August 1998, Volume 4, Number 8

Cover Art By: *Tom McKeith and Doug Smith*

ON THE COVER

FEATURES

REVIEWS

DEPARTMENTS

By Zack Urlocker

Life after Borland; Long Live Delphi!

It's been a busy, but exciting time in Scotts Valley the past several months. The company has continued to grow revenues and maintain profitability, we're shipping new products every quarter, oh, and we changed our name. I'm sure there was a collective "What?" heard when Delphi developers around the world fired up their browsers to www.borland.com and saw an announcement about INPRISE Corporation.

The Borland brand name is a good one. It's associated with high quality, high productivity development tools such as Delphi, C++Builder, and JBuilder. However, development is just one part of what INPRISE Corp. is focused on. Our mission is to radically simplify the development, deployment, and management of distributed enterprise applications. That makes development tools, including Delphi, critical to our strategy, but it also reinforces the need for additional products that support deployment and management of enterprise applications. For example, we have the Entera RPC-based middleware, the VisiBroker CORBA Object Request Broker, ITS Integrated Transaction Service, AppCenter application management, and a forthcoming Enterprise Application Server. We will make sure that these products are accessible from our development environments, including Delphi, C++Builder, and JBuilder.

The Delphi team has been particularly innovative in its support for COM, CORBA, and RPC-based middleware. Delphi 4 Client/Server Suite has all of the facilities to make it the most productive development environment for any distributed infrastructure. Because INPRISE doesn't have a platform agenda, we'll continue to support all the platforms our customers are asking for. That means you can expect ongoing support for the latest Microsoft technologies, such as Windows 98, MTS, IIS, as well as the latest cross-platform standards.

In fact, I think people will be surprised at how much innovation there is in Delphi 4. Delphi will continue growing and attracting even more new customers who want to develop distributed applications, as well as desktop and client/server applications. Certainly, no tools have made distributed development easy before, and we think Delphi's got an edge in this area that's hard to beat. At the same time, we've made sure to add some of the latest productivity features to Delphi 4, including more debugging tools, a more modern user interface with dockable toolbars and windows, a sophisticated application browser, more components, and much improved documentation, among other things.

Ongoing change. One of the things I've learned in my years at Borland is that you must keep innovating, or you'll fall by the wayside. It's as true in programming as it is in business. If you look back at programming techniques and tools five years ago and compare them with the state of the art today, it's a pretty radical change. Not only have the tools and techniques changed, but so have the applications being built. Five years ago, when the Delphi project first began, we were attempting to bring Borland Pascal into a new era of growth by solving much more difficult problems. We decided (though I had to drag the R&D team kicking and screaming) that we needed to simplify both Windows and client/server programming. It was quite a gamble, but it helped Borland launch into some new growing markets with our most successful product in years. Along the way, well beyond a million copies of Delphi were sold, and hundreds of thousands of programmers made the transition to learning client/server programming skills and became more successful in their careers. Delphi has certainly exceeded our expectations, and the Delphi team continues to amaze me with their dedication and insight into programming.

At the time, there was a lot of debate about what Delphi should be called. Many of you know the name Delphi was originally chosen as a code name. In fact, we used a lot of different code names, depending on who we were talking to. Delphi was variously known as VIP, Visual Foo, Del Mar, Del Rio, Del Fuego, VBK, and Wasabi (Don't ask!). Internally, the compiler was "Version 8" but we wanted a name that showed just how far the product had evolved from Borland Pascal. We struggled with such convoluted names as Visual App Builder, but in the end, we decided to go with the name that our beta testers liked best: Delphi.

So what's it all mean? For Delphi developers, the company name change should be pretty minor. We'll continue to create new versions of our development tools and try to have innovative features for hobbyists, individual developers, consultants, and corporate developers. The most significant aspect of the name change is that it symbolizes how much the company has changed over the last 18 months with a broader focus of integrating the enterprise. We believe the name INPRISE better reflects our overall strategy and direction than the Borland name did. That being said, the Borland brand name will continue to be used for our development tools under the INPRISE Corp. umbrella. This is similar to what other large companies have done when bringing to market multiple different product lines. For example, Network Associates has the McAfee brand of antivirus software as one of its brand names.

Delphi remains the product that I'm most proud of in my career, and you can expect Delphi will continue to play a pivotal role in our overall corporate strategy. After eight years of working at Borland International, I'm pleased to report that there is life after Borland. There's INPRISE.

Zack Urlocker is Vice President of Marketing at INPRISE Corp., developer of Borland and VisiBroker products.

table of CONTENTS

Delphi TOOLS

New Products and Solutions

ForeFront Announces ForeHelp 3 and ForeHelp Premier 98

ForeFront, Inc. announced *ForeHelp 3* and *ForeHelp Premier 98*, updated releases of the company's help-authoring program and suite of help-authoring tools. The products offer an indexing application, a development tool that keeps dialog-box help synchronized with software applications, and a content management system that allows developers to create help for Windows 95 and Windows 98 from one project database.

ForeHelp 3 offers new project management and automation features, including Topic Navigator, which provides a customizable navigation window; Reporter, which produces customized management reports; Paste Palette, which stores frequently used graphics, text, and buttons on a floating palette; a Related Topics Wizard, which creates "See Also" buttons; and a Contents Editor, which visually creates and edits a table of contents.

ForeHelp Premier 98 packages ForeFront's line of help-authoring tools into a suite for WinHelp and HTML-based help development. In addition to ForeHelp 3, three tools are introduced: Index Expert, QuickFix, and QuickContext.

ForeFront, Inc.
Price: ForeHelp 3, US$395; ForeHelp Premier 98, US$699; users of competing help-authoring suites can purchase ForeHelp Premier 98 for US$249.
Phone: (800) 357-8507
Web Site: http://www.ff.com

Catalyst Releases SocketTools 2.1

Catalyst Development Corp., provider of WinSock-compliant TCP/IP custom controls for VBX/ActiveX products, announced *SocketTools 2.1*.

SocketTools 2.1 provides 18 controls, including 32-bit ActiveX versions (supported by Delphi 2 or later). SocketTools 2.1 supports all Windows platforms, including Windows NT 4.0.

SocketTools 2.1 components include Audio Player, Domain Name Service, File Transfer Protocol, Finger Protocol, Gopher Protocol, Image Viewer, Internet Control Message Protocol, Mail Message (MIME), Network News Protocol, Post Office Protocol, Remote Access Service, Remote Command Execution, Simple Mail Transfer Protocol, Telnet Network Terminal, Terminal Emulator, Time Protocol, Web Browser, Whois Protocol, and Windows Sockets.

Catalyst Development Corp.
Price: US$247
Phone: (800) 766-3818
Web Site: http://www.catalyst.com

Enterprise ONE Releases Jaadu

Enterprise ONE, Inc. announced *Jaadu*, a Delphi component that adds Web server functionality to an application, using Delphi Web components to extend the functionality.

By employing Jaadu, developers can fully exploit built-in Delphi Web technologies, such as the WebDispatcher, PageProducer, TableProducer, and QueryTableProducer components.

The Jaadu package includes the JaaduProducer, a component that takes an HTML template and a few data sources, puts them together, and serves it to the user.

Jaadu is a solution for Delphi 3 developers who require rapid implementation of Web applications without dealing with the limitations of CGI-shell, Win-CGI, ISAPI, and/or NSAPI programming.

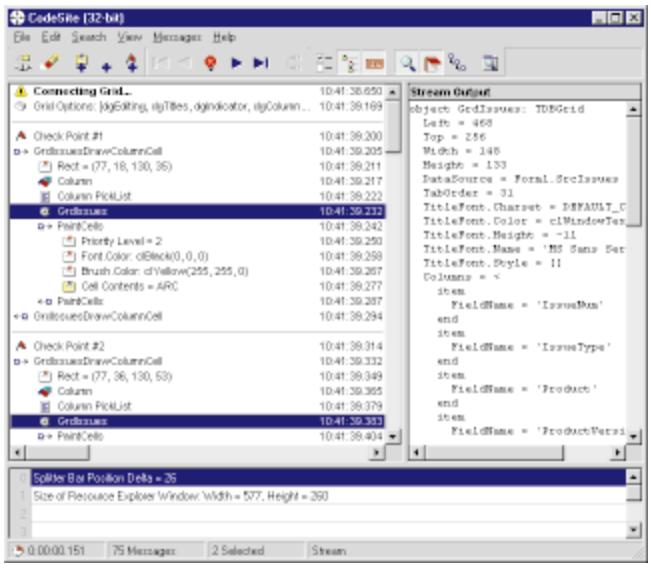
In addition, the Jaadu server component works as a multi-threaded HTTP server and programming gateway.

Enterprise ONE, Inc.
Price: US$199
Phone: (916) 947-9012
Web Site: http://www.jaadu.com

Delphi TOOLS

New Products and Solutions

Raize Software Solutions Releases CodeSite



Raize Software Solutions, Inc. announced *CodeSite*, a Delphi debugging tool that sends messages from an application to a message viewer.

CodeSite is comprised of the CodeSite Object and the CodeSite Viewer. The CodeSite Object is accessed through an interface unit, and is used to send messages to the viewer. The messages can contain objects, streams, string lists, and standard data types, such as integers and Boolean values. CodeSite also supports common data structures, such as *TColor* and *TRect*.

Each CodeSite message is associated with one of 15 types, including csmInfo, csmCheckPoint, and csmObject. In addition, each CodeSite message is time-stamped.

CodeSite offers the ability to group messages by method. Method groupings can be nested, making them effective for debugging event-driven applications and recursive methods, functions, and procedures.

Also, CodeSite provides features for CodeRush users; 25 keyboard templates and a message expert make it easier to enter CodeSite statements into source code. Also, the CodeSite Viewer integrates into the Delphi 3 IDE as a CodeRush panel plug-in.

CodeSite supports 16- and 32-bit Delphi development. CodeSite Viewers are provided for Windows 3.x and Windows 95/NT.

Raize Software Solutions, Inc.
Price: US$79.95
Phone: (630) 717-7217
Web Site: http://www.raize.com

Lingscape Announces MultLang Suite 2

Lingscape announced *MultLang Suite 2*, a set of globalization tools for Delphi and C++. MultLang is based on the Unicode 2.1 standard and works with a conversion engine to provide support for Japanese, Chinese, Arabic, Hebrew, Hungarian, Russian, and all European languages.

MultLang Suite 2 provides a quality assurance wizard that keeps projects from showing defects in user interfaces, such as overlapping text, workable shortcuts, incorrect character sets, and non-translated items. The wizard ensures that projects conform to standards defined by the developer.

MultLang Suite 2 extends the NLS API functions of Windows with support for distributed locales, making it possible to produce multi-language applications supporting several character sets on any Windows localized edition.

In addition, MultLang Suite 2 offers the Universal Language Modules (ULM), a self-describing dictionary with its own interface.

Lingscape
Price: US$998; upgrade from MultLang 1.x, US$849.
E-Mail: info@lingscape.com
Web Site: http://www.lingscape.com

South Pacific Info Services Announces TWebCompress Component 2.0

South Pacific Information Services, Ltd. announced *TWebCompress Component 2.0*, a site management component for Delphi and WebHub. TWebCompress has been rewritten to be a generic Delphi component, so it can be used for remote site/compression management in any Web site application. Source code for a WebDeploy application, which turns TWebCompress 2.0 into a stand-alone application on a Web server, is provided for managing compression/archive deploying tasks for multiple sites. A *UserSignal* property and event hook have been added to allow developer-defined remote signaling. In the WebDeploy application, this is implemented as a way to have the server run a defined application on the .EXE under remote control. A "refresh" signal is also defined, forcing TWebCompress to re-read its .INI file.

TWebCompress 2.0 is configurable (by site) for location of uploads, signal/report files, "gather" archive, encryption passwords, etc. This configuration can be changed from HTML, FTP control files, or via the panel user interface. The *Gather* function can be completely data-driven and can be used to build ad hoc LHA/WinZip-compatible archives of particular files or sets of files.

South Pacific Information Services, Ltd.
Price: Free for download; licensing conditions apply.
Web Site: http://www.spis.co.nz

News LINE

August 1998

INPRISE Releases Borland JBuilder 2

Scotts Valley, CA — INPRISE Corp. announced the release of Borland JBuilder 2, a new version of its family of visual development tools for creating pure Java business and database applications for the enterprise.

JBuilder 2 allows corporations to use 100% Pure Java technologies, including JDK 1.1.6 and JFC/Swing, to create platform-independent business applications.

The Borland JBuilder product family features JavaBean component creation, a scalable database architecture, visual "Two-Way" development tools, and the ability to produce 100% Pure Java applications, applets, servlets, and JavaBeans.

Corporations and government agencies worldwide using JBuilder include Volvo, Daiwa Securities, MicroAge, NationsBank, the Ohio Department of Transportation, Eli Lilly, Nationwide Insurance, Mercedes-Benz, and many others.

JBuilder 2 is available in three versions: JBuilder 2 Client/Server Suite, JBuilder 2 Professional, and JBuilder 2 Standard.

For more information about JBuilder 2 call INPRISE at (800) 233-2444, or visit the JBuilder Web site at http://www.inprise.com/jbuilder/.

Delphi Prefix Registry Online

Manchester, England — Steven J. Healey, an applications developer specializing in manufacturing systems, announced the Delphi Prefix Registry (DPxR), hosted by HREF. DPxR, located at http://developers.href.com/registry/dpr.htm, promotes component builders globally by registering not only prefixes, but also the components themselves. Developers can register their components, as well as find other developers' components. The registry aims to reduce the chances of naming conflicts with the use of prefixes. Conflicts can occur when developers build components in isolation.

For developers trying to locate components, the registry provides an alternative to checking Delphi newsgroups, using Web search engines, and visiting the Delphi Super Page. The site also contains links to major component builders.

Supporters of the registry and/or the use of prefixes include Ray Konopka, Bob Swart, and Marco Cantù.

Apogee Extends Services to Include AS/400

Marlboro, MA — Apogee Information Systems, Inc. announced support for Borland Delphi/400 Client/Server Suite.

The move extends the company's ability to deliver multi-tier, enterprise applications for corporations and governments by allowing the company to include native access to AS/400 databases in custom applications developed for its clients.

INPRISE's Borland Delphi/400 Client/Server Suite is a visual, high-performance Windows development tool that features visual component-based design, a 32-bit, native code compiler, and an open, scalable database architecture in an object-oriented environment.

Apogee Information Systems, Inc. is a custom application consulting and development firm specializing in INPRISE technology-based solutions. For more information on Apogee, visit http://www.apogeeis.com.

InterBase Ports InterBase 4.0 to Linux

Scotts Valley, CA — InterBase Software Corp. announced the availability of InterBase 4.0 for the Linux operating system (Red Hat 4.2) as a free download on the InterBase Web site, located at http://www.interbase.com.

InterBase 4.0, a relational database, is designed to be embedded into VAR applications.

Current InterBase customers include the United States Army, Motorola, Inc., the Philadelphia Stock Exchange, Colorado Mountain Express, and others.

InterBase 5, which is available for Windows 95/NT, Solaris, and HP-UX, will be available for Linux (Red Hat 5.0) in the third quarter, 1998.

InterBase 5 for Linux is planned to offer new SQL and server 1999 features that will give InterBase scalability, concurrency, and improved productivity, including InterClient, an all-Java JDBC driver.

Users will be able to purchase InterBase 5 for Linux via download at the InterBase Web site after its release.

Triple Point Technology Uses Delphi to Develop OutPost

Westport, CT — Triple Point Technology, Inc. introduced OutPost, a Web-enabled storage and retrieval system for users of Triple Point's TEMPEST 2000 and FRANKLIN commodity trading systems. OutPost enables traders, trading managers, and executive management to access risk management data and trading results remotely, via an intranet or the Internet.

OutPost has a three-tier client/server architecture. The client software's graphical user interface components are ActiveX controls developed using Delphi 3. The middleware software was also developed with Delphi 3 and utilizes MIDAS middleware technology.

TEMPEST 2000 is an office trading system for the physical and derivative energy commodity markets. FRANKLIN is a comprehensive commodity trading system for the electricity and natural gas markets. Using OutPost, individuals can generate a report with TEMPEST 2000 or FRANKLIN, and post it in the OutPost database. Once in the database, authorized users can access reports for previewing or printing using Microsoft Internet Explorer or Netscape Navigator browsers.

For more information, contact Triple Point Technology at (203) 291-7979, or visit their Web site at http://www.tpt.com.

INPRISE Teams with Referentia Systems for Java System

San Francisco, CA — INPRISE Corp. announced it has teamed with Referentia Systems Inc. to create Referentia for JBuilder: Volume 1, an integrated, extensible, multimedia learning system for any software development tool. Referentia for JBuilder is a separate product from JBuilder.

The system provides tips, tricks, and training using narrated lesson animations to teach JBuilder techniques. It launches from JBuilder's Help menu and includes a Try It feature, which allows users to work in JBuilder while following textual and/or narrated instructions. The system's extensibility allows users to add future volumes of content to their library of built-in JBuilder tutorials.

Referentia for JBuilder: Volume I includes tutorials authored by recognized industry experts.

For more information, contact Referentia Systems at (888) 983-6877 or (808) 396-3319.

INPRISE Announces Entera 4

Scotts Valley, CA — INPRISE Corp. announced the shipment of Entera 4, a new version of the company's RPC-based middleware for corporations building large-scale, mission-critical, distributed applications. Entera 4 adds support for Java and Delphi, and integrates and automates application management and security services.

Entera 4 supports the IBM AIX, Sun Solaris, HP-UX, and Windows NT platforms. More information on Entera 4 is available at http://www.inprise.com/entera/.

INPRISE Appoints John A. Racioppi as VP/GM

Scotts Valley, CA — INPRISE Corp. named John A. Racioppi Vice President and General Manager, US, a newly created position. Racioppi has over 17 years of sales and management experience, and will report to John Floisand, Senior Vice President of Worldwide Sales. Racioppi is responsible for INPRISE's US direct sales, channel sales, educational sales, and OEM sales activities, as well as customer service organization.

Racioppi joins the INPRISE team from DASCOM Corp., a developer of security, authorization, and scalability tools for intranet and extranet applications, where he was Vice President of Worldwide Sales. Prior to DASCOM, Racioppi spent five years with Transarc Corp., an IBM subsidiary and developer of distributed transaction processing and file systems middleware, where he was responsible for the company's international sales and OEM source licensing business units.

Racioppi's experience also includes sales and management positions with Network Systems Integrators, Liberty Management Systems, Ernst & Winney, and AT&T Information Systems.

He holds a Masters degree in business administration and a Bachelor of Arts degree from the University of Pittsburgh.

table of CONTENTS

By Bill Todd

Delphi 4 Multi-Tier

MIDAS Reaches New Levels

Using MIDAS components, Delphi 4 provides a host of new features for both multi-tier and single-tier applications. It all begins with *TClientDataSet*; you can now define aggregates for a ClientDataSet component. Figure 1 shows the Multitier page of the Delphi 4 New Items dialog box.

As you already know if you're familiar with SQL, an aggregate is the sum, minimum, maximum, average, or count of a column in an answer set. In MIDAS, aggregates are stored in the *Aggregates* property of *TClientDataSet*. Aggregate expressions can include calculations such as:

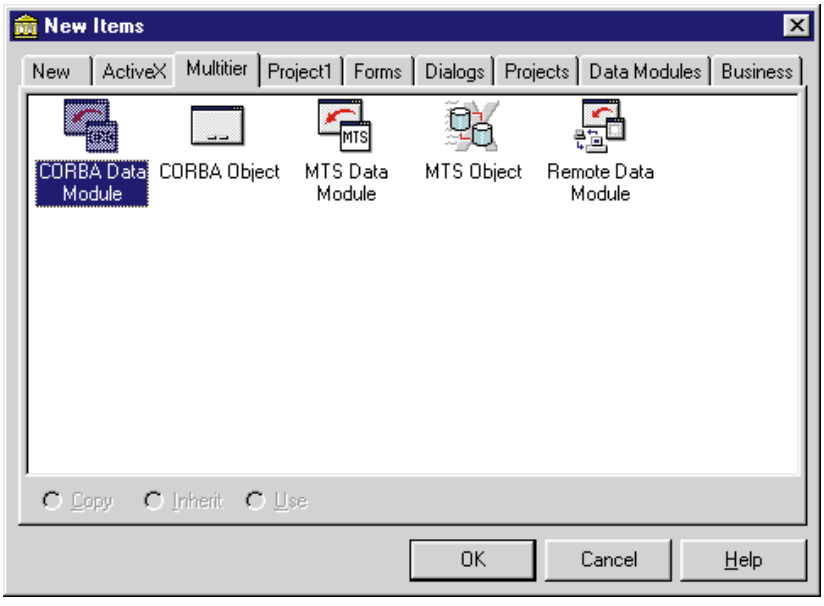


Figure 1: The Multitier page of the Delphi 4 New Items dialog box.

```
Sum(Qty * Price)

or

Sum(TotalPrice) - Sum(TotalCost)
```

Because the aggregate object's expression is a property, you can change it at run time. You can also enable or disable any individual aggregate, or all aggregates, to enhance performance. A property editor for the new *Aggregates* property of *TClientDataSet*, shown in Figure 2, lets you quickly create aggregates and set their properties at design time.

Figure 3 shows the Object Inspector with an aggregate selected. Aggregates can also be grouped on the first *N* fields of the active index. To group an aggregate, set the *GroupingLevel* property to the number of index fields to group on, and set *IndexName* to the index to use. Aggregates that group on an index that isn't active are not calculated.

Like all other datasets, *TClientDataSet* has a *Refresh* method. Calling *Refresh*, however, clears the change log so all unapplied changes are lost. The new *RefreshRecords*

method fetches a new set of records from the application server and merges them with the changes in the change log as though the changes were originally made to the new records. If any conflicts occur, the ClientDataSet triggers an *OnReconcileError* event.

Controlling Data Packet Contents

You now have three ways to control the contents of data packets retrieved from the application server — including the ability to include custom information. Using the Fields Editor to create persistent field objects for the dataset on the application server lets you control which fields are supplied to the client. Lookup and calculated fields will be returned to the client as read-only fields. The only restriction is that you must include the primary key field(s) so each record can be uniquely identified if the client will edit the data.

TProvider has a new *Options* property, shown in Figure 4, that lets you control whether BLObs and detail records will be fetched automatically, or must be specifically requested in code. You can also specify whether field properties, such as *Alignment*, *DisplayLabel*, *DisplayWidth*, *Visible*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, *Currency*, and *EditMask* are included in the data packet.

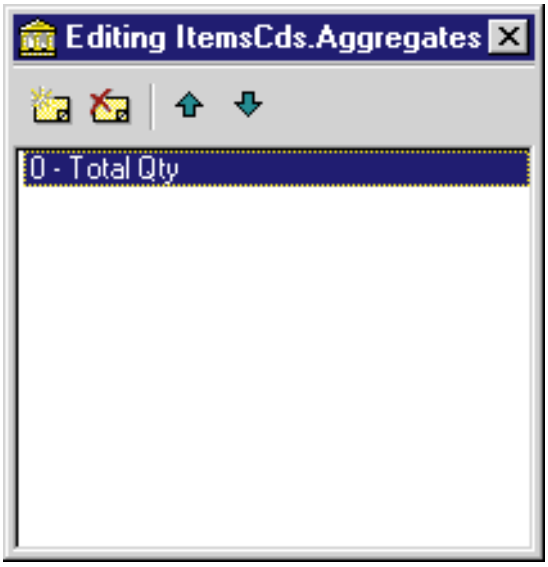


Figure 2: Aggregates in the Property Editor.

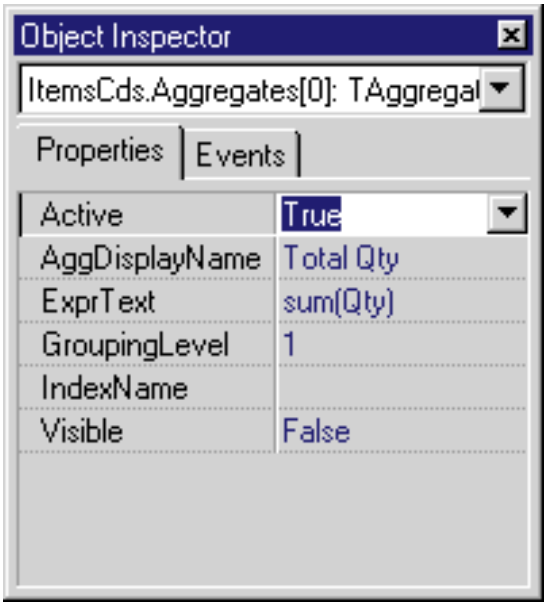


Figure 3: The properties of an aggregate.

If you're working with tables that have a referential integrity relationship, you can specify that the server automatically handle cascading updates and/or deletes. The Provider component's *Options* property also lets you declare the dataset to be read-only. This prevents the ClientDataSet from applying updates to the Provider. If you are working with a server that may modify records with triggers when they are updated, you can include the *poIncServerUpdates* option. This tells the Provider that it must automatically retransmit all updated records to the client, so the client will immediately see the final state of the records — including the effects of any triggers.

One of the more potentially powerful new multi-tier features is the ability to include custom information in the data and delta packets passed between the application server and the client. This capability is implemented through a new event, *OnGetDataSetProperties*, on the Provider side. *OnGetDataSetProperties* passes three parameters. The first is *Sender*, a pointer to the object that triggered the event. The second, *DataSet*, is a pointer to the dataset that supplies the Provider's data.

The last, named *Properties*, is an *OleVariant* output parameter. *Properties* is actually a variant array of three-element variant

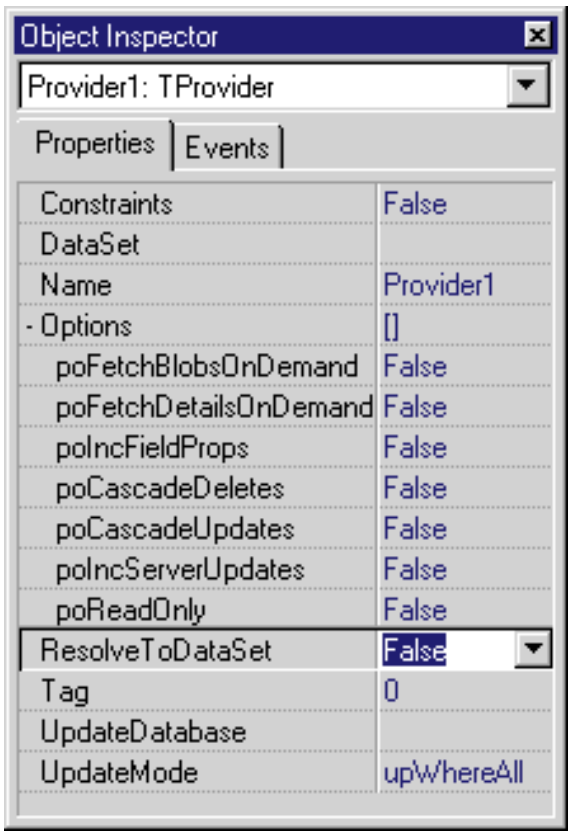
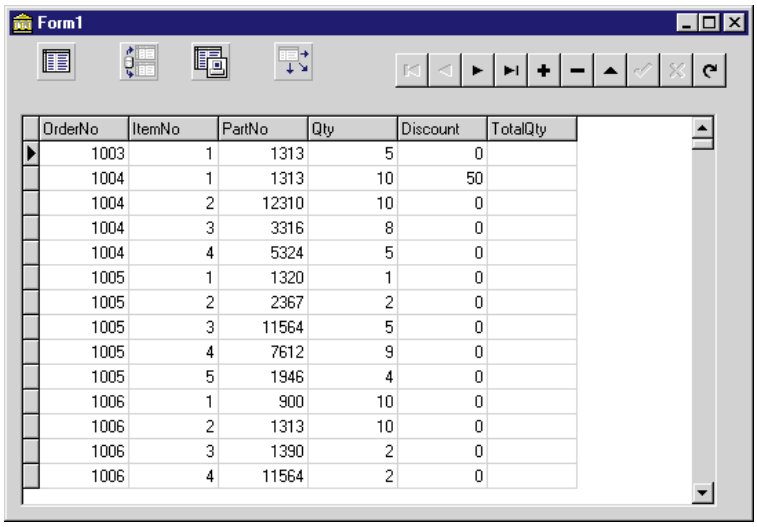


Figure 4: The *TProvider Options* property in the Object Inspector.



OrderNo	ItemNo	PartNo	Qty	Discount	TotalQty
1003	1	1313	5	0	
1004	1	1313	10	50	
1004	2	12310	10	0	
1004	3	3316	8	0	
1004	4	5324	5	0	
1005	1	1320	1	0	
1005	2	2367	2	0	
1005	3	11564	5	0	
1005	4	7612	9	0	
1005	5	1946	4	0	
1006	1	900	10	0	
1006	2	1313	10	0	
1006	3	1390	2	0	
1006	4	11564	2	0	

Figure 5: An example local table application using *TClientDataSet*.

arrays. Each element in the array contains one custom property that will be included in the data packet the Provider sends to the client. Each element in the first array is itself a variant array of three elements. The first element is the name of the property stored as a string. The second element is the value of the property. The third element is a Boolean value that tells the ClientDataSet whether this property must be returned to the Provider in the delta packet. This makes it possible for the Provider to not only send custom information to the ClientDataSet, but to also send a message to itself that will come back with any changes.

On the ClientDataSet side, the new *GetOptionalParam* method lets you retrieve the value of any optional parameter from the dataset. If you're working with a briefcase model, or a single-tier application that saves the contents of the ClientDataSet's *Data* property locally using the *SaveToFile* method, you can still take advantage of optional data packet parameters. Using the ClientDataSet's *SetOptionalParam* method, you can add custom parameters to *Data* in the client application and these parameters will be saved to disk when you call *SaveToFile*. This gives you a very flexible way to store any information that must persist between sessions. Adding parameters in the client application is useful in another way. By setting the *IncludeInDelta* member of the variant array to True, the client can send information to the application server with any updates.

The Events Are Back

One complaint of many developers using the Delphi 3 MIDAS components is that the events for the dataset components connected to *TProvider* do not fire when updates are applied. Now, setting *TProvider*'s new *ResolveToDataSet* property to True causes the Provider component to apply updates using the dataset component so that all of the dataset's events work normally. Leaving *ResolveToDataSet* set to False causes the provider to apply updates directly to the database using SQL, just as in Delphi 3.

Another new component, *TDataSetProvider*, lets you accomplish the same thing. Like *TProvider* with *ResolveToDataSet* set to True, *TDataSetProvider* applies updates using the dataset component it's connected to. However, *TDataSetProvider* is independent of the BDE, so it can apply updates to a *TClientDataSet* or a custom dataset component that does not use the BDE to connect to its underlying database.

Applying updates to BDE datasets is slower than using *TProvider* with *ResolveToDataSet* set to False, because the updates must first be processed by the dataset component and then written to the database. With *ResolveToDataSet* set to False, the updates are applied directly to the database using SQL. However, the ability to use the events of the dataset may be worth the price in many situations.

Building One- and Two-tier Applications

Building single-tier applications with *TClientDataSet* was a side effect of support for briefcase model applications in Delphi 3. It was interesting because there are still plenty of applications that don't require multi-user support, and do not deal with huge amounts of data. So the ability to create small, fast database applications that do not require the BDE was attractive. Creating these databases was laborious, however, because you could not define the database structure at design time. Creating the database had to be done with code. In Delphi 4, you can define your database at design time using the Fields Editor, and you can store multiple related tables in a single file, thanks to the support for Oracle 8's object relational database model.

Drop a *TClientDataSet* on a form or data module, double-click to open the Fields Editor, press Ctrl N to add a new field, and you can add as many fields of any type as you need. Now when you run your application, the field objects are already defined — without code. Even more powerful is the ability to add fields of type DataSet. These fields represent a linked detail dataset. You can define as many linked datasets as you need; they will be stored in a single file as part of the *Data* property of the ClientDataSet component when you call *SaveToFile*.

Using *TClientDataSet* in two-tier applications is now easier too. You can drop any dataset component, a Provider component, and a ClientDataSet component on a form or data module; set the Provider's *DataSet* property and the ClientDataSet's *Provider* property, and your application is ready to run. While some of this functionality was added in the 3.02 update, you now have complete freedom to place the dataset, Provider, and ClientDataSet components on forms or data modules — or a mixture of the two — and connect them at design time.

Figure 5 shows the main form from the sample application, TwoTier, containing Table, Provider, and ClientDataSet components linked to the sample Items table. The *ResolveToDataSet* property of *TProvider* is set to True so updates will be applied through the

Table component. The Table's *BeforePost* event handler generates a beep and calls *Abort* to prevent the update from taking place, demonstrating that *TTable*'s events now fire.

Working with Parameters

Setting the parameters for a query or a stored procedure on the application server is easier in Delphi 4, thanks to *TClientDataSet*'s new *Params* property. Using this property, you can define parameters at design time and set their values, or other properties, in the Object Inspector. You can create parameters at run time with the *CreateParams* method. Each parameter is an object, so you can change its *Value*, *ParamType*, *Name*, and *DataType* properties programmatically at any time.

Calling Methods on the Application Server

In Delphi 3 you can add custom methods to the server application's interface and call those methods from the client using the *AppServer* property of the client application's connection component. The *AppServer* property is a handle to the *IDataBroker* descendant interface added to the server's remote data module. Calls to custom methods used late binding which is slower and does not provide compile-time error checking.

In Delphi 4, you can get early binding of calls to the application server using DCOM or CORBA. To get early binding, you must cast the *AppServer* property to the interface type of the application server, and you must register the application server's type library on the client. The following call illustrates the early binding syntax:

```
TheConnection.(
  AppServer as IMyInterface).CustomMethod(aValue);
```

Although you cannot use true early binding with sockets, or OLE Enterprise, you can improve performance over late binding by using the remote data module's dispatch interface (dispinterface) to call server methods. The server's dispinterface always has the same name as the *IDataBroker* descendant interface with the string "Disp" appended. To use this method, you must add the application server's type library interface unit to the **uses** clause in the client application. You can then assign the *AppServer* property to a variable whose type is the dispinterface, and use that variable to call the server's methods, as shown here:

```
var
  dispinterface: IMyAppServerDisp;
begin
  dispinterface := TheConnection.AppServer;
  dispinterface.CustomMethod(aValue);
end;
```

New Connection Components and Features

Delphi 4 brings a more sensible architecture to the MIDAS connection components by providing a specific component for each connection type that includes only the properties applicable to the connection type. The classes for the new components are:

- *TDCOMConnection*,
- *TCORBAConnection*,
- *TSocketConnection*, and
- *TOLEEnterpriseConnection*

Both the DCOMConnection and SocketConnection components also have a new *ObjectBroker* property, which is a pointer to a descendant of the new *TCustomObjectBroker* class. This allows you to implement a custom object broker that will determine the machine the client will connect to. The new *TSimpleObjectBroker* component lets you implement automatic selection of the server the client will connect to from a list of servers using a random assignment system to provide load balancing among the servers.

A socket server service for Windows NT has also been added. With Delphi 4, it's unnecessary to have a user logged on to the NT machine to run SCKTSRVR.EXE; it runs as a service, so it starts automatically and does not require a user to be logged in. With the new Service wizard (see Figure 6) you can now also write your server application as a Windows NT service. This means you can run an application server using sockets on an NT machine, with no user logged in, to provide a much higher level of security.

Another significant addition to the sockets components is support for callbacks. While any client has been able to call custom methods on the application server in Delphi 3, you could only implement callbacks from the server to the client if you used DCOM or OLE Enterprise for the connection protocol. Delphi 4 supports callbacks using sockets in exactly the same way they are implemented using OLE Enterprise or DCOM.

Conclusion

Delphi 4 not only adds many new features for multi-tier applications, but also adds features that make using the

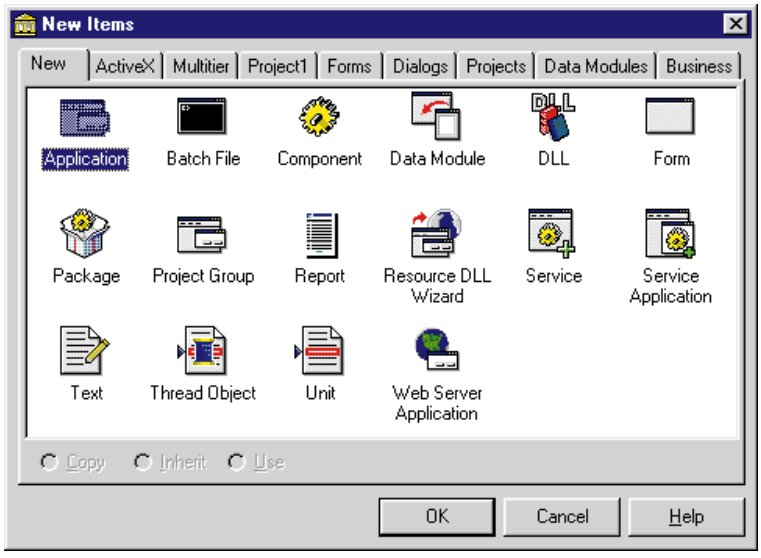


Figure 6: The New page of the Delphi 4 New Items dialog box.

On the Cover

MIDAS components for single- and two-tier applications easier. You can now easily implement business rules on a row-by-row basis, use MIDAS components in two-tier applications as an alternative to local table transactions and cached updates, and implement load balancing and callbacks for sockets connections. The MIDAS components now provide a flexible new architecture for any database application. Δ

This article is based on a prerelease version of Delphi 4. Features in the shipping version may differ or be absent.

Bill Todd is President of The Database Group, Inc., a database consulting and development firm based near Phoenix, AZ. A Contributing Editor of *Delphi Informant,* he is also co-author of four database programming books 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_Todd@compuserve.com or (602) 802-0178.

Delphi 3 / Internet Explorer 4 / DOM / HTML

By Ron Loewy

IE4's DOM Advantage

Putting the Internet Explorer WebBrowser Control to Work

One of the big enhancements to Web/Windows integration was introduced with the release of Microsoft's Internet Explorer 3.0 (IE3). IE3 exposed an ActiveX control named WebBrowser that can be embedded in Windows applications.

IE4 introduced Dynamic HTML (DHTML) and the Document Object Model (DOM), greatly expanding the amount of control a Windows application can have over a Web browser control. This article introduces some ways that a Delphi application can take advantage of IE4's programming interface.

In Brief: DHTML and the DOM

DHTML is an extension of HTML 3.2 that provides greater programmability of HTML pages with scripts. In DHTML, the HTML document is exposed as a hierarchical collection of objects, where every element (usually a tag) in the document is accessible as an object.

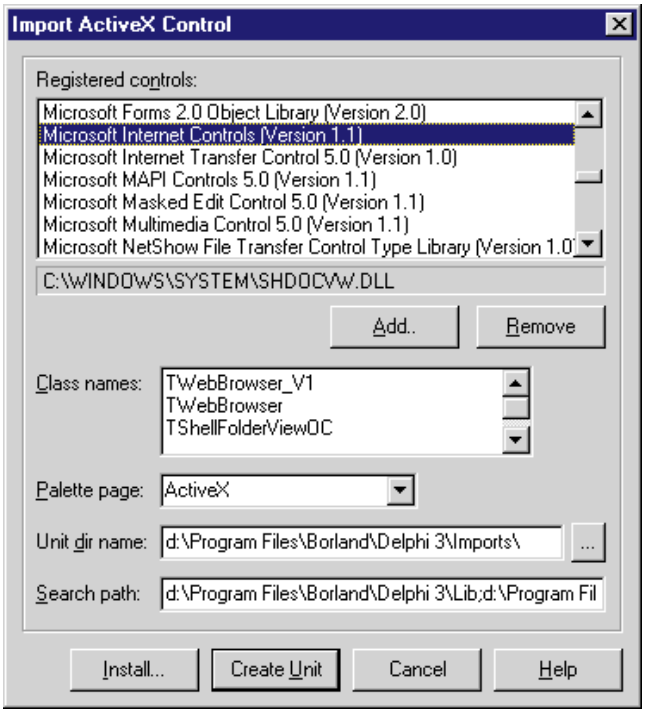


Figure 1: Installing the WebBrowser control into Delphi.

This hierarchical object representation of an HTML document is the DOM. Given access to an HTML "document," your scripts can access elements in the page, change them, and have the changes reflected immediately in the browser window.

Access to DHTML elements via scripts can be accomplished using one of several "collections" published by the document object: The *all* collection provides access to all the tags in the document; the *anchors* collection provides access to all the <A> tags; and the *scripts* collection provides access to all the <SCRIPT> elements in the document.

Individual items in a collection can be accessed using a numeric index (e.g. `document.anchors(2)` will access the second link in the document), or by the element's name defined using the ID attribute. For example:

```
<A HREF="..." ID=SecondLink>...</A>
```

can be accessed using `document.anchors.SecondLink`; both methods provide access to the same object.

DHTML offers many other features, such as absolute and relative positioning, database integration, and expanded stylesheet control. My goal in this article is not to introduce you to DHTML — there are books devoted to that subject; some of them are even good! With the assumption that you know what DHTML is and how it can be used in scripts, this article will show how the DOM can be accessed and manipulated from a Delphi application.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Flags, TargetFrameName, PostData, Headers: OleVariant;
begin
  Flags := 0;
  TargetFrameName := 0;
  PostData := 0;
  Headers := 0;
  WebBrowser.Navigate('e:\src\homepage.html', Flags,
                      TargetFrameName, PostData, Headers);
end;
```

Figure 2: Navigating to view homepage.html stored on the E: drive.

Figure 3: The result of the *Navigate* method on my homepage.html file.

The DOM via COM

When Microsoft introduced the DOM in IE4, they provided a way to access it from external applications using standard COM interfaces. Every HTML element in the DOM can be accessed using a COM interface. The hierarchy of interfaces mimics the object model that can be used in scripting code.

If you're not familiar with the DOM, it's a good idea to read some of the articles or books about DHTML and the DOM, and experiment with it using a scripting language, such as JScript or VBScript, in your HTML code. When you become comfortable with the DOM, take a look at the COM interfaces described in the Internet Client SDK (INetSDK) and those mentioned later in this article.

The MSIE WebBrowser Control

When you install IE4, it registers an ActiveX control. This browser control (implemented in shdocvw.dll) is a simple wrapper around the HTML layout engine (implemented in mshtml.dll) that provides a great way to start using IE4 in your applications.

The WebBrowser control does a lot of the low-level work of hosting an Active Document for you, but still allows you to access the DOM via COM interfaces. The INetSDK provides documentation and samples that show how the mshtml.dll layout engine can be hosted directly in your application. For the purpose of clarity, this article will not discuss the issues of hosting mshtml.dll directly. All the samples and code snippets assume that you've installed the WebBrowser control in the IDE, and that you use it to embed IE4 in your applications.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  All: IHtmlElementCollection;
  HtmlElement: IHtmlElement;
  i: Integer;
begin
  Document := WebBrowser.Document as IHtmlDocument2;
  if (assigned(Document)) then
    begin
      All := Document.All;
      Memo1.Lines.Clear;
      for i := 0 to All.length - 1 do begin
        HtmlElement := All.Item(i, 0) as IHtmlElement;
        if (Assigned(HtmlElement)) then
          Memo1.Lines.Add(IntToStr(i) + ' ' +
                          HtmlElement.TagName);
      end;
      ShowMessage('Number of elements : ' +
                  IntToStr(All.Length));
    end;
end;
```

Figure 4: Accessing the document and printing all tags to a memo control.

In Delphi 3, you'll need to use the **Component | Import ActiveX Control** menu option and add the Microsoft Internet Controls (Version 1.1) control to your component library, as shown in Figure 1. You can now create a new project, switch to the ActiveX Components tab in the component library, and drop a *TWebBrowser* control on the form.

The easiest way to start playing with the control is to call the *Navigate* method to view a document saved on your file system. I use the code in Figure 2 to view an HTML file called homepage.html stored in E:\src, as shown in Figure 3.

Accessing the DOM from Delphi

The first step to accessing the DOM from Delphi is to import the interfaces defined in the INetSDK. Unfortunately, Delphi 3's TypeLib import command cannot handle mshtml.dll where the interesting interfaces are defined. You'll need to download the 3.02 (Maintenance release 2) update from INPRISE's Delphi updates Web page (http://www.inprise.com/devsupport/delphi/downloads/dpro302download.html) and apply it on top of version 3.01. One of the modules installed by this release is tlibimp.exe in the \bin subdirectory of Delphi's installation. You'll need to execute this program from the command line and import mshtml.dll to create mshtml_tlb.pas. This file includes most of the interesting interfaces you need to access the DOM.

The *IHtmlDocument2* interface is your entry point into the DOM. The WebBrowser control exposes this interface via the *Document* property. The following code will retrieve the interface in Delphi:

```
Document := WebBrowser.Document as IHtmlDocument2;
```

where *Document* has been defined as a variable of type *IHtmlDocument2*.

Assuming you have some content in the browser control, you can now start accessing the DOM and access the document's content. The code in Figure 4 accesses the document and

prints all the tags found in the content to a memo control. It uses the *IHtmlElementCollection* and *IHtmlElement* interfaces.

The *IHtmlDocument2* interface exposes several collections you can use to access the content of the browser. This example uses the *All* property to access all the tags in the document. You can find more information on the interfaces exposed by *IHtmlDocument2* in the INetSDK documentation. Search on "IHtmlDocument2" to get the information. (You'll also notice that the *All* property of the *IHtmlDocument2* interface is the same as the *all* collection of the *Document* object, if you are using scripting code.) Figure 5 shows all the tags in homepage.html after "walking" the DOM.

Using the WebBrowser Control Events

The WebBrowser control provides several events that can be used to track the user's navigation. They're also important to your application's stability. The *OnBeforeNavigate2* event is fired before the control starts to navigate to a new URL after the user clicks on a link in the browser display. The following code provides feedback that the browser is navigating to a new URL:

```
procedure TForm1.WebBrowserBeforeNavigate2(Sender: TObject;
  pDisp: IDispatch; var URL, Flags, TargetFrameName,
  PostData, Headers: OleVariant; var Cancel: WordBool);
begin
  StatusBar1.SimpleText := 'Browser is opening ' + Url;
end;
```

The *OnDocumentComplete* event is used to inform your application that the control finished navigating to a new URL, that it finished rendering the information, and that it's safe for your application to retrieve the DOM entry point via the *Document* property. It's important you don't try to access the *IHtmlDocument2* interface before the control is in the ReadyState_Complete mode. Use code such as the following to ensure the control is ready:

```
procedure TForm1.WebBrowserDocumentComplete(
  Sender: TObject; pDisp: IDispatch; var URL: OleVariant);
begin
  StatusBar1.SimpleText := 'Finished:' + Url;
  Document := WebBrowser.Document as IHtmlDocument2;
end;
```

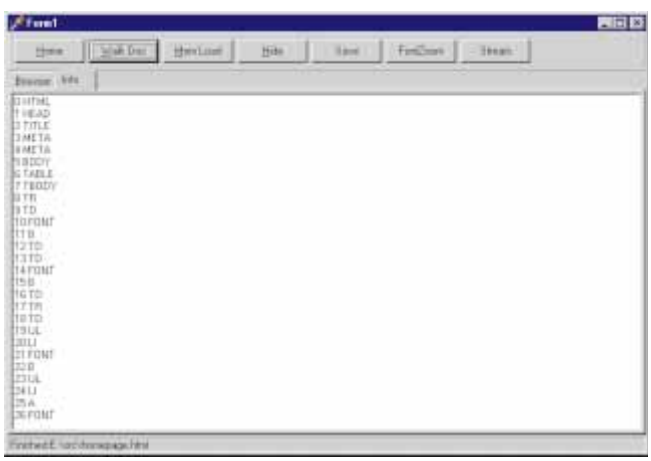


Figure 5: All the tags in the homepage.html file after "walking" the DOM.

An alternative method to ensure you do not access the DOM before the control is ready is to define *Document* as a property of type *IHtmlDocument2*, and to use a *GetDocument* read method such as the following:

```
function TForm1.GetDocument: IHtmlDocument2;
begin
  while (WebBrowser.ReadyState <> ReadyState_Complete) do
    Application.ProcessMessages;
  Result := WebBrowser.Document as IHtmlDocument2;
end;
```

Printing from the WebBrowser Control

Printing is a common task you'll want to perform from the WebBrowser control. I prefer to read information retrieved from the Web on the screen, but I can assure you that your users will want to be able to create a hard copy.

The browser control provides access to an *IOleCommandTarget* interface that can be used to perform operations such as printing:

```
procedure THtmlPageEditorForm.ViewerPrint1Click(
  Sender: TObject);
var
  VI, VO: OleVariant;
  IECommand: IOleCommandTarget;
begin
  IECommand := Document as IOleCommandTarget;
  IECommand.Exec(nil, OLECMDID_PRINT,
                 OLECMDEXEOPT_DONTPROMPTUSER, VI, VO);
end;
```

Clipboard Support

The *IOleCommandTarget* interface allows you to provide copy functionality from your application. The following statement will copy the currently selected content in the browser to the Clipboard:

```
IECommand.Exec(nil, OLECMDID_COPY,
               OLECMDEXEOPT_DONTPROMPTUSER, VI, VO);
```

If you want to copy everything to the Clipboard, make sure you select all the content before you copy it:

```
IECommand.Exec(nil, OLECMDID_SELECTALL,
               OLECMDEXEOPT_DONTPROMPTUSER, VI, VO);
```

Font Control

The WebBrowser control's default fonts might surprise you; an HTML document that doesn't use stylesheets or a <BASEFONT> tag might display differently in the WebBrowser and IE. Apparently, there are two issues you need to resolve in your code to set the default fonts:

- You need to set the size of the font to a value between 0 and 4 (0 = smallest, 4 = largest). The values correspond to the IE4 **View | Fonts** menu option. The code in Figure 6 sets the font size to 1.
- To set the default font face, you need to implement the *IDocHostUIHandler* interface. Unfortunately, I couldn't find a way to import this interface into Delphi; you'll have to translate it from C manually.

In your *IDocHostUIHandler*, you need to implement the *GetOptionKeyPath* method and provide the registry path of

```
procedure TForm1.Button3Click(Sender: TObject);
var
  IECommand: IOleCommandTarget;
  V, O: OleVariant;
const
  OLECMDEXEOPT_DONTPROMPTUSER = 2;
begin
  V := 1;
  IECommand := WebBrowser.Document as IOleCommandTarget;
  IECommand.Exec(nil, OLECMDID_ZOOM,
                 OLECMDEXEOPT_DONTPROMPTUSER, V, O);
end;
```

Figure 6: This code sets the font size to 1.

your browser defaults. The default font values should be written under the International\CodePage key (where CodePage is the code page Windows uses) in the *IEPropFontName* and *IEFixedFontName* entries.

Speeding up the WebBrowser

In my application, I need to provide fast updates to the content of the page displayed by the browser control. The WebBrowser control's functionality allows it to host Java applets and ActiveX controls, handle style sheets, and execute scripts. Unfortunately, all this functionality comes at a cost — speed. To speed the preview portion of my application, I need to extract every bit of performance from the control.

Instead of writing every change to the displayed page content to a disk file and using the *Navigate* method, I use the *IHtmlDocument* interface's *write* and *close* methods to write the content directly from memory. The code in Figure 7 demonstrates how this can be done.

It seems that Microsoft did not implement all the functionality of the *Navigate* method in *write* and *close*. For example, your font defaults implemented using *IDocHostUIHandler* will be ignored. Another problem with *close* is that the *OnDocumentComplete* event is not fired. Instead, I use a Timer component, set it to 100 milliseconds, and use the following code to set the font size after the document has been loaded into the control:

```
procedure THtmlPageEditorForm.Timer1Timer(Sender: TObject);
begin
  inherited;
  if (WebBrowser.ReadyState = ReadyState_Complete) then
    begin
      Timer1.Enabled := False;
      ZoomFont;
    end;
end;
```

ZoomFont implements the font size code I just described. My application can sometimes update the displayed page's source code several times before the control finishes the rendering of the last update. I use a virtual semaphore to signal that the control is busy and queue several changes into one rendering operation. The *ReadyState* property should be your guide for this functionality.

Hooking into the DOM Event Model

The DOM allows you to hook into events that can be hooked in scripting code, such as the *Onclick*, *Onmouseover*, and *Onmouseexit* events you can handle in JScript or VBScript.

```
procedure TForm1.MemLoadClick(Sender: TObject);
var
  v: Variant;
begin
  if (assigned(Document)) then
    begin
      v := VarArrayCreate([0, 0], varVariant);
      v[0] := '<html><head><title>Hello World</title>' +
              '</head><body>Start me up</body></html>';
      Document.Write(PSafeArray(TVarData(v).VArray));
      Document.Close;
    end;
end;
```

Figure 7: Using *IHtmlDocument*'s *write* and *close* methods to write content directly from memory.

If you look at mshtml_tlb.pas, which was created when you imported the TypeLib of mshtml.dll, you'll see a dispatch interface named *HtmlDocumentEvents*. To hook this events interface, you need to use the *IConnectionPointContainer* and *IConnectionPoint* interfaces to connect the DOM entry point to an object that implements *HtmlDocumentEvents*.

The method of connecting events using connection points is beyond the scope of this article, but you can simply copy the procedures *InterfaceConnect* and *InterfaceDisconnect* that INPRISE includes as part of OleCtrls.pas in the Source\VCL directory.

I used Delphi's Automation Object Wizard to create a *TAutoObject* descendant that implements the *HtmlDocumentEvents* interface (see Figure 8). Because this is a dispatch interface, I used Delphi's TypeLib Editor to copy the dispatch IDs to the automation object I was creating. The result is displayed in Figure 9.

My automation object implements the *OnClick* handler using the following code:

```
function THtmlDocumentEventsHandler.OnClick: WordBool;
begin
  showMessage('Hello World!');
  Result := True;
end;
```

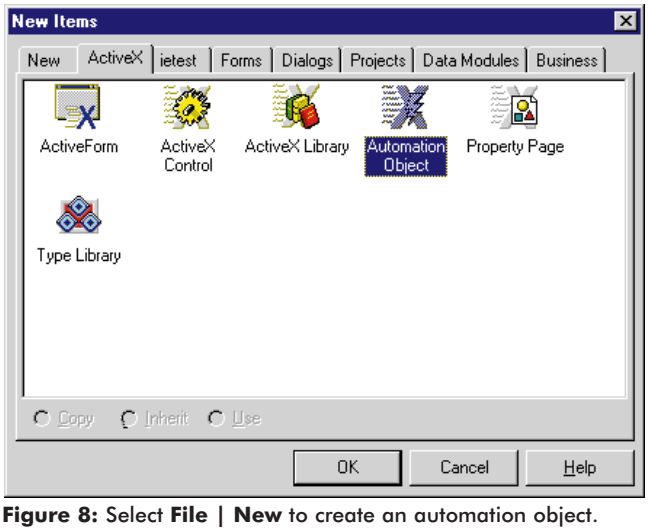


Figure 8: Select **File | New** to create an automation object.

```
IHtmlDocumentEventsHandlerDisp = dispinterface
  ['{ 016B5321-631C-11D1-A5E6-0040053BA735 }']
  function onhelp: WordBool; dispid -2147418102;
  function onclick: WordBool; dispid -600;
  function ondblclick: WordBool; dispid -601;
  procedure onkeydown; dispid -602;
  procedure onkeyup; dispid -604;
  function onkeypress: WordBool; dispid -603;
  procedure onmousedown; dispid -605;
  procedure onmousemove; dispid -606;
  procedure onmouseup; dispid -607;
  procedure onmouseout; dispid -2147418103;
  procedure onmouseover; dispid -2147418104;
  procedure onreadystatechange; dispid -609;
  function onbeforeupdate: WordBool; dispid -2147418108;
  procedure onafterupdate; dispid -2147418107;
  function onrowexit: WordBool; dispid -2147418106;
  procedure onrowenter; dispid -2147418105;
  function ondragstart: WordBool; dispid -2147418101;
  function onselectstart: WordBool; dispid -2147418100;
  function onerrorupdate: WordBool; dispid -2147418099;
end;
```

Figure 9: The result of using Delphi's TypeLib Editor to copy dispatch IDs to an automation object.

The code returns True, which tells the document object model not to bubble the event; this will ensure the default behavior of a *click* event will be carried. If we wanted to eliminate this behavior, we would have returned False; clicks on links would be ignored.

Now comes the fun part. This statement connects the event handler object to the document object model:

```
InterfaceConnect(Document, HtmlDocumentEvents,
                 DocHandler, DocConnectionID);
```

where *Document* is the *IHtmlDocument2* instance, and *DocConnectionID* is an integer variable used to keep tabs on the connections performed. *DocHandler* is the *TAutoObject* descendant instance that implements the *HtmlDocumentEvents* dispatch interface. When we're done with the document, we must disconnect from the event model using the following code:

```
InterfaceDisconnect(Document, HtmlDocumentEvents,
                    DocConnectionID);
```

Wrapping It Up

You should install the INetSDK, and acknowledge that a large portion of your time will be spent reading the documentation, looking at the samples, and wondering why no one documented the function you are interested in.

The following Usenet groups are some of the best places to find information and ask questions about the DOM — and how to handle it from Delphi:

- borland.public.delphi.activex.controls.using
- borland.public.delphi.internet
- borland.public.delphi.oleautomation
- microsoft.public.inetsdk.programming.html_objmodel
- microsoft.public.inetsdk.programming.mshtml_hosting
- microsoft.public.inetsdk.programming.webbrowser_ctl

As you can see, a lot can be done with IE4 from a Windows application if you are willing to get down and dirty with COM interfaces and incomplete documentation. The IE WebBrowser control is a very powerful tool for Internet-aware applications, and can be used to create non-traditional Internet applications.

An example of a non-browser application I wish I had the time to write is an adventure game like the Ultima games. Instead of using complex graphic code to implement the graphics and the game logic engine, you can use your favorite HTML editor and graphic editor to easily create "rooms" in the adventure, and apply the logic that takes the user in the maze using Delphi code. I'm sure you'll be able to think of other uses for the browser control. Δ

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 Rod Stephens

Vivid Equations

An Introduction to 3-D Graphics Programming

Three-dimensional graphics is a complicated subject. Entire books have been written about generating three-dimensional images. There are even books about software that generates three-dimensional images.

While you could spend years studying the topic, much of three-dimensional graphics is based on fairly simple mathematics. This article describes the equations that are the basis for three-dimensional graphics programming. It also shows how to use those equations to draw simple three-dimensional objects and surfaces.

Transformations

Transformations occur when computer operations take an object and transform it in some way. The most important transformations in three-dimensional graphics are translation, scaling, projection, and rotation. Each of these modifies the coordinates of a point in a different way. For example, translation moves a point in three-dimensional space.

Translation, scaling, projection, and rotation all preserve straight lines. After any combination of these transformations, a straight line will still be straight. This is important because it allows a program to quickly transform and display complex objects.

To transform an object, a program transforms the points that define that object. It then connects the transformed points as they are connected in the original object using straight lines. For example, to transform a rectangle, the program transforms the four corners of the rectangle, and then connects them.

Translation and scaling transformations are straightforward. Translating the point (x, y, z) by the offset (tx, ty, tz) gives the point (x + tx, y + ty, z + tz). The point is moved distance tx in the X direction, ty in the Y direction, and tz in the Z direction.

Scaling or stretching by factors of (sx, sy, sz) gives the point (x * sx, y * sy, z * sz). The new point lies sx times as far from the origin as the old point in the X direction. Similarly, it lies sy times farther in the Y direction and sz times farther in the Z direction.

A projection maps a point in a particular dimension into a lower dimensional space. Usually, this means mapping a three-

dimensional point into a two-dimensional space so the point can be displayed on a two-dimensional computer monitor.

The simplest kind of projection is a parallel projection that projects a point onto the X-Y plane by ignoring the point's Z coordinate. The point (x, y, z) in three dimensions is transformed into the point (x, y) in two dimensions. To create different parallel projections, the program can translate, scale, and rotate points before applying the simple projection onto the X-Y plane.

In practice, graphics programs often do not actually set a point's Z coordinate to zero. Instead, they simply ignore the Z coordinate when drawing the point.

Rotation

Rotation is a little more complex than translation, scaling, and projection. The simplest case is rotating a point around the origin in two dimensions, as shown in Figure 1. Here the point (x, y) lies distance R = Sqrt(x * x + y * y) from the origin.

The line from the point to the origin makes some angle α with the X axis, by the definition of the sine and cosine functions:

x = R * Cos(α)
y = R * Sin(α)

Now suppose the point is rotated by angle θ around the origin. The distance R from the point to the origin is unchanged, so the point's new coordinates (x', y') are given by:

x' = R * Cos(α + θ)
y' = R * Sin(α + θ)

These equations can be rewritten using the equations:

Cos(α + θ) = Cos(α) * Cos(θ) - Sin(α) * Sin(θ)
Sin(α + θ) = Sin(α) * Cos(θ) + Cos(α) * Sin(θ)

Making these substitutions gives:

x' = R * [Cos(α) * Cos(θ) - Sin(α) * Sin(θ)]
y' = R * [Sin(α) * Cos(θ) + Cos(α) * Sin(θ)]

Recall that the point's original coordinates were given by:

x = R * Cos(α)
y = R * Sin(α)

Then, substituting these values into the new equations gives:

x' = x * Cos(θ) - y * Sin(θ)
y' = y * Cos(θ) + x * Sin(θ)

The value θ is the angle through which the point should be rotated. If the program knows θ, it can use these equations to calculate the point's new coordinates.

These equations show how to rotate a point around the origin. To rotate a point around another point, the program can translate the point of rotation to the origin, rotate around the origin, and translate the point of rotation back to its original position. This allows the program to treat one complicated rotation as a simple translation, followed by a simple rotation, followed by another simple translation.

In two dimensions, a program rotates a point around another point. In three dimensions, it rotates a point around a line. Rotating a point around a general line would be difficult to analyze directly. Rotating around one of the coordinate axes, however, is straightforward. For example, when a program rotates a point around the Z axis, the point's X and Y coordinates change, but its Z coordinate value is unchanged. The program can simply treat the point as if it were rotating around the origin in the X-Y plane, and leave the point's Z coordinate alone.

Similarly, a program can treat rotation around the X or Y axis as rotation around the origin in the Y-Z or X-Z plane.

Rotating around lines other than the origins requires more steps, but is conceptually simple. Instead of calculating equations to rotate around the line directly, the program can build the rotation out of simpler steps.

First, the program translates the points so the line of rotation intersects the origin. Next, it rotates around the Z axis until the line lies in the Y-Z plane. It then rotates around the X axis until the line coincides with the Z axis. Now, the program can rotate around the line by rotating around the Z axis. The program finishes by reversing the rotations, and translation is used to make the line of rotation lie in the Z axis.

Homogeneous Coordinates

Each of these transformations is understandable individually, but combining them is difficult. Translation adds values to a point's coordinates, scaling multiplies; rotation uses a combination of more than one of the point's coordinates multiplied

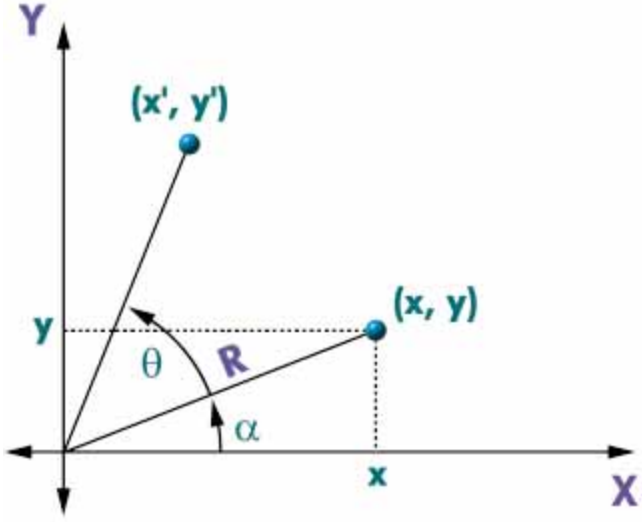


Figure 1: Rotating a point (x, y) through the angle ?.

and added in a complex way. Combining the equations for a more general transformation would be hard.

Homogeneous coordinates allow a program to treat all of these transformations in a uniform (homogeneous) way. They allow a program to combine any number of transformations easily.

In a three-dimensional homogeneous coordinate system, a point is represented by a vector of four values [a, b, c, s]. The fourth value is a scaling factor. The first three values are the point's coordinates, multiplied by the scaling factor. For example, the following vectors all represent the point (1, 2, 3):

[1, 2, 3, 1]
[10, 20, 30, 10]
[1.5, 3, 4.5, 1.5]

A vector is called normalized when its scaling factor is 1. A program can normalize a vector by dividing each of its components by the scaling factor.

Transformations are represented by 4 x 4 matrices. To apply a transformation to a point, a program multiplies the point's vector by the transformation matrix. The identity transformation, which leaves a point unchanged, is represented by the identity matrix:

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

If you multiply this matrix by the vector [a, b, c, s], you get [a, b, c, s], so the point is unchanged.

Translation by a distance of (tx, ty, tz) is represented by the matrix:

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ tx & ty & tz & 1 \end{vmatrix}$$

If you multiply this matrix by the point [a, b, c, s], you get:

$$[a, b, c, s] * \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ tx & ty & tz & 1 \end{vmatrix} = [a + 1 * tx, b + 1 * ty, c + 1 * tz, s]$$

This vector correctly represents the translation of the point. Scaling by factors of sx, sy, and sz is represented by the matrix:

$$\begin{vmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

If you multiply this matrix by the vector [a, b, c, s], you get [a * sx, b * sy, c * sz, s], as desired. Projection along the Z axis onto the X-Y plane is represented by:

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Multiply this matrix by the vector [a, b, c, s] and the resulting vector [a, b, 0, s] has Z coordinate zero. As mentioned, most programs don't bother to perform this multiplication. Instead, they simply ignore the points' Z coordinates when drawing.

Finally, rotation though the angle θ around the Z axis in the X-Y plane is represented by:

$$\begin{vmatrix} \text{Cos}(\theta) & \text{Sin}(\theta) & 0 & 0 \\ -\text{Sin}(\theta) & \text{Cos}(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Multiplying the vector [a, b, c, s] by this matrix gives:

[a * Cos(θ) - b * Sin(θ), a * Sin(θ) + b * Cos(θ), c, s]

This result agrees with the equations for rotation around the origin in two dimensions, as shown earlier.

Similarly, the matrix representing rotation around the X axis is:

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \text{Cos}(\theta) & \text{Sin}(\theta) & 0 \\ 0 & -\text{Sin}(\theta) & \text{Cos}(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

and the matrix representing rotation around the Y axis is:

$$\begin{vmatrix} \text{Cos}(\theta) & 0 & \text{Sin}(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\text{Sin}(\theta) & 0 & \text{Cos}(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

One of the nice things about these transformation matrices is that the product of two matrices represents the transformations applied one after each other. For example, if A and B are transformation matrices, and p is a vector, then:

(p * A) * B = p * (A * B)

This means a program can multiply the matrices together first, and then apply the result to the point later.

If the program is transforming a single point, this is no faster than applying the matrices to the point one at a time. However, if the program must transform hundreds or thousands of points, it can mean a large savings. Instead of applying both matrices to each point, the program need only apply the combined transformation to each point. For more complex transformations involving many rotations and translations, the savings are even greater.

Drawing in Delphi

Enough math! How can you use all this to draw three-dimensional pictures in Delphi? One straightforward

method is to create an array of points representing the end points of line segments. The program transforms the points, and then connects them on the screen.

The example program Platonic, shown in Figure 2 (and available for download; see end of article for details), uses this approach to draw the Platonic solids: tetrahedron, cube, octahedron, dodecahedron, and icosahedron. It defines the record type *Point3D* to hold a point's original and transformed coordinates:

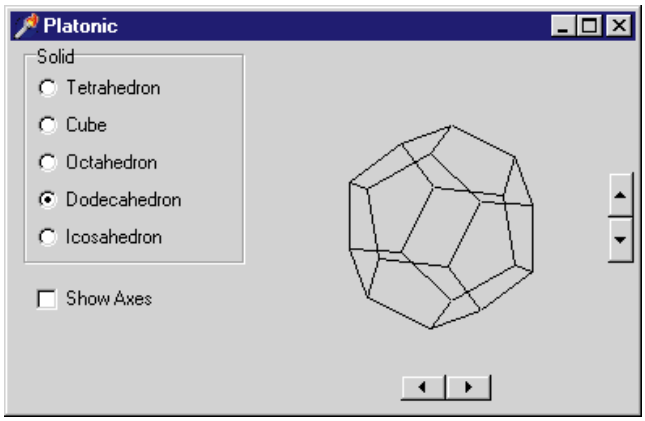


Figure 2: The example program Platonic displaying a dodecahedron.

```
TPoint3D = record
  Coord: TVector3D;  // The untransformed coordinates.
  Trans: TVector3D;  // The transformed coordinates.
end;
```

The program declares a *Segments* array to contain the segments it will draw. The variable *NumSegments* records the number of segments used:

```
Segments    : array [0..1000, 1..2] of
TPoint3D;
NumSegments : Integer;
```

The *Segment* array's first dimension gives a segment number. The second gives the start and end points for a segment. For

```
// Make M an identity matrix.
procedure TPlatonicForm.MakeIdentity(var M: TMatrix3D);
var
  i, j: Integer;
begin
  for i := 1 to 4 do
    for j := 1 to 4 do
      if (i = j) then
        M[i, j] := 1.0
      else
        M[i, j] := 0.0;
end;

// Perform matrix-matrix multiplication. Set R = A * B.
procedure TPlatonicForm.MatrixMatrixMult(var R: TMatrix3D;
                                         A, B: TMatrix3D);
var
  i, j, k: Integer;
  value: Single;
begin
  for i := 1 to 4 do
    for j := 1 to 4 do begin
      // Calculate R[i, j].
      value := 0.0;
      for k := 1 to 4 do
        value := value + A[i, k] * B[k, j];
      R[i, j] := value;
    end;
end;

// Perform vector-matrix multiplication. Set r = p * A.
procedure TPlatonicForm.VectorMatrixMult(var r: TVector3D;
  p: TVector3D; A: TMatrix3D);
var
  i, j: Integer;
  value: Single;
begin
  for i := 1 to 4 do begin
    value := 0.0;
    for j := 1 to 4 do
      value := value + p[j] * A[j, i];
    r[i] := value;
  end;

  // Normalize the point. Note value still holds r[4].
  r[1] := r[1] / value;
  r[2] := r[2] / value;
  r[3] := r[3] / value;
  r[4] := 1.0;
end;
```

Figure 3: Matrix and point manipulation procedures.

```
// Build a transformation matrix for display.
procedure TPlatonicForm.BuildTransformation(
  var T: TMatrix3D);
var
  r1, r2, ctheta, stheta, cphi, sphi: Single;
  T1, T2, T3, T4, T12, T34: TMatrix3D;
begin
  // Rotate around the Z axis until the eye lies in
  // the Y-Z plane.
  r1 := Sqrt(EyeX * EyeX + EyeY * EyeY);
  stheta := EyeX / r1;
  ctheta := EyeY / r1;
  MakeIdentity(T1);
  T1[1, 1] := ctheta;
  T1[1, 2] := stheta;
  T1[2, 1] := -stheta;
  T1[2, 2] := ctheta;

  // Rotate around the X axis until the eye lies within
  // the Z axis.
  r2 := Sqrt(EyeX * EyeX + EyeY * EyeY + EyeZ * EyeZ);
  sphi := -r1 / r2;
  cphi := -EyeZ / r2;
  MakeIdentity(T2);
  T2[2, 2] := cphi;
  T2[2, 3] := sphi;
  T2[3, 2] := -sphi;
  T2[3, 3] := cphi;

  // We could project along the Z axis here. Instead we
  // just ignore the Z coordinate when drawing.

  // Make the picture reasonably large on the form.
  // Here we scale y by -50 to reverse its sign since
  // the Canvas starts with (0, 0) in the upper left.
  MakeIdentity(T3);
  T3[1, 1] := 50;
  T3[2, 2] := -50;
  T3[3, 3] := 50;

  // Center the picture on the form.
  MakeIdentity(T4);
  r1 := SolidOption.Width + SolidOption.Left;
  T4[4, 1] := (ClientWidth - r1) / 2 + r1;
  T4[4, 2] := ClientHeight / 2;

  // Combine the transformations.
  MatrixMatrixMult(T12, T1, T2);
  MatrixMatrixMult(T34, T3, T4);
  MatrixMatrixMult(T, T12, T34);
end;
```

Figure 4: *BuildTransformation* creates a transformation matrix for viewing from point (*EyeX*, *EyeY*, *EyeZ*) looking toward the origin.

example, the first line segment connects points Segments[0, 1] and Segments[0, 2].

When it starts, Platonic loads data into the *Segments* array. *FirstSegment* is an array of integers indicating the indexes of the first segment that belong in each Platonic solid. For example, *FirstSegment[1]* gives the index in the *Segments* array of the first segment used to draw the tetrahedron. Figure 3 shows matrix and point manipulation routines used by the program.

The procedure *BuildTransformation*, shown in Figure 4, creates a transformation matrix for projecting the data. This transformation acts as if it were viewing the data from the point (*EyeX*, *EyeY*, *EyeZ*) and looking toward the origin. The procedure applies rotations until the eye coordinates lie in the Z axis, and then it projects onto the X-Y plane. It also adds a scaling and a translation to make the result fit nicely in the form's pixel coordinate system.

When the program receives a *Paint* event, it calls *BuildTransformation*. It applies the transformation to the segments that make up the selected solid. It then draws lines between the transformed points. It repeats these steps for the three segments that represent the coordinate axis if the program's **Show Axes** box is checked. Figure 5 shows the program's *Paint* event handler.

On the Surface

The example program Platonic stores both end points for each of the segments it draws. This wastes a little space since each of the segments in a Platonic solid connect to at least two others. For example, each corner of a cube is shared by three edges. For small data sets like this one, the waste is not terribly important.

The Surface example program, shown in Figure 6, stores the coordinates of each point for an array of X and Y values. This allows it to transform each point only once. Figure 7 shows how the program transforms the data and draws the surface.

Below the Surface

If you look closely at Figure 6, you will see parts of the surface overlapping each other. This picture shows the surface at a high enough angle that the overlap is not a big problem. On the other hand, when viewed closer to edge on, the picture becomes hopelessly cluttered. Removing hidden surfaces would make the image easier to understand, but hidden-surface removal in general is a difficult problem. For surfaces like this one, however, a special algorithm lets a program remove the hidden surfaces from the picture quickly and easily.

The program simply draws in order the rectangles that make up the surface, drawing the rectangles farthest from the eye coordinates (*EyeX*, *EyeY*, *EyeZ*) first. As each rectangle is drawn, it is filled with a surface color. This erases any previously drawn

```
// Draw the selected solid.
procedure TPlatonicForm.FormPaint(Sender: TObject);
var
  i, seg1, seg2: Integer;
  T: TMatrix3D;
  rect: TRect;
begin
  // Build the transformation.
  BuildTransformation(T);

  // Erase the form.
  rect.Left := 0;
  rect.Top := 0;
  rect.Right := ClientWidth;
  rect.Bottom := ClientHeight;
  Canvas.Brush.Color := Color;
  Canvas.FillRect(rect);

  // Draw the selected solid's segments.
  Canvas.Pen.Color := clBlack;
  i := SolidOption.ItemIndex + 1;
  seg1 := FirstSegment[i];
  seg2 := FirstSegment[i + 1] - 1;
  for i := seg1 to seg2 do begin
    // Apply the transformation to the points.
    VectorMatrixMult(Segments[i, 1].Trans,
                     Segments[i, 1].Coord, T);
    VectorMatrixMult(Segments[i, 2].Trans,
                     Segments[i, 2].Coord, T);

    // Draw the segment.
    Canvas.MoveTo(Round(Segments[i, 1].Trans[1]),
                  Round(Segments[i, 1].Trans[2]));
    Canvas.LineTo(Round(Segments[i, 2].Trans[1]),
                  Round(Segments[i, 2].Trans[2]));
  end;

  // Draw the axes if desired.
  if (ShowAxesCheck.Checked) then
    begin
      Canvas.Pen.Color := clGreen;
      for i := FirstSegment[0] to FirstSegment[1] - 1 do
        begin
          // Apply the transformation to the points.
          VectorMatrixMult(Segments[i, 1].Trans,
                           Segments[i, 1].Coord, T);
          VectorMatrixMult(Segments[i, 2].Trans,
                           Segments[i, 2].Coord, T);

          // Draw the segments.
          Canvas.MoveTo(Round(Segments[i, 1].Trans[1]),
                        Round(Segments[i, 1].Trans[2]));
          Canvas.LineTo(Round(Segments[i, 2].Trans[1]),
                        Round(Segments[i, 2].Trans[2]));
        end;
    end;
end;
```

Figure 5: When it receives a *Paint* event, the example program Platonic builds a transformation matrix, transforms its data points, and then draws the segments that make up the selected solid.

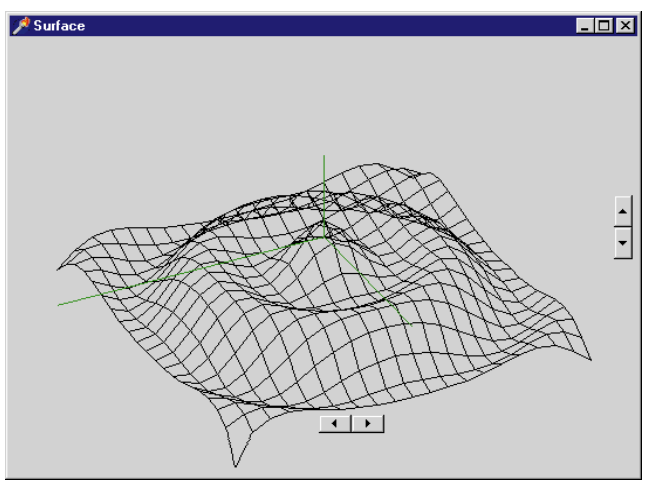


Figure 6: The example program Surface displaying a surface.

rectangles that should be hidden by the new one. As long as the rectangles are drawn in order from farthest to nearest, the newer rectangles erase the old ones that should be hidden.

The example program Hidden, shown in Figure 8, uses this method to display the same surface drawn by the example program Surface. The programs are very similar. The only difference is in how the programs draw the surface. The code used by Hidden, shown in Figure 9, does not actually sort the rectangles. It simply draws them ordered by their X and Y coordinates. If you rotate the image displayed by the program until that ordering is incorrect, you will see parts of rectangles overlapping other rectangles.

```
// Draw the surface.
procedure TSurfaceForm.FormPaint(Sender: TObject);
var
  x, y: Integer;
  T: TMatrix3D;
  rect: TRect;
begin
  // Build the transformation.
  BuildTransformation(T);

  // Apply the transformation to the points.
  for x := Xmin to Xmax do
    for y := Ymin to Ymax do
      VectorMatrixMult(Points[x, y].Trans,
                       Points[x, y].Coord, T);

  // Erase the form.
  rect.Left := 0;
  rect.Top := 0;
  rect.Right := ClientWidth;
  rect.Bottom := ClientHeight;
  Canvas.Brush.Color := Color;
  Canvas.FillRect(rect);

  // Draw the lines that are parallel to the X axis.
  Canvas.Pen.Color := clBlack;
  for x := Xmin to Xmax do begin
    Canvas.MoveTo(Round(Points[x, Ymin].Trans[1]),
                  Round(Points[x, Ymin].Trans[2]));
    for y := Ymin + 1 to Ymax do
      Canvas.LineTo(Round(Points[x, y].Trans[1]),
                    Round(Points[x, y].Trans[2]));
  end;

  // Draw the lines that are parallel to the Y axis.
  for y := Ymin to Ymax do begin
    Canvas.MoveTo(Round(Points[Xmin, y].Trans[1]),
                  Round(Points[Xmin, y].Trans[2]));
    for x := Xmin + 1 to Xmax do
      Canvas.LineTo(Round(Points[x, y].Trans[1]),
                    Round(Points[x, y].Trans[2]));
  end;

  // Transform the origin.
  VectorMatrixMult(Axes[0].Trans, Axes[0].Coord, T);

  // Transform and draw the axes.
  Canvas.Pen.Color := clGreen;
  for x := 1 to 3 do begin
    VectorMatrixMult(Axes[x].Trans, Axes[x].Coord, T);
    Canvas.MoveTo(Round(Axes[0].Trans[1]),
                  Round(Axes[0].Trans[2]));
    Canvas.LineTo(Round(Axes[x].Trans[1]),
                  Round(Axes[x].Trans[2]));
  end;
end;
```

Figure 7: The example program Surface uses this code to transform and display a surface.

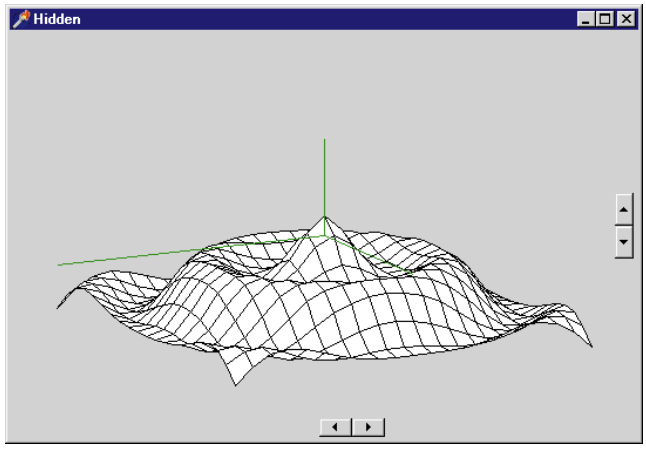


Figure 8: The example program Hidden displaying a surface with hidden surfaces removed.

```
// Draw the surface.
procedure THiddenForm.FormPaint(Sender: TObject);
var
  x, y: Integer;
  T: TMatrix3D;
  rect: TRect;
begin
  // Build the transformation.
  BuildTransformation(T);

  // Apply the transformation to the points.
  for x := Xmin to Xmax do
    for y := Ymin to Ymax do
      VectorMatrixMult(Points[x, y].Trans,
                       Points[x, y].Coord, T);

  // Erase the form.
  rect.Left := 0;
  rect.Top := 0;
  rect.Right := ClientWidth;
  rect.Bottom := ClientHeight;
  Canvas.Brush.Color := Color;
  Canvas.FillRect(rect);

  // Draw the tiles that make up the surface.
  Canvas.Pen.Color := clBlack;
  Canvas.Brush.Color := clWhite;
  for x := Xmin to Xmax - 1 do begin
    for y := Ymin to Ymax - 1 do begin
      Canvas.Polygon([
        Point(Round(Points[x    , y    ].Trans[1]),
              Round(Points[x    , y    ].Trans[2])),
        Point(Round(Points[x + 1, y    ].Trans[1]),
              Round(Points[x + 1, y    ].Trans[2])),
        Point(Round(Points[x + 1, y + 1].Trans[1]),
              Round(Points[x + 1, y + 1].Trans[2])),
        Point(Round(Points[x    , y + 1].Trans[1]),
              Round(Points[x    , y + 1].Trans[2])) ]);
    end;
  end;

  // Transform the origin.
  VectorMatrixMult(Axes[0].Trans, Axes[0].Coord, T);

  // Transform and draw the axes.
  Canvas.Pen.Color := clGreen;
  for x := 1 to 3 do begin
    VectorMatrixMult(Axes[x].Trans, Axes[x].Coord, T);
    Canvas.MoveTo(Round(Axes[0].Trans[1]),
                  Round(Axes[0].Trans[2]));
    Canvas.LineTo(Round(Axes[x].Trans[1]),
                  Round(Axes[x].Trans[2]));
  end;
end;
```

Figure 9: Hidden's *Paint* event handler.

Conclusion

This is far from the end of the three-dimensional graphics story. More advanced topics include more general hidden-surface removal, shadows, color smoothing, transparency, reflectivity, ambient lighting, and a host of other subjects. Using the techniques described here, however, you can take your first steps into the world of three-dimensional graphics. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\AUG\DI9808RS.

Rod is the author of such programming classics as *Visual Basic Graphics Programming* [John Wiley & Sons, 1997] and *Ready-to-Run Delphi 3.0 Algorithms* [John Wiley & Sons, 1998]. He also writes algorithm columns in *Visual Basic Developer* and *Microsoft Office & Visual Basic for Applications Developer*. You can find pictures created using 3-D graphics techniques at his Web site (http://www.vb-helper.com). You can reach Rod via e-mail at RodStephens@vb-helper.com.

By Cary Jensen, Ph.D.

Delphi Database Development

The BDE and the Components of RAD Database Development

I have been writing this column in *Delphi Informant* since the premiere issue. In that time, I have had the opportunity to cover a wide range of topics, from creating local database names using the Database component (the very first article), to cached updates; from OLE Automation to RTTI; from Code Insight to QuickReports. In fact, in recent months I have wondered if the column title, "DBNavigator," continues to be appropriate.

I've decided to keep the "DBNavigator" name. Furthermore, with this issue, I am beginning a series of columns focused on database-related issues. This month's column begins with an overall introduction to database development with Delphi. In coming months, I will revisit and re-examine some topics I've covered in the past, as well as introduce new database-related topics I have not yet had the opportunity to discuss.

While I still retain the right, and the desire, to cover issues that extend beyond the realm of database development, I think the timing is right for an extended look at database applications. Over the past year, I have encountered numerous developers who are new to Delphi, coming from languages such as Visual Basic, Clipper, and PowerBuilder. Consequently, although there are many Delphi developers who are completely comfortable with the issues surrounding database development, there is a growing number of programmers who are new to these issues.

Overview of Delphi Database Development

Delphi is not only a great language, it's an exceptional database development environment. This month, we begin with a general overview of the heart of database development in Delphi, the Borland Database Engine (BDE). We then continue with a discussion of the role played by many of the most common components used for data access.

The BDE

Delphi provides you with access to your data using the BDE. From time to time, you'll also see the BDE referred to as IDAPI (Integrated Database Application Programming Interface).

The BDE is a common data access layer for all of INPRISE's products, including Delphi, C++Builder, and JBuilder (through DataGateway). Several Corel products, including Paradox 8 for Windows and Quattro Pro, also use it. In addition, by default, database applications written with these products use the BDE. In other words, Delphi uses the BDE, as well as those database applications you write with Delphi.

The BDE consists of a collection of DLLs. Consequently, one installation of the BDE can be used by two or more BDE-aware applications simultaneously. As a result, each individual Delphi application is smaller in size than it would be if all data-accessing code needed to be linked into the executable. Furthermore, because two or more applications can share the same copy of the BDE loaded into memory, overall RAM usage is reduced when two or more BDE-aware applications are running simultaneously (when compared to linking the code into each .EXE).

The purpose of the BDE is to insulate you from the mundane tasks of data access. These include table and record locking, SQL construction, record updates, and basic I/O, just to mention a few. The BDE permits you to concentrate on what data you want to access, instead of how to access it. Because of the BDE, your Delphi applications can just as easily use data in dBASE or Paradox tables, files on a remote database server, and files supported by ODBC (Open DataBase Connectivity) drivers.

The BDE API consists of approximately 200 procedures and functions, all of which are available through the BDE unit. Fortunately, you almost never need to call any of these routines directly. Instead, you use the BDE through Delphi's data access components, which are found on the Data Access page of the Component palette. These components encapsulate calls to the BDE API, providing you with a much simpler interface. However, if you have special data needs not provided by these components, you can use the BDE API directly.

The relationship between Delphi (and other applications), the BDE, and underlying files is depicted in Figure 1. As you can see, the BDE is a software layout that lies between BDE-aware applications (Delphi and the database applications you create with it) and the sources of your data. It also shows that the BDE can access local tables directly, such as Paradox and dBASE. Connecting to Microsoft Access or FoxPro tables requires that you have the Microsoft DAO (Data Access Objects) DLLs installed.

Connection to other data sources requires additional drivers. For the best performance in connecting to a remote database server, such as Oracle, Microsoft SQL Server, InterBase, and so forth, use Borland SQL Links drivers. These native language drivers are provided in the Delphi Client/Server and Delphi Enterprise editions. In addition, SQL Links drivers must use an additional network protocol to connect to the remote server.

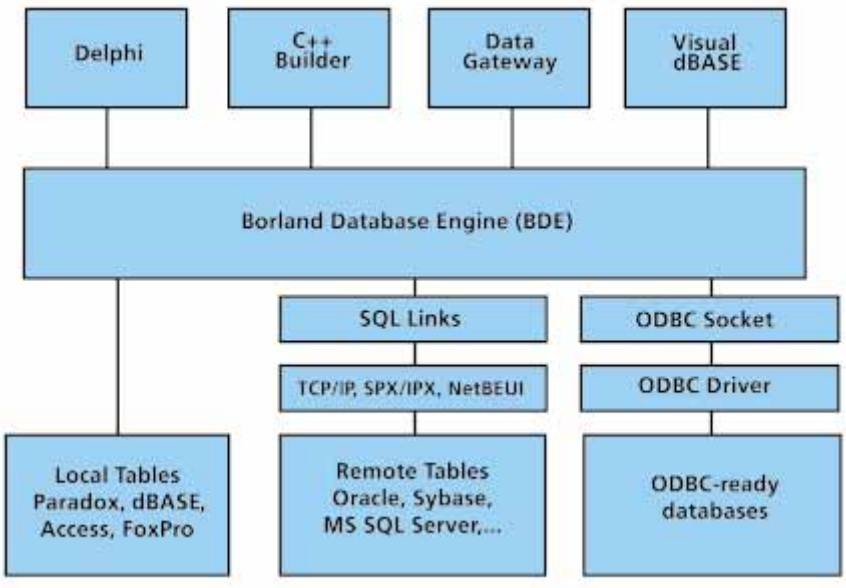


Figure 1: The relationship between BDE-aware applications, the BDE, and data sources.

Finally, the BDE also supports access to any file type for which there exists an ODBC driver. ODBC is a Microsoft standard for accessing databases, and is based on the Open SQL CLI (call level interface). To use one of these drivers, it must be installed and configured (using the ODBC 32-bit Administrator, available on the Control Panel).

Following are some of the benefits provided by the BDE:

- It provides seamless access to any data source, whether it's Paradox, dBASE, InterBase, Oracle, Sybase, Informix, Microsoft SQL Server, or data accessed through ODBC drivers.
- It provides the basic data engine for all your applications. Your clients only need one copy of the BDE on their systems, and this can be accessed by all the applications you write.
- It creates an OS (operating system)-independent layer for all your applications. The BDE manages all file I/O, network access, and memory management for data access.
- It provides enhanced performance with BLOB (Binary Large Object) data through caching services.
- It includes internal support for language drivers, providing you with an easier path to creating applications for the international market.
- It performs data translation between various data sources.
- It includes a SQL generator. Data requests, other than pass-through SQL, are translated into a common local SQL, which is a subset of ANSI92 SQL. Using the Borland SQL Links for Windows, these can be translated into the appropriate dialect of SQL, based on the SQL driver to which the BDE is sending its data requests.
- It offers access to data stored in formats supported by your installed ODBC drivers.

One of the most important benefits provided by the BDE is that your Delphi applications don't need to be written to a particular database standard. Specifically, use the same, simple interface provided by data access components to access your data. Even if later you need to change the underlying data type (e.g. from Paradox to InterBase, or Microsoft SQL Server to Oracle Server), your applications don't necessarily need to be recompiled. The description of where to find the data, and how to access the data, can be configured outside your applications using the BDE Administrator.

Understanding the BDE Administrator

You can define where your data is located, and which driver to use to access it, using the BDE Administrator. This program can be accessed from the Delphi 3 folder, but it also appears on the Control Panel.

The BDE Administrator consists of two panes. The left pane has two pages. The Databases page (see Figure 2) displays

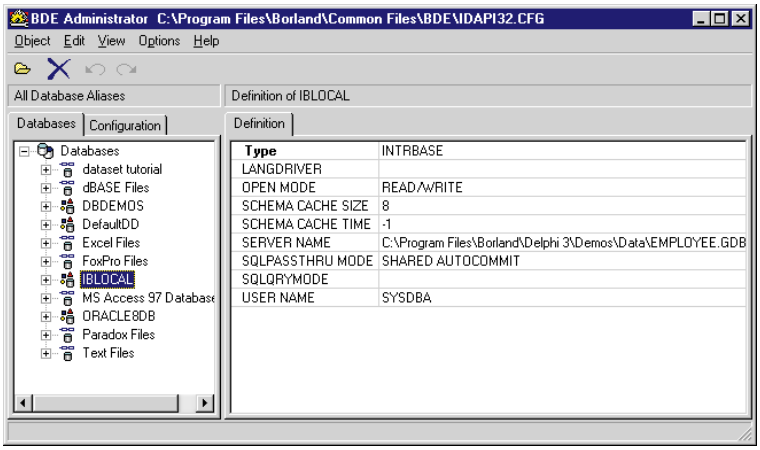


Figure 2: The BDE Administrator.

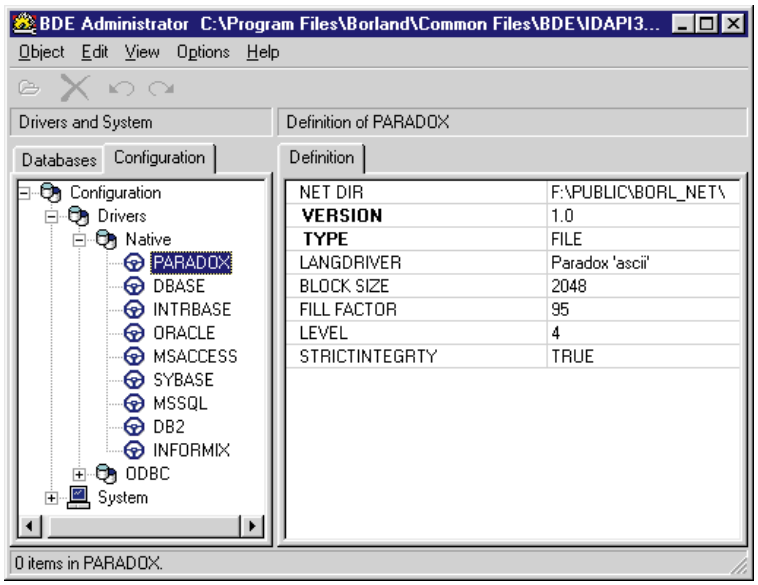


Figure 3: The Configuration page of the BDE Administrator.

aliases, and the Configuration page (see Figure 3) displays drivers and settings. The right pane contains the Definition page, which is used to display the various parameters of the selected database name, driver, or setting.

Working with Global Aliases

Aliases, or database names, are labels that reference a directory (or server) *and* a driver that can be used to access data in that directory (or server). Aliases defined on the Databases page of the BDE Administrator are available to all Delphi applications on that machine. Consequently, they are referred to as global aliases. By comparison, aliases defined within an application using a Database component are referred to as local aliases, because they are available only to that application.

Global aliases defined here are especially useful for data that must be accessed by more than one application. For example, a company may maintain a single table for storing general information about its employees. If more than one application needs to access this information, an alias can be created that points to the directory in which that table is stored. If all applications access this table using the defined alias (which is assigned to the *DatabaseName* property of the DataSet component being used for the access), a developer doesn't need to know the physical location of the table.

To create a new alias, right-click on **Databases** in the Databases page and select **New**. From the New Database Alias dialog box, select the driver that corresponds to the type of data being accessed. For local tables, including Paradox, dBASE, and ASCII, select **STANDARD**. Otherwise, select one of the other types available from the **Database Driver Name** drop-down combo box.

The BDE Administrator will generate a default name for the driver. Select the driver in the Databases page and enter the name you want to use for the driver. Then, enter the appropriate parameters for the new alias in the Definition page.

To change a given alias, select it in the Databases page. To change the alias name, click on the current name until it appears as an editable field. To change parameters, select the name, and then modify the parameters in the Definition page.

To delete a global alias, select it in the Databases page, right-click, and select **Delete**. You cannot delete an alias when it is open. (An alias is open when a green box appears surrounding the bitmap to the left of the alias.) To close an alias, select it, right-click, and select **Close**.

Configuring Drivers

Drivers are configured from the Configuration page of the BDE Administrator. To change a particular driver's configuration, select that driver, then modify its attributes in the Definition page.

Native drivers, which include both local and SQL Links, are installed when you install Delphi. You cannot add new native drivers from the BDE Administrator. These can only be installed from the installation disk supplied by INPRISE, or one created by InstallShield Express.

The driver settings you enter on the Configuration page of the BDE Administrator provide default settings for global aliases. However, when configuring a particular alias, you can override any of these parameters.

Database Applications That Don't Use the BDE

Under normal circumstances, the database applications that you create using Delphi use the BDE. However, there are alternatives that don't require the BDE. These include:

- Applications that make use of explicit file I/O. For example, you can define your own file structures, then take responsibility for reading and writing this data manually. The drawback to this type of application is that you cannot use the data access and data control

components that ship with Delphi. Similarly, if you want the data to be accessed in a multi-user environment, you are responsible for programming the necessary file- and record-locking mechanisms.

- Applications that make use of an alternative database engine. Several third-party developers have created add-on products for Delphi that provide for data access. These either support file types not supported directly by the BDE (such as Clipper or Btrieve) or have a smaller footprint than the BDE. Some of these third-party products can be used with Delphi's data access and data control components, while others provide their own component set for data access and manipulation.
- Applications that use the ClientDataSet component that ships with the Client/Server and Enterprise editions of Delphi. This component, which can be used in place of other DataSet components, permits the reading and writing of single-user flat files. The ClientDataSet component relies on a 150KB DLL named DBCLIENT.DLL, but doesn't make use of the BDE.
- Client applications that use INPRISE's MIDAS (Multi-Tier Distributed Application Services) technology. With MIDAS, your Delphi client application receives data over a TCP/IP connection, or through the use of sockets. The data is provided by an application server, which you also write using Delphi. While the application server does make use of the BDE, the client application does not. Client applications created using MIDAS are often referred to as thin clients, because they require less configuration and fewer files (specifically, no BDE).

Delphi Database Components

BDE-based database applications written in Delphi rely on both data access and data control components that ship with Delphi. The relationship between these components is depicted in Figure 4. The following sections describe each of these sets of components.

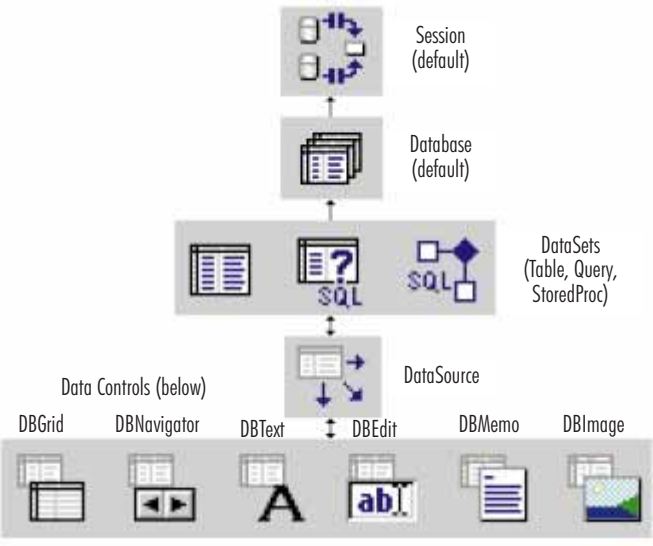


Figure 4: The relationship between the various data access components and data aware controls.

Session. The Session component represents a connection to the BDE. Conceptually, it represents the user for the purpose of file and record locking. Every database application has its own session, and Delphi creates this session automatically. (The automatically-created session component can be referenced by the instance variable named *Session*.) It is because of the session that two applications running on the same machine are seen by the BDE as two different users (different sessions equate to different users). Even if one user runs two copies of a given application, each copy will have a different session, and therefore will appear to the BDE as two users.

There is a Session component on the Data Access page of the Component palette. One of the few times you ever need to add Session components to a single project is when that project is multi-threaded and data needs to be accessed by more than one thread. Because each thread accesses data from a separate session, the BDE treats the threads as separate users. This provides a consistent mechanism for the resolution of record and table locking between threads.

Database. The Database component provides a pointer to a directory (on a stand-alone machine or a LAN), or to a remote database server. Every database application will have at least one Database component, and more if data in multiple directories and/or servers is being accessed.

If you use a DataSet, but have not specifically associated it with a Database component, Delphi will create one Database component for each directory and/or server being accessed. It will do this in response to an attempt to open the DataSet.

Unlike a Session component, you may need to use a Database component if you want to control access to a database. For example, if you want to define custom parameters for access to a database server, store a username and password, or explicitly control transactions, you will probably use at least one Database component in your application.

In a single-threaded application, all Database components are associated with a single Session component. However, many different DataSets can use one Database.

Database components, like Session components, are globally available within your application. In other words, if your Database component appears on an auto-created form or the main form, it's available to all forms and data modules in the application, without the need for a corresponding **uses** clause statement.

DataSets. DataSets, which include the Table, Query, and StoredProc components, are associated with individual tables or SQL files. (In the case of Query and StoredProc components, it's possible for these entities to be associated with more than one, or even no, underlying data file. This is the exception, however, rather than the rule.)

Each DataSet has its own cursor. Furthermore, you use the methods and properties of the DataSets to get information about — and to control — a table. For example, using a DataSet you can read data from a table, modify existing records, insert new records, or delete records.

From a developer's standpoint, DataSets greatly simplify access to data. Specifically, they encapsulate calls to the BDE from within an easy-to-use interface. For example, if you wanted to create a pointer to a record using BDE API calls, you would have to call at least five different BDE API functions. By comparison, calling a Table component's *Open* method performs the identical operation.

DataSource. The DataSource component provides for interaction between DataSets (which access data), and data controls (which provide the user interface). This interaction is two-way. For example, if a user tries to type a character into a DBGrid component, the DBGrid will first inquire through the DataSource whether the Table is in a state that permits editing (such as dsEdit or dsInsert). If the Table is not, the DataSource will attempt to place the Table in the dsEdit state. If successful, the DataSource will accept the data entered into the DBGrid. Likewise, when a Table pointer is changed to point to a new record, the DataSet informs the DataSource, which instructs the DBGrid to repaint itself.

Beginning Delphi database developers often wonder why the database functionality represented by the DataSet and DataSource components are not combined into a single component. The answer is that, from a component design standpoint, it is often better to create two simple components that perform well-defined tasks, as opposed to a more complex single component. The simpler components are easier to debug and use.

Because DataSource components are used primarily for managing the interaction between data controls and DataSets, you rarely need to use a DataSource for data access that is entirely programmatic. In other words, if you have no user interface, you probably don't need a DataSource.

Data Control Components. Data control components (i.e. those that appear on the Data Controls page of the Component palette) are used to display or manipulate data being pointed to by a DataSet. For example, using either a DBGrid or DBEdit, a user can view and edit data. Using a DBNavigator component, a user can navigate a DataSet, as well as insert and delete records.

Delphi's data access and data control components play an important role in rapid application development. Specifically, rather than having to explicitly program how a user interacts with data — including fetching, detecting changes, and saving data — you can simply place the appropriate data controls and data access components onto your form, and most of this interaction will be taken care of for you.

Conclusion

Delphi has established itself as one of the leading development tools for building database applications for the Windows platform. Essential to this role is the Borland Database Engine, which serves as an independent layout for all data access. In addition, the data access and data control components provide a seamless and easy-to-use interface to the BDE, permitting you to quickly build complex interfaces.

In next month's "DBNavigator," we'll take a look at configuring data access components, including tables and queries. Δ

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

By Major Ken Kyler and Alan C. Moore, Ph.D.

Delphi and TAPI

Part II: Building a Telephony Application

Last month, we discussed issues related to line communication in general and telephony in particular. Introducing the four essential Windows communications classes — Win32 Communications API, WAVE API, Messaging API (MAPI), and the Telephony API — we concentrated on the Telephony API, or TAPI. We examined some of the basic TAPI functions in detail, and demonstrated how to use them to initiate and manage phone calls.

This month, we'll go further, building on that foundation in several ways. We'll show you how to determine the existing capabilities of the particular TAPI implementation and monitor changes to the COMM port. We'll also show you how to access one of the dialog boxes included in TAPI. To accomplish this, we'll have to take some additional steps that were unnecessary in the first article.

Opening Lines of Communication

You'll recall that last month, we explained that initializing TAPI is a two-step process; first, you call *LineInitialize*, then you call *LineNegotiateAPIVersion*. Among other things, *LineInitialize* returns a handle to the usage instance of TAPI in its first parameter, *hLineApp*. It also sets a pointer to a callback function. The following statement from our sample application accomplishes this:

```
ErrNo := LineInitialize(@FLineApp, MainInstance,
                        LineCallback, '',
                        @FnumDevs);
```

We also need to call *LineNegotiateAPIVersion* so the application knows which version of TAPI to use. As we discussed last month, since it has several versions, each with considerably different capabilities, TAPI is one of the few Windows APIs that requires version negotiation. This will become evident when we examine some of the TAPI data structures. Now we're ready to explore some new territory.

When we open a line for communications, we could have a number of intentions: from conducting a simple voice conversation to downloading a file. The most important function in determining what communications operations our application will be able to perform is *LineOpen*. This function is declared in TAPI.pas as:

```
function LineOpen(hLineApp: HLINEAPP;
         dwDeviceID: DWORD;
  lphLine: LPHLINE; dwAPIVersion: DWORD;
  dwExtVersion: DWORD;  dwCallbackInstance:
                        DWORD;
  dwPrivileges: DWORD; dwMediaModes: DWORD;
  const lpCallParams: LPLINECALLPARAMS):
                      LONG;
```

The first parameter, *hLineApp*, is a handle. The second, *dwDeviceID*, is a line-device identifier. The third, *lphLine*, is the line device's Windows handle. The fourth and fifth parameters are negotiated: *dwAPIVersion* gets the proper API version, and *dwExtVersion* returns the extended version. The sixth parameter, *dwCallbackInstance*, is an application's instance data, which it passes to a callback function, *lineInitialize*. The next two, *dwPrivileges* and *dwMediaModes*, are probably the most important. The first determines how an application handles different kinds of calls; the latter specifies the types of calls. The final parameter, *lpCallParams*, points to a structure that provides Windows with criteria an application uses in selecting a

line. The LINECALLPARAMS record is defined in TAPI.pas, as shown in Figure 1.

You'll notice in this structure that many new options were added in version two of TAPI. Figure 2 describes the fields used by *LineOpen*. If you're making a data call, you're required to set the default values; in a voice call, this is optional. In the *FormCreate* method of TAPIU_2.pas, we initialize *FMediaMode* and *FBearerMode* as follows:

```
LPLINECALLPARAMS = ^TLINECALLPARAMS;
PLINECALLPARAMS = ^TLINECALLPARAMS;
TLINECALLPARAMS = record          // Defaults:
  dwTotalSize,                    // ---------
  dwBearerMode,                   // voice
  dwMinRate,                      // (3.1kHz)
  dwMaxRate,                      // (3.1kHz)
  dwMediaMode,                    // interactiveVoice
  dwCallParamFlags,               // 0
  dwAddressMode,                  // addressID
  dwAddressID: DWORD;             // (any available)
  DialParams: TLINEDIALPARAMS;    // (0, 0, 0, 0)
  dwOrigAddressSize,              // 0
  dwOrigAddressOffset,
  dwDisplayableAddressSize,
  dwDisplayableAddressOffset,
  dwCalledPartySize,              // 0
  dwCalledPartyOffset,
  dwCommentSize,                  // 0
  dwCommentOffset,
  dwUserUserInfoSize,             // 0
  dwUserUserInfoOffset,
  dwHighLevelCompSize,            // 0
  dwHighLevelCompOffset,
  dwLowLevelCompSize,             // 0
  dwLowLevelCompOffset,
  dwDevSpecificSize,              // 0
  dwDevSpecificOffset: DWORD;

{$IFDEF Tapi_Ver20_ORGREATER}
  dwPredictiveAutoTransferStates,
  dwTargetAddressSize,
  dwTargetAddressOffset,
  dwSendingFlowspecSize,
  dwSendingFlowspecOffset,
  dwReceivingFlowspecSize,
  dwReceivingFlowspecOffset,
  dwDeviceClassSize,
  dwDeviceClassOffset,
  dwDeviceConfigSize,
  dwDeviceConfigOffset,
  dwCallDataSize,
  dwCallDataOffset,
  dwNoAnswerTimeout,
  dwCallingPartyIDSize,
  dwCallingPartyIDOffset: DWORD;
{$ENDIF}

end;
```

Figure 1: LINECALLPARAMS, as defined in the TAPI unit.

```
FMediaMode  := LINEMEDIAMODE_DATAMODEM;
FBearerMode := LINEBEARERMODE_VOICE;
```

Then, we make certain these default values are reflected in our two combo boxes with these statements:

```
// Default to LINEMEDIAMODE_DATAMODEM.
cboxMediaMode.ItemIndex := 3;
// Default to LINEBEARERMODE_VOICE.
cboxBearerMode.ItemIndex := 0;
```

Finally, we set the following values:

```
with FLineCallParams do begin
  dwTotalSize := SizeOf(FLineCallParams);
  dwBearerMode := FBearerMode;;
  dwMediaMode := FMediaMode;
end;
```

Having initialized these and other values in the *FormCreate* method, we can now call the *LineOpen* function and get things rolling. In the expanded application included with this article (see end of article for download details), we've added this function to the *CreateCallManager* method with the following statement:

```
if cbLineMapper.Checked then
  // Automatically select the device.
  ErrNo := LineOpen(FLineApp, LINEMAPPER, @FLine, FVersion,
                    0, 0, LINECALLPRIVILEGE_NONE,
                    FMediaMode, @FLineCallParams)
else
  ErrNo := LineOpen(FLineApp, FDev, @FLine, FVersion,
                    0, 0, LINECALLPRIVILEGE_NONE,
                    FMediaMode, nil);
```

The use of the *LineOpen* function and its LINEMAPPER parameter demonstrate a very different approach from the one we took in the last article. If *cbLineMapper* is checked, we let the system select its default line; otherwise we choose from the list of available devices. LINEMAPPER identifies the first device capable of making the kind of call we want. Let's see how we determine these capabilities.

Is the Line Capable?

Once we've made this function call and have called the *CreateCallManager* method, we can go about the business of determining a line device's capabilities. Information about these capabilities is stored in another large TAPI structure, LINEDEVCAPS. This structure is defined in TAPI.pas as shown in Figure 3.

We can use the TAPI *lineGetDevCaps* function to retrieve this information. Because there can be more than one logical line on

Meaning	Use of Field
dwTotalSize	Informs Windows how much memory is available for returning TAPI information.
dwBearerMode	Indicates the general class of data transfer the line supports, including passthrough mode.
dwMinRate	Provides information bandwidth requirements; not all telephony drivers use this field.
dwMaxRate	Provides information bandwidth requirements; not all telephony drivers use this field.
dwMediaMode	Specifies the media mode the application will support. Can be one of the LINEMEDIAMODE_ constants (see TAPI.pas), or an extended media mode value defined by a service provider.
dwCallParamFlags	Flags that control the behavior of a modem when making a call.

Figure 2: *LineCallParams* fields used by the *LineOpen* function.

a computer, we call *lineGetDevCaps* for each line to identify one that supports the type of data with which we want to work. In our extended application, we call this function in the *EnumerateDevices* method.

Let's examine the structure itself. As in LPLINECALLPARAMS, *dwTotalSize* provides Windows with the amount of memory needed for the structure. Because this structure is variable in size, the next two fields, *dwNeededSize* and *dwUsedSize*, indicate how much of the structure actually contains relevant information. These two fields are set by Windows. You'll notice that several other fields are the same as, or similar to, those in LPLINECALLPARAMS, e.g. *dwBearerModes* and *dwMediaModes*.

```
LPLINEDEVCAPS = ^TLINEDEVCAPS;
PLINEDEVCAPS = ^TLINEDEVCAPS;
TLINEDEVCAPS = record
  dwTotalSize,
  dwNeededSize,
  dwUsedSize,
  dwProviderInfoSize,
  dwProviderInfoOffset,
  dwSwitchInfoSize,
  dwSwitchInfoOffset,
  dwPermanentLineID,
  dwLineNameSize,
  dwLineNameOffset,
  dwStringFormat,
  dwAddressModes,
  dwNumAddresses,
  dwBearerModes,
  dwMaxRate,
  dwMediaModes,
  dwGenerateToneModes,
  dwGenerateToneMaxNumFreq,
  dwGenerateDigitModes,
  dwMonitorToneMaxNumFreq,
  dwMonitorToneMaxNumEntries,
  dwMonitorDigitModes,
  dwGatherDigitsMinTimeout,
  dwGatherDigitsMaxTimeout,
  dwMedCtlDigitMaxListSize,
  dwMedCtlMediaMaxListSize,
  dwMedCtlToneMaxListSize,
  dwMedCtlCallStateMaxListSize,
  dwDevCapFlags,
  dwMaxNumActiveCalls,
  dwAnswerMode,
  dwRingModes,
  dwLineStates,
  dwUUIAcceptSize,
  dwUUIAnswerSize,
  dwUUIMakeCallSize,
  dwUUIDropSize,
  dwUUISendUserUserInfoSize,
  dwUUICallInfoSize: DWORD;
  MinDialParams,
  MaxDialParams,
  DefaultDialParams: TLINEDIALPARAMS;
  dwNumTerminals,
  dwTerminalCapsSize,
  dwTerminalCapsOffset,
  dwTerminalTextEntrySize,
  dwTerminalTextSize,
  dwTerminalTextOffset,
  dwDevSpecificSize,
  dwDevSpecificOffset,
  dwLineFeatures: DWORD;

{$IFDEF Tapi_Ver20_ORGREATER}
  dwSettableDevStatus,
  dwDeviceClassesSize,
  dwDeviceClassesOffset: DWORD;
{$ENDIF}

end;
```

Figure 3: The TAPI structure, LINEDEVCAPS.

Several of the other fields are quite important. The *dwNumAddresses* field contains the number of addresses a particular logical device supports; the field *dwAddressModes* indicates the manner in which the application identifies a line's address; and the field *dwStringFormat* (which we use in the sample application) indicates whether the string format is ASCII or Unicode. Because we aren't supporting Unicode in this application, we exit if we discover the format is Unicode. Otherwise, we populate the *cboxDevices* combo box with the names of the available line devices.

Monitoring a Modem: Beyond TAPI

Often, we want or need to keep track of what a modem is doing. To accomplish this, we must use lower-level communications API functions that aren't part of TAPI. The *GetCommModemStatus* function is particularly useful. This basic Windows communications function returns a modem's control-register values. It's declared in Windows.pas as:

```
function GetCommModemStatus(hFile: THandle;
  var lpModemStat: DWORD): BOOL; stdcall;
```

The parameters to *GetCommModemStatus* are *hFile*, the handle of the communications port; and *lpModemStat*, which points to the current status of the modem, stored in a control-register. Before we can call this function, we must set things up properly.

First, we need to create a thread in which to do the monitoring while the rest of our application continues to function. (See Jon Jacobs' article "Threads Simplified" in the December, 1997 issue of *Delphi Informant* for an excellent introduction of working with threads.) The simple thread class is defined in the CommStatus.pas:

```
type
  TCommStatus = class(TThread)
  protected
    procedure Execute; override;
  end;
```

In the **private** declarations of our main form class, we declare an instance of this thread as:

```
FCommStatusThread: TCommStatus;
```

We instantiate it (in suspended form) in the *Form.Create* method with the line:

```
FCommStatusThread := TCommStatus.Create(True);
```

Once we have successfully opened a line for communications, we activate the thread with the following line of code:

```
FCommStatusThread.Resume;
```

Event Constant	Description
EV_BREAK	A break was detected during input.
EV_CTS	State of the CTS (clear-to-send) signal changed.
EV_DSR	State of the DSR (data-set-ready) signal changed.
EV_ERR	A line-status error occurred (CE_FRAME, CE_OVERRUN, and CE_RXPARITY).
EV_RING	Ring indicator detected.
EV_RLSD	State of the RLSD (receive-line-signal-detect) signal changed.
EV_RXCHAR	Character received and put in input buffer.
EV_RXFLAG	Event character (specified in the device) received and put in the input buffer.
EV_TXEMPTY	Last character in the output buffer sent.

Figure 4: *lpEvtMask* modem events and their meanings.

Once we have terminated our call, we close the thread with:

```
FCommStatusThread.Terminate;
```

Within the thread, all the work takes place in the *Execute* method. Here we call two low-level communications API functions, *SetCommMask* and *WaitCommEvent*. Let's discuss each. The first, *SetCommMask*, is defined in Windows.pas as:

```
function SetCommMask(hFile: THandle;
                     dwEvtMask: DWORD): BOOL; stdcall;
```

Its first parameter, *hFile*, is the handle of the communications port. The **var** parameter, *lpEvtMask*, is a mask containing one or more events to monitor. The various modem events constants and their meanings are shown in Figure 4.

Once we've told Windows the kind of communications events to monitor, we have to wait for one to occur. (Do you see why we needed to do this in a separate thread?) The *WaitCommEvent* function waits for an event to occur on a particular communications device. The *WaitCommEvent* function is defined in Windows.pas as:

```
function WaitCommEvent(hFile: THandle; var lpEvtMask: DWORD;
  lpOverlapped: POverlapped): BOOL; stdcall;
```

The monitored events contained in the *lpEvtMask* event mask are associated with the device handle, *hFile*. The *lpEvtMask* parameter points to a 32-bit variable that receives a mask indicating the type of event that occurred. If there's an error, zero is returned; otherwise, it's one of the values shown in Figure 4.

The next parameter, *lpOverlapped*, is either **nil** or points to the address of an overlapped structure. This structure is required if *hFile* was opened with FILE_FLAG_OVERLAPPED. In that case, the *lpOverlapped* parameter may not be **nil**. It must point to a valid OVERLAPPED structure. Under these circumstances, you can use the *CreateFile* function to return the handle. The *WaitCommEvent* function can return various values. If the function fails, it returns a value of zero.

```
procedure TCommStatus.Execute;
var
  dwEvent: DWord;
  dwStatus: DWord;
begin
  dwEvent := 0;
  SetCommMask(FPort, EV_DSR or EV_CTS or SETDTR);
  repeat
    WaitCommEvent(FPort, dwEvent, nil);
    GetCommModemStatus(FPort, dwStatus);
    case dwEvent of
      EV_DSR: Form1.SetBitmap(Form1.DSR, green);
      SETDTR: Form1.SetBitmap(Form1.DTR, green);
      EV_CTS: Form1.SetBitmap(Form1.CTS, green);
    end;
    // Form1.SetBitmap(Form1.AA, green);
  until Terminated;
end;
```

Figure 5: The entire *Execute* method.

To get extended error information, you should call the *GetLastError* function.

As we've seen, the *WaitCommEvent* function monitors events on a particular communications resource. You can set or query the current event mask of the communications resource by using the *SetCommMask* and *GetCommMask* functions. If a requested overlapped operation cannot be completed immediately, the function returns False and *GetLastError* returns ERROR_IO_PENDING. This indicates that the communications operation is continuing to execute in the background. When this happens, the system will set the *hEvent* member of the OVERLAPPED structure to the not-signaled state before the *WaitCommEvent* function returns. You can call *GetOverlappedResult* to determine the success or failure of the operation. The variable pointed to by the *lpEvtMask* parameter indicates the event that occurred. In the sample application, we used the non-overlapped approach.

Putting It All Together

In the CommStatus unit, we use these API functions in the *Execute* method. As we stated earlier, *GetCommModemStatus*' second parameter, *lpModemStat*, points to the current status of the modem. The entire *Execute* method is shown in Figure 5.

You'll notice that certain communications events are associated with particular modem light indicators. We have tried to imitate the indicator lights you generally see on external modems. If you study the source code, you'll see how we initialize the modem lights and also notice that we are able to change some of them in the main file (and not the thread). Figure 6 shows the expanded application in action, with some of the lights active.

Now that we've explored some low-level communications techniques to query the status of a modem, let's return to TAPI and take a look at one of its built-in dialog boxes.

Back to TAPI: *lineTranslateDialog*

In addition to the functionality we've already discussed, TAPI also provides Windows with a number of dialog boxes for setting or viewing the properties of various devices. One of the most commonly used is the Dialing Properties dialog box.

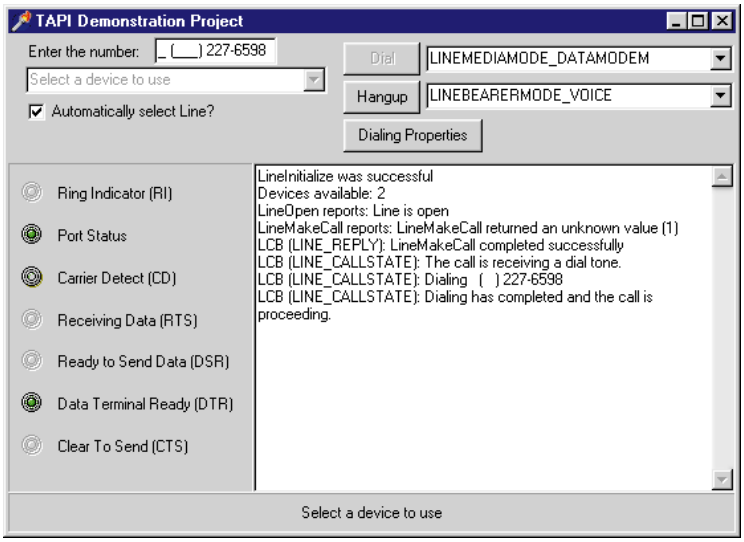


Figure 6: The new sample application with modem lights activated.

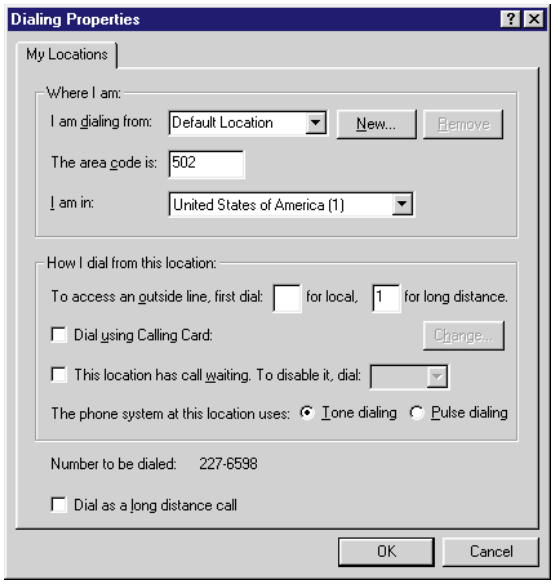


Figure 7: TAPI's *lineTranslateDialog* in action.

The *lineTranslateDialog* function displays a dialog box similar to the Dialing Properties dialog box, but includes the dialable address (see Figure 7). If you study the TAPI.pas source, you'll find this dialog function comes in two flavors: *lineTranslateDialogA* and *lineTranslateDialogW*; the former for ASCII characters and the latter for wide characters (Unicode). We use *lineTranslateDialogA* in this application; it's defined in TAPI.pas as:

```
function lineTranslateDialogA(hLineApp: HLINEAPP;
  dwDeviceID: DWORD; dwAPIVersion: DWORD; hwndOwner: HWND;
  lpszAddressIn: LPCSTR): LONG; stdcall;
```

The first parameter, *hLineApp*, is the application handle returned by *lineInitializeEx*. If the application has not yet called *lineInitializeEx*, it can set the *hLineApp* parameter to **nil**. The second parameter, *dwDeviceID*, is an identifier for the line device upon which the call will be dialed. The third parameter, *dwAPIVersion*, indicates the highest TAPI version the application supports (not necessarily the value negotiated by *lineNegotiateAPIVersion*). The parameter *hwndOwner* is the handle to the window to which the dialog box is to be attached. The next parameter, *lpszAddressIn*, is a PChar containing the phone number shown in the lower portion of the dialog box. The phone number must be in canonical format; if it isn't, the phone number won't be displayed in the dialog box. This pointer can be set to **nil**, in which case the phone number won't be displayed.

The dialog box can return various values. If it returns zero, the function was successful. A negative error number in the return values indicates an error. The possible return values are:

- LINEERR_BADDEVICEID
- LINEERR_INVALPARAM
- LINEERR_INCOMPATIBLEAPIVERSION
- LINEERR_INVALPOINTER
- LINEERR_INIFILECORRUPT
- LINEERR_NODRIVER
- LINEERR_INUSE
- LINEERR_NOMEM
- LINEERR_INVALADDRESS
- LINEERR_INVALAPPHANDLE
- LINEERR_OPERATIONFAILED

Multiple instances of this dialog box can be opened using TAPI version 2.0 or later. In earlier TAPI versions, LINEERR_INUSE is returned if the dialog box is already displayed by another application, and TAPI brings the existing dialog box to the front.

We call *lineGetTranslateCaps* after this function to get any changes the user made to the telephony address translation parameters. You can call *lineTranslateAddress* to obtain a dialable string based on the user's new selections. If you call any function related to address translation (*lineGetTranslateCaps* or *lineTranslateAddress*) and the function returns LINEERR_INIFILECORRUPT, you should call *lineTranslateDialog*. The *lineTranslateDialog* function detects the errors and corrects them. It also reports to the user the action taken.

Where to Go from Here

We've gone considerably further than in the first article, but we've still only scratched the surface. We've concentrated on simple voice calls to keep the whole discussion straightforward. There are many other types of calls (voice and data) for which TAPI provides excellent support. You can study the source code for these units, especially the TAPI.pas unit, and extend the sample application's functionality by implementing additional TAPI functions.

We're not quite finished. What we've demonstrated in the sample application is fine as an introduction. It's enabled us to test and show many basic TAPI functions. However, if we want to use this kind of functionality in more than just one application, we will find ourselves doing a lot of cutting and pasting, to say the least. What we need to do is wrap all of this functionality in a TAPI class or component that will be easily accessible in any application we write. We'll tackle that task next month. Δ

Important note: *The TAPI unit has been brought to you by the volunteers of the JEDI project. To learn more about the JEDI project, please visit http://www.delphi-jedi.org.*

The API Calls

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\AUG\DI9808AM.

Major Ken Kyler is the Air National Guard Systems Analyst for the Defense Integrated Military Human Resources System (DIMHRS). He has been programming with Delphi for two years. He is also a free-lance technical writer with articles published in several Delphi magazines. You can reach him at KylerK@PR.OSD.MIL.

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.

By Robert Vivrette

Better Coding through APIs

Napping, Driving, and Personable Cursors

It happens all the time: You're adding a feature to your latest programming work of art, when what you thought would be fairly simple quickly becomes a code morass. "There has to be a simpler way!" you exclaim. There often is. In fact, my Delphi philosophy is: "If it takes more than two or three lines of code to achieve a desired effect, you're probably doing it wrong." I live by this statement, and I have seen it hold true hundreds — if not thousands — of times. I'll complete some routine, then sit back and look at it. "Hmm ... looks a little overweight!" Then I start trimming it down: finding simpler ways to accomplish the same thing, looking for API calls to eliminate much of my code. After a few passes, pruning here and there, it happens! Two or three lines of code remain.

Windows has been around for quite a while, so it's unusual that a programmer will write something completely unique, something never coded before. The programming APIs available to Windows developers are substantial, but we can't always remember them all, and can wind up re-coding what already exists in an API.

In this spirit, this month's "At Your Fingertips" is devoted to existing API routines that sometimes fall through the cracks: simple and useful routines that, in my experience, few developers even realize exist, yet can save considerable programming effort.

Take a Nap!

Back in the days of Turbo Pascal and DOS programming, programmers had a wonderful little routine named *Delay* that would pause execution of the program for a specified period. Then Windows development came along, and *Delay* was gone. <sniff>. Or was it? It turns out the Windows API has a *Sleep* routine that performs the same function.

In a nutshell, the *Sleep* procedure pauses execution of the current thread for an indicated number of milliseconds. It takes a single DWORD parameter that specifies the number of milliseconds to wait before continuing. A value of 1000 equals one second, so to delay for three seconds, you would use *Sleep(3000)*. Specifying a value of INFINITE (i.e. $FFFFFFFF) causes the process to sleep forever (now that's useful). A value of zero causes the thread to relinquish any time left on its current time slice to any other running thread with the same priority.

Be aware of some of the behavior you may see using *Sleep*. Because you are stopping CPU cycles to your application's primary thread, your application's message processing loop will take a nap as well. This means that mouse messages won't be processed, keyboard input will be on hold, and Timers will stop working, to name just a few. In most cases, however, this is what you want anyway.

There's more. The *SleepEx* function extends this capability to allow the sleeping thread to be alerted before the completion of its sleep delay. This would enable you to put the thread to sleep pending the completion of some other supported event, such as the I/O functions *ReadFileEx* and *WriteFileEx*. There's more to the *SleepEx* function than I can go into here, but Delphi Help discusses it fairly well.

Show a Little Drive

Sometimes, developers need to obtain a list of all available drive letters on the machine. I've seen a lot of approaches to this, including building a loop from 'A' to 'Z' and looking at each drive individually. The problem with most of these approaches is that validating the existence of a drive in this manner often involves accessing the drive itself. This can generate unwanted system errors for an empty floppy drive.

You could use a DriveComboBox component and examine its *Items* property to see what drive letters it's populated with. The disadvantage here is that DriveComboBox holds a lot more code than you need. It's also a visible control, so if you aren't using it for any other purpose, you will need to stash it out of the way and set its *Visible* property to False. Sloppy!

A much better solution is to use the Win32 API functions *GetLogicalDrives* and *GetLogicalDriveStrings*. *GetLogicalDrives* returns a 32-bit value that is bitmasked to indicate the valid drives in the system. In the return value, bit position zero represents drive A, the next bit represents drive B, and so on. If a bit is set to 1, the drive is present in the system. This procedure adds the letters of all available drives to *ListBox1*:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Drvs: DWord;
  A   : Integer;
begin
  Drvs := GetLogicalDrives;
  for A := 0 to 31 do
    if (Drvs and (1 shl A)) > 0 then
      ListBox1.Items.Add(Chr(A+65));
end;
```

Note the use of the **shl** keyword to mask out only the bit position I'm interested in. By taking a '1' and shifting it left a number of positions, I can mask out just the bit position I am interested in using the **and** keyword.

The *GetLogicalDriveStrings* function is similar, but instead returns a string representing all the valid drives on the system. After each drive reference, there is a null character; at the end of the entire string, there is a pair of null characters. This procedure can be used to extract the values returned in a doubly null-terminated string:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Buffer: array[0..500] of Char;
  TmpPC : PChar;
begin
  GetLogicalDriveStrings(SizeOf(Buffer), Buffer);
  TmpPC := Buffer;
  while TmpPC[0] <> #0 do begin
    ListBox1.Items.Add(TmpPC);
    TmpPC := StrEnd(TmpPC) + 1;
  end;
end;
```

Alternatively, you could use the *GetDriveType* function to determine the type of drive that *GetLogicalDrives* and *GetLogicalDriveStrings* returns (see Figure 1).

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Buffer: array[0..500] of Char;
  TmpPC : PChar;
  Typ   : Integer;
begin
  GetLogicalDriveStrings(SizeOf(Buffer),Buffer);
  TmpPC := Buffer;
  while TmpPC[0] <> #0 do begin
    Typ := GetDriveType(TmpPC);
    with ListBox1.Items do
      case Typ of
        DRIVE_REMOVABLE : Add(TmpPC + ' (Removable)');
        DRIVE_FIXED     : Add(TmpPC + ' (Fixed)');
        DRIVE_REMOTE    : Add(TmpPC + ' (Remote)');
        DRIVE_CDROM     : Add(TmpPC + ' (CD-ROM)');
        DRIVE_RAMDISK   : Add(TmpPC + ' (RAM-Disk)');
      else
        Add(TmpPC+' (Unknown)');
      end;
    TmpPC := StrEnd(TmpPC)+1;
  end;
end;
```

Figure 1: Use the *GetDriveType* function to determine the type of drive that *GetLogicalDrives* and *GetLogicalDriveStrings* returns.

Cursors Are Nice People Too

Most good Delphi programmers know they can create custom cursors and load them into the *Screen.Cursors* array. But did you know that you can also load animated cursors? You can achieve this by using the *LoadCursorFromFile* API function. This function returns a handle to the loaded cursor, which can be a standard cursor file (*.CUR), an Animated Cursor (*.ANI), or one of the system cursors.

The technique couldn't be easier. Simply define a cursor constant (anything greater than zero and not conflicting with another cursor constant) in the **const** section of your unit:

```
const
  crMyCursor = 5;
```

Then add this code to the *FormCreate* method:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Screen.Cursors[crMyCursor] :=
    LoadCursorFromFile('C:\WINNT\Cursors\sendmail.ani');
  Panel1.Cursor := crMyCursor;
end;
```

A standard Windows 95 or Windows NT installation will have a number of animated cursor files located in the Windows\Cursors or WinNT\Cursors directory, respectively.

Conclusion

The various Windows APIs are chock-full of useful routines available to you as a Delphi programmer: It's just a matter of knowing where to look. The best way to find those interesting new functions is to look up something you're familiar with in the Win32 API Help file, and then click on the "Group" link at the top of the page. This will show you a list of routines that fall into the same general category. Scan through them and you'll be amazed at what you find.

And remember: If it takes more than two or three lines of code to do anything in Delphi, there's probably an API routine that does it all for you! Δ

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

By Steve Garland

ODBCExpress

Catch the High-speed Train to Database Access

Most of the non-Delphi world of C++, Visual Basic (VB), and PowerBuilder developers access client/server databases using ODBC. Delphi's BDE allows the same access, but why use the BDE when you can go to a pure Pascal/ODBC solution using the ODBCExpress components from Datasoft? What do you gain? Speed, control, and no DLLs to distribute with your applications. What do you lose? Over 1MB of BDE DLLs.

Why ODBC?

A couple of years ago, I was thumbing through some VB magazines (to see how the other side lives), and noticed an article on the "fastest" way to access databases. DAO and RDO were discussed, but ODBC smoked them all in leanness and performance. ODBC had earned a bad name with me because of its early, slow implementation, but I was beginning to wonder if I should give it another look. If the VB, C++, and PowerBuilder crowds were satisfied with using ODBC API calls, why couldn't I do the same?

After I started using ODBC and the BDE with SQL Anywhere, I was running into several "gotchas" with INPRISE (nee Borland) and Sybase pointing fingers at each other. One of the problems was getting SQL Anywhere to handle BLOBs greater than 32KB using Delphi/ODBC/BDE. I knew that SQL Anywhere could handle this with pure ODBC, so what was I missing? I needed to take the BDE out of the equation.

Writing directly to the ODBC API seemed like a lot of work, until I came upon a South African company that had used Delphi and ODBC to develop a document management application using Microsoft SQL Server as their back end. That company was Datasoft, and the product was ODBCExpress. I soon had SQL Anywhere eating large text files like candy, and I no longer heard that rising rumble of the BDE DLLs loading. I started to like ODBC.

ODBC was a new creature to me, so I quickly went out and found a book by the ODBC architect, Kyle Geiger, entitled *Inside ODBC* (from Microsoft Press). Although this book was published in 1995, I still highly recommend it as an excellent introduction to client/server programming. There are interesting discussions about the concept of cursors and their history, file server versus client/server computing, and a fascinating byte-by-byte account of what actually happens on a query to SQL Server. Not only the "hows" of ODBC are discussed, but also the "whys," including some interesting industry history.

I'm going to sketch out the basics of ODBC for you and, at the same time, detail the way that Datasoft implemented the ODBC API in ODBCExpress.

What Is ODBC?

ODBC is Microsoft's set-oriented API for accessing databases. The set-oriented part is important as ODBC expects data sources to be able to handle SQL. This means the data must reside in tables and have the concept of rows and columns. (As a side note, Microsoft has realized that there is other data that doesn't fit the row/column metaphor, and has created OLEDB for that. ODBC becomes a subset of OLEDB in their current scheme.)

ODBC is a specification, not an implementation, and many who remember early ODBC drivers as slow and buggy should wake up and smell the new drivers — especially the ones for Microsoft SQL Server and SQL Anywhere. The Microsoft SQL Server drivers are so fast that Microsoft uses them for performance

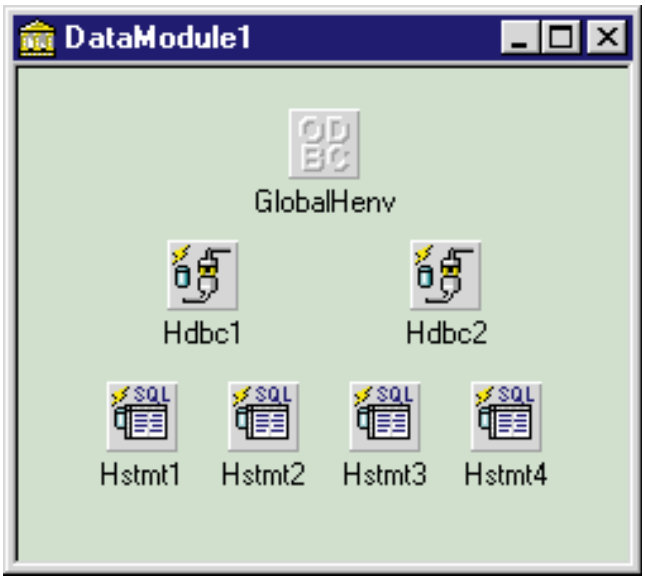


Figure 1: The building blocks for ODBC programming as implemented by ODBCExpress. Note that the GlobalHenv component is now automatically instantiated at the unit level.

benchmarks. It's also important to note that you don't need the Delphi Client/Server version to access Microsoft SQL Server using their ODBC driver. The promise of ODBC is that if you develop to the ODBC API, you can use the same code and tools against any and all ODBC drivers.

ODBC has a character all its own; it's not the BDE. If you are expecting ODBC to act as the BDE does with Paradox and dBASE tables, you'll be disappointed. This isn't to say that one is better than the other; they're just different.

ODBC uses the concept of Handles extensively. At the highest level, there is an Environment Global Handle (Henv) that is much like the Delphi Session component. The Henv Handle keeps track of the number of connections and their state, handles error messages that occur at the environment level, and helps to manage transactions. In earlier versions of ODBCExpress, there was an Henv component, but it's no longer needed, as it is created internally at unit initialization to be used by the other components (see Figure 1). Again, this is similar to the BDE's Session component.

The Connection Handle (Hdbc or Handle to a database connection) manages the network connection, or tracks directory and file information for file server databases. All function calls are routed from the back-end database through the Connection Handle. This is equivalent to the Database component of the BDE. ODBCExpress implements it in its *THdbc* class. Because they exist independently, you can use Hdbc components to access the same table isolated in a separate thread (see Figure 2).

It's at the *THdbc* level that you set the isolation level and other properties, such as cursor implementation. ODBC has had a feature called a Cursor Library since version 1.2. ODBC does this by handling the buffering of data and providing a "moving window" of buffers for the database. This enables users to "browse" their data in visual controls, such as grids. ODBC handles the buffering and allocating of memory to make this possible, so when the BDE allocates its own buffering scheme, it becomes redundant. Why use both?

The Statement Handle (Hstmt) is the worker bee of ODBC. All SQL statements and requests for metadata require an Hstmt, and each Hstmt is associated with one connection. Hstmt components handle the state information of any query activity. Hstmt is the low-level equivalent of *TQuery*. DBCExpress has a *THstmt* class that implements the Hstmt.

Pulling an Hdbc and Hstmt component onto a form allows you to query a data source or execute SQL statements (see Figure 3). At this level, you are just a thin wrapper away from the native ODBC API calls, and ODBCExpress adds little overhead to your application. Issuing update and insert statements using Hstmt components is lean and fast.

Many developers prefer to stay at this level to keep overhead down, and that's certainly a choice you have with ODBCExpress. However, many of us like to use visual data aware controls; after all, that's one of Delphi's great strengths. ODBCExpress has two sets of visual components available, but to explain them, we must first recap some Delphi VCL database history.

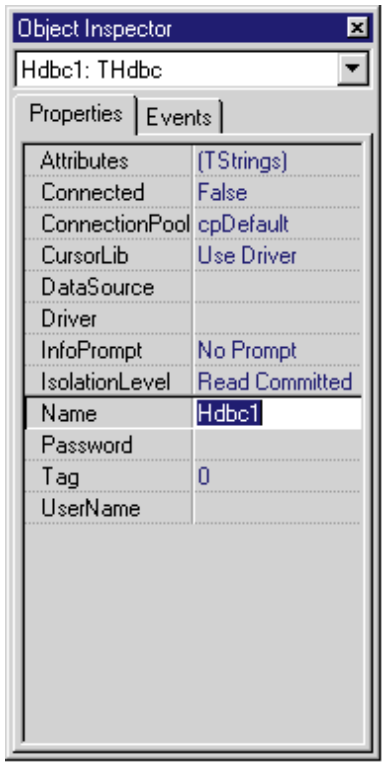


Figure 2: Object Inspector view of the *THdbc* component.

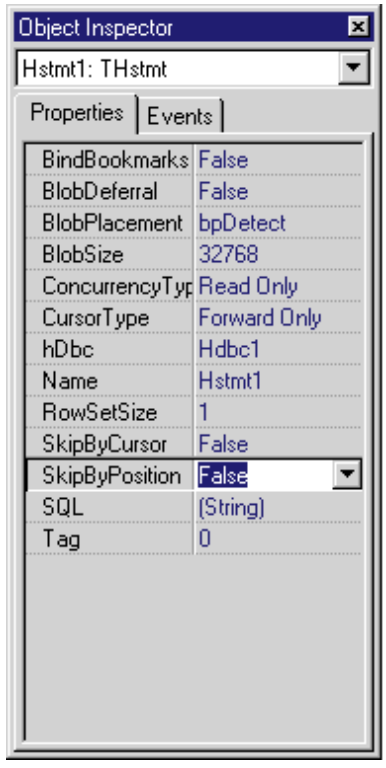


Figure 3: Object Inspector view of the *THstmt* component.

Delphi Database History

Delphi database access starts with the db unit (db.pas). In Delphi 1 and 2, there were specific BDE calls in the db unit. Therefore, third-party vendors who wanted to provide database access using engines and access to the Delphi data-aware controls other than the BDE were forced to "hack" their own versions of the db unit and distribute them as binary (dcu) files. This led to all kinds of problems with other third-party controls. This method basically intercepted the low-level BDE calls, and replaced them with proxy calls to their own engines. In addition to the BDE calls in the db unit, many parts were declared as private, and thus, impossible to extend using OOP.

When ODBCExpress 1.0 was released, there was no way to provide access to the native Delphi database controls without shipping a proprietary version of the db.dcu binary file. So, ODBCExpress came out with their own database controls, which you'll find installed on the OEControls tab of the Component palette. These controls were written from scratch by the ODBCExpress team, and let's just say that they have their own way of doing things. The grid, for instance, descends from *TStringGrid*, and, while it provides fast access, it doesn't provide design-time viewing of data, and generally doesn't behave like a native Delphi control.

So, the ODBCExpress team set out in the second version to hack the db unit and provide a way for us to use the native Delphi database controls. (They actually finished the product, but never released it.) The reasons for this are very telling in comparing ODBC and the BDE and how they differ in their philosophy. Reason one was that the BDE architecture was undocumented and extremely complex. The more interesting reason, however, is that the BDE and ODBC have different "access paradigms."

The important point is that ODBC is SQL and set-oriented, and the BDE is procedural and file-server oriented. BDE desktop database users are accustomed to using data-aware controls, and editing within a grid with no real concern for isolation-level or concurrency settings. The BDE handles it for them — including locking. This is certainly a wondrous thing, but for client/server development, one needs to make decisions as to concurrency and isolation issues, and live result sets cannot be expected at all times without some sort of trade-off. So, ODBC requires decisions to be made, as there is no "one-size fits all" philosophy. Many that come to try ODBCExpress complain that it doesn't work exactly like the BDE. Well, they're right. They are different creatures.

A lot of marketing hype promises that you can easily convert file-server databases to client/server databases, e.g. Paradox to Microsoft SQL Server. I just don't believe it. If you expect ODBC to act like the BDE, you'll be disappointed; but if you take the time to understand ODBC and use the tools as they were designed, you will find a fast, direct, and lean way to develop client/server applications.

There's a happy ending to this story. Borland/INPRISE heard many of us screaming for them to redesign the db unit, and they delivered with Delphi 3. Delphi 3 has a new and vastly improved version of the db unit that uses no BDE calls and allows developers to create their own *TDataSet* descendants that use the database controls that come with Delphi. Hallelujah! All the BDE calls were moved to the dbtables unit, and vendors (like ODBCExpress) could finally plug in their database engines by simply implementing the virtual abstract methods from the db unit. No more hacking necessary.

It All Comes Together: *TOEDataSet*

The Delphi 3 open dataset architecture paved the way for the ODBCExpress team to use their core ODBC components to provide a component that would make a Delphi database developer feel right at home. It's very similar to the Delphi Query component with a *SQL* property and a *Params* property with which to use stored procedures. But this is ODBC, so there is also an *Hstmt* property in case you need more granular control of the *TOEDataSet* (see Figure 4).

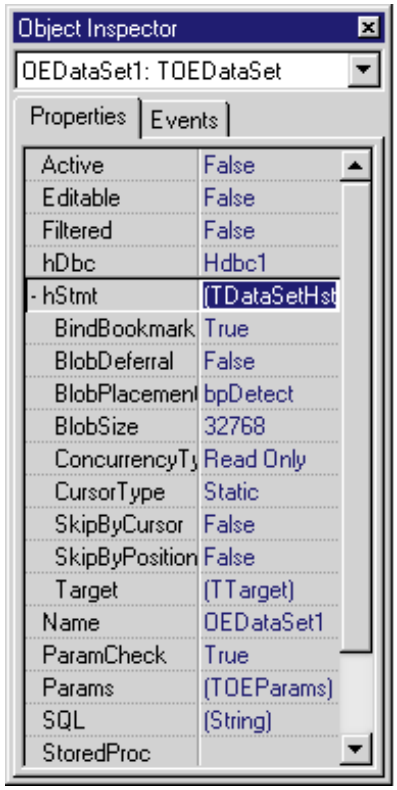


Figure 4: Object Inspector view of the *TOEDataSet*.

Pull a *THdbc* component, a *TOEDataSet*, a Delphi DataSource and Delphi DBGrid, and connect them just like you would a Query component. What could be easier? Populate the *SQL* property, set *Active* to True, and the DBGrid is populated. Use the Fields Editor to change display labels or other *TFields* of the DataSet.

All the data-aware controls can be used, including DBEdit and DBMemo. The difference is that *TOEDataSet* provides a thinner layer between you and ODBC than does the BDE. The ODBCExpress documentation states that by providing ODBC Driver cursor support, the *TOEDataSet* "eliminates the need for a resource-heavy front-end cache as used by existing database engines." By implementing all the low-level ODBC controls (*Thdbc*, *THstmt*), then providing a *TDataSet* descendant in the *TOEDataSet*, ODBCExpress gives you the best of both worlds: ease-of-use and granularity. At the ODBC level, you can still control the way you do updates, inserts, and deletions. You still get complete control over transactions, and can set all the properties of the *THstmt* that are part of the grid.

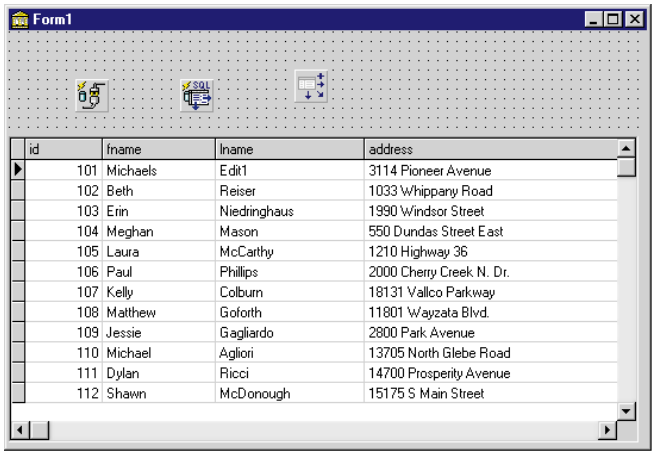


Figure 5: Design-time view of *THdbc*, *TOEDataSet*, and a *TDataSource* hooked to a SQL Anywhere query.

However, with power comes responsibility. There are two methods that can be used to insert, update, or delete a row from a data source. You can create and execute a full SQL statement, or you can do a positional insert, delete, or update of a row in a result set. I prefer the former method, and I create a separate *THstmt* to execute a SQL string that I create. There is very little overhead to this method, and I have found it to be very fast. Using the *positional* method will depend on whether the result set is modifiable; there are extensive discussions of these issues in the ODBCExpress documentation.

One problem I have found with the *TOEDataSet* is appends. Because the *TOEDataSet* represents a result set, when you perform an insert, you must re-fire the statement if you want the inserted row to appear in the result set. This means the *TOEDataSet* must be closed and then opened. Because any bookmarks are destroyed when the dataset is closed, you lose your position. ODBCExpress suggests you save the primary key values of the inserted row, then search the new result set to reposition the cursor. A procedure for this has been promised in a future release.

Documentation

ODBCExpress comes with an online Help file, a text file of FAQs, a manual, a tutorial in Word format, and several directories of sample programs. Much of the documentation explains the lower-level ODBC controls and the older ODBCExpress proprietary controls. Many of the demonstrations use the older controls also. There is only a brief discussion on using the *TOEDataSet*, and its use in the samples is limited. I would hope that the future sample and documentation would focus more on the *TOEDataSet*. Again, ODBC has its own way of operating, and it's worthwhile to understand how they handle things at the API level.

Other ODBC Components

TOESchema lets you create tables and views. Pull the component on the form, connect it to the *THdbc* component, and right-click on the component to go into a Schema Editor where you can access all the tables and indexes to add fields or change field attributes.

TOECatalog encapsulates many of the ODBC calls to get information about tables and views. For example, it allows you to easily get a list of all views or stored procedures available in your database.

TOEAdministrator gives you access to the ODBC Administrator's list of data sources, so you can add, delete, and change them. It is equivalent to the BDE Administrator.

TOESetup helps in setting up and configuring ODBC itself, as well as its drivers and translators.

There is another tab of components installed on the Delphi palette under the OEControls tab. These are the old-style ODBCExpress controls that preceded *TOEDataSet*. I recommend you avoid these controls; it's much easier to use the Delphi controls with the updated Delphi 3 open dataset architecture (see Figure 5).

Third-party Support

This release of ODBCExpress also provides support for InfoPower and Orpheus controls, or any other third-party component library that supports the open dataset architecture. Also, you can use the supplied IPREG unit to install a *TWwOEDataSet* that can be used with the popular InfoPower control (see Figure 6). I was able to run the InfoPower Locate Dialog, Filter Dialog, and Record View Dialog with no problem. Although there is no *Filter* property on the *TOEDataSet*, there is an *OnFilterRecord* event that the InfoPower component uses. I was impressed that it worked so easily, but curious enough to dig into the InfoPower source to find that they call the *OnFilterRecord* event.

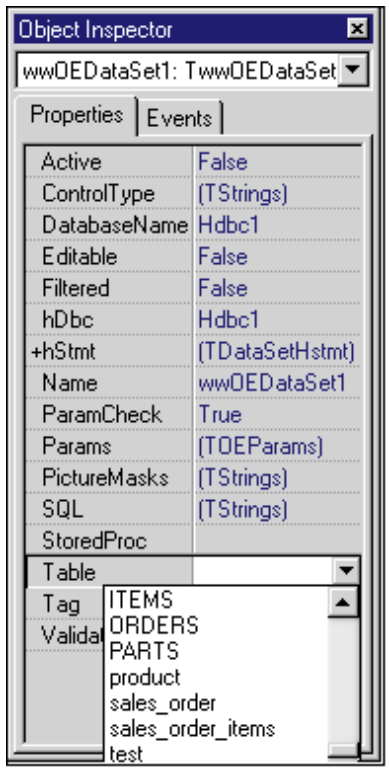


Figure 6: InfoPower-enabled ODBCExpress DataSet, *TwwOEDataSet*.

Technical Support

ODBCExpress has an Internet user forum at http://www.odbcexpress.com. They also provide unlimited support via e-mail, but you are encouraged to use the public forums. I have found responses from the development team to be prompt and complete. Datasoft is a South African company, so phone support can be expensive.

INFORMANT FACT FILE

ODBCExpress is a Delphi component library that allows access to any ODBC data source, and compiles directly into an executable. The Borland Database Engine is not required, and client/server applications can be written without the Delphi Client/Server Suite. Version 4 of ODBCExpress uses a Delphi *TDataSet* descendant that allows developers to use the normal Delphi data-aware components, as well as third-party tools.

Datasoft (Pty) Ltd.
P.O. Box 44633
Claremont
7735
South Africa

Phone: + 27 21 683 4680
Fax: + 27 21 683 4695
Web Site: http://www.datasoft.co.za
Price: ODBCExpress, US$299; full-source version, US$699.

The product is distributed electronically. There are versions for Delphi 1, 2, and 3, as well as a version for C++Builder. Version 3 supports Delphi 1 and 2. Version 4.5 is the current version for Delphi 3. ODBCExpress has shareware (evaluation) and purchased versions, with only a "nag" screen as the difference between them. Applications developed with ODBCExpress can be compiled directly into an executable and distributed royalty-free.

By the time you read this, the next version of ODBCExpress should be released. It will include enhancements such as calculated and look-up fields, multiple field look-up, and the ability to locate other improvements to the *TOEDataSet*. It will also fully implement the ODBC 3.0 API and include new ODBC 3.0 features, such as connection pooling.

Conclusion

I guess it's not hard to see that I like ODBCExpress. I find it stays true to the ODBC API character, and allows me to drop down to the API level when I want to make my own ODBC calls, or use the *TOEDataSet* with the native Delphi database controls. I've used these components from version 1 to version 4, and have been impressed with how they've evolved, the speed with which that evolution has occurred, and the openness and availability of the ODBCExpress development team. Δ

Steve Garland is a long-time Delphi developer who lives in Boise, ID with his wife, 18-month-old son, and four wild pug dogs. He is the principal of Hyper Logic Resource Group, a Boise-based custom software company. He is also involved in developing thin-client technology for the Asta Technology Group (http://www.astatech.com). He can be reached at steveg@hyper-logic.com.

Nathan Wallace's Delphi 3 Example Book

Don't buy *Nathan Wallace's Delphi 3 Example Book* [Wordware, 1998], with the intention of reading it. Sitting down with the book, hoping to read it cover to cover, you'll quickly find the text monotonous and the format repetitive. On the other hand, *do* buy this book to write notes, carefully highlight often-used methods, mark pages with Post-Its for your current project, and to flatten out on your desk so often that the spine becomes unrecognizable. The true value of this book is identified with its pragmatic approach to the function of Delphi and the consistency of its layout. It is highly likely that once this fine book is discovered and understood, it will not leave the side of any working Delphi 3 programmer. Mr Wallace, in explaining his intentions for writing *Example*, describes his vision of a "coffee rings book," one that has its cover and pages branded by coffee cup rings from being beside the computer all the time. *Example* definitely achieves this goal.

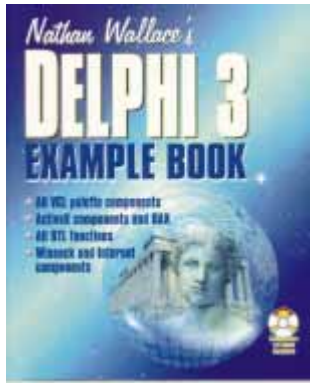
Wallace's book is a welcome throwback to a bygone era in the programming profession when each compiler shipped with pounds and pounds of documentation. The books would line the programmer's shelf, the pages of the function and procedure volume dog-eared and worn, the spine of the book cracked throughout. Sadly, we rarely get this opportunity for enlightenment from the compiler vendor any longer; the library of books has been replaced by a set of files on CD-ROM. A different approach is required in contrasting online and printed documentation — and the printed word retains a distinct advantage in some areas.

Example segments the documentation and examples into logical chapters organized around the flow of the VCL. A perusal of the introduction to each chapter prepares the reader for that section. Topics such as the Application object, and the procedures and functions that are a part of the RTL, are listed alphabetically. Chapters discussing the component library are listed in the order in which they appear on the specific tab. With the introductory comments quickly out of the way, the reader is lead directly to the meat of the chapter.

Consistency plays a large part in making this reference so valuable. The presentation of each component or object follows the same format and structure. A literate description of the object is followed by a schedule of the properties, methods, and events, each presented alphabetically. For each item, the identifier is listed, followed by its data type and a short contextual discussion of the object. Separating the book further from the online documentation is the inclusion of a usage example with every item, a feature lacking in the software version.

The consistency advantage extends to the component families that originated outside of Borland. The best example is found in the NEWT Internet component tab. The ActiveX components have Help files that differ in layout from the Borland-generated files. This book irons out the differences.

Reality catches up to this paper-bound nirvana when it comes to the sheer depth and breadth of the Delphi environment. The physical limitations of publishing a



paper book must be considered when attempting to keep the work manageable and affordable, the same reasons given by vendors justifying their skimpy documentation. Rather than choose to not cover some aspects of the tool, Mr Wallace has supplied chapters covering several of the VCL component tabs and objects as HTML files on the CD-ROM. The author carefully chose the items that would receive HTML treatment, selecting those that had encountered little change from earlier releases of the product or less-used objects. Unfortunately, the advantages pointed out earlier with respect to the paper version disappear when the text moves to HTML for the last six chapters of the book. Rather than retaining the concise, quick-reference

format of the printed page, the designer went on a button binge, requiring the programmer to drill down through multiple interface layers to retrieve information. Simple text pages with well thought out links would be preferable to this difficult tool.

The example files provided for each chapter are a minor disappointment. The component examples reviewed are nothing more than the component placed on a form. There are no event procedures showing the use of the component, nor any properties set programmatically. A bit button linked to the *ShellExecute* API function is featured that caused the compilation to fail on every example attempted; the program needed the *ShellAPI* unit included in the **uses** directive. When that correction is made, the examples will compile and, upon execution, the user can click on this button. It will execute a Web browser and attempt to connect to Mr Wallace's support site. Unfortunately, the site has undergone considerable renovation, and the user will encounter Page Not Found errors with the URL provided. Attempting to manually locate any updated files was also nonproductive, as the support pages for this book appeared to remain under construction.

Mr Wallace has done an outstanding job assembling this resource. The Delphi community is well served by this much-needed effort, which, in tandem with the recently published *Tomes of Delphi 3* series [Wordware, 1998], brings an additional measure of respectability to the Delphi community. Books of this nature are commonplace in C/C++ circles. By documenting the depth of the Delphi environment, the book positions the tool on the same professional level as the more prevalent environments — an important consideration because there are still developers who immediately dismiss any Pascal-based tool.

Nathan Wallace's Delphi 3 Example Book belongs on the desk of all serious Delphi programmers, those who work under the pressure of being consistently productive, and those with a more leisurely approach to development. Nathan Wallace has produced an extraordinary book, full of accurate and helpful information and in a format that is eminently useful. While the HTML files are a disappointment in comparison to the quality of the rest of the book, the usefulness of the collected data is excellent.

— *Warren Rachele*

Nathan Wallace's Delphi 3 Example Book by Nathan Wallace, Wordware Publishing, Inc., 2320 Los Rios Blvd., #200, Plano, TX 75074, (800) 229-4949, http://www.wordware.com.

ISBN: 1-55622-490-7
Price: US$54.95 (851 Pages, CD-ROM)

The Dawn of Distributed Computing

As you read this issue of *Delphi Informant*, hopefully you are in Denver, the site of BorCon98. This annual conference provides a chance to receive technical training, network with fellow developers, and meet some of the people from INPRISE. Historically, this is also the time that INPRISE paints a picture of what it intends to do for the coming year. The entire conference is surrounded with an air of optimism and enthusiasm for users of Borland products. To add to the ambiance, I thought I'd share my views on why I think distributed computing will be a good thing for developers and INPRISE.

Regardless of how you feel about the company's name change, the focus of Borland/INPRISE has been clear over the last 18 months: Distributed computing is in the spotlight. Products such as MIDAS, AppCenter, and OLEnterprise, the focus on SAP and AS/400 integration, and the acquisition of Visigenics are clear examples of the commitment to this emerging arena.

These developments were not proposed and developed in a vacuum. The reason paradigm shifts occur in the software development industry is due to programmers who demand more from their compiler vendor, tool, and/or language. Think of it as survival of the fittest for development products. As a result of this Darwinian evolution of development tools, we no longer have to use things like command-line editors and tools. In fact, we have the ability to become even more productive because of things like integrated development environments (IDE), GUI tools, and visual editors. How many people would really rather be using EDLIN instead of a visual editor?

Borland captured the OOP market nine years ago and spurred its growth by remaining on the leading edge of technology. Essentially, an escalating war between language and compiler vendors was fought to regain the ground Borland had clearly taken. Today, it's physically impossible to develop software without seeing some passing reference to OOP. Borland didn't invent OOP, but they certainly brought it to the forefront.

With the advent and necessity of distributed computing, INPRISE has identified the next technology to show as much promise as OOP did. Borland products can continue to grow and flourish for both small-company development shops and enterprise developers alike, but the reality is that the computer industry has shifted more toward enterprise development and, as a result, distributed multi-tier computing.

Rick LeFaivre, Senior Vice President of Research and Development at INPRISE, has a write-up of the waves of technology available at http://www.inprise.com/about/executive/rickmultitier.html. The common thread through all these waves is innovation. In each case, both end-users and developers clamored for more, and the development tools industry responded by providing what the developers needed. The shift to distributed computing is no exception.

Distributed computing is viewed by some as a return to mainframe mentality, but I see this as a golden opportunity to share information across previously impassable boundaries. Data can be made available around the world easily using existing infrastructures — and at a very reasonable cost. The benefits of this model are many: centralized business logic, thin clients, fault tolerance, and load balancing to name just a few.

The market for developing distributed computing solutions is ready to explode. The dawn of distributed computing is here. You owe it to yourself, and your customers, to find out how distributed computing can make your applications better. Isn't that what all of us really want? Δ

— Dan Miser

Dan Miser is a Design Architect for Stratagem, a consulting company in Milwaukee. He has been a Borland Certified Client/Server Developer since 1996, and is a frequent contributor to Delphi Informant. *You can contact him at http://www.execpc.com/~dmiser.*

HTML as a Second Language

For Delphi developers, the primary language is obviously Object Pascal. For many, assembler and SQL are important second languages. Increasingly however, HTML is becoming important for all Windows programmers, and its implications go beyond the Web; HTML is becoming popular as a language for writing Windows Help files. Like Rich Text Format (RTF), HTML consists of tags. Among other things, these tags indicate the formatting of the text. Recently, with the aim of sharpening my own skills and understanding, I had an opportunity to examine several new HTML books. I'll start with a brief overview of those references and then suggest how HTML might be useful in your Delphi programming.

HTML references. Among the books I examined were two comprehensive references: Lois Patterson's *Using HTML 4* [QUE, 1997], and *HTML 4 Unleashed* [Rick Darnell and John Pozadzides, SAMS, 1997]. The former includes an excellent listing of HTML tags in one of its appendices; the latter introduces the tags, systematically by function, throughout the text. *Using HTML 4* includes a short appendix describing the tags new to HTML 4. A third work by Larry Aronson and Joseph Lowery, *HTML 3.2 Manual of Style* [Ziff Davis Press, 1997], is somewhat dated and considerably shorter than the previous two. Nevertheless, it provides a wonderful introduction to basic HTML.

One of the newer developments in HTML relates to building dynamic Web pages that change in response to a surfer's actions. The new extensions to HTML that enable this kind of flexibility are generally referred to as Dynamic HTML. While each book mentioned so far deals with dynamic Web pages to some extent, there are two additional volumes that concentrate on these new developments: *Dynamic Web Publishing Unleashed, Second Edition* [Shelley Powers, et al., SAMS, 1997], and *Dynamic HTML Unleashed* [Rick Darnell, et al., SAMS, 1998]. As you might expect, neither of these provides the same comprehensive introduction to HTML the earlier books do. However, *Dynamic Web Publishing Unleashed, Second Edition* includes useful appendices on HTML 4 and Cascading Style Sheets, one of the important HTML extensions.

While these two books cover similar ground, their emphasis is slightly different. *Dynamic Web Publishing Unleashed, Second Edition* provides an overview of the issues involved with creating a dynamic Web site; *Dynamic HTML Unleashed* puts more emphasis on the techniques needed to accomplish that goal. Besides HTML, publishing on the Web often involves additional technologies and languages, including Java, ActiveX, JavaScript, CGI, VRML, and more. While some of these are specific to one of the two major browsers, others are universal. Appropriately, another issue raised in many of these volumes concerns the proprietary tags specific to a particular browser. But how does any of this relate to Delphi programmers?

The Delphi connection. There are several obvious ways we can use Delphi to work with HTML files: to generate HTML code, to parse and interpret HTML code, and to display HTML code in our own browser. All these topics have come up in Delphi books and some in articles. In the chapter "Introduction to CGI Programming with Delphi" in his *Delphi 2 Unleashed* [SAMS, 1996], Charles Calvert shows how to generate HTML code. In *Mastering Delphi 3* [SYBEX, 1997], Marco Cantù demonstrates how to produce HTML tables with Delphi. He also discusses CGI and ActiveX in building a "Web Application" with Delphi. Finally, in *Delphi 2 Multimedia Adventure Set* [The Coriolis Group, 1996], Chris D. Coppola, et al. devote several chapters to building a multimedia Web browser. All these examples have one important thing in common: They use Delphi to create applications that read or write HTML files.

Hopefully, I've convinced you of the need to develop an understanding of HTML, which, after all, isn't that difficult. By learning "another language" you increase your ability to work effectively in a field that is quickly becoming more multi-dimensional. Δ

— Alan C. Moore, Ph.D.

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

Book Title	HTML Tags	Browser Issues	Script Languages	Related Technologies	Cascading Style Sheets
HTML 4 Unleashed	Excellent	Excellent	Excellent	Very Good	Excellent
Using HTML 4	Very Good	Good	Excellent	Very Good	Good
HTML 3.2 Manual of Style	Excellent	Good	Fair	Fair	Good
Dynamic HTML Unleashed	Fair	Excellent	Very Good	Good	Good
Dynamic Web Publishing Unleashed, Second Edition	Good	Good	Excellent	Excellent	Good