September 2000, Volume 6, Number 9

Cover Art By: *Arthur Dugoni*

ON THE COVER

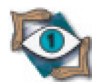
6 Dynamic Delphi
Word Control: Part I — Ron Gray
Mr Gray presents the Microsoft Word object model, demonstrates how to control it using run-time and compile-time Automation, and provides sample source code for Delphi 4/5 and Word 97/2000.

FEATURES

10 Distributed Delphi
The Gold Standard, MIDAS & COM: Part II — Bill Todd
Completing his two-part series, Mr Todd presents two additional ways for a COM server to call a COM client's methods: via Automation, and through a callback interface.

14 OP Tech
Database Persistent Objects: Part I — Keith Wood
Mr Wood shows how classes can be set up to automatically store themselves to, and retrieve themselves from, a standard relational database, with minimal effort using RTTI.

21 The API Calls
Raw API Programming — Andrew J. Wozniewicz
Mr Wozniewicz demonstrates how to code without the VCL, when an executable's size and speed are critical considerations, e.g. real-time data acquisition, restricted memory/CPU, etc.



28 First Look
InterBase 6 — Bill Todd
Famous for being open source, Mr Todd explains there's much more that's new with IB6, including large exact numerics, new administrative tools, replication, and much more.

REVIEWS

32 ReportBuilder 5.0 Enterprise
Product Review by Tim Sullivan

DEPARTMENTS

2 Delphi Tools
5 Newsline
36 Best Practices by Clay Shannon
37 File | New by Alan C. Moore, Ph.D.

Delphi
TOOLS

New Products and Solutions

dtSearch Launches Version 6.0 of dtSearch Web and Text Retrieval Engine

dtSearch Corp. announced ***dtSearch Web 6.0*** and the ***dtSearch Text Retrieval Engine***, which feature comprehensive support for XML, including indexing and searching of multi-layered nested fields.

Version 6.0 supports the Linux platform, and enhances support for Windows 2000/CE. Other new features include Java support through a JNI interface to the dtSearch Text Retrieval Engine; Unicode support to enhance existing support of European-based languages and to add support for double-byte character sets, such as Chinese and Japanese text; and added integration with and support for Microsoft Office 2000 and Corel Office 2000.

Proprietary indexing and searching algorithms allow for faster indexing and searching performance even over large databases and other diverse collections of documents. dtSearch offers over two dozen indexed and unindexed text search options. Search features include a scrolling list of indexed words; fuzzy search level, adjustable from 1 to 10 using a proprietary fuzzy search algorithm; concept searching (including a customizable, extensive built-in thesaurus); natural-language searching with advanced relevancy ranking by "hit" density and rarity; boolean (and/or/not); proximity; variable term weighting; stemming; field; and range.

dtSearch Web uses "point-and-click" set-up to let users add instant searching to a Web site. Built-in proprietary HTML file conversion supports searching of word processor, database, spreadsheet, ZIP, and other file types. After a search, dtSearch Web displays retrieved documents in the user's browser with highlighted hits, while preserving all HTML links and images. dtSearch Web also provides full PDF support, including display of PDF files in Adobe Acrobat Reader with highlighted hits.

dtSearch Text Retrieval Engine allows developers to add searching to any PC, network, or Internet/-intranet product. It's both a COM object and a DLL. In addition to Java support, it includes sample source code in Delphi, C++, Visual C++, Visual Basic, and Active Server Pages (ASP).

Finally, the dtSearch Text Retrieval Engine also includes sample source code to dtSearch Web in both ASP and ISAPI-based versions.

dtSearch Corp.
Price: US$999 per server.
Phone: (800) IT-FINDS
Web Site: http://www.dtsearch.com

Digital Metaphors Releases Learning ReportBuilder

Digital Metaphors Corp. announced the release of ***Learning ReportBuilder***, an interactive learning tool designed to teach non-developers how to use reporting solutions built using ReportBuilder, the reporting tool for Delphi.

Learning ReportBuilder provides documentation for the end users of applications built using ReportBuilder. Learning ReportBuilder includes a 125-page guide in PDF format, a stand-alone reporting application, and an online Help file, all intended to aid end users in learning the intricacies of reporting in general, and ReportBuilder in particular.

Digital Metaphors Corp.
Price: Free
Phone: (972) 931-1941
Web Site: http://www.digital-metaphors.com/learnrb/learnrb.exe

InfoCan Management Announces Internet Development with Delphi

InfoCan Management announced ***Internet Development with Delphi***, a new course in Web-based development training. It is a five-day, hands-on course that utilizes the Internet tools Borland provides in Delphi for Web-based development.

The course targets developers with a working knowledge of Delphi who need to scale their applications to the Internet. It is strongly recommended to have Delphi 5 Client/Server Foundations, or equivalent working knowledge on topics covered, as a prerequisite.

The course provides practical information on creating an integrated Web application that utilizes all of the Delphi Web-based technologies. The courseware provides students practical applications and scenarios, including Internet Components, ActiveForm, and Web Servers Applications — CGI, ISAPI, Borland WebBroker Technology, State and Persistence, Active Server Pages, Advanced Security, and InternetExpress. It also covers the pros and cons of techniques under different situations.

InfoCan Management
Price: CDN$2,750 (approx. US$1,850).
Phone: (888) INFOCAN
Web Site: http://www.infocan.com

Bowne Global Solutions Launches ExtraGLOBAL

Bowne Global Solutions launched ***ExtraGLOBAL***, an extranet that speeds the development, testing, and delivery of multi-language products to technology users and markets worldwide.

Working with Palm, Bowne Global Solutions provides internationalization, localization, and testing for all Palm III-, V-, and VII-series handheld systems for use in seven languages: traditional Chinese, French, German, Italian, Japanese, Korean, and Spanish, including the operating system, desktop applications, programming tools, and marketing and customer support materials. Bowne also localized and tested the Palm IIIc, Palm Vx, and Palm Operating System into French, Italian, German, Japanese, and Spanish.

Bowne Global Solutions
Price: Contact Bowne for information.
Phone: (323) 866-1000
Web Site: http://www.bowneglobal.com

ProtoView Launches ActiveX Component Suite 8

ProtoView Development released *ActiveX Component Suite 8*. The ActiveX Component Suite is comprised of the Data Explorer, DataTable, ScheduleX, and TreeViewX products. Tipping the scales at thirty-one components, version 8 includes three new additions: Property Browser, Color Combo, and Image Combo.

The ProtoView Property Browser allows developers to add a properties/style sheet to their applications, giving end users the ability to edit the look and feel of their application and components in the interface at run time. The Property Browser can also be used as a data-entry form for application users to have access to domain-specific properties of server-side components.

The Property Browser supports color and font options and can include any user-defined property. It uses standard string (text), numeric, and drop-down lists to select style options on properties. Drop-down lists can consist of text or can display "..." buttons to bring up custom dialog boxes when editing a field.

The Property Browser also includes Color Combo and Image Combo for color and image property editing. Color Combo and Image Combo can be used for in-cell editing of properties in the Property Browser, or as stand-alone components in other areas of application development. Both support feature-rich properties pages that allow developers to set the majority of styles for this component without code.

Version 8 also adds improvements to "browser-exclusive" features. The DataTable grid component now provides a method to allow developers to get the common print dialog box. This allows developers to add functionality to get the common print dialog box for Internet Explorer, allowing end users to set printer specifications at run time.

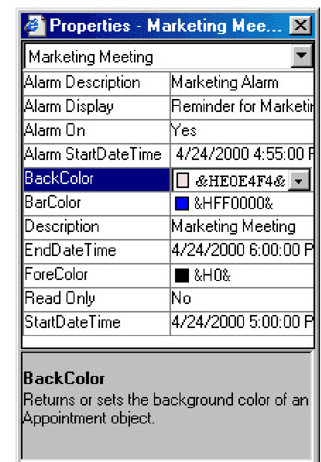
Tabs have also been added with version 8. With tabs, the Outlook Bar and Data Explorer UI component can have tabs that consist of multiple trees and forms attached to them. Using the Data Explorer UI component and the Outlook Bar, developers create the user interface "shell" commonly found in applications like Microsoft Outlook. With the addition of tabs, developers can bind to any number of data sources or load multiple forms on the same node click. Images can be set for tabs, and the orientation of tabs can be placed on any side of the window (top, bottom, left, and right).

DataTable 8 includes optimizations to existing sorting and printing features. For sorting, the addition of a BubbleSort option has been added for working with common data. For printing, additional print methods have been added to customize headers and footers. Using this method, developers can take DataTable grid data and insert it into existing print reports and forms.

The ScheduleX product now includes Date Edit and Time Edit (also found in DataTable). Date Edit has been enhanced to support the ScheduleX calendar for its drop-down display. The ScheduleX calendar supports multiple lines of text in date cells with word wrapping. User interface features, such as Outlook-style time display and "all day" appointments, have also been added to the DayView.

For TreeViewX, multiple selection drag-and-drop has been added. With this feature, end users can select any number of nodes and drag-and-drop them to another treeview.

ProtoView Development
Price: US$695. Products in the ActiveX Component Suite are also sold separately: Data Explorer 8, DataTable 8, and ScheduleX 8, US$395 each; TreeViewX 8, US$199.
Phone: (800) 231-8588 or (609) 655-5000
Web Site: http://www.protoview.com/order



Netronic Releases VC-XGantt

Netronic Software and its US affiliate company, **American Netronic, Inc.**, released *VC-XGantt*, an ActiveX control that adds interactive Gantt-style functionality into all programming environments that accept OCX controls. The custom control was developed to relieve programmers of the tedium and complexity of coding high-level graphical objects to represent bars and milestones below a time ribbon. The North, Central, and South American releases are effective immediately.

VC-XGantt allows programmers to simply define a control region on a form (or Window) to create an instant Gantt charting application. Custom property pages allow programmers to choose from hundreds of features. Interactive, simultaneous histograms are available, and all printing is handled automatically by VC-XGantt. The control can be deployed on a Web page, and various views can be exported to a compact Web-viewer format for management-level reporting directly from the Web.

Netronic Software/American Netronic, Inc.
Price: US$3,495 (includes the software development kit on CD-ROM, hard-copy and electronic documentation, sample source code examples, and 30-day free technical support).
Phone: (800) 447-2633
Web Site: http://www.netronic-us.com

LEAD Technologies Announces LEADTOOLS 12

LEAD Technologies, Inc. announced the release of *LEADTOOLS 12*. Version 12 provides new and enhanced features in all five of LEAD's imaging engines (Raster, Document, Multimedia, Vector, and Medical).

New features in LEADTOOLS' Raster Imaging engine include the ability to load and save TIFF files with CMP compression; faster JPEG compression and decompression; the ability to load and save uncompressed SGI files and RLE compressed SGI files; a magnifying glass tool; a new Magic wand that allows a region to be created from an X-and-Y location and a color tolerance; and a picturized algorithm that provides more options for selection and use of images.

New features in LEADTOOLS' Document Imaging engine include loading and saving TIFF files with JBIG compression; faster loading of 1-bit TIFF files; new document clean-up features, such as hole-punch removal, line removal, border removal, invert text, smooth filter, and dot removal; and an updated barcode engine.

New features in LEADTOOLS' Multimedia Imaging engine include added support for saving multimedia files via DirectShow in Multimedia toolkits; enhanced NT compatibility in multimedia tools; more control over saving audio streams; and programmatic configuration of video/audio codecs.

New features in LEADTOOLS' Vector Imaging engine include support for DWG and DWF; support for saving vector images as a DXF inside a TIFF file; a comprehensive drawing toolkit; improved layer support; and the ability to maintain 3D text as compressed vector data.

New features in LEADTOOLS' Medical Imaging engine include support for the latest version of the DICOM 3.0 specification; support for additional DICOM classes, including Radiotherapy Beams Treatment Record Storage, Radiotherapy Brachy Treatment Record Storage, Visible Light Endoscopic Image Storage, and more; optimized DICOM image load processing; and a caching mechanism between disk and memory to optimize speed.

Other important global changes in version 12 include a reduction in the number of TLS slots (thread local storage) used by all LEADTOOLS DLLs. LEADTOOLS version 12 now uses up to two TLS slots, freeing up system resources and reducing the likelihood of conflicts.

LEAD Technologies, Inc.
Price: From US$495 (visit Web site or call for more information).
Phone: (704) 332-5532
Web Site: http://www.leadtools.com

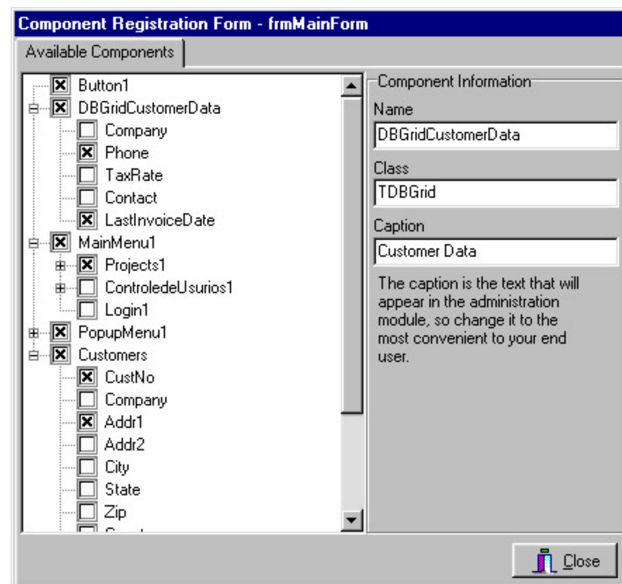
Tools&Comps Announces *TUsersCS* Security Component 1.5

Tools&Comps announced *TUsersCS Security Component 1.5*, the latest upgrade to the company's end-user security administration tool. *TUsersCS* was designed for applications that run in a client/server environment and need flexibility in the security control. *TUsersCS* makes it easier to implement end-user control in small or corporate applications.

The component allows a developer to control access to an application through a login screen, where a user name and password are requested. The component allows a developer to grant or deny a user access to various components inside an application, such as buttons, panels, DBnavigators, DBgrids, fields, toolbars, menu items, and virtually any *TControl* descendant. The component works by disabling or making invisible the components that the user has no permission to use, view, or modify.

The *TUsersCS* package comes with another component, *TUsersCSReg*, which allows the developer to register at design time the components he wants to protect. The developer activates the Component Registration Form and checks the most relevant components. The developer can also change the caption value to change the text that will be shown in the User Administration Module.

The User Administration Module shows a list of the application's forms and, to the selected form, shows its components in a *TTreeview* that imitates the form's component hierarchy.

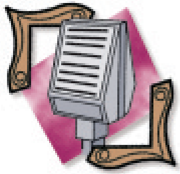


Tools&Comps
Price: US$249.95 for a single license.
Phone: 55 27 99602760
Web Site: http://www.toolsandcomps.com

News

L I N E

September 2000



Inprise/Borland and Corel Terminate Proposed Merger

Scotts Valley, CA — Inprise/-Borland Corp. announced its merger agreement with Corel Corp. has been terminated by mutual agreement of the two companies without payment of any termination fees. The reciprocal stock option agreements have also been terminated.

Dale Fuller, Inprise/Borland interim president and CEO said, "Much has changed since the merger was agreed to more than three months ago, and our board concluded that it would be best to cancel the merger on an amicable basis."

In January of 2000, Inprise/-Borland and Corel entered into a confidentiality agreement that included a standard three-year standstill covenant. That agreement remains in effect.

Inprise/Borland Offers JBuilder Handheld Express

Scotts Valley, CA — Inprise/-Borland released a preview version of JBuilder Handheld Express, an extension for the mobile market to Inprise/Borland's JBuilder 3.5, a Java technology development tool. The preview version of JBuilder Handheld Express enables users to develop Java solutions using the Java 2 Micro Edition (J2ME) software development kit, and deploy their application to the Palm OS. The preview software is available for download at no charge via http://www.borland.com/jbuilder/hhe.

JBuilder provides features that dynamically adapt to any J2ME profile, including the Mobile Information Device Profiles currently being developed through the Java Community ProcessSM.

The product is based on J2ME Connected Limited Device Configuration (CLDC), a Java run time and virtual machine optimized for consumer and embedded devices, such as cell phones, pagers, and Personal Digital Assistants. CDLC version 1.0 is available directly from Sun's Web site at no charge for development purposes. Applications can be written and run on any device that contains a CLDC-compatible run time.

Links to the JBuilder Handheld Express tool, available for free download, can be found at http://www.borland.com/jbuilder/hhe. Inprise/Borland also encourages developers to use the Palm OS Emulator found on Palm, Inc.'s Web site at http://www.palm.com.

Inprise/Borland Announces Support for Enterprise Application

San Francisco, CA — Inprise/-Borland announced the next version of its Application Server and AppCenter products. Inprise Application Server 4.1 supports the J2EE standard and combines the benefits of EJB and CORBA. AppCenter 4.0, now with added support for EJB and CORBA, enhances the management capabilities of the application server market and enables customers to better handle the increased demands of the Internet economy.

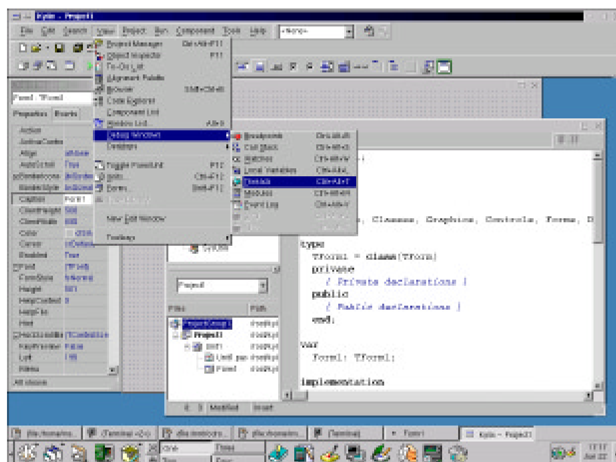
AppCenter 4.0 provides enterprise development and operations teams with software-based centralized tools to model, monitor, and manage distributed applications running on multiple hardware and software platforms. AppCenter 4.0 is capable of managing the complete Enterprise JavaBeans environment, including EJBs and EJB servers and containers.

A list of additional AppCenter 4.0 features is located at http://www.borland.com/appcenter/feaben/.

Inprise Application Server 4.1 includes enhanced EJB transaction support with full two-phase commit; a high-performance transaction manager supporting JDBC 2.0/XA; integration of VisiBroker messaging service with support for the Java Messaging Service; and a new deployment wizard to simplify rolling EJB applications out to multiple containers. For more, visit http://www.borland.com/appserver.

VisiBroker 4.1 for Java is the latest version of the object request broker designed to facilitate the development and deployment of distributed applications that are scalable, flexible, easily maintained, and based on industry standards. A list of features in VisiBroker for Java is located at http://www.borland.com/visibroker.

Inprise/Borland Offers Visual Basic Developers Fast Path to Linux



The latest build of Kylix as it appears on KDE.

Scotts Valley, CA — Inprise/-Borland announced a solution that gives Visual Basic developers worldwide an entry into the Linux market by taking advantage of the cross-platform capabilities of Delphi. Inprise/-Borland is launching a worldwide awareness campaign (including seminars, direct mail, electronic and print advertising, and special offers) that will focus on showing Visual Basic developers how Delphi can be used to create applications on Windows and port those applications to Linux. (For more information, please visit http://www.borland.com/vb/.)

Evans Marketing, an independent research firm providing market research focused on the software development community, polled 468 Visual Basic developers worldwide regarding Linux development and other topics. The results indicated that 36 percent of Visual Basic developers in North America and 53 percent outside of North America are interested in creating Linux applications in 2000.

By Ron Gray

Word Control

Part I: Microsoft Word as an Automation Server

Many applications require word-processing capabilities. To provide basic functionality, Delphi developers can, and do, customize the Memo or RichEdit control. But with Microsoft Word available — as it usually is — it just makes good sense to exploit it for your application's word-processing needs.

Any Delphi application can offer Word's advanced editing and publishing capabilities with just a few lines of code. Word exposes its properties and methods just as any other Automation server or Delphi component does. In fact, in Delphi 5, Word *is* a component.

Of course, Word's capabilities go far beyond the basic word processing offered by the Memo and RichEdit controls, and you can take advantage of them all. Documents can contain complex tables, graphics, and hyperlinks. Word's drawing tools can be used to create graphics and other special effects. The powerful Visual Basic for Applications (VBA) programming language allows for the creation of everything from macros for automating simple tasks, to complete custom applications. The Web-authoring tools provide an HTML editor. Data can be merged into Word documents to automate letters, produce mail merges, or print envelopes and labels. Word's file converters give access to a wide variety of file formats. You can even use the spell checker to check the spelling of text from an Edit or Memo control. Documents can be linked to your application or embedded and stored in BLOB fields as binary objects. All of this can be done programmatically thanks to Automation.

This article is the first of a two-part series. It begins with a review of Automation, the technology that makes integration possible, and then looks at the Word object model. Part II will continue examining the Delphi 5 Word components, look at OLE (Object Linking and Embedding), and will offer suggestions for completely integrating Word into your Delphi applications.

A Brief Review of Automation

Automation (formerly OLE Automation) is a mature Microsoft technology that enables one application (the controller) to directly manipulate the objects of another application (the server). Automation is built on Microsoft's Component Object Model (COM), which defines a binary interface standard for implementing objects independent of any platform or programming language. This means that a Delphi application can set properties and invoke methods of objects in Microsoft Word, even though Word is written in another language.

As an example, consider a set of classes written in Delphi that performs a specific function. The classes can be wrapped in a component, but the component can only be used in Delphi, because it must be compiled into the application. Even if placed in a DLL, other languages still could not use it (at least not easily), because they don't support the same syntax. COM allows you to wrap the classes into a component that allows other applications to manipulate it in an object-oriented fashion, regardless of the language. Automation allows the component to be physically wrapped in a DLL or EXE.

Automation makes inter-application communication a reality by allowing applications to implement component-oriented objects designed to perform specific tasks — like a powerful word processor. Just as object-orientation facilitates the development of reusable objects, Automation lets you build reusable components. Going one step further, DCOM (Distributed Component Object Model) removes machine boundaries, and allows components to communicate openly, regardless of their location on the network.

The two key elements of Automation are Automation servers and Automation controllers (also known as clients). In general terms, an Automation server is an application that can be controlled programmatically by another application. Specifically, it is a COM component that implements the *IDispatch* interface.

An Automation server can be in-process or out-of-process with respect to the client. An in-process Automation server is implemented as a DLL, and runs in the same process space as the client. An out-of-process Automation server is an EXE that runs in its own address space. Word is an out-of-process Automation server. Its content and functionality are available programmatically through exposed objects. Everything you can do in Word through the user interface can be done programmatically by invoking methods of these objects. That's quite a feat.

An Automation server must be registered in the Windows registry (under HKEY_CLASSES_ROOT\CLSID*GUID*) before it can be used (see Figure 1). The entry for Word has four keys:

- **InprocHandler32** is used to specify a (possibly custom) handler. Its default value is Ole32.dll.
- **LocalServer32** contains the full path to the local server, e.g. "C:\PROGRA~1\MICROS~1\OFFICE\Winword.exe /Automation."
- **ProgID** is an identifier associated with a CLSID. The format is *Vendor.Component.Version*. Its value is "Word.Application.8" for Word 97, or "Word.Application.9" for Word 2000.
- **VersionIndependentProgID** is the human-readable name of the application's class, and remains constant over versions. For Word it's "Word.Application."

Because the Automation server exposes its methods and properties, it can be manipulated programmatically by controller applications. An Automation controller is simply an application that can access and control objects of an Automation server. The language of the Automation controller must support the ability to create an instance of the server (as an object) and call its methods. A common example of a Delphi Automation controller is using data from a table to generate a mail merge or catalog using Word.

Automation in Delphi

Delphi applications can be Automation servers, Automation controllers, or both.

Automation control is supported in two ways:

- Run-time Automation handling
- Compile-time Automation handling

Run-time Automation Handling

When the Delphi application has no compiled information about the Automation server's properties or methods, it must resolve type information at run time (known as late binding). In other words, a Delphi application can contain code that references methods of a Word object without the compiler really knowing anything about the object. How is this possible? Through the Variant, a data type derived from Visual Basic that contains another type. The *OleVariant* is a COM-compatible type used to manipulate Automation objects when their properties are not known at compile time, i.e. calls to the object's properties and methods are ignored by the compiler and are late bound at run time.

Use the *CreateOLEObject* function to instantiate an Automation object. It accepts a string that represents a program ID (described previously), and returns a reference to the identifier of the program ID's *IDispatch* interface, which is used to communicate with the object. The methods, properties, and variables of the Automation server are available through this reference.

The following example uses Word as an Automation server to create a document, insert some text, then print the document (see Figure 2):

```
var
  oWord: OLEVariant;
begin
  oWord := CreateOLEObject("word.application.8");
  oWord.Visible := True;
  oWord.Documents.Add;
  oWord.Selection.TypeText('Hey now!');
  oWord.ActiveDocument.PrintOut;
end;
```

The code begins by declaring a variable and creating a reference to the Word object. As previously mentioned, the Word object is declared as an *OleVariant* and is created using *CreateOLEObject*. The program ID "word.application.8" is the class name for Word 97 (winword.exe). The program ID "word.application" is also registered to return the same object. Some earlier examples of automating Word use "word.basic" for the program ID to retrieve a WordBasic object. Word Basic was replaced with VBA in Office 97.

The next statement in the example displays the Word application by setting the object's *Visible* property to True. This is important because Word, like most Automation servers, is not visible by default. This makes sense because the Automation server can be used to perform tasks without the user knowing it. There are times when displaying Word isn't necessary, as in the previous example; because the code simply prints a document, Word doesn't need to be visible. In fact, there are many examples when Word can be used without the user even knowing it, such as printing a label or spell-checking some text. The remaining lines of code simply invoke methods of the

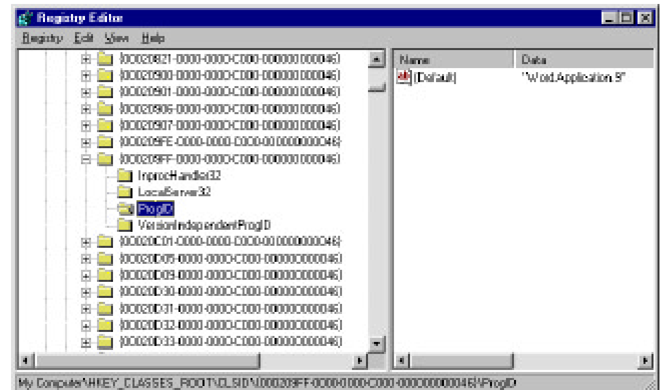


Figure 1: Registry keys for Microsoft Word.

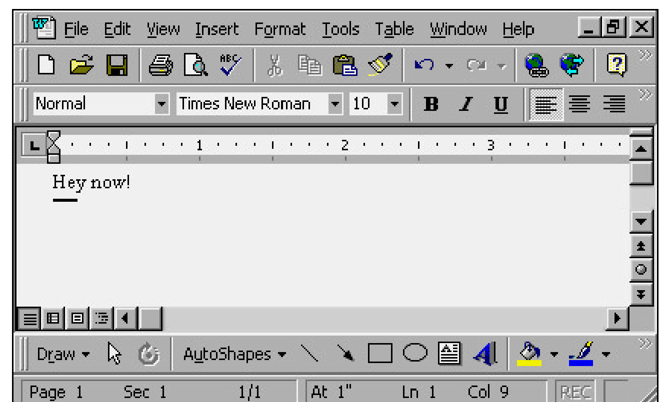


Figure 2: A simple example of automating Word from Delphi.

Application object, which is described in more detail later in the article.

There are two drawbacks to using run-time Automation. First, the compiler cannot perform error checking, because it doesn't know the methods or properties of the object. This leaves more room for errors at run time. Second, there is overhead in determining type information because it must be resolved through the *IDispatch* interface. Fortunately, both issues can be resolved by using a pre-generated class of the Automation server.

Compile-time Automation Handling

Most Automation servers provide type information about their exposed objects, properties, and methods in the form of type libraries. These are usually stand-alone files with a .tlb or .olb extension. Delphi can import an Automation server's type library and automatically generate classes that include a dispatch interface wrapper. The main advantage to using pre-generated classes is that type information is resolved at compile time (i.e. it's early-bound), which helps reduce the chance of run-time errors.

You don't need to import the type library in Delphi 5 because it's already been done. To import the type library for Word in Delphi 4:

1) Choose **Project | Import Type Library**.
2) If you're using Word 97, select **Word (Version 8.0)** from the list. If it isn't on the list, click the **Add** button to locate the MSWord8.olb type file, located in C:\Program Files\Microsoft Office\Office, or on the CD. The dialog box should look like the one shown in Figure 3. If you're using Word 2000, select **Microsoft Word 9.0 Object Library (Version 8.1)** from the list.
3) Choose **OK**.

The Import Type Library tool reads the server's published interface, and creates a wrapper class of the Automation server. The import utility creates a unit named Word_TLB.pas (in the \Imports directory by default). Include the unit in the **uses** clause, and write code to control the server with a dual interface or a dispatch interface. The dual interface offers advantages over the dispatch interface, so it's the preferred method.

As mentioned earlier, Automation servers implement the *IDispatch* interface. A dispatch interface, or dispinterface, uses the functions of *IDispatch* (*GetTypeInfoCount*, *GetTypeInfo*, *GetIDsOfNames*, and *Invoke*) to indirectly call functions of the Automation server. Before calling a function, the controller application must call *GetIDsOfNames* to get the ID of the function, then use the *Invoke* function to call it. A dual interface is a COM interface that inherits from *IDispatch*, so although functions are still available through the dispatch interface, they are also available through vtable binding, a way of storing pointers to functions for fast, nearly direct, access for calling them.

Once Word_TLB.pas has been added to the **uses** clause, you can create and initialize an *Application* object using the *_Application* dual interface and CoClass client proxy class, *CoApplication_*, as follows:

```
var
  oWord: _Application;
begin
  oWord := CoApplication_.Create;
  oWord.Visible := True;
  oWord.Documents.Add(EmptyParam, EmptyParam);
  oWord.Selection.TypeText('Hey now!');
  oWord.ActiveDocument.PrintOut(EmptyParam,
    EmptyParam, EmptyParam, EmptyParam, EmptyParam,
    EmptyParam, EmptyParam, EmptyParam, EmptyParam,
    EmptyParam, EmptyParam, EmptyParam,EmptyParam,
    EmptyParam);
end;
```

Notice that the *oWord* variable is now declared as *_Application* rather than *OleVariant*. The interface is initialized with the CoClass client proxy class, *CoApplication_*. A CoClass is a class that implements one or more interfaces — in this case, the *_Application* interface defined in Word. CoClass classes are generated automatically when the type library is imported. The rest of the code calls the same methods as in the example of run-time Automation handling, except that each method must now be called with the correct parameters. Use the *EmptyParam* variable to indicate that a particular parameter isn't used.

As mentioned earlier, controlling an Automation server with a dual interface offers advantages over run-time Automation handling. Besides reducing the overhead of type checking, it also enables the Code Insight features, such as Code Completion and Code Parameters. More importantly, syntax checking is performed at compile time, which helps reduce errors. However, this approach is still relatively cumbersome and low-level. For example, to receive events from the Word object requires writing an event sink, an interface through which the Automation server can communicate with the client.

Delphi 4 offers wrapper classes to the Word object in Word97.pas, located in the Demos directory in \ActiveX\OleAuto\Word8. Although the file is for demo purposes, the classes encapsulate some of the complexity, including receiving events.

TWordApplication

Delphi 5 further simplifies Word Automation by wrapping Office in a collection of components that are available right off the Servers page of the Component palette. Of course, it still uses compile-time Automation handling as described previously, but the imported type library is completely encapsulated in Delphi components. These components will be described in detail in next month's article.

General Considerations

Automating Word presents a few interesting caveats. First, remember that Word is a separate application that can be controlled by the user, other applications, or both. Word might already be running when a Delphi application needs to use it. In this case, should the application use the existing object, or run another instance of Word?

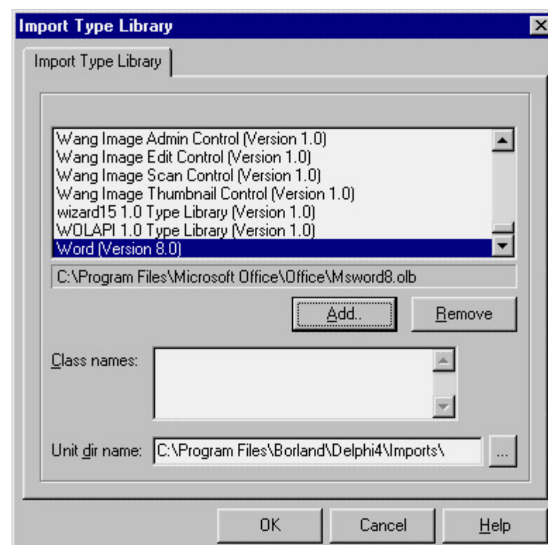


Figure 3: The Import Type Library dialog box is used to import the Word type library into Delphi 4. (This step is unnecessary in Delphi 5.)

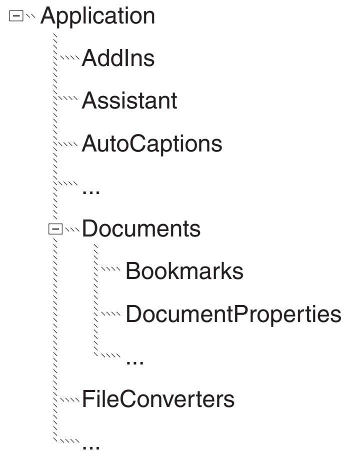


Figure 4: A small portion of the Microsoft Word object model.

User Interface	Programmatically
Start the Word application.	Get a reference to the Word *Application* object.
Open a document.	Use the *Documents* collection to open a document.
Select a specific document.	Get a reference to a specific *Document* object.
Select a range of words from the document.	Get a reference to a Word *Range* object.

Figure 5: Steps to select text in a Word document.

Obviously, memory constraints limit the number of instances that can be running simultaneously. But sometimes creating a new instance of Word is required to get "exclusive" access, so you don't need to worry about user intervention.

Second, when Word is visible, the user can switch documents, run macros, quit Word at any time, and generally cause havoc if your Delphi code isn't robust enough to handle these eventualities. Keep in mind that you have complete control over the menus and toolbars to limit or enhance a user's access.

Third, be careful of dialog boxes that Word displays as a result of an action or set of actions. For example, the previous sample code creates a document, then inserts some text. If another line is added to quit Word, it might not be executed because Word displays a dialog box for saving the new document. If Word isn't visible, the application could appear hung when it's actually waiting for input from the user. If Word is visible, the user might not answer the dialog box the way your application expects. One possible solution is to minimize or eliminate dialog boxes. For example, the previous code should be modified to either save or close the new document before exiting Word. Another way is to manage the dialog boxes directly (with the *Dialogs* collection object) to preset options and check return values.

The Word Object Model

An object hierarchy or object model describes how objects of an application relate to each other, along with how the application's functionality is divided among them. As developers, we're familiar with objects and how to use them, but when they relate to each other, it helps to see them visually in a hierarchical diagram, such as the one shown in Figure 4. The Word object model is vast — nearly 200 objects with over one thousand properties and methods. Objects are often accessible only through other objects in a parent/child relationship. Child objects can have child objects of their own. For example, text in a Word document is four levels deep.

Before working with Word, it's important to understand how Word's content and functionality are defined in the object model. The object model is documented in the Help file vbawrd8.hlp, which is on the Microsoft Office CD, but is not installed by default. To install it, run the Microsoft Office Setup program, and select Add/Remove components. Select Help for Visual Basic under the Help options.

The overall structure of the object model resembles the structure of the user interface. This means that performing an action programmatically usually requires the same steps to perform the same action with the user interface. The object model is easier to follow if you keep this in mind. For example, the top-level object — the *Application* object — is the Word application itself. All Word functionality is under this object. The *Document* object is a child of the *Application* object. It is comprised of objects that make up the content and functionality of the document. The table in Figure 5 lists the steps to select text in a document with the user interface, and how to do the same steps programmatically.

The Word object model also uses the *Collection* object. A *Collection* is simply an object that contains a set of related objects. Microsoft Office has a naming convention for dealing with such objects: The name of the *Collection* object is usually the plural of the name of the objects it contains. For example, the *Documents* object is a collection containing *Document* objects. Consider the user interface. Word can contain multiple documents. Each document is a *Document* object, and all the *Document* objects are contained in a *Documents* collection. Just as a document can only be accessed from Word, a *Document* object is accessed from the *Documents* object.

Most collections provide a *Count* property and *Item* method to navigate and select specific objects. This example uses both to loop though all open documents and set the first word:

```
Application.Documents.Item(1).Words.Item(1).Text = "Hello."
```

Conclusion

This article discussed controlling Microsoft Word using run-time and compile-time Automation in Delphi. It also presented the Word object model and explained some of its conventions. The included sample source code will run in Delphi 4 and 5, and with Word 97 or 2000.

Next month, this series will continue by examining the Word components in Delphi 5, and how they simplify the Automation process. It will also look at how to link or embed documents into a Delphi application, and save them to tables. ∆

The sample application referenced in this article is available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\SEP\DI200009RG.

Ron Gray is a software developer specializing in business database applications. He has written numerous articles using different languages and is the author of LookUp Manager, a collection of components for visually managing lookup codes and abbreviations in applications. He can be reached at rgray@compuserve.com.

By Bill Todd

The Gold Standard, MIDAS & COM

Part II: Calling Servers and Deployment

Last month's article demonstrated building an application consisting of multiple modules that share a common database connection using MIDAS. Communication between the modules was implemented via COM, with the application's main module acting as a COM client, and the other module acting as a COM server. Communication from the server to the client was provided by events added to the server's dispatch interface and imported into the client using the Type Library Import wizard.

This month, we'll look at two other ways to allow the COM server to call methods of the COM client. To avoid confusion, I will refer to the application's main module and COM client as the customer module, and the application's COM server module as the orders module (both are available for download; see end of article for details).

The first technique is to add an Automation object to the customer module, so it can also function as a COM server with the orders module acting as its client. The second method will add a callback interface to the orders module. The customer module will include an object that implements the callback interface. When the customer module opens the orders module via COM, it will also create an instance of the callback interface and pass it to the orders module. The orders module can use this interface reference to call methods implemented in the customer module.

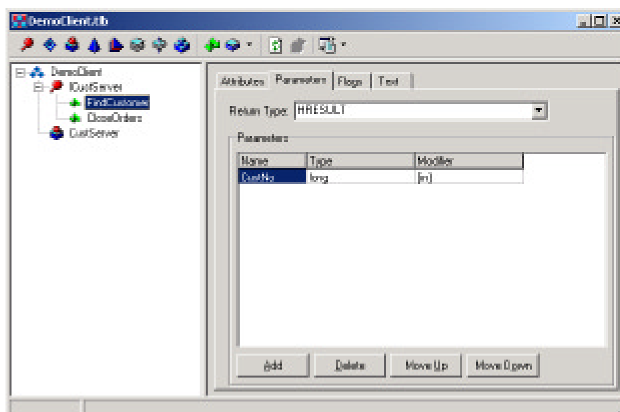


Figure 1: The *ICustServer* interface as it appears in Delphi's Type Library editor.

Everyone's a Server and Everyone's a Client

At first, it appears that the easiest way to allow two applications to communicate using COM — and have the ability to initiate an event in the other application — is to make each application a COM client and a COM server. And this technique works as long as you're careful. We'll start with last month's customer and orders modules as they stood before any events were added to the orders module's dispatch interface, and before the orders module's type library was imported, and the wrapper component was added to the customer module.

The first step is to open the Object Repository, go to the ActiveX page, and start the Automation Object wizard. Add an Automation object named *CustServer*, and then use the Type Library editor to add the *FindCustomer* and *CloseOrders* methods to the *ICustServer* interface (see Figure 1). Add a single parameter named *CustNo* of type long to the *FindCustomer* method.

Next, add the code for the *FindCustomer* and *CloseOrders* methods:

```
procedure TCustServer.CloseOrders;
begin
  OrderServer := nil;
end;

procedure TCustServer.FindCustomer
           (CustNo: Integer);
begin
  CustomerDm.FindByCustNo(CustNo);
  CustomerForm.Show;
end;
```

The next step is to switch to the orders module and add the code that calls back to the customer

module, starting with the *OnDestroy* event handler of the order form (see Figure 2).

When the user closes the order form, the order form has to notify the customer module, so it can close its Automation connection to the orders module. First, the code in Figure 2 frees the orders data module, *OrderDm*. Then it opens an Automation connection to the customer module by calling the *CoCustServer.Create* method added by the Automation Object wizard. Next the code calls the customer module's *CloseOrders* method and immediately closes the Automation connection. Notice also that the call to *CloseOrders* is protected by a **try..finally** block to ensure the Automation connection is closed.

This illustrates the one disadvantage of this technique. The orders module can only connect to the customer module for very brief periods of time, because a user could try to close the customer form at any time. If a user does try to close the customer form when another module has an open Automation connection to it, the warning message in Figure 3 will appear.

As you can see, this dialog box warns the user not to close the application while it has open COM connections, implying dire consequences if the application is closed. The only way to ensure the user never sees this warning is to make sure an Automation connection to the customer module persists for a very short period of time and only while the user is interacting with some part of the application other than the customer form. The need to open and close the Automation connection, and use a **try..finally** block for each call, also increases the amount of code you must write.

The final step is to add the following code to the *OnClick* event handler of the View | Customer menu choice. This code is identical to that in Figure 2, except it calls the *FindCustomer* method and passes the customer number from the current order record as a parameter:

```
procedure TOrderForm.Customer1Click(Sender: TObject);
begin
  CustServer := CoCustServer.Create;
  try
    CustServer.FindCustomer(
      OrderDm.OrdersCdsCustNo.AsInteger);
  finally
    CustServer := nil
  end; // try
end;
```

```
procedure TOrderForm.FormDestroy(Sender: TObject);
begin
  { Free the Orders data module. }
  OrderDm.Free;
  { Tell the customer form to close its connection
    to the orders form. }
  CustServer := CoCustServer.Create;
  try
    CustServer.CloseOrders;
  finally
    CustServer := nil;
  end; // try
end;
```

Figure 2: The orders form's *OnDestroy* event handler.

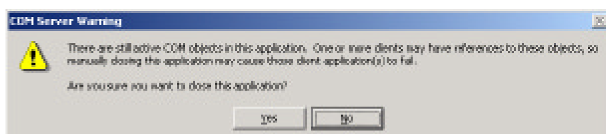


Figure 3: The open COM connections warning dialog box.

Using a Callback Interface

Allowing the orders module to use a callback interface to call methods in the customer module is a straightforward concept consisting of the following steps:

1) Add an interface to the orders module to define the methods to be called in the customer module.
2) Add the methods to be implemented in the customer module to this interface.
3) Add a method to the orders module's interface that receives and saves a reference to the callback interface.
4) Add the orders module's type library interface unit to the customer module.
5) Declare a type in the customer module that implements this new interface.
6) Create an instance of this type, and pass the interface reference to the orders module.

Figure 4 shows the Type Library editor after adding the *IOrderServerCallback* interface, and its *OnCloseOrders* and *OnFindCustomer* methods. Figure 4 also shows the *Connect* and *Disconnect* methods that were added to the *IOrderServer* interface. Note that the *Connect* method takes a single parameter of type *IOrderServerCallback*.

Here is the code for the *Connect* and *Disconnect* methods:

```
procedure TOrderServer.Connect(
  const Callback: IOrderServerCallback);
begin
  ClientCallback := Callback;
end;

procedure TOrderServer.Disconnect;
begin
  ClientCallback := nil;
end;
```

The *ClientCallback* variable used by the *Connect* and *Disconnect* methods is declared as a global variable of type *IOrderServerCallback* in the **interface** section of the *OrderServer* Automation object's unit. The *Connect* method receives a reference to the *InOrderServerCallback* interface, and saves this value in the *ClientCallback* variable. This interface reference is used by the methods in Figure 5 to call the *OnCloseOrders* and *OnFindCustomer* methods.

Now let's look at what is required on the client side to implement the callback interface. The first step is to add the orders module's type library interface unit, *DemoOrders_TLB*, to the project. In the

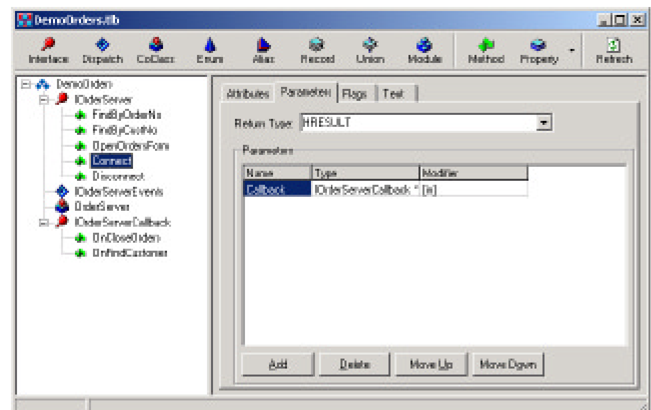


Figure 4: The orders module's type library with the callback interface.

customer form's unit, the declaration for the *TOrderEventHandler* class is added to the type declaration section just before the declaration for the *TCustomerForm* type:

```
type
  TOrderEventHandler = class(TAutoIntfObject,
                             IOrderServerCallback)
    procedure OnCloseOrders; safecall;
    procedure OnFindCustomer(CustNo: Integer); safecall;
  end;
```

This class implements the *IOrderServerCallback* interface and its methods. *TOrderEventHandler* descends from *TAutoIntfObject*, which has no class factory so it can only be instantiated by calling its constructor. This makes it private to this application since it cannot be instantiated through COM.

The real work on the client side is performed by the *OpenOrderServer* method of the customer form (see Figure 6). This method is called whenever the customer module needs to open the orders module via Automation. It begins by calling the *LoadRegTypeLib* Windows API function to load the *OrderServer* type library and save a reference to the type library in the *TypeLib* variable. Next, this method creates an instance of the *TOrderEventHandler* class, declared in the previous code snippet, and saves the reference in the *OrderEvent* handler, a private member variable of the *TCustomerForm* class. The third step is to open the Automation connection to the orders module, and save the interface reference in the *OrderServer* variable. *OrderServer* is a public member of the *TCustomerForm* class. Finally, the *Connect* method of the server is called and the *OrderEventHandler* reference is passed to it as a parameter.

```
procedure TOrderForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  { Free the Orders form when it closes. }
  Action := caFree;
  ClientCallback.OnCloseOrders;
end;

procedure TOrderForm.Customer1Click(Sender: TObject);
begin
  ClientCallback.OnFindCustomer(
    OrderDm.OrdersCdsCustNo.AsInteger);
end;
```

Figure 5: Calling methods through the callback interface.

```
procedure TCustomerForm.OpenOrderServer;
var
  TypeLib: ITypeLib;
begin
  if not Assigned(OrderServer) then begin
    { Load the orders module's type library. }
    OleCheck(LoadRegTypeLib(
      LIBID_DemoOrders, 1, 0, 0, TypeLib));
    { Create an instance of the TOrderEventHandler class. }
    OrderEventHandler := TOrderEventHandler.Create(
      TypeLib, IOrderServerCallback);
    { Open the automation server. }
    OrderServer := CoOrderServer.Create;
    { Call the Connect method of the OrderServer Automation
      object and pass the IOrderServerCallback interface
      reference to it. }
    OrderServer.Connect(OrderEventHandler);
  end; // if
end;
```

Figure 6: The *OpenOrderServer* method.

```
procedure TOrderEventHandler.OnCloseOrders;
begin
  CustomerForm.CloseOrderServer;
end;

procedure TOrderEventHandler.OnFindCustomer(
  CustNo: Integer);
begin
  CustomerDm.FindByCustNo(CustNo);
  CustomerForm.Show;
end;
```

Figure 7: The *OnCloseOrders* and *OnFindCustomer* methods of *TOrderEventHandler*.

The following code shows the *CloseOrderServer* method, which closes the Automation connection to the orders module by setting *OrderServer* to **nil**. Then it destroys the instance of *TOrderEventHandler* by setting *OrderEvent* handler to **nil**:

```
procedure TCustomerForm.CloseOrderServer;
begin
  OrderServer := nil;
  OrderEventHandler := nil;
end;
```

CloseOrderServer is called from the *OnCloseOrders* method of *TOrderEventHandler* (see Figure 7). It also shows the code for the *OnFindCustomer* method.

Which Method Is Better?

You have now seen three different techniques for enabling an Automation server to call back to its client. Using the event interface added by the Automation Object Wizard and the wrapper component generated by the Type Library Import wizard is certainly easy. The only disadvantage to this method is that considerable effort is required if you need to add additional methods or events after you have generated the wrapper component. This is because you must recreate the wrapper component. You also have to install the wrapper component on the palette to work with the application at design time.

The second approach, making both applications an Automation client and server, is very easy. However, you must limit the connection to the main application to a very brief interval, so a user will not try to close the program while the connection is open and get the open COM connection warning dialog box shown in Figure 3.

The last method, using a callback interface, requires a bit more code in the client, but is very easy to use and modify if you need to add additional callback events at any time. It doesn't require any special components on the palette, and adding additional callback events is as simple as adding the methods to the type library, writing the code for each method, and recompiling your application.

Deploying an Application

Because the architecture for modular applications described in these articles uses two technologies, MIDAS and COM, the rules for deploying these applications is a combination of the rules for deploying MIDAS applications and COM applications. Deploying a MIDAS application requires that you install and register MIDAS.DLL on the machine running the MIDAS client, and install and register both MIDAS.DLL and STDVCL50.DLL on the computer running the MIDAS server. You must also register the MIDAS server with Windows. Any other COM servers in your application must also be registered with Windows.

Commercial installation programs, including InstallShield Express, will automatically register COM servers for you. If the server is an EXE, you can also register it by running it once, or by running it with the /REGSERVER command-line switch. To register COM servers from your own application, see the *TRegSvr* sample application that comes with Delphi.

Conclusion

The marriage of MIDAS and COM provides an architecture for large applications that support multiple independent modules, rather than a single EXE with all of the modules sharing a common database connection through MIDAS. The benefit? This makes team development easier by allowing different programmers to work on different modules. It makes customization and updates easier by allowing you to modify or update a single module. It also allows you to build modules that can be shared among multiple applications without change. And who wouldn't want life to be easier? ∆

The files accompanying this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\SEP\DI200009BT.

Bill Todd is president of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is co-author of four database programming books, including *Delphi: A Developer's Guide*. He is a Contributing Editor to *Delphi Informant Magazine*, and is a frequent speaker at Borland Developer Conferences in the US and Europe. Bill is also a member of Team Borland and a nationally known trainer; he has taught Delphi programming classes across the country and overseas. He can be reached at bill@dbginc.com.

table of CONTENTS

OP TECH

RTTI / Delphi 5 / Persistence / Databases

By Keith Wood

Database Persistent Objects

Part I: Automatic Storage for Objects

Object Pascal provides full support for object-oriented programming. We can create new classes to extend any of the existing ones, adding whatever functionality we desire. Interfaces provide a way to enhance abilities outside the constraints of the class hierarchy. However, the objects we instantiate from these classes only exist for the duration of the application.

Delphi allows objects descended from *TComponent* to be written out to a stream to preserve their state. This is the basis of the form files (.dfm) that accompany many units. However, the content of these files is not easily manipulated outside of a Delphi program. It would be great if objects could be saved to a standard format that would allow for processing and management by other applications.

In this article, we explore how classes can be set up to automatically store themselves to, and retrieve themselves from, a standard relational database with minimal effort on our part. All we need to do is derive our new component from a particular base class, and publish the properties we want to be stored. We achieve this through the magic of Delphi's Run-time Type Information (RTTI).

Objects to Tables

Objects and relational databases have a natural symmetry. The class declaration defines what data fields are available for use in instances of that class, and acts as a template from which we can construct new objects. Similarly, the definition of a table in a relational database lays out which columns exist, along with their types, and allows us to insert records following this pattern. Objects that we create from a class definition correspond to individual rows inserted into a database table, and properties of a class map onto columns in the table. In a normal relational database, there is no provision for attaching behavior (methods) to a row, but — hey! — this is why we're using Delphi!

One of the benefits of object orientation is the ability to inherit attributes from a parent class. This inheritance can be ignored, treating each class as a single entity with the combined properties of all the classes, and mapped onto a single table. Alternately, it can be modeled in a relational database by having tables for each class in the hierarchy. At each level, we only have the properties that were introduced at that level. The tables are linked in a one-to-one relationship by some unique key.

In an application, classes usually interact to provide the desired functionality. These relationships can be stored as foreign keys in the relational database, with the object referred to being saved in its own table. In this way we can use the standard join abilities of the database to reconstruct the linkages. One-to-many relationships would be modeled just by storing the reference from the child back to the parent. Many-to-many relationships require a linking table in a relational database, reducing the arrangement to two one-to-many connections. Duplicating this in a class hierarchy would ease the mapping, while decreasing the coupling between the two original classes.

Although database tables generally require a primary key, objects need no such identifier. Typically, they're uniquely referenced via a pointer held in a variable. Due to the mapping we're attempting to implement, we'll require each class to have at least one field that is designated as the primary identifier for each object. This is used to save and retrieve individual records from the database.

RTTI

Delphi keeps track of the objects it manipulates, and makes a lot of that information available to us through the use of RTTI. From this we can determine the names and types of any published properties of a class (among other things), which we use to automatically enable the database persistence mechanism.

For RTTI to be available, the class of interest must be derived from *TPersistent*. RTTI can then provide

details about all the published properties of such classes. Recall that published properties are the same as public ones in that they can be accessed from anywhere the parent class can be seen, but that they also appear in the Object Inspector (for component-based classes). Delphi itself uses RTTI to achieve this. Similarly, we can use RTTI to inspect an arbitrary class and access its published property definitions. We derive our base class for database persistence, *TDBPersistent* (found in the DBPersist unit), directly from *TPersistent*. The TypInfo unit needs to be included in the **uses** clause, because it provides the access to RTTI.

Because the list of properties is related to the class itself, rather than any particular instance of it, we preprocess the class to extract all the necessary information and store it in an external variable, allowing faster access to it later. As Object Pascal has no class initialization mechanism, we must invoke this process in some other manner. The obvious place is in the **initialization** section of a unit. This guarantees that the properties are available as soon as the normal code starts executing. The *RegisterDBPersistentClass* procedure shown in Listing One (beginning on page 18) is for exactly this purpose: taking the class as its parameter.

First up, we check to see whether this class has already been registered. If so, we immediately exit because we only need to extract the information once. Otherwise, we register the class with Delphi's streaming system. This allows us to find the class later by name, and to construct a new instance of it on demand. Next, we create a specialized list and attach it to a global property list under the class' name. Note that "global" here means that the object exists externally to any instances of classes, but it is only accessible within this unit because we declare it in the **implementation** section. We need to use an external list because we only want one instance of it, regardless of the number of persistent classes and their objects.

To enable the class hierarchy to be reconstructed for use in creating a matching table structure, we must determine which properties were introduced in which parent classes. We cycle through the class information using *ClassParent* to move up the hierarchy, stopping when we reach *TDBPersistent*. At each level we call *GetPropList* (from the TypInfo unit) with the *tkProperties* parameter to retrieve the number of published properties declared there. Passing **nil** as the final parameter to this function simply returns the number of entries in the property list, rather than the list itself. Remember that published properties can't be un-published later. So once they're declared, they're available in all descendants. These counts are stored in a list for later use.

Next, we reserve enough space for the RTTI for the properties of this class, and load it with the details — again, using *GetPropList*. As we step through each entry in the list, we create a corresponding entry in the global property list. We map Pascal data types into standardized SQL types, as listed in *TDBPropType*:

```
{ Types supported in the database. }
TDBPropType = (ptSmallInt, ptInteger, ptEnumeration, ptSet,
               ptFloat, ptDate, ptTime, ptDateTime, ptChar,
               ptString, ptClass);
```

Where different types map into the same basic data type, we use the actual type name for further clarification, as can be seen with *TDateTime*, which is held as a floating point value. For string data we also need to determine the maximum length of the string. This is done by retrieving the type data information and extracting the desired value.

Note that we can handle Pascal sets and enumerated types and save these to the database. This is possible because Pascal stores them internally as integers. Similarly, we can store class references (also stored as integers, but with a different meaning), provided that the class also derives from *TDBPersistent*. We need this last restriction because we can't store the class references directly. They're actual memory addresses that would be meaningless when reloaded, and must instead store those properties defined as primary keys.

Primary key fields are very important, because they uniquely identify each object/record. To denote which properties are to be used as the primary key, we need some sort of flag that can be passed through RTTI, because this is all we have access to when building up the property list. There is a stored flag that indicates whether that property is retained, but it's not a good idea to subvert this for a different use. So we're left with the property name or the type name.

Because we'll be working with the property names throughout our code, we don't want to fiddle with these. Hence the indicator for a primary key is passed through the type name and consists of the first three characters, "TPK." This is pre-pended to the name of the underlying type, allowing that to be retrieved as necessary. So an integer property that is also the primary key would have a type of *TPKInteger*. Declarations for the primary key versions of the basic types are available in the DBPersist unit.

Finally, we look back through the list of parent classes and their properties to determine where each property was introduced. We do this by comparing them with the *NameIndex* value from RTTI. The resulting class name is transformed into a table name, and is stored in the property list with the other details we just extracted. For convenience, a list of the names of the primary key fields is also constructed as we go, and is saved as the final entry in the global property list.

Accessing all this information from each class is achieved through a number of class methods that select the appropriate property list from the global list, and then map into it directly. A class method is one that belongs to the class itself, rather than any particular instance of that class. This means it can be accessed through the class, without having to create an object. However, it also means that we cannot access any properties or methods specific to an instance, thus the need for the external property list. Having the keyword **class** in front of their declarations identifies these methods.

Database Mapping

Now that we have a list of the properties of our persistent class, we need to be able to map those into fields within a database table, and to submit actions to the database to save and retrieve them. We use SQL to manipulate the database, because this is a widely used standard allowing access to a large variety of relational databases.

Although we're using relational databases and accessing them through SQL, there are enough differences between the various dialects that we need to cater for specialized mappings for each one. To achieve this, we create a database manager, *TDBManager*, which handles all the interactions with the database. It's passed a *TDatabase* object to use for its communications, and creates a *TQuery* object internally to submit the actual requests.

To generate the required SQL code, it needs a reference to a mapper as specified by the *IDBMapper* interface. This serves to isolate the manager from the idiosyncrasies of each database. A standard implementation of this interface is provided by *TStandardDBMapper*, which

specifies the most common version of SQL for each type of request. This is then extended by specialized subclasses for each database. Implementations for Oracle, InterBase, and Paradox are included.

The actions we need to be able to perform for each object are: to save it to storage, read it back, alter it in storage, and remove it. These correspond to the standard INSERT, SELECT, UPDATE, and DELETE actions provided by SQL. For added convenience, we also implement methods that allow us to automatically create the table(s) necessary to store instances of a class, and to remove those tables when they're no longer required.

To be able to create an appropriate table for storing an object, we need to map the Pascal data types into those provided by the underlying database. This area is the one with the greatest diversity between databases (see Figure 1). Recall that we've already classified the Pascal types as standardized SQL types in the property lists. From here we need to map those standard types onto the actual types. The *GetType* method from the *IDBMapper* interface provides this transformation.

The names of the tables are taken directly from the class names (minus the initial "T"). Similarly, the names of the fields are taken directly from the property names. Following normal Pascal naming conventions creates names that are accepted by most databases. The only real limitation is the maximum length of a name, which — of course — varies from one database to another and is encapsulated in the *GetMaxNameLength* method.

Whenever we encounter a property that is a reference to another *TDBPersistent* object, we need to do some extra work. Because we can't store the class reference directly, we need to include a foreign key reference instead. We do this by locating the property list for the other class, and creating fields in the table corresponding to the other's primary key fields. Naming these fields requires combining the name of the property in the current class with the primary key property name(s) from the other. This is necessary because there may be multiple references to that foreign class in the current one, and/or the foreign property names may conflict with those in the current one.

As well as creating the table, we need to create its primary key. In standard SQL, this is specified as part of the table creation as a constraint (although provisions are made for constructing the index separately through the mapper interface). Furthermore, the fields that make up the primary key often need to be treated differently from normal fields. In particular, it's usually necessary to make these fields not nullable.

TDBPropType	Paradox	InterBase	Oracle
ptChar	CHARACTER(1)	CHARACTER	CHAR
ptClass	*	*	*
ptDate	DATE	DATE	DATE
ptDateTime	TIMESTAMP	DATE	DATE
ptEnumeration	INTEGER	INTEGER	NUMBER
ptFloat	NUMERIC	NUMERIC	NUMBER
ptInteger	INTEGER	INTEGER	NUMBER
ptSet	INTEGER	INTEGER	NUMBER
ptSmallInt	SMALLINT	SMALLINT	NUMBER
ptString	VARCHAR	VARCHAR	VARCHAR2
ptTime	TIME	DATE	DATE

Figure 1: Mapping Pascal types to SQL types. The asterisks denote class references replaced by foreign key fields.

A further complication is introduced when we follow a class hierarchy by mapping inherited classes into separate tables. In this case, we generate a number of table creation statements, one for each class in the hierarchy. For ease of processing, all the create statements are placed into a string list, and are then executed in turn. If there is no hierarchy, or we're not duplicating one in the database, then there will only be a single statement in the first list entry. The *UseTableHierarchy* variable in the DBPersist unit allows us to control the hierarchy behavior. It's set to True by default, causing any class hierarchy to be mapped into separate tables.

To create the table(s), all that's necessary is to instantiate an object of the appropriate type, and call its *CreateTable* method. We must have an object, because we need the link through to the database provided by the *DBManager* property. Removing the table(s) is just as easy, by calling the *DeleteTable* method of the object.

There and Back

The main interactions with the database are through saving and retrieving individual objects. Through the mapping interface, we can construct INSERT, SELECT, UPDATE, and DELETE statements.

INSERTs, UPDATEs, and DELETEs generate lists of statements when a class hierarchy is being used, one for each table. DELETEs are the simplest, needing only the table name(s) and a WHERE clause constructed from the primary key values. INSERTs build two lists for each table — one for the field names, and one for the values — before combining them into the complete statement. References to other classes are translated into a collection of fields corresponding to the primary key of that other class. If a class reference is **nil**, then nothing is inserted (defaulting to null). In the UPDATE statements we need to include all the fields, even unassigned class references. Setting the latter to null, when appropriate, is necessary to ensure the integrity of the database.

Obtaining the value of a property is achieved by using the property name, and one of the *GetXXXProp* functions from the TypInfo unit (as appropriate for the particular property type). The values are then converted into strings through the *ValueAsString* property of *TDBPersistent*, and can then be incorporated into a SQL statement.

Retrieving the values for an object is a little different from retrieving the values from the other actions. Although we could work through a class hierarchy and retrieve details from each table separately, there is no need to do this. Within a single SELECT statement, we can link all the appropriate tables, and return the composite record in one try. This reduces the traffic between the program and the database. We step through all the properties for a class, noting which table (class) they came from, and generate the corresponding SQL for them. Using table aliases, we can reduce the coding by labeling tables as "T*n*," with *n* ranging from zero through whatever is needed.

When the fields have been returned from the database, we extract them by name and map them back into the object, as shown in Listing Two (on page 19). Using the *ValueAsString* property allows us to set each one as a string value, making the final conversion internally, based on its actual type. Again, we rely heavily on RTTI. After determining the property type from the list for this class, we convert from the string value, and call one of the *SetXXXProp* procedures to transfer that value to the named property.

Another complication arises when we encounter a class reference property. Remember that we're storing the primary key from the

other class as a foreign key in this one. So, to restore the original link, we need to create an object of the appropriate type, set its primary key values from the foreign key fields retrieved, and then load it from the database — without prior knowledge of what the class is or what properties it contains.

Again it's RTTI to the rescue, along with some other abilities of the *TPersistent* class. First, we obtain the foreign key values from the current record by stepping through the primary key properties of the other class. These are concatenated into a single string for passing to the *ValueAsString* property. Here the property is recognized as a class reference, and passed to an internal function: *CreatePersist*. The attached class is located from its name (hence the need to register it with Delphi's streaming system earlier on), and a new object is created with the *NewInstance* method.

We then step through the properties for that class, and — for the primary key fields — transfer the values returned from the database into the appropriate fields. Finally, we can ask our new object to load itself from the database. This causes a big problem, because we're already in the middle of one query and are attempting to start another, whereas the database manager only uses a single query object. So we ask the manager to save the current query state, perform the new one, and then restore the original. It does this by placing the current one into a list and instantiating a new *TQuery* object.

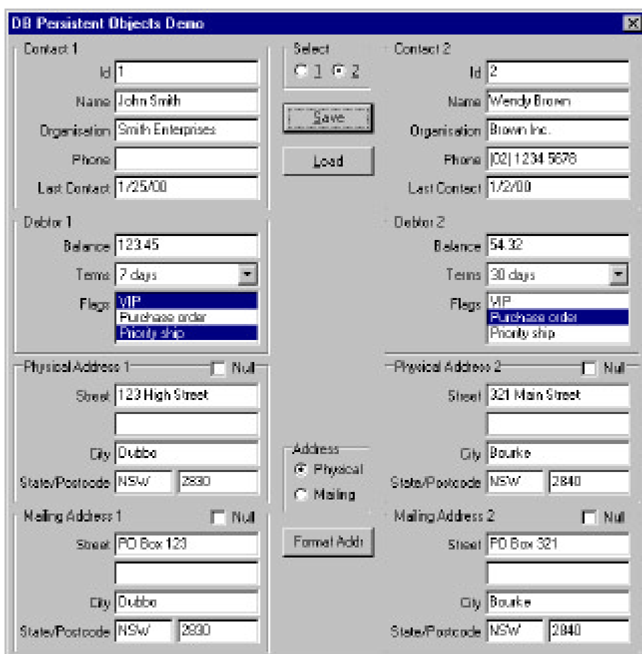


Figure 2: Demonstration of persistent objects.

Contact	ContactId	LastContact	MailingAddr_AddressId	Name	Organisation	PhysicalAddr_AddressId
1	1	1/25/00	102	John Smith	Smith Enterprises	101
2	2	1/2/00	202	Wendy Brown	Brown Inc.	201

Debtor	ContactId	Balance	Flags	Terms
1	1	123.45	5	1
2	2	54.32	2	2

Address	AddressId	City	Country	PostCode	State	Street1	Street2
1	101	Dubbo	Australia	2830	NSW	123 High Street	
2	102	Dubbo	Australia	2830	NSW	PO Box 123	
3	201	Bourke	Australia	2840	NSW	321 Main Street	
4	202	Bourke	Australia	2840	NSW	PO Box 321	

Figure 3: The objects in the database.

When the object has been loaded, a reference to it is passed back from the *CreatePersist* function, and is set as the property value. Obviously this process isn't perfect. If we have several references to the same object, we end up with multiple copies of it under this scheme. We need some way of locating a single master copy of each object and transferring all references to it there. This is an area for future expansion.

Demonstration

To show how all this works in practice, look at the project that accompanies this article (see end of article for download details). In the DBPDemo2 unit shown in Listing Three (beginning on page 19), three database persistent classes are declared: *TContact*, *TDebtor*, and *TAddress*. These include properties as examples of most of the available Pascal types, including references to address objects from contacts. The debtor class illustrates how class inheritance works in conjunction with database persistence.

Database access is established through one of the *TDatabase* components on the main form, *DBPDemo1*. These are set up for the InterBase IBLocal (*dbsIB*), or Paradox DBDemos (*dbsPdx*) databases. The uncommented line from the *FormCreate* method determines which one is used:

```
{ Create the database manager. }
dbm := TDBManager.Create(dbsPdx);   { Paradox database. }
{ dbm := TDBManager.Create(dbsIB);  { InterBase database. }
```

The database manager automatically instantiates the correct mapper for the supplied database. Feel free to use whichever is appropriate, or to substitute an Oracle database.

On start up, the main form creates the necessary database tables before displaying two *TDebtor* objects, along with their addresses (see Figure 2). The tables are dropped when the application terminates. Normally the tables would be created and retained to provide the necessary persistence.

The objects can be saved to the database (a good first step), and then viewed through a tool such as Database Desktop (see Figure 3). Changes made externally can be reloaded into the program. Note how the non-SQL types are stored: as sets, enumerated types, and class references.

Conclusion

Creating objects that model the real world make it easier to program interactions with and between them. Although Delphi provides for persistence of objects, this is only available in a binary format, and can't be easily used outside a Delphi program. By mapping an object into one or more tables in a relational database, we can achieve the desired result of persistence, along with a format that can be processed through other applications using standard SQL.

The classes just described allow an arbitrary class to be automatically saved to a relational database. Simply derive from *TDBPersistent* and publish its properties. The process is insulated from the specifics of a particular database through the database manager class and the mapper interface. Although the scheme we just described isn't ready for production work — as it has several outstanding issues — it does illustrate how the mapping can be done, and provides a solid base on which to work.

Although the persistence scheme described here has been designed to be as easy to use as possible, we can provide further assistance through a wizard that generates the necessary class unit. This is the subject of next month's article. ∆

This article is based on ideas presented by Claude Duguay in his article "Object/Relational Database Mapping" from the January 2000 issue of *JavaPro*.

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\SEP\DI200009KW.

Keith Wood is an analyst/programmer with CCSC, based in Atlanta. He started using Borland's products with Turbo Pascal on a CP/M machine. Often working with Delphi, he has enjoyed exploring it since it first appeared. You can reach him via e-mail at kbwood@compuserve.com.

Begin Listing One — *RegisterDBPersistentClass*

```
{ Extract property details for later use. }
procedure RegisterDBPersistentClass(
  clsDBP: TDBPersistentClass);
var
  slsAncestors: TStringList;
  slsPrimaryKeys: TStringList;
  clsClass: TClass;
  iIndex, iInherit, iMaxLength,
    iPropCount, iListSize: Integer;
  pplProps: PPropList;
  ppiInfo: PPropInfo;
  ptdData: PTypeData;
  ptType: TDBPropType;
  sPropName, sPropType, sTableName: string;
  bPrimaryKey: Boolean;
  lstProps: TPropertyList;
begin
  { Have properties already been processed into a list? }
  if slsGlobalProps.IndexOf(clsDBP.ClassName) > -1 then
    Exit;

  { Register the class with Delphi so it can be retrieved
    by name. }
  Classes.RegisterClass(clsDBP);

  lstProps := TPropertyList.Create;
  slsPrimaryKeys := TStringList.Create;
  slsAncestors := TStringList.Create;
  try
    { Add to the global list of properties. }
    slsGlobalProps.AddObject(clsDBP.ClassName, lstProps);
    { Find ancestor objects and their number
      of properties in order of inheritance. }
    clsClass := clsDBP;
    while clsClass.ClassName <> TDBPersistent.ClassName do
      begin
        slsAncestors.InsertObject(0, clsClass.ClassName,
          Pointer(GetPropList(clsClass.ClassInfo,
          tkProperties, nil)));
        clsClass := clsClass.ClassParent;
      end;

    { Load list of published properties for this object. }
    iPropCount :=
      Integer(slsAncestors.Objects[slsAncestors.Count-1]);
    iListSize := iPropCount * SizeOf(Pointer);
    GetMem(pplProps, iListSize);
    try
      GetPropList(clsDBP.ClassInfo,tkProperties,pplProps);
      { Translate this list into a more useable format. }
      for iIndex := 0 to iPropCount - 1 do begin
        ppiInfo := pplProps^[iIndex];
        sPropName := GetPropertyName(ppiInfo);
        sPropType := GetPropertyClassName(ppiInfo);
        { See if this property is a primary key field
          -- type has 'TPK' prefix. }
        bPrimaryKey :=
          (Copy(sPropType,1,Length(sPKPrefix))=sPKPrefix);
        if bPrimaryKey then
          begin
            slsPrimaryKeys.Add(sPropName);
            Delete(sPropType, 1, Length(sPKPrefix));
          end;
        { Translate the property type into
          ANSI SQL type (sort of). }
        iMaxLength := 0;
        case GetPropertyClassKind(ppiInfo) of
          tkInteger:
            if (sPropType = 'Byte') or
               (sPropType = 'ShortInt') or
               (sPropType = 'SmallInt') then
              ptType := ptSmallInt
            else
              ptType := ptInteger;
          tkFloat:
            if sPropType = 'TDateTime' then
              ptType := ptDateTime
            else if sPropType = 'TDate' then
              ptType := ptDate
            else if sPropType = 'TTime' then
              ptType := ptTime
            else
              ptType := ptFloat;
          tkChar, tkWChar:
            ptType := ptChar;
          tkLString, tkString, tkWString:
            begin
              ptType := ptString;
              ptdData := GetTypeData(ppiInfo^.PropType^);
              iMaxLength := ptdData^.MaxLength;
            end;
          tkEnumeration:
            ptType := ptEnumeration;
          tkSet:
            ptType := ptSet;
          tkClass:
            begin
              ptdData := GetTypeData(ppiInfo^.PropType^);
              if ptdData.ClassType.InheritsFrom(
                   TDBPersistent) then
                ptType := ptClass
              else
                raise EDBPersist.Create(Format(
                  sUnsupported, [sPropName, sPropType]));
            end;
          else
            raise
              EDBPersist.Create(Format(
                sUnsupported,[sPropName, sPropType]));
        end;

        { Find the object that introduced this property. }
        for iInherit := 0 to slsAncestors.Count-1 do begin
          sTableName := slsAncestors[iInherit];
          if ppiInfo^.NameIndex < Integer(
               slsAncestors.Objects[iInherit]) then
            Break;
        end;
        Delete(sTableName, 1, 1);
        { Add property details to the list. }
        lstProps.Add(TDBProperty.Create(
          GetPropertyName(ppiInfo), sTableName, ppiInfo,
          ptType, iMaxLength, bPrimaryKey));
      end;
      { Add list of primary keys to the list. }
```

```
      lstProps.Add(slsPrimaryKeys);
    finally
      FreeMem(pplProps, iListSize);
    end;
  finally
    slsAncestors.Free;
  end;
end;
```

End Listing One

Begin Listing Two — *TDBPersistent.LoadFromDB*

```
{ Transfer database fields to this object's properties. }
procedure TDBPersistent.LoadFromDB(dstData: TDataSet);
var
  iIndex: Integer;

  { Return the concatenated keys for this class. }
  function GetClassKey(sPropName, sForeignClass: string):
    string;
  var
    iProps: Integer;
    sPropField: string;
    lspProps: TPropertyList;
  begin
    Result := '';
    lspProps := GetPropertyList(sForeignClass);
    for iProps := 0 to lspProps.PropertiesCount - 1 do
      with lspProps.Properties[iProps] do
        if PrimaryKey then begin
          sPropField := Copy(sPropName + '_' + Name,
            1, DBManager.DBMapper.GetMaxNameLength);
          Result := Result + '`' + dstData.FieldByName(
            sPropField).AsString;
        end;
    Delete(Result, 1, 1);
  end;

begin
  for iIndex := 0 to GetPropertyCount - 1 do
    with GetProperty(iIndex) do
      if PropType = ptClass then
        ValueAsString[Name] :=
          GetClassKey(Name, GetPropertyClassName(PropInfo))
      else
        ValueAsString[Name] :=
          dstData.FieldByName(Name).AsString;
end;

{ Set the value of the property from a string. }
procedure TDBPersistent.SetValueAsString(
  sName, sValue: string);
var
  iIndex: Integer;

  { Create an instance of the nominated type and load it. }
  function CreatePersist(sForeignClass, sKeyValue: string):
    TDBPersistent;
  var
    iProps, iPos: Integer;
    clsPersist: TDBPersistentClass;
    lspProps: TPropertyList;
    sField: string;
  begin
    { Create a new object of the specified type. }
    try
      clsPersist :=
        TDBPersistentClass(FindClass(sForeignClass));
      Result := TDBPersistent(clsPersist.NewInstance);
      Result.FDBManager := DBManager;
      lspProps := GetPropertyList(sForeignClass);
    except
      raise EDBPersist.Create(Format(sUnknownForeign,
                                     [sForeignClass]));
    end;
    { Load up its key fields. }
    for iProps := 0 to lspProps.PropertiesCount - 1 do
      with lspProps.Properties[iProps] do
        if PrimaryKey then begin
          iPos := Pos('`', sKeyValue);
          if iPos > 0 then
            begin
              sField := Copy(sKeyValue, 1, iPos - 1);
              Delete(sKeyValue, 1, iPos);
            end
          else
            sField := sKeyValue;
          Result.ValueAsString[Name] := sField;
        end;
    { And read it from the database. }
    DBManager.SaveQuery;
    Result.SelectRecord;
    DBManager.RestoreQuery;
  end;

begin
  for iIndex := 0 to GetPropertyCount - 1 do
    with GetProperty(iIndex) do
      if Name = sName then begin
        { Set value based on property type –
          converting from string. }
        case PropType of
          ptEnumeration, ptInteger, ptSet, ptSmallInt:
            SetOrdProp(Self, PropInfo, StrToInt(sValue));
          ptFloat:
            SetFloatProp(Self, PropInfo,
                         StrToFloat(sValue));
          ptDate, ptDateTime, ptTime:
            SetFloatProp(Self, PropInfo,
                         StrToDateTime(sValue));
          ptChar:
            SetOrdProp(Self, PropInfo, Ord(sValue[1]));
          ptString:
            SetStrProp(Self, PropInfo, sValue);
          ptClass:
            if sValue = '' then  { Null. }
              SetOrdProp(Self, PropInfo, Integer(nil))
            else
              SetOrdProp(Self, PropInfo, Integer(
                CreatePersist(GetPropertyClassName(
                PropInfo), sValue)));
        end;
        Exit;
      end; // if
end;
```

End Listing Two

Begin Listing Three — *TDBPersistent* demonstration classes

```
type
  String3  = string[3];
  String15 = string[15];
  String30 = string[30];
  String50 = string[50];

  TDebtorTerms = (dtCOD, dt7Days, dt30Days);

  TDebtorFlag = (dfVIP, dfPurchaseOrder, dfPriorityShip);
  TDebtorFlags = set of TDebtorFlag;

  TAddress = class(TDBPersistent)
  private
    FAddressId: TPKInteger;
    FCity: String50;
    FCountry: String30;
    FPostCode: String15;
    FState: String3;
    FStreet1: String50;
    FStreet2: String50;
  public
    constructor Create(dbmManager: TDBManager;
```

```
    AddressId: TPKInteger);
  procedure Assign(Source: TPersistent); override;
  procedure FormatAddress(slsAddress: TStrings);
published
  property AddressId: TPKInteger
    read FAddressId write FAddressId;
  property Street1: String50
    read FStreet1 write FStreet1;
  property Street2: String50
    read FStreet2 write FStreet2;
  property City: String50 read FCity write FCity;
  property State: String3 read FState write FState;
  property PostCode: String15
    read FPostCode write FPostCode;
  property Country: String30
    read FCountry write FCountry;
end;

TContact = class(TDBPersistent)
private
  FContactId: TPKInteger;
  FLastContact: TDate;
  FMailingAddr: TAddress;
  FName: String50;
  FOrganisation: String50;
  FPhysicalAddr: TAddress;
  FPhone: String15;
public
  constructor Create(dbmManager: TDBManager;
    ContactId: TPKInteger);
  procedure Assign(Source: TPersistent); override;
  procedure FormatAddress(slsAddress: TStrings;
    bMailing: Boolean);
published
  property ContactId: TPKInteger
    read FContactId write FContactId;
  property Name: String50 read FName write FName;
  property Organisation: String50
    read FOrganisation write FOrganisation;
  property Phone: String15 read FPhone write FPhone;
  property PhysicalAddr: TAddress
    read FPhysicalAddr write FPhysicalAddr;
  property MailingAddr: TAddress
    read FMailingAddr write FMailingAddr;
  property LastContact: TDate
    read FLastContact write FLastContact;
end;

TDebtor = class(TContact)
private
  FBalance: Double;
  FFlags: TDebtorFlags;
  FTerms: TDebtorTerms;
public
  procedure Assign(Source: TPersistent); override;
published
  property Balance: Double read FBalance write FBalance;
  property Terms: TDebtorTerms read FTerms write FTerms;
  property Flags: TDebtorFlags read FFlags write FFlags;
end;
```

End Listing Three

By Andrew J. Wozniewicz

Raw API Programming

When Size and Speed Are Paramount

Developing Windows applications in Delphi without the use of the VCL is a technique as old as Windows itself. It requires no additional tools or libraries beyond the standard Windows and Messages units. However, it does require a lot of patience and perseverance, because it's very tedious to develop applications this way.

In some cases, however, the result is more appealing for reasons of efficiency or interoperability. This article explores this time-honored, raw-API-style of developing Windows applications. By the end, you'll be able to decide if it's something you'll dare to undertake.

To VCL, or Not to VCL

At its core, Delphi owes its awesome power largely to its native collection of components, classes, and subroutines known as the Visual Component Library, or simply the VCL. The VCL is at the heart of the RAD capability of Delphi, and provides a framework for just about any type of Windows application — from a simple "Hello World" application, to a sophisticated network server, to a distributed application suite.

There is a price to pay for this power, however. Even the simplest Delphi/VCL application has a several-hundred-kilobyte footprint, and grows rapidly with the addition of any component, form, or data module. Moreover, the VCL defines a complex dispatch mechanism for routing Windows messages to the VCL objects. This mechanism allows for a VCL object like *TForm* to be seamlessly coupled with its corresponding Windows object (the actual physical window denoted by its window handle), but it introduces additional layers of handling and processing. These extra levels of indirection, in turn, cause delays in handling the messages received by the application.

Even more importantly, the extra programming layers introduce their own idiosyncrasies on top of the Windows API for a programmer to learn. As if there weren't enough complexity already! Sometimes cutting through these Baroque embellishments is the best course of action.

The problem of disk space, memory, and efficiency of execution may be totally irrelevant for most business applications you are likely to write. Far too many programmers spend their precious time worrying about petty efficiency issues, when they should spend it polishing the functionality of their applications, or ensuring their applications' robustness. With the prices of secondary storage falling steadily and with the processor speeds reaching into gigahertz ranges, it seems that most of these worries should be a thing of the past.

Yet some special applications will definitely benefit from a direct, close-to-the-API approach. It's for these rare occasions that this article was written (see the sidebar "Why on Earth Would You Do This?" on page 27 for a further discussion). Using this approach will also give you a new appreciation of the programming power placed at your disposal by the creators of the VCL. Note that what we're creating is a true, GUI-mode application with a main window, caption bar, menu, etc., not a character-mode console application, which is another issue altogether.

Figure 1: The sample program at run time — a minimum Windows application.

The Main Window

Figure 1 shows what our API-style application is going to look like at run time. A minimum Windows application shows a blank window, possibly with a caption, i.e. title bar.

Figure 2 shows the complete source of the raw API Hello World program written in Object Pascal (available for download; see end of article for details). The listing shows the contents of the .dpr file. Many Delphi programmers don't even know that you can mess around with the main project source (Project | View Source from the Delphi menu). You can — as long as you follow the rules.

Figure 2 gives you an idea of how to create a very simple raw-API application, but let me warn you that an application that actually does something would take a lot more work. For one thing, we do not concern ourselves here with details like menus, dialog boxes, toolbars, accelerators, and myriad other important elements.

Details Explained

When you compile and run the sample program from Figure 2, you'll notice it behaves the way you would expect a Windows application to behave. You can move its window around the screen or resize it with the mouse. You can maximize or minimize the window to the task bar via the system menu. You can also close the window to terminate the program. All of this default functionality is provided by Windows "for free," or for the cost of actually creating the window with a function call and setting up a so-called message loop. Let's now examine the code to discover how it accomplishes the feat of being a well-behaved Windows GUI application.

The code begins with the **uses** clause listing Messages and Windows — two standard Delphi units. The Windows unit contains most of the Windows API procedure and function declarations, and the Messages unit has a lot of constants identifying standard Windows messages. Once inserted into our **uses** clause, these two units make all these API declarations available to our program.

The *WndProc* function definition that follows is a callback function traditionally called a "window procedure." We'll discuss it later. Let's first concentrate on the contents of the main program block.

Registering the Window Class

Before it can create any visible window, a Windows application must register a window class. A window is always created based on a particular window class. The class determines certain characteristics of all windows based on it, while the individual windows may define some additional characteristics at the time of their creation, as you will see shortly.

Please note that window classes have nothing to do with Delphi object classes. A window class is a system-level (Windows) entity. A Delphi class is an object-oriented programming (Object Pascal) construct. The two are not related for any practical purposes. When programming with the standard VCL, you get the illusion that a Delphi object (a *TForm* instance) is the same thing as a visible window that shows up on the screen. The reality is that it takes the complex machinery of the VCL to create this convenient illusion.

It's time for some good news and some bad news back in the realm of the raw API. The good news is that more than one window can be created based on a single window class. The bad news is that even the simplest window like ours must have a registered class, so we have to go through all the motions of registering it.

```
program NonVCLHelloWorld;

uses
  Messages, Windows;

{ $R *.RES. }

const
  AppName: array[0..10] of Char = 'HelloWinApp';

function WndProc(HWnd: HWND; Msg: UINT; wParam: WPARAM;
  lParam: LPARAM): LRESULT; stdcall;
begin
  case Msg of
    WM_DESTROY:
      begin
        PostQuitMessage(0);
        Result := 0;
      end
    else
      Result := DefWindowProc(HWnd, Msg, WParam, LParam);
  end;
end;

var
  C: TWndClass;
  Handle: HWND;
  Msg: TMsg;
  Result: Integer;
begin
  with WClass do begin
    style := CS_VREDRAW or CS_HREDRAW;
    lpfnWndProc := @WndProc;
    cbClsExtra := 0;
    cbWndExtra := 0;
    C.hInstance := HInstance;
    hIcon := LoadIcon(HInstance, IDI_APPLICATION);
    hCursor := LoadCursor(HInstance, IDC_ARROW);
    hbrBackground := GetStockObject(WHITE_BRUSH);
    lpszMenuName := nil;
    lpszClassName:= @AppName;
  end;
  RegisterClass(WClass);
  Handle := CreateWindow(@AppName, 'Hello World!',
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, CW_USEDEFAULT, 0, 0, HInstance, nil);
  ShowWindow(Handle, SW_NORMAL);
  UpdateWindow(Handle);
  while GetMessage(Msg,0,0,0) do begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
  end;
end.
```

Figure 2: The source code of the raw API Hello World program.

The class must be registered with Windows with the *RegisterClass* function call. There also is an *UnregisterClass* API function that does the opposite: removes the class registration. However, its use is not strictly necessary in a 32-bit Windows program, since all windows classes registered by the application will automatically be unregistered when the application terminates.

The *RegisterClass* function registers a window class for subsequent use in calls to the *CreateWindow* or *CreateWindowEx* functions. The *RegisterClass* function takes a single parameter of type *TWndClass*. This parameter is actually a record that must be filled in with the appropriate class attributes before being passed to the function. Our main program begins by filling in this record structure, which, in our case, is called *WClass*.

The *TWndClass* Record Structure

The declaration of the *TWndClass* record is found in the Windows.pas unit and is reproduced in Figure 3.

```
type
  tagWNDCLASSA = packed record
    style: UINT;
    lpfnWndProc: TFNWndProc;
    cbClsExtra: Integer;
    cbWndExtra: Integer;
    hInstance: HINST;
    hIcon: HICON;
    hCursor: HCURSOR;
    hbrBackground: HBRUSH;
    lpszMenuName: PAnsiChar;
    lpszClassName: PAnsiChar;
  end;

  TWndClassA = tagWNDCLASSA;
  TWndClass = TWndClassA;
```

Figure 3: The declaration of the *TWndClass* record.

Notice that the declaration is indirect and that several aliases for this record type are created. The actual structure of the record is given by the *tagWNDCLASSA* type declaration. The extra names are introduced merely for the convenience of someone translating code literally from C. In Pascal programs, we will stick with the *TWndClass* type.

Filling in the style field of the *TWndClass* record can specify a window class style. You can use any combination of the available style values by combining them with a bit-wise **or** operator. The values available for this field are listed in the Delphi WinAPI online Help (search for *RegisterClass* first, then follow the link to the WNDCLASS structure). Examples include CS_DBLCLKS, CS_GLOBALCLASS, CS_NOCLOSE, etc.

In our simple application, we're only interested in a small subset of available styles. We are setting the class style to the combination of (CS_VREDRAW **or** CS_HREDRAW), which means that the entire client area of the window will be redrawn whenever the size of the window changes in a horizontal or vertical dimension, or both.

The *lpfnWndProc* field of the *TWndClass* record is an address of the window procedure for all windows of that class. We'll be discussing the window procedure a little later. For now, suffice it to say that we must provide this procedure in our application if we want to be able to process Windows messages sent to our window. Since we're interested in processing some messages, we created the *WndProc* function in our program. Here, we set the *lpfnWndProc* field of the class structure to the address of this function.

Continuing with the process of filling in the *TWndClass* structure, we set the *cbClsExtra* and *cbWndExtra* fields to zero. These fields define the class extra bytes and window extra bytes, respectively. In practice, these parameters hold the keys to the kingdom of application frameworks, since they may be used to associate an object (such as an instance of a Delphi class) with the window class or the individual window. For our purposes, they aren't needed right now, so we set them to zero, meaning that we don't need any extra bytes for data to be associated with our class, or any of its windows. The *hInstance* field identifies the application instance. We simply pass whatever value the standard *System.HInstance* variable contains.

The *hIcon* field identifies the class icon. A class icon is used when a window of that class is minimized. This field must be a handle of an icon resource. An icon resource is not the same as a *TIcon* VCL object. Here we explicitly use the:

```
LoadIcon(HInstance, IDI_APPLICATION)
```

call to retrieve the handle of the standard default icon for a Windows application. The IDI_APPLICATION constant is also defined in Windows.pas.

In a similar manner, we set the *hCursor* field of the class structure to a handle denoting the default mouse cursor shown when it hovers over the window's client area. In our case, we request the default arrow cursor handle from Windows by calling the *LoadCursor* API function with our application's instance handle and IDC_ARROW arguments.

The *hbrBackground* field indicates the handle to a brush to be used to paint the background of a window. We make use of the *GetStockObject* function to retrieve one of the Windows system brushes, which is the white brush in our case. You can experiment with other values to *GetStockObject*, such as LTGRAY_BRUSH, BLACK_BRUSH, or even NULL_BRUSH. You can also provide your own custom brush created with a *CreateBrushIndirect* function call.

As an aside, notice how the concept of a handle frequently applies to raw API programming. A window handle, an icon handle, and a cursor handle are but a few examples of API handles. Handles are 32-bit token numbers that allow us to refer to system resources. Internally, these values are indeed pointers to structures describing the resources within Windows, but we use them in an abstract way, passing them around to various API functions. Windows hides the details of its internal structures from us behind the façade of the API calls. These handles survive even at the level of VCL objects in the form of a *Handle* property, e.g. *TFont.Handle*, *TForm.Handle*, etc.

It is also worth noting that the *hbrBackground* field could, instead of a handle to a physical brush, be set to a color value from a user-predefined palette of system colors. For example, we might have specified:

```
hbrBackground := COLOR_BTNFACE + 1;
```

to obtain the color effect identical to that employed by a standard Delphi form. The COLOR_XXXX system constants are also declared in the Windows unit. The "plus one" part must be there to ensure consistency with the constant's meaning (ever heard of the "off by one" programming error?).

We're almost done filling in the *TWndClass* structure; only two fields remain. The first is *lpszMenuName*, which we set to **nil** because our simple application has no menu defined in a resource file. If we were to enhance it, this could be the name of the main menu resource for the application.

The second field is *lpszClassName*, a pointer that needs to receive the address of the zero-terminated string with the name of the class. The name we invent is HelloApp; it's simply declared as character array constant *AppName*. Object Pascal makes it easy to deal with even the C-style zero-terminated strings required by many API functions. We simply pass the address of the character array thus defined as the value of the *lpszClassName*, and we are ready to register the class.

The *RegisterClass* Call

Fortunately, the entire complexity of registering the class resides in correctly filling the *TWndClass* structure. The call to *RegisterClass* is trivial in comparison.

If the *RegisterClass* function succeeds, we get back a unique ATOM that identifies the class. An ATOM is a Word-sized token value that identifies the unique class name within the system; it's just another type of a system handle. We can simply ignore the return value here. A more sophisticated application may wish to check for it being zero, meaning that the class registration failed.

Creating the Window

Now that we have the class registered, we can finally deal with the business of creating our window. The window class we so diligently defined specifies general characteristics of a window, thereby allowing the same window class to be used for creating many individual windows.

Remember, individual windows are always created based on registered classes, so the effort expended in registering our class was really necessary. We need to do more work, however, to create the actual window. We'll start by specifying the parameters to the *CreateWindow* function.

As you know by now, every window in Windows has a unique handle, i.e. a number that identifies it to the system. If your program creates multiple windows, each of these windows will have its own, different handle. The call to the *CreateWindow* function returns a window handle:

```
Handle := CreateWindow(@AppName, 'Hello World!',
  WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
  CW_USEDEFAULT, CW_USEDEFAULT, 0, 0, HInstance, nil);
```

The returned handle is a key aspect of a window, allowing the system to communicate messages to our window, and also allowing us to identify our window in any of the numerous system function calls. We tuck it neatly into a global variable *Handle*.

The *CreateWindow* Function

The *CreateWindow* function is declared in the Windows unit as follows:

```
function CreateWindow(lpClassName: PChar;
  lpWindowName: PChar; dwStyle: DWORD;
  X, Y, nWidth, nHeight: Integer; hWndParent: HWND;
  hMenu: HMENU; hInstance: HINST; lpParam: Pointer): HWND;
```

Its job is to create an overlapped, pop-up, or child window. It specifies the window class, window title, window style, and (optionally) the initial position and size of the window. The function also specifies the window's parent or owner, if any, and the window's menu.

Not surprisingly, the first parameter of the *CreateWindow* function specifies the window class. All that hard work with registering the class didn't go to waste after all! The class name can be any name previously registered with the *RegisterClass* function, or any of the predefined controls: *Button*, *Edit*, *Combobox*, etc. Of course, it makes very little sense to attempt to use a button as an application's main form. But it's worth observing that many standard controls you know as Delphi components are actually true windows in their own right, with their own window procedures and all. When they appear in a Delphi application, they result from a *CreateWindow* call buried somewhere deep inside the VCL.

Let's return to the *CreateWindow* function. The third argument is *dwStyle*, which allows us to specify the window style. The possible values for window styles are too numerous to list here, but you can find them in the Windows.pas unit as WS_XXXX constants, such as WS_OVERLAPPED, WS_CAPTION, WS_BORDER, etc. They're also listed in the online Help (search for the *CreateWindow* function in the Win32 API Help). Similar to the case with the class styles, you can combine several styles together with the **or** operator.

Please note that a window style and a class style are not the same thing. Each of these "style" settings controls slightly different aspects of the window's appearance and behavior. While the class styles apply to all instances of that class, i.e. all windows of a given class, the window styles only apply to a specific window identified by a window handle.

The next two arguments to *CreateWindow* — *X* and *Y* — specify the initial position of the left, top-most corner of the window, in screen pixels, relative to the entire screen. You can either specify actual values here, which will ensure that your window is always displayed at exactly the designated coordinates on the screen, or you can let Windows determine the default placement by passing CW_USEDEFAULT instead — a more common approach. This is exactly what we did. Hence the window will appear each time in a position determined by Windows.

Similarly, the *nWidth* and *nHeight* arguments are either the desired width and height of the window in pixels, or we can also pass the CW_USEDEFAULT constant for each, to let Windows determine the initial vertical and horizontal size.

Next is the *hWndParent* argument. It gives us the opportunity to create our window as a child window. When you pass an existing window's handle in place of this argument, you would have created a dependent child window. Child windows always appear on the surface area of their parents. For example, all controls — such as buttons, edit boxes, and list boxes — are children. Their *CreateWindow* calls must include handles to their own parent windows. In our case, because the top-most window of any application doesn't typically have a parent, we simply pass a zero.

The *hMenu* parameter allows us to specify the main menu for the window. It must be a handle of a menu created with a call to *CreateMenu*, or it must be *LoadMenu* API functions. It does seem redundant, since we had an opportunity to specify a menu when we were registering the class. However, the default menu for the window may only have come from a static resource bound to our application. Here, at the time of the window creation, we also have an opportunity to create a new custom menu on-the-fly, or even to load a menu resource defined in a separate executable, such as a DLL or a similar plug-in.

We have already encountered the next parameter, *hInst,* when registering the class. It's the same old *HInstance* value from the System unit that identifies the instance of our running application.

Last but not least, we have the *lpParam* that we can pass to the *CreateWindow* function. This argument is an open invitation to bind the window to a user object, such as an instance of a Delphi class. It essentially allows us to pass an arbitrary 32-bit value, such as a pointer or an object reference, to the window procedure so the value will be retrieved when processing the WM_CREATE message. Here, we don't bother to use it, so we pass a **nil** value.

Congratulations! After all this work, you have succeeded in creating a window. You can tell that the creation was successful by examining the *Handle* returned by the *CreateWindow* call. A zero returned for the handle indicates an error.

Showing the Window

Creating the window with the *CreateWindow* call is not sufficient. It creates the window internally in the system, but the window does

```
type
  tagMSG = packed record
    hwnd: HWND;
    message: UINT;
    wParam: WPARAM;
    lParam: LPARAM;
    time: DWORD;
    pt: TPoint;
  end;

TMsg = tagMSG;
```

Figure 4: The *tagMSG* record structure.

not yet appear on the display. We must explicitly make the window appear via a call to *ShowWindow*:

```
ShowWindow(Handle, SW_NORMAL);
```

As you can tell, *ShowWindow* takes the window handle we carefully preserved in the previous creation step, and then an argument that tells it whether to show the window in a minimized, maximized, or normal state.

Although it's not crucial for our very simplistic application, we also call *UpdateWindow* to force an initial repainting of the Window's client area. Since we are not painting anything in the client area yet, this call is not strictly necessary, but we better make sure it's not forgotten later.

The Message Loop

We've finally arrived! After the *UpdateWindow* call, our main application window is up and visible on the screen. The time has come for our program to make itself ready to interact with the user by accepting mouse and keyboard events. Windows maintains a message queue for each program running. When an event such as a keystroke or a mouse movement occurs, Windows translates that event into a message that is being placed in the relevant application's message queue.

The program is now ready to enter a so-called message loop in which one by one, the messages placed by the system in our message queue will be retrieved and processed. The basic pattern of a message loop is as follows:

```
while GetMessage(Msg,0,0,0) do begin
  TranslateMessage(Msg);
  DispatchMessage(Msg);
end;
```

It consists of a loop, during which consecutive messages are being retrieved from the message queue via calls to *GetMessage*. The first parameter of *GetMessage* is another record structure defined in the Windows unit (see Figure 4).

The definition in Figure 4 includes a *TPoint*, which is itself another API structure:

```
TPoint = record
  x: Longint;
  y: Longint;
end;
```

When we look at the declaration of *GetMessage* in the Windows unit, the *Msg* structure appears as a by-reference **var** parameter:

```
function GetMessage(var lpMsg: TMsg; hWnd: HWND;
  wMsgFilterMin, wMsgFilterMax: UINT): BOOL; stdcall;
```

A call to *GetMessage* causes the *lpMsg* parameter to be filled with the current message data. In our program, the second, third, and fourth parameters are set to zero to indicate that we want to process all messages generated to all windows in our program, so no filtering should take place.

When a call to *GetMessage* returns, Windows would have filled in the fields of the message record with the contents of the next message from the message queue. The *hWnd* field of the message structure is the handle of the window to which the message is directed. Since our program only creates one window, this value is always the same as the value we stored in the global *Handle* variable.

The *Message* field contains the integer identifier of the message, for example WM_CLOSE, WM_LBUTTONDOWN, WM_QUERYENDSESSION, etc. The identifier distinguishes one kind of message from another. For each kind of Windows message, there is a corresponding message identifier constant defined in the *Windows* unit, each starting with the WM_ prefix, for "Windows message."

Each message may carry additional information specific to that message. For example, WM_LBUTTONDOWN, which is a message sent to a window when the user clicks the left mouse button over the client area, provides additional information about the location of the click in *x* and *y* client-relative coordinates. Any such additional information is passed via a 16-bit or 32-bit *lParam*.

The time field of the message structure carries the information about the system time when the message was placed in the queue. The *pt* field has the mouse coordinates at the time the message was placed in the queue.

The whole message loop works as follows: If the *Message* field of the *TMessage* structure retrieved from the message queue is anything but WM_QUIT, then *GetMessage* returns a non-zero value. If the message happens to be WM_QUIT, *GetMessage* returns zero, and the **while** loop terminates, thereby terminating the whole program. Inside the message loop, the statement:

```
TranslateMessage(Msg);
```

passes the message back to Windows for some pre-processing. What happens is that keyboard keystroke messages get translated into character messages according to a country-dependent keyboard configuration. The correct translation from a physical key to a particular character code becomes especially important with non-English versions of Windows. The second statement inside the message loop:

```
DispatchMessage(Msg);
```

sends the newly retrieved message to the window procedure of the target window for processing. We'll talk more about the window procedure momentarily. Once the window procedure is finished processing the message, control returns to *DispatchMessage*, then back to the message loop, and the message loop continues with the next *GetMessage* call. This all happens many times during the course of a typical Windows application session, and ends whenever *GetMessage* returns zero, i.e. when the message is WM_QUIT.

The Window Procedure

Everything described up to this point is standard boilerplate code

that gets a simple Windows application up and running — overhead, in short. So far, the window class has been registered, the window has been created and displayed on the screen, and the message loop is actively retrieving messages from the queue. The real application-specific action within any program takes place inside the window procedure, which we will dissect right now. The window procedure in our Hello World program is called *WndProc*. When registering the window class for our main window, we have indicated *WndProc* to be the default window procedure. Every window procedure must be declared in exactly the same way:

```
function WndProc(HWnd: HWND; Msg: UINT; wParam: WPARAM;
  lParam: LPARAM): LRESULT; stdcall;
```

The name of the window procedure that you declare in your program is unimportant. We could have named our window procedure *MainWindowProc*, for example, and it would work just fine. What is important, however, is that the "procedure" (which, by the way, is really a Pascal function returning an integer value) must declare the exact types of the parameters and the return value just like the previous code. Names of these parameters are again unimportant, but their types and the order in which they are listed do matter. Furthermore, the window procedure must be declared with a standard-call calling convention.

The role of the window procedure is to process any messages sent to the window. We do not call this subroutine explicitly in our code; instead, we let *DispatchMessage* do this job — to call our window procedure indirectly. This indirection is crucial to understanding the flow of control in a Windows program: The window procedure is the lowest-level "event handler" for any events affecting a given window. It is typically a piece of code that just sits there waiting to be called by Windows in response to a user action.

In contrast, the main program block behaves more like a traditional, top-down application. First it runs its initialization code, then it does its processing, and finally it terminates, thereby halting the entire application. The apparent disconnect between the main program block executing sequentially and the window procedure, which just waits to be briefly called whenever there is a message to process with Windows intervening in the middle, is at the very foundation of the cooperative, event-driven architecture of Windows.

Inside a Window Procedure

Let's now examine the structure of a window procedure. I've already mentioned that the term "window procedure" is really a misnomer. The window subroutine really is a function, returning a 32-bit integer value. The meaning of the return value could be different for different messages, but generally is zero when the window procedure processes the message, and non-zero otherwise. (Yes, it's possible to ignore messages sent to a window; more about that shortly.)

A window procedure declares an *HWnd* parameter, which is the handle of the window for which the message was intended. Again, our program only deals with one window total, but more complex applications may have several windows of the same class open at the same time. Each of these windows would typically be associated with the same window procedure.

The second parameter to a window procedure is the message identifier, such as WM_CLOSE or WM_LBUTTONDOWN. The remaining two parameters are the extra bits of data specific to each message that is being passed along to provide more information about the event that occurred. They are called message parameters.

Processing Window Messages

A unique number, passed as the *Message* parameter to the window procedure, identifies each message that a window procedure may receive. The message identifiers are declared as constants such as WM_CLOSE, WM_QUIT, etc. Typically, a **case** construct inside the window procedure would determine the logic for processing each type of message.

Our window procedure, since it belongs to a minimalist application, is concerned with only one message: WM_DESTROY. This is a message that the system sends to each window just before destroying that window's internal data structures. This, in turn, gives the window procedure an opportunity to clean up any resources that may have been associated with that window.

We also intercept the WM_DESTROY message to indicate the termination of the entire program. The call to *PostQuitMessage* will eventually terminate the message loop in the main block of the application. Without this call, the main window would close and be destroyed, but the message loop would never terminate, since there would be no WM_QUIT message to cause *GetMessage* to return zero. The now windowless process would simply linger in the background until it was manually killed from the Task Manager's Close Program dialog box.

As a side note, it is possible to write programs without a main window, or any window at all. A GUI program without a user interface? This isn't necessarily an oxymoron, since this is precisely how some service applications, also known as daemons, are written.

Unprocessed Messages

What happens with the multitude of other messages that a window receives during the course of its life, but is not interested in processing? Some of these messages must be processed, lest the application be totally unresponsive to the user. For example, we want the standard responses to the user resizing the window by dragging its frame, or to the user moving the window by dragging its caption bar. Fortunately, these kinds of standard system-level actions are best delegated to the system to process; we don't need to worry about them.

The only precaution we must take is to ensure that the system gets a chance to process any messages we aren't interested in handling. This is done by calling *DefWindowProc*, declared in the Windows unit:

```
function DefWindowProc(hWnd: HWND; Msg: UINT;
  wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
```

That this declaration matches exactly the one prescribed for a window procedure is not a coincidence, of course: *DefWindowProc* is a window procedure. It is the default window procedure supplied by Windows for applications to automatically process any leftover messages an application may not be interested in processing. We put a call to it in the catch-all **else** clause of the **case** statement inside the *WndProc*.

That's all there is to it. The application is ready to go. Congratulations! You have successfully implemented a simple, raw WinAPI application.

Why on Earth Would You Do This?

So why would you ever want to build a pure API application? A good concrete example would be in a real-time data acquisition application, such as a medical application (EEG, ECG), trading application (tick data), etc., where every millisecond counts. Raw API programming makes a program more predictable within time-critical sections. Windows isn't a good real-time system, and it needs all the help it can get sometimes.

Another example is when dealing with restricted memory and/or CPU power. A raw API application outperforms a VCL counterpart, while having only a fraction of the memory and CPU requirements. Any application developed the old-style way is simply more nimble, and much more responsive in a way that a user can feel immediately.

Yet another example is when creating language-independent extensions, plug-ins, and the like. Raw API modules have a much better chance to be interoperable across different programming and operating environments (C, C++, Visual Basic, Delphi, and Smalltalk on the one hand, and Windows 3.1/95/98/2000/NT on the other).

Finally, the understanding of raw API programming helps even when using the VCL. Sometimes, due to bugs in the VCL, or simply because there is no other way, one has to resort to low-level API calls. Knowing how to do that makes it possible to program around the limitations of a framework. It also makes it possible to understand the framework better, so that one can extend it and tweak it to the task at hand.

— Andrew J. Wozniewicz

Conclusion

Typically, a more ambitious Windows application would take care of processing many more kinds of messages. My goal here was to show the bare minimum application that you can implement without the help of the VCL. For good or for ill, the art of creating raw Windows API applications is now almost forgotten. With the advent of application frameworks, regardless of the programming language of choice, one rarely engages in the pursuit of raw API calls nowadays. Fortunately, the basics of an API-style GUI program are relatively simple, as you hopefully were able to see. The explanations took us a while, but the essence could be captured in a 50-line program listing.

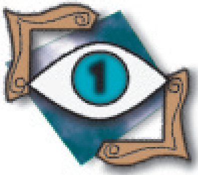
There are many other details about building non-VCL Delphi applications that extend beyond the scope of this article. If you are interested in more information on this topic, please refer to http://www.delphi-resource.com.

Even if you don't go to the extreme of building a pure API application, understanding the technologies underlying the VCL is well worth the effort, and pays off in reducing the time to find and work around bugs, and in allowing you to program things that would otherwise be difficult to program using the framework. Raw API programming is just one more tool for your programming toolkit. Use it wisely, but don't hesitate to use it when needed. ∆

The sample application referenced in this article is available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\SEP\DI200009AW.

Andrew J. Wozniewicz is the president of Optimax Corp. (http://www.optimax.com), a company specializing in Delphi consulting and training. He is the author of the original *Teach Yourself Delphi in 21 Days* (SAMS, 1995) and numerous technical articles. He can be reached at andrew@optimax.com.

table of CONTENTS



By Bill Todd

InterBase 6

More Than Open Source

The feature of InterBase 6 that has received the most attention is its change to being an open source product. However, InterBase 6 also provides a host of important new features for database developers. This article provides a brief description of many of the new features of InterBase 6, including: large exact numerics, new administrative tools, replication, a more powerful ALTER TABLE command, and much more.

Large Exact Numerics

Perhaps the most important new feature for business application developers is called *large exact numerics*. In prior versions, InterBase implemented the Numeric and Decimal data types in an unusual way. Numeric and Decimal are fixed-decimal data types. When you declare them, you must specify two numbers called precision and scale. Precision is the total number of digits the column can contain; scale is the number of digits to the right of the decimal point. Precision is limited to 18 digits. For example, a column declared as NUMERIC(15,2) would hold a total of 15 digits, and two of those 15 digits would be to the right of the decimal point. The difference between the Numeric type and the Decimal type is that the Numeric type always stores precision digits and the Decimal type stores precision digits or more.

The SQL standard defines the Numeric and Decimal data types primarily to provide precise storage for money amounts, but they are equally useful for any amount that requires a fractional part to be stored accurately. This contrasts with floating point data types, which store fractional amounts approximately. This imprecision can lead to errors that are unacceptable in financial applications. For example, if you store monetary amounts in floating-point format, it's possible to add a large number of values and have the result be off by a penny. In prior versions of InterBase, Numeric and Decimal columns whose precision was nine or less were stored as integers; however, if the precision was greater than nine they were stored in floating-point format, with the resulting lack of precision in the fractional part.

InterBase 6 stores all Numeric and Decimal values as scaled integers, so the numbers are always stored precisely. Columns with a precision up to four are stored as 16-bit integers. Those with a precision of five through nine are stored as 32-bit integers and columns whose precision is greater than nine are stored as 64-bit integers. Another important use of the Numeric and Decimal data types in InterBase 6 is for primary keys, because generators now return a 64-bit integer value instead of the 32-bit integer returned in prior versions. Since there is no 64-bit integer data type you must use Numeric or Decimal to store integer values beyond the range of a 32-bit integer.

Dialects

InterBase 6 introduces the concept of dialects as a way to implement new features required by the SQL 92 standard that conflict with features in previous versions of InterBase. InterBase 6 implements three dialects. Dialect one is compatible with older versions of InterBase; dialect two is a diagnostic mode; and dialect three introduces support for quoted identifiers, large exact numerics, and the SQL Date, Time, and TimeStamp data types.

In dialect one a string constant in a SQL statement can be delimited by single or double quotes. In dialect three, string constants must be enclosed in single quotes. Double quotes are reserved for the names of database objects which are reserved words, contain spaces, contain non-ASCII characters, or are case-sensitive. Although quoted identifiers can include spaces followed by other characters, they cannot include trailing spaces.

Until now, the Date data type, which contains both date and time information, was the only way to store date and time information in InterBase. Dialect three introduces the SQL 92 Date, Time, and TimeStamp types. Date fields store date information only, Time fields store time information only, and TimeStamp is equivalent to the dialect one Date type and stores both the date and time. If you back up a version 5 or older database and restore it in version 6, all Date columns and domains will be converted to TimeStamp. InterBase 6 also introduces the CURRENT_DATE,

CURRENT_TIME, and CURRENT_TIMESTAMP functions to retrieve the current date, time, or date and time, respectively. Also new is the EXTRACT function for extracting information from the data types that contain date and/or time information. For example:

```
EXTRACT(YEAR FROM SomeDate)
```

would return the year portion of the date from a Date or TimeStamp value.

InterBase Express

InterBase Express for InterBase 6 adds a new InterBase Admin tab to the Component palette. The Admin tab contains 11 new components that provide an interface to the new InterBase Service, Install, and Licensing APIs. With these components, you can install InterBase or the InterBase client, and administer every aspect of your InterBase server and database.

The IBBackupService and IBRestoreService components allow you to back up and restore InterBase databases, on a client machine or on the database server. Using the IBConfigService component, you can start and stop the InterBase service and determine if the service is running. You can also shut down a database, or bring a database on line. IBConfigService also provides methods to set all of the parameters of a database, including read-only, the SQL dialect, the number of page buffers, the sweep interval, the asynchronous write mode, and whether space for record versions is automatically reserved in the database pages.

Use the IBValidationService component to validate your database and recover any transactions that are in limbo. The IBStatisticalService reports various statistics about your database. The IBLogService is a handy diagnostic tool that retrieves the contents of the InterBase log file from the server. If you need to add or delete users, or make changes to the security settings for a user, use the IBSecurityService component. The IBLicensingService component lets you add license certificates, and the IBServerProperties component reports server licensing and configuration information. Finally, the IBInstall and IBUninstall components let you easily install or uninstall InterBase from your application.

IBConsole

The most noticeable change in InterBase 6 is that Server Manager and WISQL have been replaced by IBConsole. IBConsole, shown in Figure 1, is a Windows application you can use to administer InterBase servers and databases on any platform. The IBConsole user interface is divided into a tree pane, on the left, and a work pane, on the right. The tree pane provides a view of your InterBase servers, databases, and database objects. Figure 1 shows two servers, each with one database, with all nodes in the tree expanded. The tree pane lets you select and work with servers, databases, backups, license certificates, the server log, and users. For each database you have access to domains, tables, views, stored procedures, external functions (also called user-defined functions), generators, exceptions, BLOB filters, and roles.

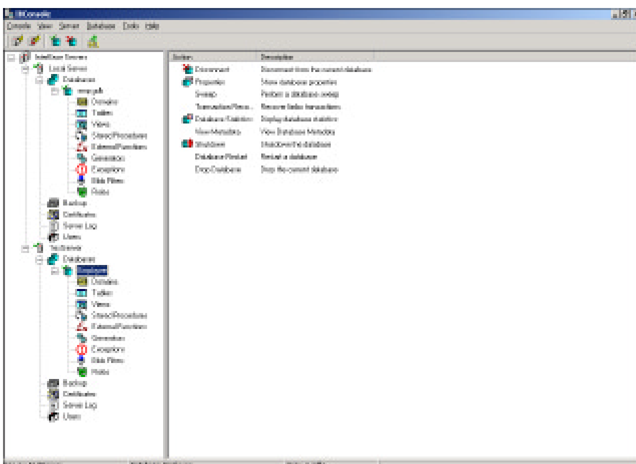


Figure 1: The tree pane and work pane of the IBConsole user interface.

You can work with servers and databases in two ways. The easiest way is to right-click the server or database, and make a choice from the context menu. The alternative is to select a server or database, and use the main menu. Before you can work with a server, you must right-click on **InterBase Servers**, the root entry in the tree pane, and choose **Register** to register the server. Figure 2 shows the Register Server and Connect dialog box. Simply enter the server's name, choose a network protocol, enter an alias for the server (which will appear in the tree view) and enter a user name and password. Click **OK** and the server will appear in the tree pane. Right-click on the server to register an existing database or create a new one.

What appears in the work pane depends on what you select in the tree pane. Figure 3 shows the work pane with a database selected. The items in the work pane offer most of the same options you will see on the context menu if you right-click the database in the tree pane. To perform any task in the work pane, simply double-click it.

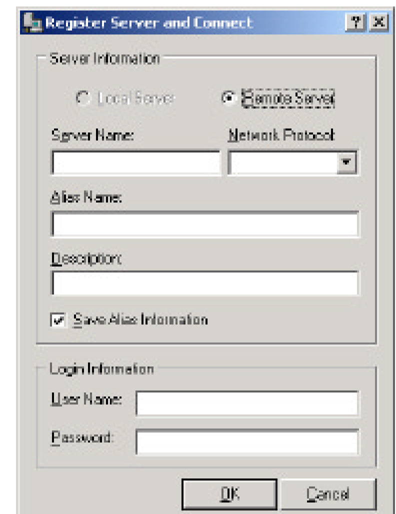


Figure 2: The Register Server and Connect dialog box.

When you expand a database you can double-click on **Domains**, **Tables**, **Views**, **Stored Procedures**, **External Functions**, **Generators**, **Exceptions**, **Blob Filters**, or **Roles**. Figure 4 shows the result of double-clicking the Customer table. The Properties tab lets you see the structure of the table, and the Metadata tab shows the CREATE TABLE statement for the table. Permissions

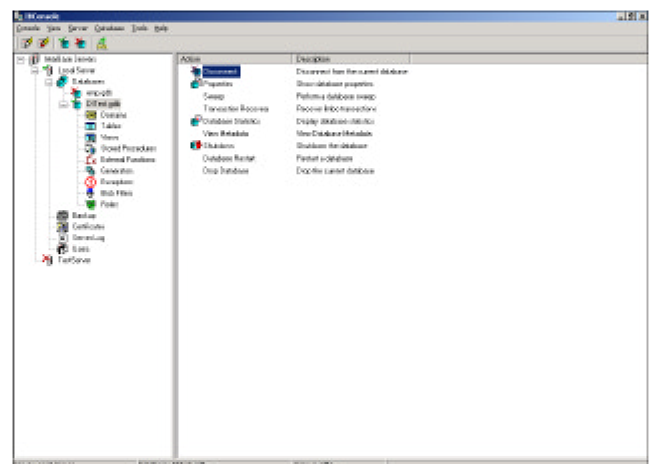


Figure 3: The work pane with a database selected.

lists all users with access to the table and their rights. Using the Data tab you can browse the table's data, and Dependencies lists any objects in the database that depend on the table. Double-clicking on other database objects displays similar information.

Unfortunately, most of the information displayed in IBConsole is read-only. To make changes you must still write your own SQL. Click the SQL toolbar button, or choose **Tools | Interactive SQL** from the menu to open the SQL Editor shown in Figure 5. The top pane shows your SQL script. You can enter your SQL from the keyboard, or load an existing SQL script and execute it. Any SQL you enter can also be saved as a script file. The bottom pane shows the result of executing the script, or, if the script includes a SELECT statement, the bottom pane displays the returned data.

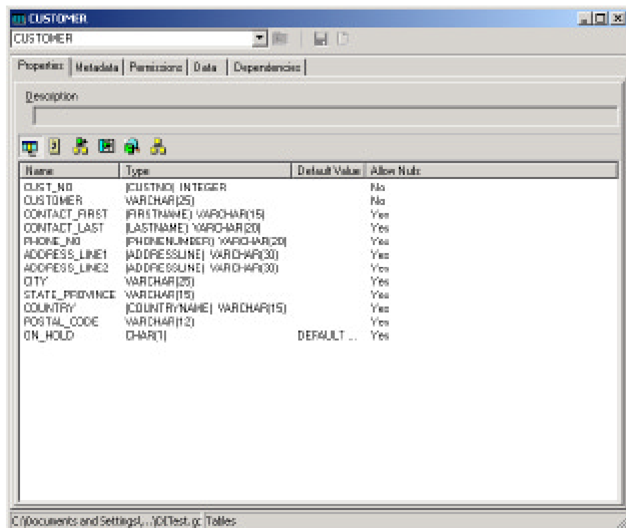


Figure 4: The properties of a table.

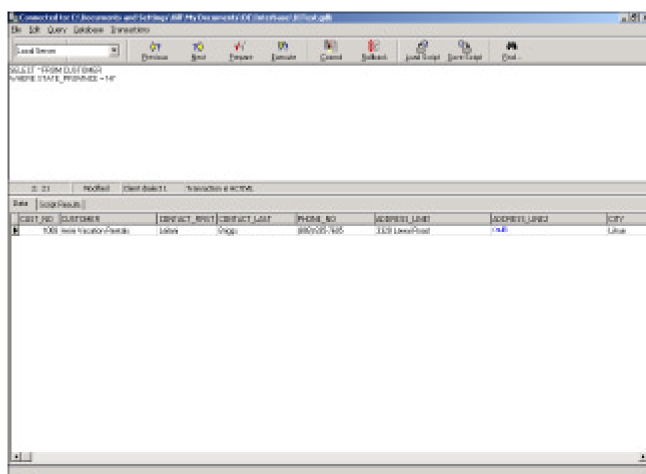


Figure 5: The SQL editor.

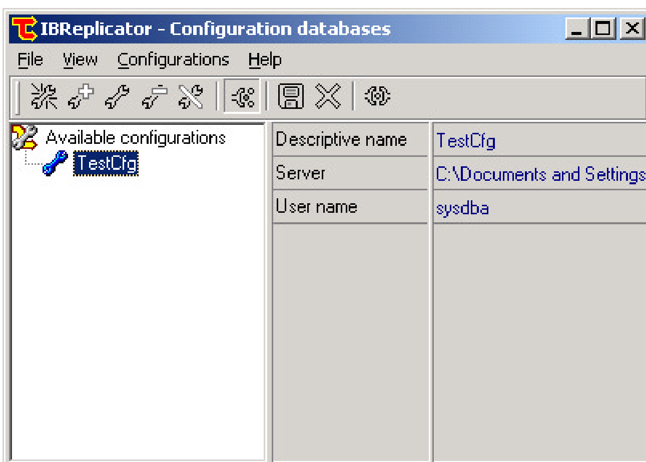


Figure 6: The Replication Manager's Configuration databases form.

Replication

With InterBase 6 you can replicate a source database to any number of target databases. The targets can have different table structures and different column names. You can replicate different tables, rows, and columns to each target database.

Replication can be initiated by a request from a client application, at timed intervals configured in the replication scheduler, or by an InterBase event. You can also use synchronous replication to propagate each change to the target database(s) as soon as it occurs.

Replication would be simple if the source database was the only one being changed. However, if both the source and target databases are being changed, some mechanism must be provided to resolve conflicts. Conflict resolution can be configured to any of three modes. Using priority-based resolution, the database with the highest priority takes precedence. Suppose the source database has the highest priority. In this case, an update to a record that doesn't exist in the target is automatically converted to an insert. An insert where the target already contains a record with the same primary key is converted to an update. A delete where the record doesn't exist in the target is ignored. Using time-stamped resolution, the change with the latest timestamp has precedence. Using master/slave resolution, the source database takes precedence.

For each source/target database pair you must create a configuration database using the Replication Manager, as shown in Figure 6. While this is a bit more work if the replication and structure is identical for all of the targets, it provides virtually infinite flexibility if they are not. The next step is to add the source and target databases using the Databases tab of the Replication Manager's main form (see Figure 7).

Use the Replication Manager's Replications page, shown in Figure 8, to create the replication schemata. The schemata identifies the source

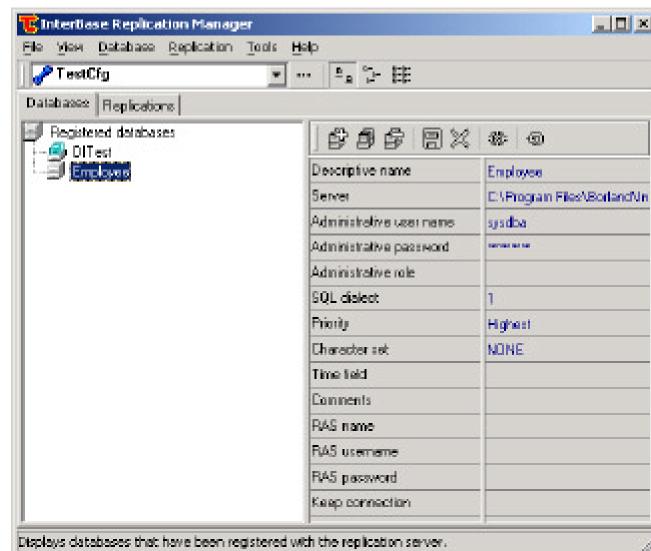


Figure 7: Adding databases in the Replication Manager.

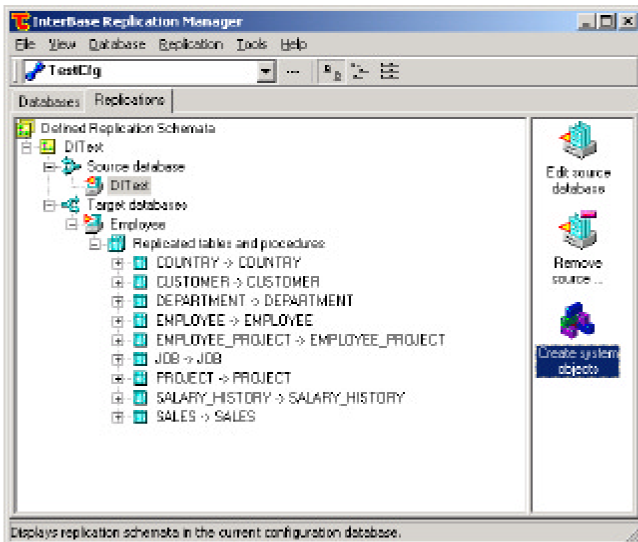


Figure 8: The Replications page of the Replication Manager.

and target databases and the tables, rows, and columns that will be replicated. The final step is to double-click the **Create system objects** icon in the Replication Manager's control panel at the right side of the form. This step automatically creates the log table and stored procedures required for replication in the source database. Note that no changes are made to the target database.

Read-only Databases

InterBase 6 databases have two modes: read-write and read-only. All databases are created in read-write mode, but can be changed to read-only using gbak, gfix, or IBConsole. Once a database has been changed to read-only mode you can copy it to a CD-ROM, or any other read-only media, and access the data. The only restrictions are that you cannot change the data or metadata, and you can only access generators to get their current value. For example:

```
SELECT GEN_ID(EMP_NO_GEN, 0) FROM EMPLOYEE
```

will work, and will return the current value of the `EMP_NO_GEN` generator. However:

```
SELECT GEN_ID(EMP_NO_GEN, 1) FROM EMPLOYEE
```

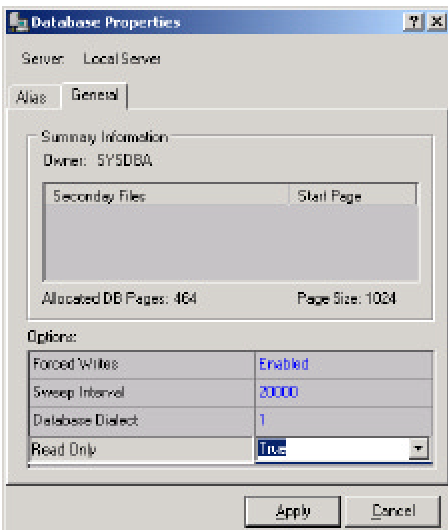


Figure 9: The Database Properties dialog box.

will fail because the generator cannot be incremented. To make a database read-only using IBConsole, right-click the database and choose **Properties** from the context menu to display the Database Properties dialog box shown in Figure 9. Click **Read Only**, then click the drop-down arrow and choose **True**. Finally, click the **Apply** button to change the mode to read-only. To change the mode with gbak, back up the database, then restore it in read-only mode using the following command:

```
gbak -create -mode read-only employee.gbk employee.gdb
```

To change the mode using gfix use the following command:

```
gfix -mode read_only employee.gdb
```

To change the mode to read-only or read-write, you must be the owner of the database or sysdba, and you must have exclusive use of the database.

ALTER TABLE and ALTER DOMAIN

The SQL ALTER TABLE statement now allows you to change the name, data type, or position of an existing column in a table. For example:

```
ALTER TABLE CUSTOMER
ALTER POSTAL_CODE TO ZIP_CODE
```

will change the name of the POSTAL_CODE field to ZIP_CODE, and:

```
ALTER TABLE CUSTOMER
ALTER POSTAL_CODE TYPE VARCHAR(14)
```

will change the data type to VARCHAR(14). In InterBase 6, ALTER DOMAIN also lets you change the type of a domain.

GBAK

The gbak utility now includes the functionality provided by gsplit in InterBase 5, allowing you to back up to multiple files in a single step. The new -service switch lets you run a backup on the server, without copying the data across the network to the client. This can mean substantially faster backups with less network traffic load.

Conclusion

InterBase 6 is a major upgrade. With the addition of large exact numerics, replication, and the InterBase Express Admin tools, you now have a full-featured SQL database server — available at no cost — that lets you create BDE-free database applications that support any requirement from a single user, with a modest amount of data to hundreds of users with many gigabytes of data distributed across multiple databases. It's even a great database for over-the-counter software that requires a large database on a CD-ROM.

InterBase 6 will be available for download in late June 2000 (certainly by the time you read this) from http://www.InterBase.com. Packaged CD editions will follow shortly thereafter. Editions with printed documentation will cost more. Δ

Bill Todd is president of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is co-author of four database programming books, including *Delphi: A Developer's Guide*. He is a Contributing Editor to *Delphi Informant Magazine*, and a frequent speaker at Borland Developer Conferences in the US and Europe. Bill is also a member of Team Borland and a nationally known trainer; he has taught Delphi programming classes across the country and overseas. He can be reached at bill@dbginc.com.

NEW & USED

By Tim Sullivan

ReportBuilder 5.0 Enterprise

Reporting on a Higher Level

I began using ReportBuilder a couple of years ago with version 3.5. Since then, Digital Metaphors has listened carefully to its users, and the product has matured significantly as a result. With the latest version, 5.0 Enterprise, they've managed to create something that blows its competitors away.

ReportBuilder is now offered in three editions. At the low end, there is the Standard version, which offers an excellent design-time-only reporting environment. It includes support for non-BDE tables and — using their Just-In-Time components — support for reports that don't have databases behind them. At the next level up, the Professional edition allows you to let your end users design and save queries and reports. Finally, the Enterprise edition adds Digital Metaphor's secret weapon: RAP.

ReportBuilder offers all the features you would expect from a good report writer: band-based design, support for various types of data (memos, rich text, .gif/.jpeg/.bmp images), grouping, etc. However, it's at the higher levels that it really shines, with things like AutoSearch, AutoJoin, and RAP. In addition, the architecture is remarkably well designed, and makes extending or enhancing the default forms easy, without changing the source code.

ReportBuilder also includes a lot of extras that make it stand out from other packages. These include: a Report Wizard that helps you quickly build reports; a Label Wizard that supports a majority of Avery label formats; and a Report Explorer that provides a convenient way for end users to store reports. It looks and feels like Windows Explorer, so users understand it quickly.

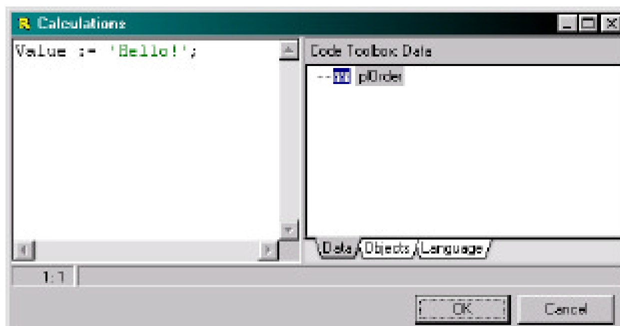


Figure 1: Using the ReportBuilder 5.0 variable component.

What the Heck Is RAP?

RAP stands for Report Application Pascal, and is probably the most significant enhancement to ReportBuilder since 4.0. While there have been numerous beta versions available for use with the 4.x series, 5.0 is its first "official" release. It allows end users to write event handlers for their reports in a way that is similar to how developers write event handlers for Delphi. This gives ReportBuilder the ability to handle extremely powerful and complex calculations, colorings, or just about anything else your end users might want.

At a simple level, you can drop a variable component onto the report, right-click, and choose Calculations. Simply assign a result to Value and go (see Figure 1).

In addition to the ability to create variables, you also have the power to hook the same events as in Delphi at design time. The event handler you write in here can be as complex as required. You can change labels to be bold (or red) if they are above or below some level. In short, if you can think of it, you can do it. This brings ReportBuilder to a level that surpasses even Crystal Reports in terms of sheer power.

One of the things I like about RAP is how easy it makes doing things such as building phone numbers and concatenating names. For example, I separate the area code from the phone number in my data structures. Joining these can be difficult with a query, but it's a breeze with RAP. This code

properly formats the fax number, whether or not the end user has specified an area code:

```
if (plClients['Fax Area'] <> '') then
  Value := '(' + plClients['Fax Area'] + ') ' +
           plClients['Fax']
else
  Value :=  plClients['Fax'];
```

AutoSearch

Another of the advanced features of ReportBuilder is AutoSearch, which is Digital Metaphors' term for run-time parameters. In all the report writers I've used since starting with Delphi, I've longed for one of the features that Access has: the ability to ask the user for input. ReportBuilder does Access one better by asking for all parameters at once.

For example, let's take a report that displays a list of job applicants. Certain end users might want to see only the applicants entered between certain dates. Or, alternately, they might want to find a certain last name. Figure 2 shows the AutoSearch items.

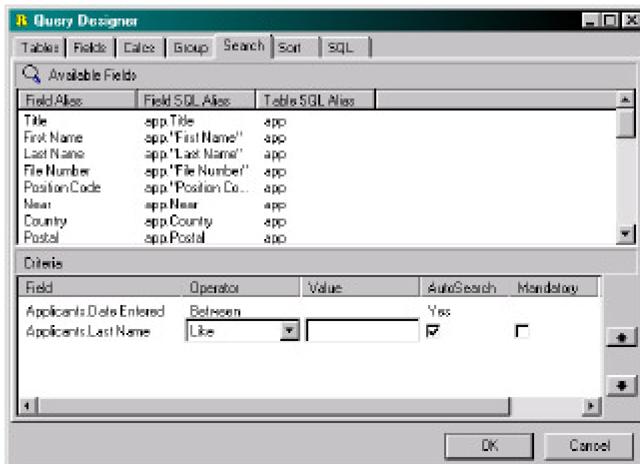


Figure 2: ReportBuilder's AutoSearch asks for all parameters at once.

The AutoSearch dialog box pops up when the report is run. It has several panels, one for each item in the query designer (see Figure 3). One of the coolest things about this is that you can easily customize the panels. For instance, you may notice that the dialog box doesn't have date edits on the Date Entered. For my application, I want a formatted date input for my users. I use the Orpheus date edit controls everywhere else in my application, so I want to register a new class of panel that will be used instead.

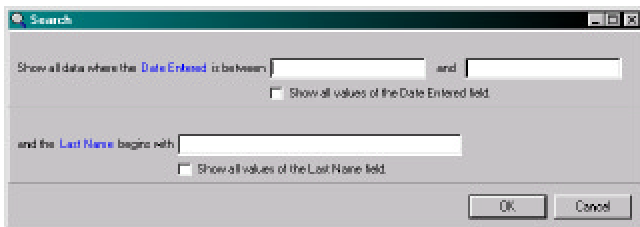


Figure 3: The AutoSearch dialog box contains several panels, one for each item in the query designer.

To create this custom panel, we need to define a new class for the AutoSearch dialog box and override the *GetPanelClassForField* method. This method is what the AutoSearch dialog box uses to determine what panel to display for the specified AutoSearch item, depending on whether you're searching between two dates, or for dates greater than a certain value. This method has a parameter, *APanelClass*, that returns the required panel.

In the previous case, where I want to have my Orpheus date edits, I define my AutoSearch dialog box as shown in Figure 4.

```
type
  { TuilAutoSearchDialog }
  TuilAutoSearchDialog = class(TppAutoSearchDialog)
  protected
    procedure GetPanelClassForField(
      AField: TppAutoSearchField;
      var APanelClass: TppAutoSearchPanelClass); override;
  end;

...

procedure TuilAutoSearchDialog.GetPanelClassForField(
  AField: TppAutoSearchField;
  var APanelClass: TppAutoSearchPanelClass);
begin
  // In this case, we are going to deal with only date
  // fields between two values. This can easily be expanded
  // to handle as many different possibilities as you want.
  if (aField.DataType in [dtDate, dtTime, dtDateTime]) and
     (aField.SearchOperator = soBetween) then
    APanelClass := TuilBetweenDatesSearchPanel;
end;
```

Figure 4: We need to define a new class for the AutoSearch dialog box, and override the *GetPanelClassForField* method.

Next, we need to define the panel we're going to return. We start with one of the base AutoSearch panel types, *TppBetweenSearchPanel*. We then create our class, as shown in Figure 5.

The most important method we need to override is *Init*. This allows us to take the default panel, hide the controls we don't need, and add the ones we do. In this case, we're hiding the standard *TEdits*, and replacing them with our *TOvcEdits* (see Figure 6).

The final step is to register the new dialog box class with ReportBuilder. This is done in the **initialization** section of the unit, by calling:

```
ppRegisterForm(TppCustomAutoSearchDialog,
  TuilAutoSearchDialog);
```

Digital Metaphors has applied this design philosophy to just about every aspect of ReportBuilder, so it's extraordinarily easy

```
type
  TuilBetweenDatesSearchPanel =
    class(TppBetweenSearchPanel)
  private
    FOvcDateEdit1 : TOvcDateEdit;
    FOvcDateEdit2 : TOvcDateEdit;
    // Required for Orpheus controls.
    FController   : TOvcController;
  protected
    procedure ShowAllValuesClickEvent(
      Sender: TObject); override;
  public
    constructor Create(aOwner: TComponent); override;
    destructor Destroy; override;
    procedure Init; override;
    function  Valid: Boolean; override;
  end;
```

Figure 5: Declaring the *TuilBetweenDatesSearchPanel* class.

to replace just about any screen of the report writer, from the AutoSearch dialog box, to the Print Preview window, without altering the existing source code.

```
procedure TuilBetweenDatesSearchPanel.Init;
var
  lValues: TStrings;
begin
  inherited Init;
  // Hide the default edit boxes.
  EditControl.Visible := False;
  EditControl2.Visible := False;

  FController := TOvcController.Create(nil);

  // Create the first date edit.
  FOvcDateEdit1 := TOvcDateEdit.Create(Self);
  FOvcDateEdit1.Controller := FController;
  FOvcDateEdit1.Parent := Self;
  FOvcDateEdit1.Left := EditControl.Left;
  FOvcDateEdit1.Top := EditControl.Top;
  FOvcDateEdit1.Width := 150;
  // Move the AND label so it sits after the DateEdit.
  AndLabel.Left :=
    FOvcDateEdit1.Left + FOvcDateEdit1.Width + 5;
  // Create the second date edit.
  FOvcDateEdit2 := TOvcDateEdit.Create(Self);
  FOvcDateEdit2.Controller := FController;
  FOvcDateEdit2.Parent := Self;
  FOvcDateEdit2.Left := AndLabel.Left + AndLabel.Width + 5;
  FOvcDateEdit2.Top := EditControl.Top;
  FOvcDateEdit2.Width := 150;
  // Get current values (if there are any).
  lValues := TStringList.Create;
  try
    ppParseString(Field.SearchExpression, lValues);
    if (lValues.Count > 0) then
      FOvcDateEdit1.Date := ppStrToDateTime(lValues[0]);
    if (lValues.Count > 1) then
      FOvcDateEdit2.Date := ppStrToDateTime(lValues[1]);
  finally
    lValues.Free;
  end;
end;
```

Figure 6: Overriding the *Init* method.

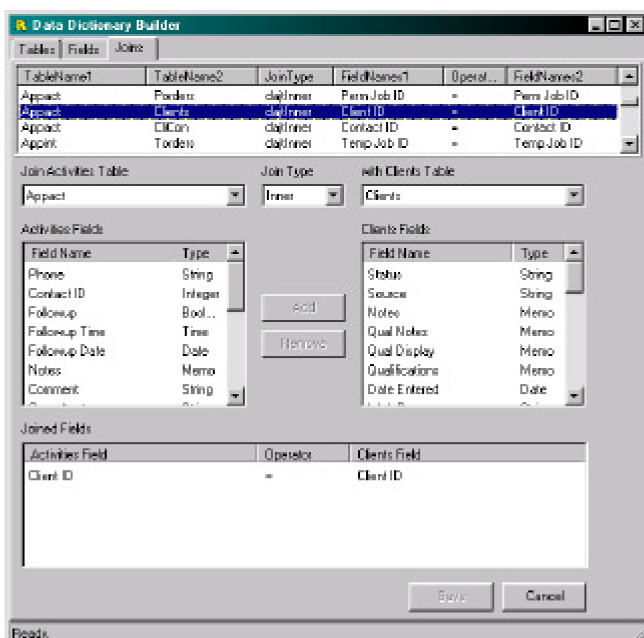


Figure 7: ReportBuilder's Data Dictionary Builder.

It's Like Webster's for Your Data

One of the things my end users hate more than anything else in the world is having to learn the relationships between the tables in their application. They inevitably call me to find out if this field links to that field, or what the AppQList table stores. ReportBuilder has a solution: the Data Dictionary, shown in Figure 7. With this, you provide detailed descriptions of the tables, fields, and relationships to ReportBuilder, and it does all the linking automatically for your users.

I can almost hear you whimpering about what a nightmare setting this up must be. Well, don't! It's easy, easy, easy. Double-click on the DataDictionary component, and it brings up the dictionary editor. You can then generate the list of tables, fields, and links. It guesses what fields link up based on name; you simply go through and delete the tables and links that don't make sense. In my application, with about 90 tables, it took me less than an hour to develop the data dictionary.

The dictionary also allows you to specify what fields are selectable or searchable, giving you a great amount of control over what your end users are allowed to do. End users like it because they don't need to know the relationships between the tables in order to build powerful queries, and I like it because the users don't need to come to me any more for that information.

End-user Assistance

One of the most difficult things for many users is getting their minds around reporting. Most of my end users barely understand the concept of a record, let alone writing reports. Digital Metaphors has come up with a great solution: Learn ReportBuilder. This includes a stand-alone version of the report writer, some sample data, and a 130-page tutorial-style guide. It's a great tool for end users to learn about report writing, and lets them play around with data that is external to your application, learning general report-writing techniques instead of ones specific to your application. This provides them with a much more solid idea of what is involved with writing reports, before attempting anything specific to their industry.

Informant Fact File

ReportBuilder is the most powerful Delphi report writer available on the market, hands down. Though there are other capable systems available, ReportBuilder offers not just a fantastic designer, but unprecedented control over altering and enhancing its look and feel. Add to that an entire programming language, and you have something other reporting tools simply can't touch.

Digital Metaphors Corp.
16775 Addison Road, Suite 613
Addison, TX 75001

Phone: (972) 931-1941
Web Site: http://www.digital-metaphors.com
Price: Standard, US$249; Professional, US$495; Enterprise, US$749.

Conclusion

ReportBuilder is the most powerful Delphi report writer available on the market, hands down. While there are other capable systems available, ReportBuilder offers not just a fantastic designer, but unprecedented control over altering and enhancing its look and feel. Add to that an entire programming language, and you have something other reporting tools simply can't touch.

Although the price for the Enterprise version may seem a little steep, it is well worth it for a royalty-free, end-user report writer that's fully integrated into your software. I heartily recommend it. ∆

Tim Sullivan is the president of Unlimited Intelligence Limited and author of both the UIL Security System and the freeware UIL Plugin System. He can be reached at tim@uil.net.

The Delphi Hall of Fame

If you watch much football, you're probably familiar with the All-Madden Team: football players of the past and present deemed worthy by former coach and current announcer John Madden to a roster spot on his cross-time "virtual team."

Each sport has its Hall of Fame, wherein standout players are immortalized. Why not have one for Delphi, even though it doesn't have a physical presence? Delphi isn't as old as football, but its wooden anniversary is indeed upon us. Yes, it has been five years since Delphi was released. It is traditional to bear gifts of wood on the fifth anniversary. In honor of this, it might be fitting to recognize those from the Delphi community on paper (after all, a wood-based product) for their contributions and accomplishments. Like John Madden, I have the audacity to name an "All Guru" team. Note that in some cases, organizations, or even virtual organizations, are included.

So sit back, relax, and imagine a long, drawn-out drum roll — followed by a dramatic pause — as the names of the winners are handed to the presenter of the awards. An inscrutable grin appears on the face of the presenter as he sees who has been included in the first-ever induction into the Delphi Hall of Fame. The envelope, please:

- The Delphi development team, for giving us the world's greatest development tool. Especially noteworthy are: Anders Hejlsberg, the original chief architect of Delphi; Danny Thorpe, author of *Delphi Component Design*; and Chuck Jazdzewski, current chief architect of Delphi.
- Team B, for their help and moderating influence on the newsgroups.
- *Delphi Informant Magazine* (http://www.DelphiZine.com).
- *The Delphi Magazine* (http://www.itecuk.com).
- Wordware Publishing, for publishing a plethora of Delphi books (http://www.wordware.com).
- Robert Vivrette, for his electronic newsletter *Unofficial Newsletter of Delphi Users* (http://www.undu.com).
- Robert Czerwinksi, for the Delphi Super Page (http://delphi.icm.edu.pl).
- TurboPower Software, for their Delphi programming tools, such as SysTools, Abbrevia, AsyncPro, etc. (http://www.turbopower.com).
- Jeff Duntemann, for his book *Delphi Programming Explorer* (co-written with Jim Mischel and Don Taylor) and his (unfortunately now defunct) magazine *Visual Developer*.
- Charlie Calvert, for his *Delphi X Unleashed* books and his "Tech Corner" on the Borland Web site.
- Ray Lischner, author of *Secrets of Delphi 2*, *Hidden Paths of Delphi 3*, and *Delphi in a Nutshell*.
- Ray Konopka, author of *Delphi 3 Component Design*, creator of Raize components and CodeSite, and "Delphi by Design" columnist for *Visual Developer*.
- Mark Miller, for his work on the Delphi programming productivity tools CDK (Component Development Kit), reAct (component tester and debugger), and CodeRush (editor).
- Marco Cantù, for his *Mastering Delphi* books and *Delphi Developer's Handbook* (co-written with Tim Gooch and John Lam).
- Dr Bob Swart, for his Web site (http://www.drbob42.com), free tools (such as IntraBob), and for naming everything but his children after himself.
- Steve Teixeira and Xavier Pacheco, for their *Delphi Developer's Guide* books.
- Neal Rubenking, author of *Delphi Programming Problem Solver* and *Delphi for Dummies* and columnist for *PC Magazine*.
- John Ayres, for his book *The Tomes of Delphi: Win32 Core API* (co-written with a host of others).
- Alan C. Moore, Ph.D. for heading up the JEDI (Joint Endeavor of Delphi Innovators) Project (http://www.delphi-jedi.org).
- David Intersimone, AKA David I, "the Last Barbarian," "no, I'm not related to Jerry Garcia," etc. for his perseverance, substance, and style.
- Steve McConnell, for his books on programming and project management (*Code Complete*, *Rapid Development*, *Software Project Survival Guide*, and *After the Gold Rush*).

— *Clay Shannon*

Clay Shannon is a Delphi developer for eMake, Inc. in Post Falls, ID. Having visited 49 states (all but Hawaii) and lived in seven, he and his family have finally settled in northern Idaho, near beautiful Lake Coeur d'Alene. The only spuds he has seen in Idaho have been in the grocery, and most of those are from Oregon and Washington. Clay has been working (almost) exclusively with Delphi since the release of version 1, and is the author of *Developer's Guide to Delphi Troubleshooting* [Wordware, 1999]. You can reach him at BClayShannon@aol.com.

Learning Windows 2000

With the three columns I wrote about Delphi and Linux last Spring, I'll bet some readers are wondering if I'll abandon Windows programming altogether. Not quite yet. I recently purchased two operating systems: Red Hat Linux and Windows 2000. To prepare to install and work with the latter, I decided to examine a number of new books. In this column, I'd like to share some of my observations about them.

Regrettably, most of these books are aimed at system administrators, not programmers. Still, as developers, it's helpful for us to know as much as possible about any new version of Windows for which we intend to create applications. These books fall into three general categories: Windows 2000 Professional, Windows 2000 Server, and Special Topics. I suspect the book in the last category will be of most interest to readers of this journal.

Windows 2000 professional books. With the exception of the final specialized title, all of these books cover basic topics, including the installation and configuration of the new operating system. However, if your main concern is simply installation and setup, I recommend *Microsoft Windows 2000 Professional Installation and Configuration Handbook* by Jim Boyce [QUE, 2000, ISBN: 0-7897-2133-3]. As with the other books I'll discuss, this one provides an overview of the new Windows 2000 features. However, it goes into greater detail on planning and executing your setup of Windows 2000. Incidentally, all of these books recommend a certain amount of planning before the actual installation. If you're nervous or uncertain about setting up security or the file system, working with user accounts and groups, or working with specific types of hardware, this may be the first book with which you want to become familiar.

If you would like to learn about Windows 2000 from a respected expert, take a look at *Peter Norton's Complete Guide to Microsoft Windows 2000 Professional* by Peter Norton, et al. [SAMS, 2000, ISBN: 0-672-31778-8]. This work also provides a good deal of information on installation, along with strong sections on memory management and the various file systems. It also broaches such important topics as working with new technologies like DCOM and ActiveX, working with related Microsoft applications (such as Outlook Express), and multimedia (a favorite of mine!). The tips and warnings we have come to expect from this software guru are scattered throughout the book, as well.

Similar in content to Peter Norton's treatise, Paul Cassel's *Microsoft Windows 2000 Professional Unleashed* [SAMS, 2000, ISBN: 0-672-31742-7] covers additional topics that will be of interest to some readers. Topics include a detailed exposition of the command-line interface, using Windows 2000 on laptops, and a particularly interesting chapter on using Windows 2000 and Linux together on the same system.

If you are not satisfied with a work under 1,000 pages and want a great deal of detailed information on the topics I've mentioned, consider *Special Edition Using Microsoft Windows 2000 Professional* by Robert Cowart and Brian Knittel [QUE, 2000, ISBN: 0-7897-2125-2]. This book is particularly strong in its discussion of built-in Windows 2000 applications. It even gets into a discussion on programming, albeit only in VBScript. Of more interest to us is the brief section on the Windows 2000 registry, which provides a nice overview.

Windows 2000 Server books. Just as Windows 2000 comes in several versions, some of the previously mentioned books are also available in *Server* editions. I'll discuss the two I've examined. Surprisingly, *Peter Norton's Complete Guide to Microsoft Windows 2000 Server* by Peter Norton, et al. [SAMS, 2000, ISBN: 0-672-31777-X] is thinner than the *Professional* edition. Although there is a certain amount of overlap between the two, there is not as much as you might expect. Both versions are aimed at fairly advanced users, but *Windows 2000 Server* provides more information on some of the exciting new features of Windows 2000, such as Active Directory. It's geared specifically at system administrators, with a good deal of information on successfully managing a Windows 2000 network. However, it also includes topics such as TCP/IP Networking, and Remote Access Server (RAS).

Microsoft Windows 2000 Server Unleashed by Todd Brown, Chris Miller, and other contributors [SAMS, 2000, ISBN: 0-672-31739-7], is also a bit shorter than the *Professional* version. Like the Norton Server edition, this volume includes sizable sections on Active Directory and RAS. However, rather than having a separate chapter on TCP/IP as Norton does, this essential protocol is discussed in various sections throughout the book. Also, there is a bit more information covering installation issues.

Special category: Active Directory. *Active Directory Programming* by Gil Kirkpatrick [SAMS, 2000, ISBN: 0-672-31587-4] is the book with which I spent the most time, and it's also the book that I believe will be of most value to those readers who plan to write applications specifically for Windows 2000. At the core of Windows 2000 is a new structure, Active Directory, which includes configuration information for the majority of Windows 2000 administrative services. Kirkpatrick points out that this pervasiveness will expand in the future to encompass even more aspects of Windows 2000, as well as other Microsoft products (such as BackOffice version 5).

After providing an excellent and extremely detailed introduction to Active Directory, Kirkpatrick devotes the remaining three-quarters of this impressive work to presenting two of the programming interfaces: the Active Directory Services Interface (ADSI), and the Lightweight Directory Access Protocol (LDAP) API. He first provides a nice comparison between the two, discussing the appropriateness of each to different environments, and then provides a plethora of programming solutions to many common Active Directory tasks.

For the reader who is wondering: When is he going to write about Delphi books? The answer is: next month! As the year 2000 ends, I plan to write about the recent Inprise/Borland Conference and Delphi 6, as well as present another interview with a top Delphi person and return to the Delphi Toolbox theme. Until next time ...

— *Alan C. Moore, Ph.D.*

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

table of CONTENTS