a result of buffer misalignments between the two computers, and the fact that the audio buffer is always recorded well before it's sent.

Conclusion

By now, you should be able to understand at least something about how NetSound code works. Of course, some simple things are left out, but, by looking at the code and reading the comments on the actual source code files, you should be able to work things out.

As for improvements, there are many things you could do better. For example, you could combine these two programs, and give two people the ability to communicate with each other (full-duplex operation). Or, you could add support for multiple users and mix the audio channels on the server side. The possibilities are almost endless, so please use your imaginations! Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JUL\DI9907JJ.

Jani Järvinen does contract programming using Delphi, concentrating on Windows NT and Internet solutions.

Begin Listing One — *TNetSoundServerForm.TCPDataArrival*

```
procedure TNetSoundServerForm.TCPDataArrival(
  Sender: TObject; bytesTotal: Integer);
var
  Dummy   : OleVariant;
  I, J, K : Integer;
begin
  Log(Sender, 'Arrival of data, ' + IntToStr(
    (Sender as TTCP).BytesReceived) + ' bytes in buffer.');
  if (BytesTotal < PlayBufferSize) then
    (Sender as TTCP).GetData(Dummy, varArray + varByte, 0)
  else begin
    if not Receiving then begin
      Receiving := True;
      try
        I := BytesTotal;
        J := BufToUse;
        repeat
          Log(Sender,'Reading bytes, ' + IntToStr(I) +
            ' bytes left.');
          Dummy := Unassigned;
          (Sender as TTCP).GetData(
            Dummy, varByte+varArray, PlayBufferSize);
          repeat
            Application.ProcessMessages;
          until ((BufToUse<>J) or PlayerThread.Terminated);
          if PlayerThread.Terminated then
            Exit;
          with TvarArray(
                TvarData(Dummy).VUnknown^) do begin
              K := Bounds[0].ElementCount;
              if (BufToUse = 1) then
                Move(Data^,Buf1,K)
              else
                Move(Data^,Buf2,K);
            end;
            Log(Sender,'Wrote to Buf' + IntToStr(BufToUse) +
              ' ' + IntToStr(K) + ' bytes.');
            J := BufToUse;
            Dec(I,PlayBufferSize);
            if (I < PlayBufferSize) then begin
              { Check if new data arrived while we were
                processing old data. }
              I := (Sender as TTCP).BytesReceived;
              if (I > PlayBufferSize) then
                Log(Sender,
                  'New data arrived while processing, now ' +
                  IntToStr(I) + ' bytes left.');
            end;
          until (I < PlayBufferSize);
        finally
          Receiving := False
        end;
        Log(Sender,IntToStr((Sender as TTCP).BytesReceived) +
          ' bytes still left.');
      end;
    end;
end;
```

End Listing One

Begin Listing Two — *TPlayerThread.Execute*

```
procedure TPlayerThread.Execute;
begin
  WaveOutPrepareHeader(WaveOut, @WaveHeader1,
                       SizeOf(TWaveHdr));
  WaveOutWrite(WaveOut, @WaveHeader1, SizeOf(TWaveHdr));
  NetSoundServerForm.Log(nil, 'Thread.Play Buf1');
  Playing := True;
  repeat
    with NetSoundServerForm do begin
      if (BufToUse = 1) then begin
        FillChar(Buf2, SizeOf(Buf2), 0);
        WaveOutPrepareHeader(WaveOut, @WaveHeader2,
                             SizeOf(TWaveHdr));
        WaveOutWrite(WaveOut, @WaveHeader2,
                     SizeOf(TWaveHdr));
        Playing := True;
        BufToUse := 2;
      end
      else begin
        FillChar(Buf1, SizeOf(Buf1), 0);
        WaveOutPrepareHeader(WaveOut, @WaveHeader1,
                             SizeOf(TWaveHdr));
        WaveOutWrite(WaveOut, @WaveHeader1,
                     SizeOf(TWaveHdr));
        Playing := True;
        BufToUse := 1;
      end;
      while Playing do { nothing. } ;
      Log(nil, 'Thread.Play Buf' + IntToStr(BufToUse));
      if (BufToUse = 1) then begin
        WaveOutUnprepareHeader(WaveOut, @WaveHeader2,
                               SizeOf(TWaveHdr));
        FillChar(Buf2, SizeOf(Buf2), 0);
        WaveOutPrepareHeader(WaveOut, @WaveHeader2,
                             SizeOf(TWaveHdr));
        WaveOutWrite(WaveOut, @WaveHeader2,
                     SizeOf(TWaveHdr));
      end
      else begin
        WaveOutUnprepareHeader(WaveOut, @WaveHeader1,
                               SizeOf(TWaveHdr));
        FillChar(Buf1, SizeOf(Buf1), 0);
        WaveOutPrepareHeader(WaveOut, @WaveHeader1,
                             SizeOf(TWaveHdr));
        WaveOutWrite(WaveOut, @WaveHeader1,
                     SizeOf(TWaveHdr));
      end;
    end;
  until Terminated;
  WaveOutReset(WaveOut);
  NetSoundServerForm.Log(nil, 'Thread.Terminated');
end;
```

End Listing Two

Begin Listing Three — *TPlayerThread.Execute*

```
procedure TRecorderThread.Execute;
begin
  NetSoundClientForm.Log(nil, 'Thread.Record Buf1');
  WaveInStart(WaveIn);
  WaveInPrepareHeader(WaveIn, @WaveHeader1,
                      SizeOf(TWaveHdr));
  waveInAddBuffer(WaveIn, @WaveHeader1, SizeOf(TWaveHdr));
  Recording := True;
  repeat
    with NetSoundClientForm do begin
      if (BufToUse = 1) then begin
        FillChar(Buf2, SizeOf(Buf2), 0);
        WaveInPrepareHeader(WaveIn, @WaveHeader2,
                            SizeOf(TWaveHdr));
        WaveInAddBuffer(WaveIn, @WaveHeader2,
                        SizeOf(TWaveHdr));
        Recording := True;
        BufToUse := 2;
      end
      else begin
        FillChar(Buf1, SizeOf(Buf1), 0);
        WaveInPrepareHeader(WaveIn, @WaveHeader1,
                            SizeOf(TWaveHdr));
        WaveInAddBuffer(WaveIn, @WaveHeader1,
                        SizeOf(TWaveHdr));
        Recording := True;
        BufToUse := 1;
      end;
      while Recording do { nothing. };
      Log(nil, 'Thread.Record Buf' + IntToStr(BufToUse));
      if (BufToUse = 1) then begin
        WaveInUnprepareHeader(WaveIn, @WaveHeader2,
                              SizeOf(TWaveHdr));
        SyncBuf := @Buf2;
        SyncBufNum := 2;
        Synchronize(SendBuffer);
        FillChar(Buf2, SizeOf(Buf2), 0);
        WaveInPrepareHeader(WaveIn, @WaveHeader2,
                            SizeOf(TWaveHdr));
        WaveInAddBuffer(WaveIn, @WaveHeader2,
                        SizeOf(TWaveHdr));
      end
      else begin { Buf1 is now full of data. }
        WaveInUnprepareHeader(WaveIn, @WaveHeader1,
                              SizeOf(TWaveHdr));
        SyncBuf := @Buf1;
        SyncBufNum := 1;
        Synchronize(SendBuffer);
        FillChar(Buf1, SizeOf(Buf1), 0);
        WaveInPrepareHeader(WaveIn, @WaveHeader1,
                            SizeOf(TWaveHdr));
        WaveInAddBuffer(WaveIn, @WaveHeader1,
                        SizeOf(TWaveHdr));
      end;
    end;
  until Terminated;
  WaveInStop(WaveIn);
  WaveInReset(WaveIn);
  NetSoundClientForm.Log(nil, 'Thread.Terminated');
end;
```

End Listing Three

By Steve Griffiths

When Every Bit Counts

Compact Storage Using Integer Bitfields

In many situations, a user may be required to select between 0 and *n* items from a list, e.g. languages spoken, past illnesses, etc. A ListBox, CheckListBox, or collection of checkboxes can easily be used to display the selection to the user, but storing the selected items in a database table requires consideration.

It would be simple to provide a Boolean field for each item in the list, but this approach leads to wide tables, and wouldn't allow an end user to add an item to the list. Alternatively, a child table could be used to write a record for each item that was selected. This would work and would allow an extendable list, but it would require coding to update the list each time the master record changes and to save changes to the child table.

A far more compact method is to store a numeric reference to each selected item in a single integer field. This article explains the use of a technique called *bit shifting* to achieve this, and also details a data-aware CheckListBox that implements the bit-shift mechanism.

Bits Broken Down

A four-byte integer contains 32 bits, each of which can be individually addressed and used as a switch. This means that with a little trickery, a single integer can be used to determine the selection status of up to 31 items. (The reason 31 is the maximum — and not 32 — is that most databases use a signed integer type, and use the most significant bit — bit 31 — to store the sign.) All we need to do is convert each selected item in the list to an equivalent bit number in the integer and turn it on.

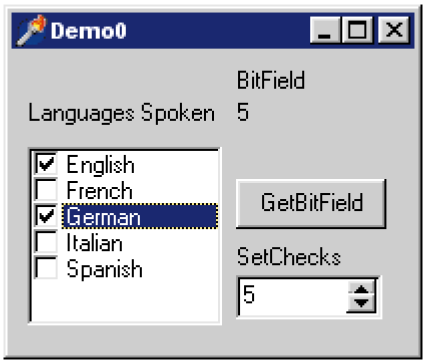


Figure 1: A demonstration form with a CheckListBox.

Back to Binary

The binary system uses a base of 2, with each bit representing either 0 or 2 to the power of the bit position (zero-based). Binary values are read right-to-left; for example:

binary 110101
= 2^0 + 0 + 2^2 + 0 + 2^4 + 2^5
= 1 + 4 + 16 + 32
= 53

List Indexing

Each item in a StringList (e.g. *ListBox1.Items*) has an index value between 0 and the number of items in the list minus one. To store a reference to each selected item in single integer, we need to convert the index value of the item to the number that sets the equivalent bit in the integer; selecting items 2 and 5 in the list should set bits 2 and 5 in the integer. To do this, we use bit shifting.

One of the Boolean operators provided by Delphi is **shl**. This stands for "Shift Left," and, when applied, moves each bit of an integer one step to the left. For example, 53 **shl** 1 = 106. In binary representation, 0110101 **shl** 1 = 1101010. As you can see, each bit has been moved to the left. To convert an ordinal value to

```
// If Item is selected, set corresponding Result bit.
function TForm1.GetBitField: Integer;
var
  i : Integer;
begin
  Result := 0;  // Initialize Result.
  with CheckListBox1 do
    for i := 0 to Items.Count - 1 do
      if Checked[i] then
        Result := Result or (1 shl i);
end;
```

Figure 2: Obtaining an integer bitfield from a list.

a bit equivalent, start with 1, and shift it left by the ordinal value, e.g. 1 **shl** 4 = 16. In binary representation, 00001 **shl** 4 = 10000.

Truth in Logic

Since we want to represent more than one item in the integer, we cannot simply assign the converted index value; this would erase any previous entry. For our purposes, Delphi provides the **or** Boolean operator. When two numbers are **or**-ed together, the resulting number will consist of any bit that is set in the first or second number. For example, 1 **or** 4 = 5. In binary representation:

	0001
or	0100
=	0101

This is not the same as addition. For example, 2 **or** 2 = 2. In binary:

	0010
or	0010
=	0010

By use of the **or** operator, we are able to iterate the items in a list, convert the index value of each selected item, and set the equivalent bit of an integer.

An Example

Figure 1 shows a form with a CheckListBox containing five languages. The index values are from 0 to 4. Items 0 and 2 have been selected. In addition to the *Items* property, the CheckListBox contains a *Checked* property, which maintains the checked status of each item. To assess the checked status of an item, use the item's index value. For example, *Checked[0]* is True; *Checked[1]* is False.

The *GetBitField* function (see Figure 2) will iterate the *Checked* list, test each to see if it is checked, and if so, will update the integer field. Once obtained, the result can be stored to any integer field in a table.

Getting It Back

To set the item's *Checked* property based on the contents of a bitfield, we must use a technique known as *bit masking*. Delphi provides the **and** operator, which is useful for this purpose. When two numbers are **and**-ed together, the resulting number will only have a bit set if that bit is set in both numbers. For example, 1 and 2 = 0. In binary representation:

	1001
and	1010
=	1000

```
// For each item, mask its equivalent bit.
// If true (non-zero), set checked true.
procedure TForm1.SetChecks(BitField : Integer);
var
  i : Integer;
begin
  with CheckListBox1 do
    for i := 0 to Items.Count - 1 do
      Checked[i] := LongBool(BitField and (1 shl i));
end;
```

Figure 3: Setting the checked items from a bitfield.

To set the *Checked* property of the items in the list, iterate the list and mask the integer against the bit equivalent of the item's index value. The *SetChecks* function shown in Figure 3 demonstrates how this works. Most of the work here is done by one statement:

```
Checked[i] := LongBool(BitField and (1 shl i));
```

First, 1 is shifted left by the iterator value to give a number that represents the iterator's equivalent bit setting. This value is then **and**-ed against a bitfield, and the result is cast as a LongBool. A result of zero (0) is considered False, and a non-zero result is True. A LongBool is used because a Boolean type only occupies one byte, and will essentially ignore any bits in the remaining three upper bytes, whereas a LongBool occupies four bytes and will evaluate the entire integer. Finally, the *Checked* property for the item being referred to by the iterator is set to the resulting Boolean value.

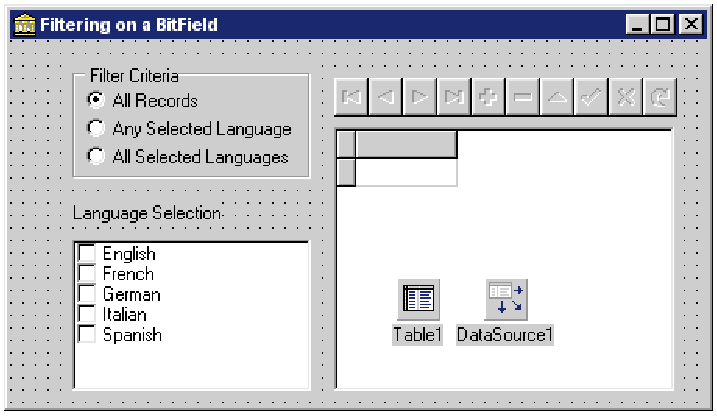
Filtering on a Bitfield

Now that we have a mechanism for representing a selection of items as an integer, it would be nice if we could filter a DataSet based on the value of that number. The same Boolean techniques discussed previously can be used to provide a flexible filtering mechanism.

Figure 4 shows a form containing several components: a DBGrid, a RadioGroup, a CheckListBox, a Table, and a DataSource. The Table contains two fields: one for a name, and the other for an integer representing the languages spoken by the person. The RadioGroup is used to select the filter type, and the CheckListBox is used to select which languages are to be selected by the filter. Notice that the RadioGroup can be used to display matches for any selected language, or only those records where the person speaks all selected languages.

The Code

There are four event handlers used to control the filters. Two of these are used to filter the data, one is used to obtain a bitfield, and the last


Filtering on a BitField

Filter Criteria
(● All Records) ○ Any Selected Language ○ All Selected Languages

Language Selection
☐ English
☐ French
☐ German
☐ Italian
☐ Spanish

Figure 4: The second demonstration form.

```
// Re-evaluate bitfield whenever a language is checked
// or unchecked.
procedure TForm1.CheckListBox1ClickCheck(Sender: TObject);
var
  i : Integer;
begin
  with CheckListBox1 do begin
    Tag := 0;  // Clear Result.
    // Iterate list and set checked bits in Tag property.
    for i := 0 to Items.Count - 1 do
      if Checked[i] then
        Tag := Tag or (1 shl i);
  end;
  // Refresh Filter.
  Table1.Filtered := (RadioGroup1.ItemIndex > 0);
end;
```

Figure 5: The *OnClickCheck* event handler.

is used to dynamically assign one of the two *OnFilterRecord* event handlers to the table. Whenever a language is checked or unchecked, the *OnClickCheck* event handler iterates the items list and updates the *Tag* property with the new bitfield value (see Figure 5). The *Filtered* property of the table is then set to True or False, depending on the selection in the RadioGroup. This will refresh the table and update the grid contents to reflect the new selection.

Changing the filter criteria in the RadioGroup will fire its *OnClick* event handler (see Figure 6). This handler will assign the appropriate event handler for the table's *OnFilterRecord* event, and set the table's *Filtered* property appropriately.

```
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  with RadioGroup1 do
    case ItemIndex of
      // If Filtered is False, the event handler is
      // irrelevant as it will not be called.
      0: Table1.Filtered := False;
      1: begin
        Table1.OnFilterRecord := FilterAny;
        Table1.Filtered := True;
      end;
      2: begin
        Table1.OnFilterRecord := FilterAll;
        Table1.Filtered := True;
      end;
    end;
end;
```

Figure 6: Setting the correct filter event handler.

The *FilterAny* event handler performs a Boolean **and** with the bitfield value contained in the CheckListBox's *Tag* property. If the result is non-zero, at least one of the selected languages is contained in the table's Languages field. When cast to a *LongBool*, this non-zero value will return True, and the record will be accepted by the filter:

```
// Accept records containing any of the selected languages.
procedure TForm1.FilterAny(DataSet: TDataSet;
  var Accept: Boolean);
begin
  Accept := LongBool(CheckListBox1.Tag and
              Table1.FieldbyName('Languages').AsInteger);
end;
```

The *FilterAll* event handler also performs a Boolean **and** operation, but instead of assessing if the result is True or False, the numeric result is compared with the bitfield value. If the two numbers are identical, all the selected languages are contained in the Languages field, and the record is selected:

```
// Only accept records that contain all selected languages.
procedure TForm1.FilterAll(DataSet: TDataSet;
  var Accept: Boolean);
begin
  Accept := (CheckListBox1.Tag and
    Table1.FieldbyName('Languages').AsInteger =
      CheckListBox1.Tag) and (CheckListBox1.Tag > 0);
end;
```

When writing code, it is often useful to use the *Tag* property of components for storage of arbitrary values, as the storage is essentially free. However, in this instance, it created a small problem. As the form closes, the *OnFilterRecord* event was being fired. The two *OnFilterRecord* event handlers refer to the *Tag* property of the CheckListBox. The CheckListBox had already been destroyed, resulting in an Access Violation error. A line in the form's *OnCloseQuery* event handler solves the problem by unassigning any *OnFilterRecord* event handler from the table. An alternative is to use a variable for the bitfield value:

```
procedure TForm1.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  Table1.OnFilterRecord := nil;
end;
```

Do It Once

The routines we've discussed provide a straightforward mechanism for storing and retrieving an integer representing the selected items in a list — specifically a CheckListBox, in this example. If we wish to store this information in a table, there still remains the issue of calling these functions from the correct event handlers — typically, a *DataSource.OnChange* to update the CheckListBox, and a *Table.BeforePost* event to save the changes to the table. There is also the issue of informing the table that the CheckListBox contents have changed. Adding this small piece of code wherever it's necessary to store the checked item status can become tedious.

In true Delphi fashion, the cleaner alternative is to create a data-aware descendant of *TCheckListBox* that incorporates the previous routines. This way, the mechanics of storage and keeping the CheckListBox synchronized with the table can be written once and used code-free.

TSgDbCheckListBox

The *TSgDbCheckListBox* component is a descendant of *TCheckListBox* that adds *DataSource* and *DataField* properties to the standard *TCheckListBox*. It contains the functionality to store and retrieve the selected items status, and, as a nicety, contains a procedure to ensure the selected *DataField* is of *ftInteger* type. (Making a standard component data-aware is described in the Delphi developer's guide and is not discussed in detail here.)

The *DataField* selected to store the bitfield must be of type Integer, and, to track 31 items, it cannot be a Shortint. The *SetDataField* procedure calls the *CheckFieldType* procedure. If the selected field is not the right type, an *EInvalidFieldType* exception is raised. This procedure is also called when the DataSet becomes active (see Figure 7).

Most of the code is fairly standard for making a component data-aware. What we need to do is make sure that when the record changes, the checkmarks reflect the selected items, and that when

```
type
  EInvalidFieldType = class(Exception);

// Ensure selected field is of Type FtInteger; if it is any
// other type, an EInvalidFieldType exception is raised.
procedure TSgDbCheckListBox.CheckFieldType(
  const Value: string);
var
  FieldType : TFieldType;
begin
  if (Value <> '') and
     (FDataLink <> nil) and
     (FDataLink.DataSet <> nil) and
     (FDataLink.DataSet.Active) then
    begin
      FieldType :=
        FDataLink.DataSet.FieldByName(Value).DataType;
      if FieldType <> ftInteger then
        raise EInvalidFieldType.Create(
  'TSgDbCheckListBox.DataField must be of type ftInteger');
    end;
end;
```

Figure 7: Making sure the selected datafield is an integer type.

```
procedure TSgDbCheckListBox.DataChange(Sender: TObject);
begin
  if FDataLink.Field <> nil then
    SetCheckMarks;
end;

procedure TSgDbCheckListBox.SetCheckMarks;
var
  i : Integer;
begin
  FDataLink.OnDataChange := nil;
  for i := 0 to Items.Count - 1 do
    if (FDataLink.Field.Value and
       (1 shl i)) = (1 Shl i) then
      Checked[i] := True
    else
      Checked[i] := False;
  FDataLink.OnDataChange := DataChange;
end;
```

Figure 8: Setting the checkmarks when the data changes.

```
procedure TSgDbCheckListBox.UpdateData(Sender: TObject);
var
 i : Integer;
 Value : Integer;
begin
  Value := 0;
  for i := 0 to Items.Count - 1 do
    if Checked[i] then
      Value := Value or ((1 shl i));
  FDataLink.Field.Value := Value;
end;
```

Figure 9: Creating a bitfield for storage.

the user changes the selection in the CheckListBox, the table is updated. When a record changes in a DataSet, it fires the *DataSource OnDataChange* event, which is relayed to our component via the *TFieldDataLink*. The handler calls the *SetCheckMarks* procedure, which updates the checkmarks as described in Figure 8.

The *UpdateData* event handler is provided for the *TFieldDataLink*'s *OnUpdateData* event. This iterates the items in the CheckListBox to build the bitfield, and assigns the result to the *DataField*'s *Value* property (see Figure 9).

Putting It All Together

Figure 10 shows an extended version of the filter demonstration. A *TSgDbCheckListBox* component has been placed onto the form. Its *DataSource* property is set to *DataSource1*, and its *DataField* is set to the Languages field of the table. The *Items* property is initialized to the same list of languages as the language selection CheckListBox. (In the spirit of reuse, the items were pasted in from the other CheckListBox.)

When the program is run, the filter section works as before, and now the SgDbCheckListBox displays the languages spoken by the selected person. The languages spoken may be edited by checking and unchecking the selections and posting the record. Notice that the table automatically goes into edit mode when a selection is changed.

Extending the Component

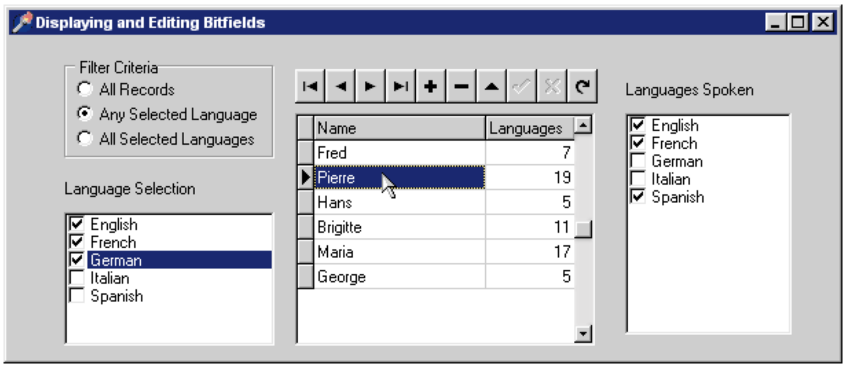
This component uses a 32-bit integer for storage, and provides storage for a total of 31 items. Although this is fine for most visual lists, as in user-interface terms, multiple selection of more than this number of items in a single control can be a little daunting, and it's not easy to display all selected items without having to scroll. However, if more items are required, the component may be extended by adding a second (or third) *DataField*, and breaking the *Items* list into groups of 31. For a truly dynamic storage system, the items should be grouped into a collection of seven-item groups, with the result for each group being cast as a char and concatenated into a string for storage to the table.

Another enhancement is the addition of a Lookup Source and Lookup Field so the list items may be provided from an external table.

Wrapping Up

It's easy to represent the selected status of multi-selection lists as an integer, and the techniques described here can be readily adapted to other list-type classes, both visual and non visual. As always, a little bit of planning can save a great deal of cutting and pasting. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JUL\DI9907SG.


Displaying and Editing Bitfields

Filter Criteria
( ) All Records
(•) Any Selected Language
( ) All Selected Languages

Language Selection
[✓] English
[✓] French
[✓] German
[ ] Italian
[ ] Spanish

Name	Languages
Fred	7
Pierre	19
Hans	5
Brigitte	11
Maria	17
George	5

Languages Spoken
[✓] English
[✓] French
[ ] German
[ ] Italian
[✓] Spanish

Figure 10: An extended version of the filter demonstration.

Steve Griffiths has been programming with x-base languages for over 20 years, switching to Delphi as his primary development tool from version 1. He is currently working for NuMedics Inc., writing medical software with an emphasis on diabetes and disease management. He can be reached via e-mail at sgrif@teleport.com.

By Ron Loewy

XML from Delphi

Introducing XML and Using It from a Delphi Application

If you've read any computer trade magazines in the last year, you've run across stories about XML, and how it will change the Web and the Internet. This article provides a short description of what XML is and isn't, provides some information about possible uses of XML, and presents a primer to XML document parsing from Delphi applications.

XML vs. HTML

Many make the false assumption that XML is a new version of HTML that will replace it as the language of choice on the Web. The truth is that XML is a different animal; it uses a syntax resembling HTML, but has a different goal.

HTML (HyperText Markup Language) started as a markup language for document encoding and display. The popularity of Mosaic, and Netscape Navigator after it, accelerated the development of HTML; it was heavily modified to support display across the Internet infrastructure. Today's HTML is a document display technology that's proven its worth on millions of Web pages.

XML (Extensible Markup Language) is a meta-language — a language used to describe other languages specific to a narrow domain. Microsoft used XML to create the Channel Definition Format (CDF), an XML implementation for the creation of push sites in Internet Explorer. Similarly, the Chemical Markup Language (CML) is an XML implementation specifically for use in documents that describe chemical structures.

```
<?XML version="1.0"?>
<TForm Name="Form1" Caption="Hello XML World"
     Left=50 Width=200 Top=50 Height=400>
  <TLabel Name="Label1" Left=8 Top=12
       Caption=" Hello XML World"/>
  <TPanel name="Panel1" align=alBottom Top=30>
   <TMemo Name="Memo1">
    This data will appear in the memo control.
   </TMemo>
   <TButton Name="OkBtn" Caption="&Ok" Left=100 Width=30/>
   <TButton Name="CancelBtn" Caption="Cancel"
         Left=50 Width=30/>
  </TPanel>
</TForm>
```

Figure 1: One way of using XML to describe a form and its components.

I like to compare HTML and XML using the following description: An HTML document is easy to read (and create) by humans in its native format (before being rendered by a browser), but, because it doesn't impose structure on a document, it's hard for an application to read and analyze. On the other hand, an XML document is a domain-specific description of information that is easy to read (and create) in its native format by humans and applications, but has no display characteristics applied to it.

An XML Document

Before continuing our discussion of what XML is, and what it can be used for, let's examine an XML document. Let's consider an XML document that describes a Delphi form. One way we might use XML to describe the form and its components is shown in Figure 1. In this sample, the form is described in a hierarchical way. It includes one label and one panel; this panel is the parent of two buttons.

As you can see, the syntax of this XML document resembles an HTML document. We use tags to describe the different components. In one case, we have some text encapsulated by the <TMemo> and </TMemo> tags, and we have attributes for the tags.

A Well-formed XML Document

An XML document is a hierarchical collection of elements (tags) and their content. Every XML document has one root element that must enclose all other elements in the document. In the example in Figure 1, <TForm> is our root element.

Every element in XML documents must have starting and closing tags. Between these tags, the document can include other tags and textual content. Sometimes you don't need to enclose any more content in a tag. In this case, you can create an "empty" tag and use:

```
<TagName ... attributes ... />
```

as a shortcut to close it, rather than:

```
<TagName ... attributes ...></TagName>
```

Attributes can appear only in the open tag of the element. Every XML document must start with an XML processing tag:

```
<?XML version="nnn"?>
```

A Valid XML Document

A document like the one we used for our sample is a well-formed XML document, i.e. it follows all the rules we described. XML provides a more strict definition of a document, called a Valid document, which is a document that describes its type by referencing a domain-specific description called a DTD (Document Type Definition) and implements the rules of this DTD.

Applications that need to read and process data stored in XML format use an XML parser. XML parsers come in two flavors: validating parsers that check the document against a DTD, and non-validating parsers that ensure the document is well-formed, but don't validate it against a DTD.

A DTD is a collection of rules that describes what the root element of the document must be, what other XML elements can appear in the document, and what elements can be included in other elements. For example, an XML document that describes an order will include elements for the order number, the items purchased in the order, and customer information. Let's assume a sample of the customer information can look as follows:

```
<Customer Name="Aharon Green">
  <Address>
    <Street>1259 NW Pickle Rd.</Street>
    <City>Stamform</City>
    <State>CT</State>
    <Zip>12345</Zip>
  </Address>
  <Phone>508 765 1234</Phone>
</Customer>
```

Our DTD can ensure the Street, City, State, and Zip elements will appear in the Address element, but can't appear in the Customer or Phone elements. As a developer, you might think of a DTD as a COM interface definition, and an XML document that implements this DTD as a COM object that creates one instance of it.

When we said earlier that XML is a meta-language used to create domain-specific markup languages, we were talking about the task of creating a DTD that describes the structure of documents that store and describe information specific to our needs. The issue of DTD creation and manipulation is beyond the scope of a single article introducing XML. For the purpose of this article, we'll discuss only well-formed XML documents, their uses, and how to parse such documents from a Delphi application.

Other XML Objects

We've discussed XML documents as a collection of elements (tags) that can enclose other tags and raw text. Every tagged object in an XML document is called an element; raw text is usually referred to as PCDATA. In addition to these, XML documents can include comments, non-XML streams, XML "processing" instructions for the application handling the document, and XML entities (macros that are expended in the XML stream). The DTD definition has a syntax of its own, but I'll try to keep things simple for this article, and leave the description of DTDs for another time.

Uses for XML

As you may have already noticed, XML allows you to store and describe information in a hierarchical fashion. We often use a hierarchical metaphor in applications, from the Windows Explorer to the folders you use to store your e-mail. We can mimic this hierarchical structure with master-detail tables in our databases, or store the information in objects in memory that describe this hierarchical nature of the data. With XML, we gain a platform-independent standard way to describe data in a hierarchical fashion, and share it between applications.

Because XML is easy to read and understand by applications, it's a perfect method to publish data that needs to be read and manipulated by different applications. Consider search engines on the Web. Go to your favorite, and search for a book about Delphi by typing the query "Delphi+Book." On one search engine I received 382,780 hits, many of which weren't related to Delphi or Delphi books. If the data was searched against a collection of XML documents, the search engine could only look at data that appears in <Book> tags for example, and minimize the number of hits returned.

XML data is easy to publish over the Internet and is platform independent. If you want to share data with users on Windows, Macs, UNIX machines, and IBM mainframes, you're limited to propriety data, or non-structured data formats. With XML, it's easy to create applications on all these platforms (or on the same platform with different development tools) that exchange this information. In addition, many XML language implementations are being created by different organizations and working groups, so you might be forced to work with XML data sooner than you expect.

Related Technologies

The world of XML includes several more pieces that make XML usable in the real world. A short description of the important pieces follows, but due to space constraints, we won't explore them in detail in this article.

Displaying XML documents with XSL. XML documents, as we have seen, are easy to write and read by developers and applications. However, unlike HTML, you can't use XML to display the information in a nicely formatted way in a browser. The solution to this problem is Extensible Style Language (XSL). An XSL filter defines a set of rules that convert an XML document to a display language like HTML. An XSL filter is, in fact, an XML implementation. By associating an XSL filter with XML documents, you can define a way to display the information as HTML documents in a browser (see Figure 2).

Consider a Web application that uses XML data to return queries over the Internet or an intranet. Because the data is stored in XML format for easy searching and analysis, you would want to display it nicely formatted. Usually, an XSL processor that converts the XML data into an HTML display document using the rules in the XSL filter document is activated.

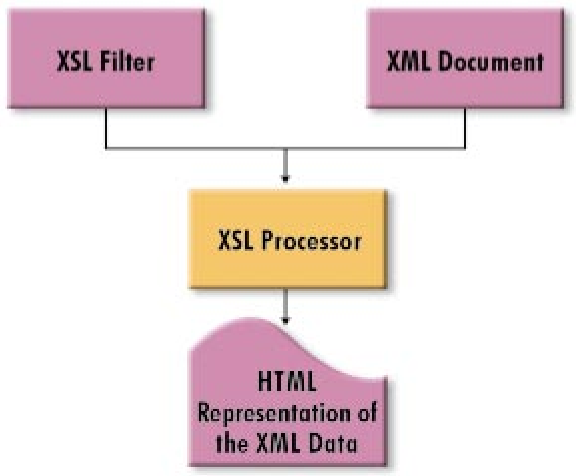


Figure 2: An XSL processor combines the XML data with the XSL filter rules to create a browser-ready HTML document.

```
var
  XMLDoc      : OleVariant;
  RootElement : OleVariant;
...
XMLDoc := CreateOleObject('msxml');
try
  XMLDoc.Url := 'd:\path\to\xml\document.xml';
  RootElement := XMLDoc.Root;
  ...
finally
  XMLDoc := varNull;  // Release the parser.
end;
```

Figure 3: Encapsulated calls to the XML parser.

Constant	Value	Description
XMLELEMTYPE_ELEMENT	0	XML Element
XMLELEMTYPE_TEXT	1	Text
XMLELEMTYPE_COMMENT	2	Comment
XMLELEMTYPE_DOCUMENT	3	Document
XMLELEMTYPE_DTD	4	DTD
XMLELEMTYPE_PI	5	Processing Instruction
XMLELEMTYPE_OTHER	6	Non-XML Data

Figure 4: Supported XML object types.

Linking XML documents with XLL and Xpointer. In HTML, we reference one document from another using the <A HREF> tag. We can also define bookmarks in a document with the <A NAME> tag. XML doesn't offer this functionality as a basic part of the language, but the XML Linking Language and the Xpointer definition are used to extend XML documents, and provide ways to reference one document from another.

XLL and Xpointer go even further. They allow you to create multi-directional links, references to entire subsets of documents, and a mechanism to traverse the element hierarchy of a document and target a specific element or subset of elements in a document. For most applications, XLL and Xpointer aren't important. However, if your application requires linking XML documents, look for information about these technologies.

Things to come. XML is a relatively young technology with a bright future. It's understandable that new developments related to XML usage in the real world will occur as it becomes more popular. Among the interesting things to notice are the official W3C Document Object Model (DOM) for XML, which provides a standard way to access the hierarchy of elements in an XML document from applications; XQL, a query language for XML data; and XML Islands in HTML documents, a technique that allows you to embed XML data in HTML documents.

Parsing XML Documents with Delphi

Parsing XML documents requires an XML parser. The options we have are creating a parser from scratch, purchasing a third-party solution, or using a "system" tool. I'm not a big believer in the need to invent every piece of code that goes in my application; I prefer to concentrate on the logic of my applications, and use as many industry-standard components as possible. Fortunately for us, Microsoft's Internet Explorer 4.0 and later (which is also a part of Windows 98 and the upcoming Windows 2000) includes an XML parser component.

The file, msxml.dll, that IE installs into your system directory is this XML parser. The IE4 version is a non-validating parser that doesn't provide support to the W3C's new DOM standard for XML documents. The new XML parser that will ship with IE5 (in beta at the time of this writing) has many advantages, including support for the new DOM standard. However, because it can change by the time IE5 ships, I prefer to use the IE4 version, which is still supported in the current IE5 beta (and will most likely continue to be supported when IE5 ships). Although it doesn't offer all the bells and whistles of the latest XML development, it's still a useful tool that will get you started in a hurry.

Creating Support Files

To keep the code simple, I will use the XML automation object in this article. Although this won't be as fast as linking via the vtable, it will keep the code simple. However, I will use the type library of the XML parser object to take advantage of the constants it defines.

Locate msxml.dll in the system directory and use Delphi's tlibimp.exe tool (in the \Bin directory of your Delphi installation), e.g.:

```
tlibimp c:\winnt\system32\msxml.dll
```

My experiments indicate that Delphi 3.02 tlibimp.exe doesn't create a correct import Pascal unit. Delphi 4.02 creates an import library that is usable in both Delphi 3.02 and 4.0x. However, notice that if you try to import msxml.dll from the first public IE5 beta release, even Delphi 4's tlibimp.exe will create an un-compilable import unit.

The results of the import process are the files msxml_tlb.pas and msxml_tlb.dcr. I moved these files to Delphi's \Imports subdirectory.

Using the *msxml* Object

The *msxml* object is instantiated like other automation objects — via the *CreateOleObject* method from Delphi's *ComObj* system unit. Most calls to the XML parser will be encapsulated (see Figure 3). Notice the way the root element of the document is obtained via the document's root element. The object should be capable of accessing XML documents over HTTP; use a correct URL to access a document over the network, or a simple file name to parse a local file.

If you're interested in more information about the capabilities of the XML document object created by the parser, refer to the *IXMLDocument2* interface definition in the typelib import unit you created earlier.

The Hierarchical Element Tree with the *msxml* Object

The Root element, and every other XML element represented in the tree created by the parser, has a *Children* property that returns a col-

```
procedure TForm1.ParseDocBtnClick(Sender: TObject);
var
  XMLDoc      : OleVariant;
  RootElement : OleVariant;
begin
  ElementsBox.Items.Clear;
  XMLDoc := CreateOleObject('msxml');
  try
    if (OpenDialog1.Execute) then begin
      ElementsBox.Items.Add(
        'Parsing ' + OpenDialog1.FileName);
      ElementsBox.Items.Add('');
      Caption := 'XML Parser - ' + OpenDialog1.FileName;
      XMLDoc.URL := OpenDialog1.FileName;
      RootElement := XMLDoc.Root;
      WriteElement(RootElement, 0);
    end;
  finally
    XMLDoc := VarNull;
  end;
end;
```

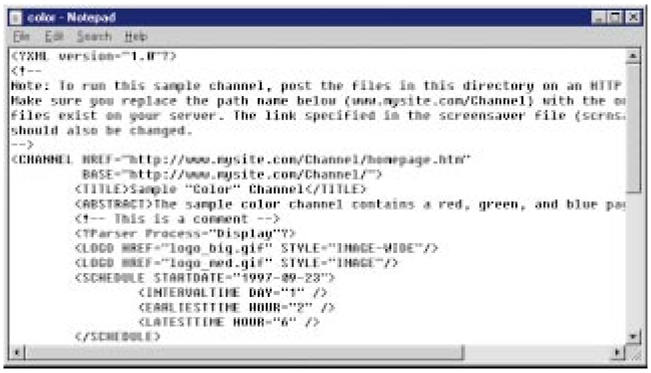
Figure 5: The *OnClick* event handler for the **Parse Document** button.

```
procedure TForm1.WriteElement;
var
  s              : string;
  t              : WideString;
  Count, i       : Integer;
  ChildElement   : OleVariant;
  eColl          : OleVariant;
begin
  s := '';
  for i := 1 to level do
    s := s + '  ';
  case AnElement.Type of
    XMLELEMTYPE_ELEMENT:
      begin
        t := AnElement.TagName;
        s := s + 'E:' + t;
        if (AnElement.Text <> '') then
          s := s + ' (' + AnElement.Text + ')';
        ElementsBox.Items.Add(s);
        eColl := AnElement.Children;
        if (assigned(TVarData(eColl).VDispatch)) then begin
          Count := eColl.Length;
          for i := 0 to Count - 1 do begin
            ChildElement := eColl.Item(i, 0);
            WriteElement(ChildElement, 1 + level);
          end;
        end;
      end;  // XML element.
    XMLELEMTYPE_TEXT:
      begin
        s := s + 'T:';
        ElementsBox.Items.Add(s);
      end;  // Text.
    // Check for the other object types.
    ...
  end; // case
end;
```

Figure 6: The *WriteElement* procedure.

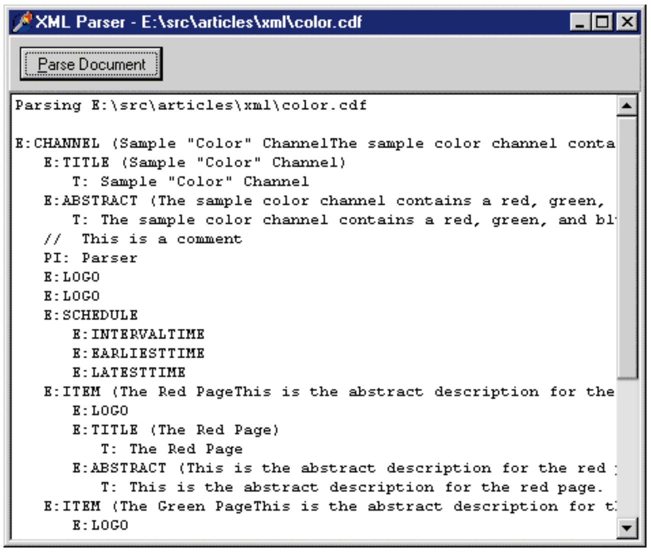
lection of all the objects it encloses. Unfortunately, Microsoft used the *element* phrase to describe every object in the XML document hierarchy — not just XML elements. Although the *Children* property of an XML element object is defined as an *ElementCollection* object, it can hold objects like comments, DTD information, text, etc. The table in Figure 4 indicates the object types supported.

To illustrate the parsing process, I created a simple form with a button to activate the parser, and a list box that will hold a hierarchical representation of the parsed XML file. The event handler for the button is shown in Figure 5. We create the XML parser object and obtain the root element as indicated before. The interesting bit of information is the call to the *WriteElement* procedure, which has the following declaration:



```
<?XML version="1.0"?>
<!--
Note: To run this sample channel, post the files in this directory on an HTTP
Make sure you replace the path name below (www.mysite.com/Channel) with the ac
files exist on your server. The link specified in the screensaver file (scrns.
should also be changed.
-->
<CHANNEL HREF="http://www.mysite.com/Channel/homepage.htm"
         BASE="http://www.mysite.com/Channel/">
        <TITLE>Sample "Color" Channel</TITLE>
        <ABSTRACT>The sample color channel contains a red, green, and blue pa
        <!-- This is a comment -->
        <?Parser Process="Display"?>
        <LOGO HREF="logo_big.gif" STYLE="IMAGE-WIDE"/>
        <LOGO HREF="logo_med.gif" STYLE="IMAGE"/>
        <SCHEDULE STARTDATE="1997-09-23">
                <INTERVALTIME DAY="1" />
                <EARLIESTTIME HOUR="2" />
                <LATESTTIME HOUR="6" />
        </SCHEDULE>
```

Figure 7: Part of the Color.CDF sample that ships with the Internet Client SDK (*INetSDK*) with minor changes to show special object handling.



```
Parsing E:\src\articles\xml\color.cdf

E:CHANNEL (Sample "Color" ChannelThe sample color channel conta
   E:TITLE (Sample "Color" Channel)
      T: Sample "Color" Channel
   E:ABSTRACT (The sample color channel contains a red, green,
      T: The sample color channel contains a red, green, and bl
   //  This is a comment
   PI: Parser
   E:LOGO
   E:LOGO
   E:SCHEDULE
      E:INTERVALTIME
      E:EARLIESTTIME
      E:LATESTTIME
   E:ITEM (The Red PageThis is the abstract description for the
      E:LOGO
      E:TITLE (The Red Page)
         T: The Red Page
      E:ABSTRACT (This is the abstract description for the red
         T: This is the abstract description for the red page.
   E:ITEM (The Green PageThis is the abstract description for t
      E:LOGO
```

Figure 8: The Color.CDF object tree displayed by our sample application.

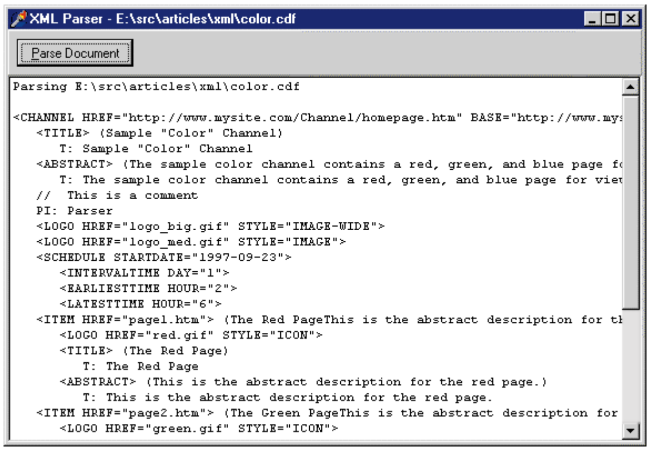
```
procedure WriteElement(
  AnElement: OleVariant; Level: Integer);
```

The implementation of this procedure is shown in Figure 6. The *WriteElement* code prefixes the object based on its level, determines the type of the object, and creates the string that will be displayed in the printed tree.

For an XML element object, get and display the tag name of the element (see Figure 7). If there is text enclosed in the element, we display this text in parentheses, and call *WriteElement* recursively for every object in the element's *Children* collection property (see Figure 8). Notice the technique we use to ensure that the collection property holds a valid automation object. We cast it to *TVarData*, and check the *VDispatch* member to ensure it's assigned.

Reading Attribute Values with the *msxml* Object

If you inspect the *IXMLElement2* interface in msxml_tlb.pas, you'll notice it exposes an *Attributes* property via the *Get_Attributes* method. This property (like the *Children* property) returns an element collection object. Unlike the *Children* property, the objects available through this collection are XML *Attribute* objects (repre-



```
Parsing E:\src\articles\xml\color.cdf

<CHANNEL HREF="http://www.mysite.com/Channel/homepage.htm" BASE="http://www.mys
   <TITLE> (Sample "Color" Channel)
      T: Sample "Color" Channel
   <ABSTRACT> (The sample color channel contains a red, green, and blue page fo
      T: The sample color channel contains a red, green, and blue page for viev
   //  This is a comment
   PI: Parser
   <LOGO HREF="logo_big.gif" STYLE="IMAGE-WIDE">
   <LOGO HREF="logo_med.gif" STYLE="IMAGE">
   <SCHEDULE STARTDATE="1997-09-23">
      <INTERVALTIME DAY="1">
      <EARLIESTTIME HOUR="2">
      <LATESTTIME HOUR="6">
   <ITEM HREF="page1.htm"> (The Red PageThis is the abstract description for th
      <LOGO HREF="red.gif" STYLE="ICON">
      <TITLE> (The Red Page)
         T: The Red Page
      <ABSTRACT> (This is the abstract description for the red page.)
         T: This is the abstract description for the red page.
   <ITEM HREF="page2.htm"> (The Green PageThis is the abstract description for
      <LOGO HREF="green.gif" STYLE="ICON">
```

Figure 9: The Color.CDF sample displayed with XML element attribute information.

sented in the code with the *IXMLAttribute* interface). To parse the *Attributes*, we need to add the following code to the *WriteElement* procedure for the XMLELEMTYPE_ELEMENT case clause:

```
aColl := AnElement.Attributes;
if (assigned(TVarData(aColl).vDispatch)) then begin
  t := '';
  for i := 0 to aColl.Length - 1 do begin
    AnAttr := aColl.Item(i, 0);
    t := t + ' ' + AnAttr.Name + '="' + AnAttr.Value + '"';
  end;  // Loop through all the attributes.
  if (t <> '') then
    s := s + t;
end;
```

For this example, I also changed the way an XML element is displayed in our list box to look like a tag (instead of the E: prefix of the previous example). If your application knows the attributes expected for an element, you could use the *Attribute* property to gain access to the attribute value directly without iterating through all the attributes as this sample does.

Other Capabilities of the *msxml* Object

The *msxml* object includes other methods and properties you can use if you want to build the XML document in memory using the hierarchical object model the object exposes. You can use methods such as the *AddChild*, *RemoveChild*, *GetAttribute*, *SetAttribute*, and *RemoveAttribute* to change the document. The document's *CreateElement* method is used to create an unconnected object, and the *AddChild* method of an element is used to insert it into the hierarchy (see Figure 9).

More information is available on Microsoft's SiteBuilder Web site in the Internet Client SDK documentation, and in the code created for msxml_tlb.pas.

Conclusion

XML is a platform-independent, Internet-ready standard for the creation of domain-specific markup languages. XML excels at storing hierarchical information and application-neutral data sharing. Microsoft, IBM, Sun, and most other software corporations are busy creating XML tools, documentation, sample code, and definitions for different uses. XML technology is being developed at a rapid pace, and the chances are good that, sooner or later, your application will be able to take advantage of XML data.

This article provides a brief introduction to XML and some of its related technologies, as well as demonstrates how to use the IE4 XML parser object from Delphi applications. The XML 1.0 definition and DOM definition have been released by the W3C. XSL is currently still in draft mode, and we've witnessed a lot of changes between the first and later drafts. I've decided to wait for XSL to mature in the W3C development process before I invest time in writing or learning to use an XSL processor.

I recommend getting a good XML book to learn more about XML document creation, XML applications, and DTDs. As usual, the Internet is full of good resources; http://www.microsoft.com/xml includes information about Microsoft's offerings, which are of interest to Delphi developers. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JUL\DI9907RL.

Ron Loewy is a software developer for HyperAct, Inc. He is the lead developer of eAuthor Help, HyperAct's HTML Help authoring tool. For more information about HyperAct and eAuthor Help, contact HyperAct at (515) 987-2910, or visit http://www.hyperact.com.

By Aleksandr Gofen

object vs. class

Fewer Pointers, Less Double Thinking

Delphi classes are hidden pointer variables. This requires special thinking and attention. This article discusses objects that have nothing to do with pointers, objects that behave as normal global or local variables in a high-level programming language.

By this I mean objects declared as the "old fashioned" Object Pascal **object** type. Variables of this type may be either global/local, or dynamic, i.e. reside on the heap. What is called the "old" object calling convention is actually a mechanism of referring to an object:

```
SomeObject = object
```

via a pointer:

```
SomeObjectPtr = ^SomeObject;
```

Because of the convention that variables of type **class** in Delphi are hidden pointers, this way of calling the object becomes obsolete, but not the **object** type itself. The linear memory model implemented since Windows 95 and NT, together with the new features of Delphi 2, 3, and 4, leave cases when the **object** type provides a better solution.

Pitfalls of the class Type

In general, pointers are in conflict with fundamental principles of high-level languages, such as:

- the rule of scope, and
- one-to-one correspondence between a variable and its instance (memory image).

Classes and pointers don't work this way. Not only do we need to declare and initialize them like other variables, we also have to take two additional actions: to create and later destroy the instances (to allocate and de-allocate the memory). Thus, although a pointer:

```
var p: ^T
```

is visible and exists only within its scope, its instance *p*^ (some structure of type *T*) doesn't necessarily exist within this scope, and doesn't necessarily disappear outside it. Moreover, it may happen that several pointers:

```
var p1, p2, p3: ^T
```

point to the same instance of type *T*, or conversely, there may be instances of *T* to which no pointer points at all. Pointers can also point to a wrong place.

Hidden pointers (type **class**) are even more peculiar. For objects:

```
var a, b: TSomeClass
```

neither the assignment `a := b`, nor the condition `a = b` has its usual mathematical meaning. Instead, special methods, *Copy* and *Equal*, must be designed to perform these functions. In addition:

```
procedure Any1(x: TSomeClass, ...)
```

as well as:

```
procedure Any2(const x: TSomeClass, ...)
```

can actually change the instance of *x*, despite the fact that it looks like it's called by value.

One more example: imagine that you have to design a new type, say *TMatrix*, and operations on values of that type, such as *Sum* and *Product*. Normally, you need to declare:

```
function Sum(const A, B: TMatrix): TMatrix;
function Product(const A, B: TMatrix): TMatrix;
```

This allows you to use expressions such as `Product(Sum(A, B), C)`, but not if *TMatrix* is a class. Where do the functions and the implicit variable *Result* (all of which are pointers) point to? Who is responsible for creating and destroying the instances of the *Result* of these functions? Here it's better not to bother with functions at all, but to declare procedures instead and give up the opportunity to write the previously mentioned expressions:

```
procedure Sum(const A, B: TMatrix; var C: TMatrix);
procedure Product(const A, B: TMatrix; var C: TMatrix);
```

Again, because *TMatrix* is a class, **const** and **var** have no usual meaning and are only used to show the input parameters and the output in the procedures.

(Note: An undocumented feature of Delphi 2-4 syntax even allows omitting the carat (^) in pointer expressions, e.g.:

```
p1^.SomeField or p2^[i, j]
```

can be replaced with:

```
p1.SomeField or p2[i, j]
```

This means that pointers can be hidden not only for classes now; therefore the provocative confusions between instances and their pointers are very real.)

This shows that explicit or implicit pointer variables are somehow unnatural for high-level languages, because they require a sort of "double-thinking" in the design phase, and can cause even more pain while debugging. The crucial issue here is to distinguish between logical and technical reasons to use pointer variables. Logically, pointers are needed in Pascal for designing special data structures such as linked lists, graphs, etc., which cannot be done in another way.

There are also technical reasons that have influenced programming style for a decade. Here are at least four such reasons.

One. Memory access limitations for IBM PCs required that no data structure exceed 64KB. Thus, the stack size was usually much less, typically 16KB (program size plus stack size could not exceed 64KB). This limited the space for global and local variables. Since Delphi 2, this is no longer true. Now the stack size is controlled by the compiler directive's minimum stack size ($MINSTACKSIZE, default 16KB), and maximum stack size ($MAXSTACKSIZE, default 1MB). The stack grows incrementally as needed. Thus, memory allocation on the heap is no longer more efficient than allocation on the stack.

Two. The only way to use upper memory before the advent of 32-bit operating systems was by using the heap and pointers. Again, this is no longer true.

Three. For arrays of largely variable size, it was reasonable in standard Pascal to allocate memory on the heap rather than to reserve the maximum possible array size (allocated on the stack). Since the introduction of variant arrays in Delphi 2 and dynamic arrays in Delphi 4, this reason has become less important. Variables and fields of these types require the memory according to their current length (plus overhead), and programmers may treat them similar to ordinary local or global variables. (Note: Variant arrays that aren't "locked" are less efficient, but this only matters for performance-critical parts of applications.)

The previously mentioned technical reasons are no longer valid, but the following one still is.

Four. The Visual Component Library (VCL) is based on the Windows API, which makes heavy use of classes and pointers. Thus, if users want to build a hierarchy of objects inheriting from the VCL, these objects must also be of type **class**, i.e. pointers.

Nevertheless, designing new structures that aren't descendants of the VCL, we can find better solutions than using classes. In the next section, we will describe programming with type **object** as an alternative to **class**.

Where Type object Is Better Than Type class

In the Delphi literature, we cannot find much about type **object** except that it's still supported to be backward-compatible with Object Pascal. Therefore, we use the syntax given in Object Pascal before the introduction of Delphi. We can also use properties. The relationship of type **object** with type **class** is such that a hierarchy based on inheritance from type **object** cannot mix with that of **class**. Nevertheless, fields of **object** or **class** may be of any type.

We are going to use the **object** type for declaring global or local variables without using pointers. In practice, we usually want them to be global, because they don't lose their values outside scope. To avoid the clutter of many global variables concentrated in one place, we are free to design as many units as we logically need so each unit declares its own global variables.

The biggest advantage of the **object** type is demonstrated when we define structures of fixed length at design time. Then there's no need to allocate dynamic memory to certain fields and structures. (Otherwise, the **class** type would be more appropriate.)

While designing a hierarchy of objects without virtual methods, we don't need to declare and use constructors and destructors. (Constructors are still needed for virtual object variables, not to allocate memory to them, but to initialize the virtual call mechanism, i.e. late binding.) See Figure 1 for some examples.

Given the types shown in Figure 1, the following functions perform operations on complex numbers:

```
function AlgOper(const z1: TAlgCmplx; const op: Char;
  const z2: TAlgCmplx): TAlgCmplx;
function ExpOper(const z1: TExpCmplx; const op: Char;
  const z2: TExpCmplx): TExpCmplx;
```

and functions:

```
function ExpToAlg(const z: TExpCmplx): TAlgCmplx;
function AlgToExp(const z: TAlgCmplx): TExpCmplx;
```

transform one format into another. With these functions, it's easy to use arithmetic expressions with complex numbers. For example, the mathematical expression (z1 - z2)/(z3 + z4) may look like:

```
AlgOper(AlgOper(z1, '-', z2), '/', AlgOper(z3, '+', z4));
```

Another example defines an object that provides bit-to-bit access within one byte:

```
TBit = [0..1];
T8Bits = object
  BitStore: Byte;
  function  GetBit(const i: Byte): TBit;
  procedure PutBit(const i: Byte; const r: TBit);
  property Bit[const i: Byte]: TBit
    read GetBit write PutBit;
end;
```

With this definition and:

```
var s: T8Bits
```

we can write for example:

```
s.Bit[3]:= 0;
```

or

```
with s do Bit[2] := Bit[1];
```

(Note: It seems reasonable to specify *Bit* as a default property in this example, but all versions of the Delphi compiler up to 4 do not support the property specifier, **default**, for the **object** type.)

The advantages of using type **object** over type **class** in the previous examples are obvious. Besides that the variables of the defined types may be treated simply as non-pointer variables, the type *T8Bits* occupies exactly one byte, not four, as it would if it were a class. Also, if complex numbers were defined as a class, it would be impossible to define the operations as functions and to use them in expressions.

So far, the examples have been rather simple, involving no hierarchy, inheritance, nor polymorphism. Let's consider some examples with all these features.

```
// Defines complex numbers in form  a + ib.
TAlgCmplx = object
  Re, Im: Extended;
  // z.Conj returns the conjugation to z.
  procedure Conj;
  // z.Init(x,y) means z = x + iy.
  procedure Init(const x, y: Extended);
end;

// Defines complex numbers in form r*exp(i*Arg).
TExpCmplx = object
  r, Arg: Extended;
  // z.Conj returns the conjugation to z.
  procedure Conj;
  // z.Init(rad, fi) means rad*exp(i*fi).
  procedure Init(const rad, fi: Extended);
end;
```

Figure 1: No need to declare and use constructors and destructors when designing a hierarchy of objects without virtual methods.

First, a general note: Given the definitions:

```
TObj1 = object { some object };
TObj2 = object(TObj1);
TObj3 = object(TObj2);
TCls1 = class { some class };
TCls2 = class(TCls1);
TCls3 = class(TCls2);
```

the respective variables:

```
var Obj1: TObj1; Obj2: TObj2; Obj3: TObj3;
    Cls1: TCls1; Cls2: TCls2; Cls3: TCls3;
```

are assignment-compatible from left to right, i.e. pointer *Cls1* may point to instances of any of the types *TCls1*, *TCls2*, or *TCls3*. If all three class types have a virtual method:

```
procedure Same(...); virtual;
```

and a constructor, `Create(...)`, then:

- if `Cls1 := TCls1.Create(...)`, `Cls1.Same(...)` calls the version of *Same* for *TCls1*;
- if `Cls1 := TCls2.Create(...)`, `Cls1.Same(...)` calls the version of *Same* for *TCls2*;
- if `Cls1 := TCls3.Create(...)`, `Cls1.Same(...)` calls the version of *Same* for *TCls3*.

Thus, the variable *Cls1* always calls the correct versions of the method according to the one that created the instance of *Cls1* at run time. This is how polymorphism works for the **class** type.

In contrast, when we deal with variables of type **object** directly, it's impossible to not know at compile time which of the objects in the hierarchy calls the method. For example, `Obj2.Same(...)` calls exactly the version of the method for the type *TObj2*. Thus, for the **object** type, the usual form of polymorphism never occurs, at least not with a direct method call.

However, it may happen when one method calls another. Suppose there is a method belonging only to *TObj1*, say *TObj1OnlyMethod*, and in its body there is a virtual method, *Same* (*Same* may belong to *TObj1*, *TObj2* or *TObj3)*. Then, at compile time, it cannot be known what version of *Same* must be

```
type T3D = array[1..3] of Extended;
T3x3 = array[1..3, 1..3] of Extended;

T3DPoint = object
  x: T3D;  // 3D vector.
  procedure Init(const a: array of Extended);
  procedure Apply(var v: T3D);  // Just calls Move.
  procedure Move(var v: T3D); virtual; // Here it's empty.
end;

// The inherited field x is intended for displacement.
TDisplacement = object(T3DPoint)
  constructor Init(const d: array of Extended);
  // Applies displacement x to vector v.
  procedure Move(var v: T3D); virtual;
end;

TGenTransfrm = object(TDisplacement)
  A: T3x3;  // Matrix of rotation.
  constructor Init(const d, B: array of Extended);
  // Applies rotation A and displacement x to vector v.
  procedure Move(var v: T3D); virtual;
  procedure Prod(const B: T3x3);  // A := A*B
end;

var
  v: T3D;
  Shift: TDisplacement;
  GenTrans: TGenTransfrm;
```

Figure 2: An example of **object** types.

called. With that in mind, let us consider the example of **object** types shown in Figure 2.

(Note: For virtual methods of **object** type, the **virtual** directive should be used instead of **override**.) Given these definitions, the variable *v.x* represents some point in 3D space, *Shift.x* is intended to keep a displacement, and *GenTrans* represents a general transformation with displacement *GenTrans.x* and rotation *GenTrans.A*. Then, the call *Shift.Move(v)* performs the displacement of *v*, the call *GenTrans.Move(v)* performs the rotation, and then displacement of *v*. In these cases, the caller is known at compile time.

To demonstrate the case when the caller is unknown at compile time, the ancestor type *T3DPoint* introduces two dummy methods: *Move*, which does nothing; and *Apply*, which only calls *Move*. Contrary to the *Move* method, there is only one version of *Apply*, inherited by all descendants. Thus, *Shift.Apply(v)* calls *TDisplacement.Move(v)*, and *GenTrans.Apply(v)* calls *TGenTransfrm.Move(v),* i.e. the appropriate version of the method *Move* is called in each case. So, polymorphism can work with the **object** type, although its usage is more limited.

Finally, a few notes on using a "mixture" of the **class** type, say *SomeClass*, having certain fields of the **object** type. Such combination can be useful if no other field of *SomeClass* is of the **class** type. Then, the methods *Create* and *Destroy* (inherited from the VCL's *TObject*) will automatically allocate and de-allocate the right amount of memory, so users don't need to bother to design their own *Create* and *Destroy* methods to perform these operations for every field of *SomeClass*.

Conclusion

Since version 5.5 of Borland's Turbo Pascal, objects defined as type **object** can be referred to directly or via pointers. Direct reference is still necessary. The **object** type is a complementary structure to Delphi's **class** type in the sense that **class** is always a hidden pointer to the respective object. Both are similar, but because Delphi is still evolving as a language, there are few differences in syntax, which will hopefully disappear in the later versions.

The practical advantages of the **object** type increased due to the linear memory model, and the potentially unlimited stack implemented in Delphi 2, 3, and 4. The **object** type provides better and safer solutions in situations where developers need structures of a short length, or structures of a long fixed length, or of a long length that varies only in a narrow range.

I wish to express my deep appreciation to Dr Manfred Mackeben, who helped to essentially improve this text. ∆

Aleksandr Gofen is a programmer at the Smith-Kettlewell Eye Research Institute in San Francisco, where he has worked for the last three years. Previously, he was a Senior Researcher for the Institute of Computer Science (Academy of Sciences, Russia) and Hydro-Meteorological Center in Moscow, Russia. Gofen has been developing scientific applications in all versions of Delphi and Inprise's Pascal, varying from the Numeric Weather Forecast and the Taylor Solver, to the Macula Mapping Test (Eye Research Institute). He can be reached via phone at (415) 561-1644 or e-mail at galex@skivs.ski.org.

By Warren Rachele

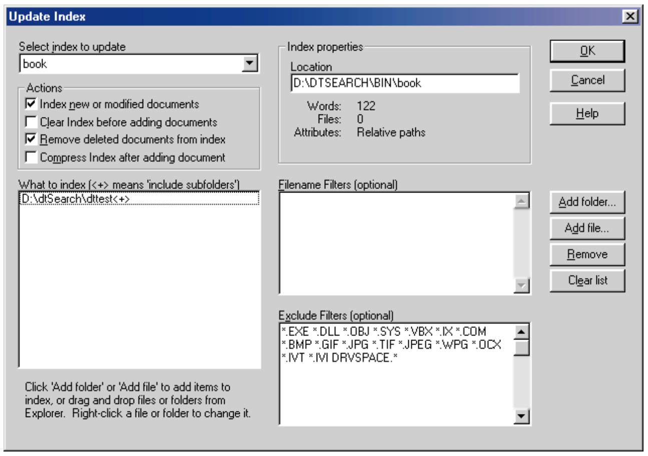
dtSearch

The Search Is Over

The volume of data being stored in electronic format today is unimaginable, and growing every second. Every fact or half-truth you would ever want to know is stored somewhere. The trouble, as demonstrated on a minor scale by our own untidy hard drives, is finding those facts. Sifting through collections of documents and files with such intriguing names as Document13.doc or weasels.zip to find the name of one of David Horowitz' "fellow travelers" is a time-consuming and sometimes fruitless process.

To the rescue comes dtSearch by DT Software, Inc. dtSearch is a blindingly fast text search and retrieval product, currently shipping version 5.1. This tool will build indexes of the files on your storage medium, then allow you to build complex search requests that operate against the index files rather than the source files themselves. The results are amazing. Complicated Boolean or natural language requests are answered in seconds, showing the source document contents with the location of the matching data highlighted.

In addition to the dtSearch end-user product, DT Software also markets the core of their software in the form of the dtSearch Text Retrieval Engine. This is a DLL-based product with APIs for Delphi, C/C++, and Visual Basic. The engine can be incorporated, royalty-free, into your software to provide the same functionality offered by the dtSearch product.


Update Index

Select index to update: book

Actions
- [✓] Index new or modified documents
- [ ] Clear Index before adding documents
- [✓] Remove deleted documents from index
- [ ] Compress Index after adding document

Index properties
Location: D:\DTSEARCH\BIN\book
Words: 122
Files: 0
Attributes: Relative paths

What to index (<+> means 'include subfolders'): D:\dtSearch\dttest<+>

Filename Filters (optional):

Exclude Filters (optional): *.EXE *.DLL *.OBJ *.SYS *.VBX *.IX *.COM *.BMP *.GIF *.JPG *.TIF *.JPEG *.WPG *.OCX *.IVT *.IVI DRVSPACE.*

Click 'Add folder' or 'Add file' to add items to index, or drag and drop files or folders from Explorer. Right-click a file or folder to change it.

Figure 1: Defining the contents of the index by inclusion or exclusion of specific file types.

dtSearch

To appreciate the dtSearch engine's capabilities, the vendor recommends using the dtSearch tool itself. The recipient of numerous flattering reviews since its original release in 1991, this search tool has continued to expand and refine its capabilities. The product is eye-blink fast in seeking out instances of specified text. It uses one or more indexes based upon the contents of the documents you want to search to find and highlight the instances of the search request, displaying the documents for you if desired.

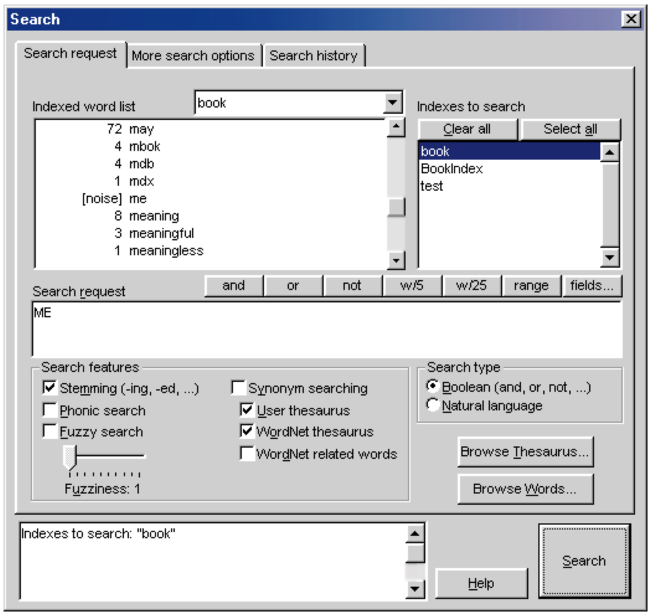
Putting the product to use couldn't be simpler. dtSearch requires an index to perform its searches, so this is where you must start. Building an index is straightforward. After naming the index and electing to add files to it, the window shown in Figure 1 is displayed. This dialog box is used to define the contents of the index by the inclusion or exclusion of specific file types. The user has the option of indexing specific files by placing discrete file names in the **What to index** list, or creating more general indexes by specifying entire directories of documents to be included. Every file, including ZIP, HTML, PDF, and major word processor, spreadsheet, and database files can be indexed directly by dtSearch, giving you enormous flexibility in your use of the product. A default list of excluded file types is displayed in the lower right and can be modified at any time.

Once initiated, the indexing process is *fast*. The test directory included the manuscript for a 500-page book and a special zip file that contained the same files. dtSearch completed this task in a few seconds and awaited my search request. The document index that's created is a database containing the locations of the words in the documents that were selected. The index doesn't include noise words such as "but" and "if." Memory requirement for the index-building process is disk space about equal to that of all the documents to be indexed while the indexing process is running. Once completed, this temporary storage is released, and the resultant index is about one quarter of the size of the total document space.

The indexes created are infinitely manageable, and dtSearch provides a suite of utilities to perform these functions. All indices can be updated and modified at will, and can be merged to the advantage of the search processes. A library manager is provided for network installations to manage the sharing of index files.

Searching for text is the reason that dtSearch exists, and it excels at this task. dtSearch performs combinations of Boolean, proximity, numeric range, soundex, etc. shaped search requests alone or in tandem. The program supports concept searching, a method that utilizes a thesaurus to show hits based on synonyms to the words selected, and user-adjustable fuzzy searching to locate words even if they're misspelled in the source documents. Natural language searching is also a feature of the dtSearch software, allowing the user to develop query-by-example searches.

To initiate a simple search, the user selects **Search** from the menu and is presented with the dialog box shown in Figure 2. Selecting the index or indexes to be used, a list of cataloged words paired with the number of occurrences is displayed from that index. Typing a single word into the **Search request** field and clicking the **Search** button is the simplest form of a search. The search is performed, and a list scheduling the documents in which the word


Search

Search request | More search options | Search history

Indexed word list: book

72 may
4 mbok
4 mdb
1 mdx
[noise] me
8 meaning
3 meaningful
1 meaningless

Indexes to search: Clear all | Select all

book
BookIndex
test

Search request: and | or | not | w/5 | w/25 | range | fields...

ME

Search features
- [✓] Stemming (-ing, -ed, ...)
- [ ] Phonic search
- [ ] Fuzzy search
- Fuzziness: 1
- [ ] Synonym searching
- [✓] User thesaurus
- [✓] WordNet thesaurus
- [ ] WordNet related words

Search type
- (•) Boolean (and, or, not, ...)
- ( ) Natural language

Browse Thesaurus... | Browse Words...

Indexes to search: "book"

Help | Search

Figure 2: Type one word into **Search request** and click the **Search** button for the simplest search.

appears is displayed, matched by a representation of the first document in that list having each of the occurrences highlighted. By default, you'll quickly notice the Stemming feature has located additional variants of the word you requested. For example, a search on the word "Sort" also located "Sorts," "Sorted," and "Sorting."

Extensive search options are available to the dtSearch user. Searching is not limited to utilizing the indexes only; files can be Boolean-searched without the benefit of the index. In addition, the file set can be limited by file name, date, or size. These options are applicable whether you select a Boolean, or a natural language search. The simple search detailed in the previous paragraph is easily expanded using the Boolean operators AND, OR, or NOT, and providing a second operand. There is no limit to the number of connected conditions that can be used in a search statement, which can also be expanded to include proximity qualifiers. Proximity measures the distance, indicated in words, between the indicated word and a specified marker in the file, such as the first or last word of the document. Natural language searching builds search requests from combinations of words, phrases, or sentences. An unstructured search request ignores Boolean operators and interprets the statement in a more natural manner. The returned documents are ranked by their relativity to the request.

dtSearch doesn't stop with the search capabilities, but offers equally extensive display options for the search results. The "hit list" and matching documents can be viewed in a split window or separate windows altogether. The documents can be printed and formatted, and text cut and moved to other documents, making this tool something text workers will find they can't do without.

dtSearch Web

dtSearch Web is an ISAPI-based Internet search engine based on the dtSearch tool. Offering multiple search options, dtSearch Web makes it a simple process for a Webmaster to quickly add instant text search options to any NT-based Internet or intranet site. The match results are displayed in the user's browser, including marked hits inside the HTML documents. dtSearch Web adds "hit" navigation to the browser display. A link at the top of the document gives the user access to the first hit; clicking the highlight markings before and after each hit takes the user to the next or previous hit in the Web page. If the original document in which dtSearch locates the specified text is not in HTML format, dtSearch will automatically convert the native text into HTML.

The dtSearch Text Retrieval Engine

The dtSearch Text Retrieval Engine allows the developer to incorporate all the features discussed in connection with the dtSearch end-user product into their own Delphi applications. The engine API allows the product to be extended to support proprietary file formats, allows your application to supply text directly to the engine for indexing non-file data, allows the engine to perform in-memory searches, and supports multi-threaded searching.

Using all this power is not a trivial matter, however. Integrating the dtSearch Text Retrieval Engine is not a simple matter of dropping a component onto your project and setting a few properties. You'll have to code all the functionality you need, and link it back to the API. This is where usage becomes tough, as the documentation is somewhat bereft of examples for the Delphi programmer. There's an excellent sample application you can use,

```
procedure TForm1.FormCreate(Sender: TObject);
var
  initInfo : dtsInitInfo;
  debuglog : array[0..FilenameLen] of Byte;
  reply    : Word;
  currDir  : string;
begin
  StrPcopy(@debuglog,'dtsearch.log');
  _dtssDebugLogEx(@debuglog, dtsLogCommit);

  Init_dtsInitInfo(initInfo);
  GetDir(0, currDir);
  StrPCopy(@initInfo.PrivateDir, currDir);

  initInfo.pShowErrorFn := @ReceiveMessage;
  initInfo.pShowInfoFn := @ReceiveMessage;
  initInfo.pAskYesNoFn := @ReceiveQuestion;

  _dtssDoInit(initinfo, reply);
  if (reply <> 0) then
    begin
      MessageDlg('Errorcode from dtsDoInit is ' +
        IntToStr(initInfo.errorcode) +
        ', failure to init.', mtError, [mbOk], 0);
      Close;
    end
  else
    MessageDlg('Engine initialized', mtInformation,
               [mbOk], 0);
end;
```

Figure 3: An example of the programming required to integrate the dtSearch engine into your program.

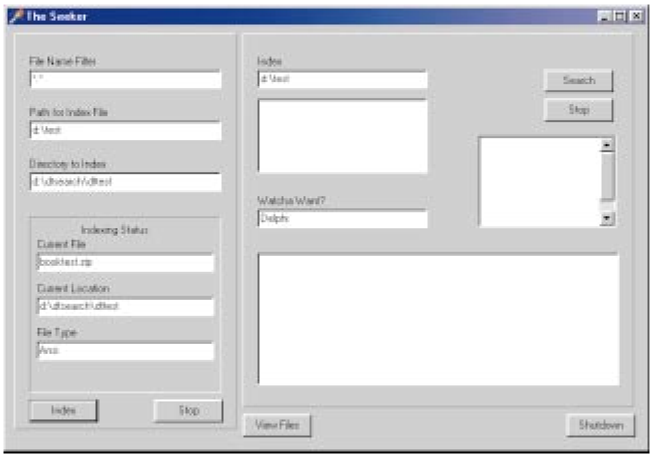
but you're required to dig deeply into the included text files to locate basic information needed to get started. For example, to determine that you must include the DTSearch.pas file with your program's units, you would need to dig into the reame_dtengine.txt file.

Using the engine is separated into four steps when integrated into your program (dtSearch maps an API function call to each):

1) Initialize the engine with *dtssDoInit*.
2) Create an index of the documents with *dtssDoIndexJob*.
3) Perform the search request with *dtssDoSearchJob*.
4) Close the engine with *dtssDoShutDown*.

Before diving into the code, the Delphi programmer must be fluent in addressing DLLs written in C/C++ through their native interface. Though the structures you'll be addressing are defined in DTSearch.pas, many of them require pointers to functions and objects. You must understand the referencing and dereferencing of pointers to successfully use this product.

An example of the programming required to integrate the dtSearch engine into your program is shown in Figure 3. This procedure is the initialization phase of the process, and starts by defining and allocating a debugging log. The *dtsInitInfo* structure is initialized and filled with the appropriate values before calling the initialization function. This structure is passed to the function to supply information to the search engine about the directories to be used when indexing and searching, and to provide the names of the functions to call to display error messages. Note that the callback functions, such as those assigned to



File Name Filter: *.*
Path for Index File: d:\test
Directory to Index: d:\dtsearch\dtest
Current File: booktest.zip
Current Location: d:\dtsearch\dtest
File Type: Ansi
Index: d:\test
Watcha Want?: Delphi

Figure 4: As shown in this demonstration, the indexing functions return real-time status messages during the process.

pShowErrorFn, *pShowInfoFn*, and *pAskYesNoFn*, must be declared as *cdecl*.

As with the end-user product, an index must be created before issuing the search requests. Incorporating this functionality into your program follows much the same path as the initialization process; initialize the appropriate structure, input the values needed to specify the process, and issue the function call to create the index. As illustrated in the demonstration program shown in Figure 4, the indexing functions return real-time status messages during the process. A callback function retrieves and displays those messages to the user. All the indexing options available in the end-user product are exposed through the engine. The options are set through the members of a structure — *dtsOptions* — and set or retrieved by calls to *dtssSetOptions* or *dtssGetOptions*.

All this activity is merely set up for the main course: the searching capabilities. The simple steps required for a search request are to specify the index or indexes to be used, define a search request, and let it go. Providing a container for the results allows the user to immediately see the results of their search. The parameters for the search are defined into the *dtsSearchJob* structure, just as they are in all other dtSearch tasks. In the Seeker application, the results are directed to an external file that is later read to fill the results listbox. It's important to remember that the results are returned in the form of a number of hits and the file names in which those hits reside. This process doesn't manipulate the source files.

The files containing the matches can be displayed in HTML format by using the function *dtssConvertFile*. When this function is run, it will insert hit markers around the located text and add header and footer markers to the file. These facilitate the navigation of the document in a Web browser or other HTML display tool. This conversion process doesn't affect the underlying file; it creates a new file for display purposes.

Usage Notes

There is a price to be paid for all this unbridled power. To use the dtSearch Text Retrieval Engine in your application can be quite a challenge. This is a complicated product with a non-Delphi interface, and you must be prepared to spend the time

INFORMANT FACT FILE

The searching capabilities of dtSearch and the dtSearch Text Retrieval Engine are nothing short of astounding. Complicated Boolean or natural language requests are answered in seconds, showing the source document contents with the location of the matching data. However, there is a price to be paid for all this unbridled power; this is a complicated product with a non-Delphi interface, and you must be prepared to spend time understanding the integration requirements.

DT Software, Inc.
2101 Crystal Plaza Arcade, Suite 231
Arlington, VA 22202

Phone: (800) IT-FINDS or (703) 413-3670
E-Mail: sales@dtsearch.com
Web Site: http://www.dtsearch.com
Price: dtSearch, US$199; dtSearch Text Retrieval Engine, US$999.

necessary to understand the requirements of the integration. Because of the pointer references used extensively throughout the functions and structures, the use of the engine will not be as straightforward as a component-based implementation. (Note: DT Software might welcome the development of a component wrapper for these functions.)

The documentation consists of two wire-bound manuals: one for the end-user aspects of the products, and a 168-page *Programmer's Reference*. The dtSearch manual tops out at 63 pages, covering everything from the installation of the product to the scanning of documents, and the addition of batch commands for indexing and searching. It would benefit greatly from a rewrite. The material doesn't seem to fit together logically, and I found myself skipping around the manual quite a bit. A number of topics could also benefit from some expansion; many of the items that I was able to locate had terse descriptions at best.

The *Programmer's Reference* is to the point and brief — almost too brief. It needs more examples in languages other than C/C++ to get the users of those languages off to a faster start. I wouldn't have been able to implement the product without the sample files, but this approach leaves a number of questions unanswered. Further study of the DTSearch.pas and experimentation would have made some of the functionality clearer, but is this a reasonable expectation of your tool users? Many programmers will purchase a package of components or tools hoping to have an immediate impact on their projects. Asking them to invest an inordinate amount of time in understanding your interface might be enough to rein in the popularity of a package.

Conclusion

The searching capabilities of the dtSearch product and, by extension, the dtSearch Text Retrieval Engine, are nothing short of astounding. Their search algorithms, in combination with a proprietary index format, allow colossal amounts of text to be managed quickly and efficiently. The DT Software Web site lists a number of commercial products that have integrated the engine into their code to the user's benefit. If your text management needs dictate a search product, dtSearch is an excellent choice, and the dtSearch Text Retrieval Engine is an equally excellent choice if your next project requires these functions. ∆

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.

Ready-to-Run Delphi 3.0 Algorithms

Many books aimed at programming using modern visual development tools have left the fundamentals of algorithm design and selection behind. New programmers enter the field with the impression that complete, commercial programs can be simply assembled from various components. While this might be true in some limited circumstances, it's hardly a realistic basis for large-scale development. Rod Stephens' *Ready-to-Run Delphi 3.0 Algorithms* comes to the rescue. This book is packed with 398 pages of fundamental and advanced information about the core of programming: algorithm design and data-structure construction.

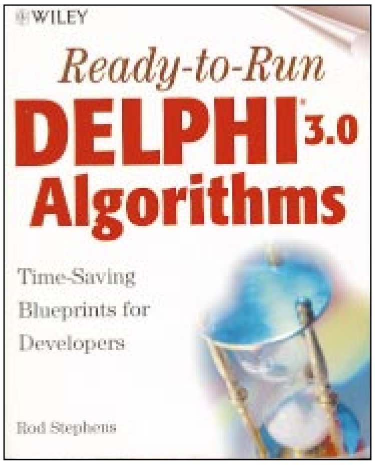
First, ignore the title! Unfortunately titled in 1998 with version number 3, *Ready-to-Run Delphi* is a book for the ages. Rod Stephens' work belongs on any programmer's shelf alongside that of Donald Knuth, Al Stevens, and Herb Schildt. As readers of this magazine are aware, Stephens has provided lucid and informative explanations of implementing fundamental and complex algorithms in magazine-length works. This book allows him to expand these thoughts and provides an invaluable learning and reference tool.

A publishing decision labeled this book with the 3.0 moniker in the age of version 4 and the upcoming version 5 of Delphi, but it makes little difference to the content. The only segment of the book that's affected by the version change is the five pages devoted to resizing arrays, a feature supported natively by version 4. The rest of the book is filled with the kind of information normally taught in computer science. This book has nothing to do with the interface or components of a program, and everything to do with the development of the fundamental building blocks that should be a part of every programmer's repertoire.

This isn't a book that needs to be approached chapter by chapter. After absorbing the algorithm analysis material in Chapter 1 and the list algorithms in Chapter 2, you can go where your interests lead you. Because the list data structure is fundamental to so many other, more complex structures in programming, it's important that you understand this material well. Chapter 2 also gives the reader a good introduction to the author's style and approach. In a concise 27 pages, Stephens provides an understandable and exceptionally readable presentation on the various forms of the list, from the simple array to a threaded, doubly linked list.

These topics sometimes occupy over 100 pages in a computer science textbook with loftier ambitions than simply informing the programmer and teaching the subject. However, the fast approach makes some important assumptions regarding pointers that the beginning or intermediate programmer might find troubling. When the caret suddenly appears in the code examples and the linked list structure is introduced with its next-item pointers, readers might find themselves in uncharted territory. Unfortunately, there is no safety net here; the author assumes you have knowledge on this subject or will gain it elsewhere.

Ready-to-Run Delphi provides thorough discussions of nearly every process and structure a programmer could ever need. Readers will encounter quick-but-complete discussions on the data structures that should be a



part of every programmer's kit. Chapters cover stacks and queues, arrays, trees, balanced trees, and decision trees. The various flavors of each are discussed and contrasted within each chapter. With study, you will easily grasp the pros and cons of each structure and be able to choose those appropriate for your next project.

Rather than a simple accessory, the CD-ROM included with this book is critical to the reader's understanding of the material. The code in the book, the focus of the material, is presented only in snippet form. To fully cement your understanding of the concepts, it's a good idea to load each applicable project into Delphi to place the snippets in context within the larger project. The projects themselves are excellent presentations. Stephens didn't create contrived examples. Rather, the sample programs are wholly focused on the concept being explained. He takes the examples one step further when presenting concepts that lend themselves well to comparison, such as sorting. One of the projects for the chapter discussing sorting includes a number of alternate sorting algorithms and approaches. When this program is run, the reader can see directly the comparative differences in sorting efficiency between the various designs. This approach makes the differences much easier to distinguish than individual demonstrations of the algorithms.

Unfortunately, the author occasionally succumbs to the temptation to produce a sample program that shows what can be done with the language, but doesn't lend itself to developing the readers' understanding of the topic. An example of this miscue is found in the chapter on recursion. Stephens selects the presentation of Hilbert and Sierpinski curves to discuss the use of recursion. While the purpose of including these demonstrations falls within the context of deciding whether recursion is appropriate, their innate complexity complicates the decision. *Ready-to-Run Delphi* matches the abundance of material dealing with data structures with a number of chapters discussing the various fundamental processes that a programmer will need. Topics include recursion, sorting, searching, and hashing. Each of the chapters is as successful as the structure chapters. The layout of the chapters follows the established approach of concisely explaining the numerous options available, when and where they're best used, and the pros and cons of each.

The book concludes with a solid presentation of object-oriented techniques that starts well, but doesn't pack the punch of the other material. The material is presented in a clear and straightforward manner, but tends to trail off just as it gets started. Stephens gives clear explanations of object-oriented vocabulary, such as encapsulation and polymorphism. What's lacking is an expansion of the ideas with examples that would really cement the concepts.
If you're a programmer who wants to go beyond the assembly of components and the setting of properties, *Ready-to-Run Delphi 3.0 Algorithms* will start you down the path to developing your core programming skills. Rod Stephens is an excellent writer with a to-the-point style that makes his work easy to read and understand. Long after you've read this work, you'll find yourself drawn to it when a sticky development situation rears its head.

— *Warren Rachele*

Ready-to-Run Delphi 3.0 Algorithms by Rod Stephens, John Wiley & Sons, Inc., 605 Third Ave., New York, NY 10158, (212) 850-6011, http://www.wiley.com.

ISBN: 0-471-25400-2
Price: US$49.99
398 Pages, CD-ROM

table of CONTENTS

An Interview with Eagle Software's Mark Miller

Mark Miller is President of Eagle Software (http://www.eagle-software.com), an independent producer of Delphi tools. In that role he has been the chief architect of its products, including the Component Development Kit (CDK), reAct, and CodeRush (formerly code-named Raptor). Mr Miller is also a frequent and popular speaker at international developer conferences and user groups.

DI: What are the advantages and disadvantages of being an independent developer?

Miller: The disadvantage is that your job security is directly proportional to the market. The advantage is that I am able to pursue my passion — developing tools — with intense focus and speed.

DI: What advice would you give the developer starting out who wants to develop tools for other programmers?

Miller: I assume you mean the developer wants to make a living at this, right?

DI: Yes, primarily developing tools as the main source of livelihood.

Miller: Pick a niche area that isn't covered well. You don't want to pour your heart and soul into a product only to find someone in another part of the world is giving away something that is essentially the same. We've already canceled two products that were in development because of this.

Be prepared for a gradual start. At least, that's how it was for us. When we placed our first ads for CDK, my estimates were really high. I was expecting 30 orders the next day (I think we got three). Although the CDK 1.0 was an excellent product, Delphi was new and most developers thought of component development as an advanced topic that was slightly out of reach. As time progressed and word spread about the CDK, our sales increased.

Make something that people will talk about — something really amazing or really useful. Then developers will talk about it, and help you get the word out about your product. Make a product that is of extremely high quality. Developers are finicky, and each developer has his or her own individual high standards of quality. Pleasing developers is much harder than pleasing general consumers.

Make a product that's simple and easy to use. Include as much assistance for the user wherever you can. At the translation company, the translation product we created had many complex features, and it was often in the hands of technically challenged managers. As a result, the average tech support call lasted about 15-20 minutes, with a few running several hours. This was extremely expensive (of course, so was our product so everything evened out). When creating the CDK, I swore this would not be the case.

To reduce support calls, I added a feature called "Mr. CDK", which is essentially an agent that watches activity in the CDK, and alerts users of any potential problems. Although the Mr. CDK agent was not animated (he was a static cartoon drawing of a developer), he was in a shipping product at least a year before Microsoft put the infamous paper clip in Office. On every page of the CDK we added "I need more help" buttons, and the CDK was complemented with an extensive online help file and manual, that covered not only the basics of using the CDK, but took both novice and expert developers through the several advanced component building issues. As a result of all of this work, tech support on the CDK was, and continues to be, extremely low. We consider all of this effort a success.

DI: Delphi is clearly at the center of your programming universe. What do you consider its most important strength? And without giving away any trade secrets about Delphi tools you might be planning, what is the one area of Delphi you feel needs to be improved the most?

Miller: Delphi's greatest strength is its RTTI, its component architecture (especially *TComponent* and *TForm*), and the fact that Delphi is built with Delphi. All of these things combine to allow us to get under the sheets with Delphi, enhancing it in just about any way we see fit. It's impossible to get the same kind of integration with Visual C++ or any other IDE.

As far as improvement, most are things that we can do. However, there are a few things that the Borland R&D team could do in improving the Tools API that would make some amazing things possible. Additionally, I would like them to add the concept of an "AutoVar" to the language. Any variable declared in the "AutoVar" section of a method is automatically created and destroyed (in appropriate try/finally blocks) within the scope of the method. The compiler would invisibly implement the creation and try/finally/free code. I would also like to see packages enhanced to allow the concept of interface and implementation applied to their list of contained units. This could solve a lot of distribution problems associated with packages. Other than that, I would like to see a version of Delphi that supported Linux.

DI: Often I see discussions on the Internet about Inprise in general and Delphi, focusing on the direction in which this company is moving and the implications for its flagship product. Could you share some of your observations and views? What will the future hold for Delphi?

Miller: I think the Inprise/Borland.com split is the best decision they've made in years. At every Borland conference I've been to, the keynotes heavily focused on what Borland was doing for the enterprise. As a developer, I felt they were completely missing the point; I don't care about those things. I just care about how productive I can be in their environment. I think that's what other developers care about as well, even developers working at the enterprise level. Everybody wants to reduce expense and time to market.

As far as the future of Delphi is concerned, that's hard to tell right now. For Delphi to succeed and survive in the long term (10+ years), it has to continue to be significantly better than the alternatives. When I say "better" I mean that teams using it must consistently deliver products faster, on spec, at a lower cost, and with lower long-term maintenance costs than teams using Visual Basic, C++, or Java. Right now that's the case, and management is realizing that, but for Delphi to survive this must continue to be the case. Additionally, either Delphi must evolve to support other platforms, or Windows must continue to be the dominant platform.

DI: Windows is very popular right now as a computing environment, particularly in the home computing market. Do you see any indications this will change?

Miller: I see some small indications in the form of Linux and Java, but I don't believe we've really seen Microsoft's full response yet. You have to remember that Microsoft is a company with a lot of cash, employing a large number of very smart people. With those resources, they should be able to continue to dominate the market for quite some time — seven to 10 years at least.

DI: What do you feel was your biggest challenge as a developer?

Miller: Getting CodeRush 4 to do what is does inside Delphi 4 is pretty challenging. It's like writing an application where you don't have half the code. If there's a bug in the half you don't have, you have to somehow fix it or implement a work-around in the other half.

Delphi 4 introduced a few architectural changes that presented major roadblocks into porting CodeRush 3 technology across. For one of these issues (dockable forms), we had to create a descendant to a class that we had no source code to, and we had to do this at run time. Just to be clear, I'm not talking about creating an object dynamically; I'm talking about creating a new *class* dynamically. We had to override methods in this new class, and then pass an instance of it to a registration procedure located somewhere in a Delphi unit to which we did not have the source. Finding the solution involved a great deal of failed attempts and dead ends before we ultimately got it right, but the result is pretty impressive. Users can now create true Delphi dockable forms containing whatever they want in a matter of minutes. Delphi believes the forms are its own, so persistence and streaming to the desktop file are built in.

DI: What's the most exciting project you are working on right now?

Miller: Without a doubt it's CodeRush. CodeRush represents my vision of the future of programming. It's what programming environments will be like for everyone four to five years from now. I'm extremely excited about high-speed coding, and I know that Delphi plus the appropriate third-party tools is unbeatable in terms of productivity. Nothing else comes half as close.

I'm so confident of this that I openly challenge any team of up to 10 VB, C++, or Java programmers against a team of only three Delphi programmers in a high-speed programming duel to the death. I figure we could do one of those 24-hour caged events on pay-per-view, where four teams walk in and only one walks out. Winners would be judged on architecture, execution speed, user interface design, and projected ease of maintenance. Even outnumbered, we'd stomp all over the other teams so much it would be embarrassing. (Tissues would be freely available to all non-Delphi programmers in attendance who will no doubt be crying at the end.) It would definitely be an eye-opener for anyone who's never seen high-speed programming in action. I've actually tried to get the Software Development conference folks to allow Delphi into their annual C++ Super Bowl competition, but they've denied my requests twice.

DI: Having seen you in action, doing live coding in your presentations at conferences, I have no doubt you'd be hard to beat. Mark, on behalf of *Delphi Informant*, its readers, and the Delphi community, thank you for agreeing to do this, and for your thoughtful and candid responses. ∆

— Alan C. Moore, Ph.D.

Note: This is an abridged version (approximately half) of the interview. The complete interview is available at the Informant Web site: http://www.informant.com.

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.

table of CONTENTS