



Cover Art By: Darryl Dennis



## ON THE COVER

### 6 Greater Delphi

**CORBA** — Dennis P. Butler

After a brief introduction to the Common Object Request Broker Architecture, Mr Butler quickly moves on to building an assortment of CORBA servers and clients from the ground up, even constructing a client with JBuilder to emphasize CORBA's language independence.



## FEATURES

### 16 Undocumented

**RTTI Gets Easier** — Bill Todd

Although it remains undocumented, a new set of functions added to TYPINFO.PAS in Delphi 5 make Delphi's Run-time Type Information capabilities easier to use — as Mr Todd explains.



### 20 DBNavigator

**The Data Module Designer** — Cary Jensen, Ph.D.

Delphi 5 is full of new features; one of the more important for database developers is its Data Module Designer. Dr Jensen provides us with an introduction and a detailed description of the new tool.



### 25 Sound + Vision

**Extending TAPI** — Robert Keith Elias and Alan C. Moore, Ph.D.

Mr Elias and Dr Moore review TAPI concepts, overview the TAPI and Multimedia API, and explain the code needed to play .WAV files to, and record them from, a phone line — and much more.



### 31 Columns & Rows

**An AS/400 Skeleton Key** — G. Bradley MacDonald

Delphi is the best client/server development environment extant, but the AS/400 can prove exotic ground for even seasoned Delphi programmers. Mr MacDonald demystifies Big Blue's popular mini.



## REVIEWS

### 35 Raize Components 2.1

Product Review by Ron Loewy

## DEPARTMENTS

2 Delphi Tools

5 Newslite

38 File | New by Alan C. Moore, Ph.D.





## RSW Offers e-TEST Suite 3.1

RSW Software, Inc. announced *e-TEST Suite 3.1*. e-TEST Suite automates the testing of business-critical Internet and intranet applications throughout the application lifecycle.

The 3.1 release includes several enhancements and upgrades, including the e-Reporter module and optimization for the BroadVision One-To-One family of Internet applications.

e-TEST Suite was designed to address the unique needs of Web application developers and testers with an integrated, development-through-deployment solution for Web testing. It includes e-LOAD for load/scalability testing, e-TESTER for functional and regression testing, and e-MONITOR for monitoring the performance and continuous availability of live Web applications. All three are powered by RSW's Visual Script

technology, which drives all facets of testing with a common set of scripts and requires no programming.

e-TEST Suite 3.1 is optimized for popular Web development environments, including NetDynamics, Microsoft ASP, WebObjects, ColdFusion, and the BroadVision One-To-One

environment.

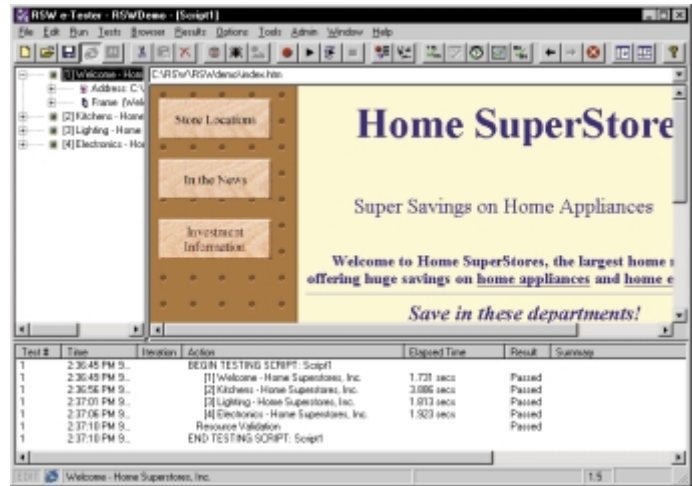
e-TEST Suite 3.1 also features Visual Script extensibility and enhanced support for JavaScript and VBScript.

**RSW Software, Inc.**

**Price:** From US\$4,995

**Phone:** (508) 435-8000

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



## Premia Announces CodeWright 6.0

Premia Corp. announced the release of *CodeWright 6.0*, a new version of the programmer's editing system. With the addition of tools optimized for code completion, code reuse, and code storage management, CodeWright 6.0 has been built into a full-featured editing system custom-ready to accelerate the programmer's role in the development cycle.

The new release incorporates CodeSense technology, a familiar "in-place" popup window for

completing C functions. One may invoke a list to complete a partially typed symbol, and when the mouse passes over a symbol found in the CodeSense database, its definition is displayed in a pop-up ToolTip. In addition to listing functions in the popup window, CodeSense uses visual go-to buttons that open the document containing the function prototype. Long symbol lookups run in their own thread, allowing the user to continue to work while lookup

is in progress. The user can specify entirely new libraries for CodeSense to adopt, and CodeSense never forces completion; when one wants completion, the user invokes CodeSense.

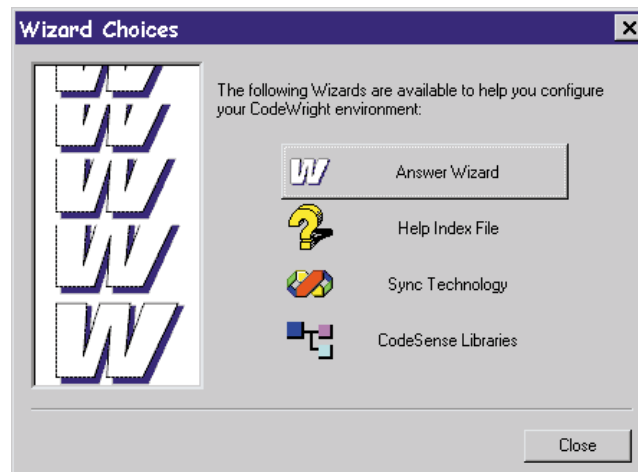
Another new feature is CodeFolio, which allows code to be stored in an Explorer-style directory within a dedicated window. When a named code segment is dropped from the CodeFolio window into a target document, CodeFolio allows macro expansion and prompted input at the insertion point. Entire processes (such as a large Perl script) can also be applied at the moment of code placement. Moreover, when the working folder resides on a network, CodeFolio gives teams a shared view of an intelligent, reusable code repository.

**Premia Corp.**

**Price:** US\$299 for a single-user license; upgrades and multi-user licenses are available.

**Phone:** (800) 547-9902

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





## Arsenal Launches Arsenal Word Processor Toolkit

Arsenal Inc. announced *Arsenal Word Processor (AWP) Toolkit*, a text processor control that enables developers to build into any application the possibility to view, generate, edit, and print documents in most popular for-

mat and to solve a number of development problems concerning document processing.

AWP Toolkit can be used to make an application independent from external document processing programs, establish the full

program control over the work with the text, create special edit tools with unusual functions (e.g. "save in compressed form" or "notify after close automatically," etc.), and generate complex reports and save them in any supported format.

A unique AWP Toolkit feature is the tag support — you can put any number of tags in the document and set your own properties and handlers.

You can choose between an ActiveX control or a plain COM interface in your application.

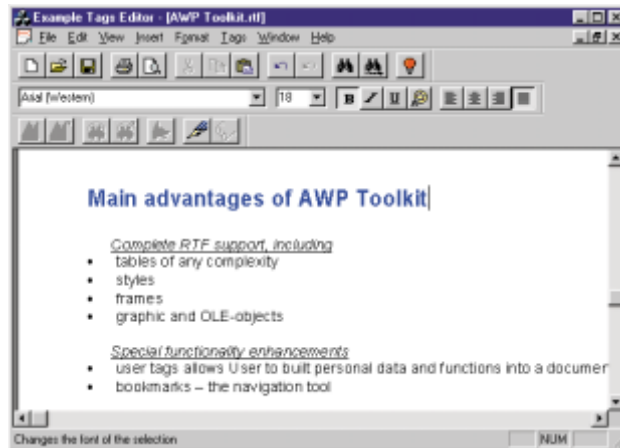
### Arsenal Inc.

**Price:** AWP Toolkit, US\$445;

AWP Toolkit/ActiveX, US\$599.

**E-Mail:** [sales@arssoft.com](mailto:sales@arssoft.com)

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



## DBI Technologies Announces Component Toolbox OCX 4.0

DBI Technologies Inc. announced *Component Toolbox OCX 4.0*, a collection of 54 dynamic, 32-bit ActiveX components for Delphi, Visual Basic, Visual C++, Visual FoxPro, Access, PowerBuilder, and HTML.

Additions to Toolbox include ctImage, a single compact component with functionality and sup-

port for HTML environments; ctList, which supports list item sub-text, picture clips, checkboxes, and automatic column sorting; ctListBar, an Outlook-style ListBar component that includes horizontal and vertical presentations, mouseover events, support for drag-and-drop, tool tips for each item, word-wrapping, and Image List; ctHTML, a compo-

nent for quick display of documents and application-specific HTML browsing; and ctTree, which includes support for multiple column sorting and in-line label editing.

### DBI Technologies Inc.

**Price:** US\$299 for a single developer license.

**Phone:** (204) 985-5770

**Web Site:** <http://www.dbi-tech.com>

## KRFTech Announces WinDriver

KRFTech Ltd. announced *WinDriver*, a device driver development toolkit designed to enable developers to create high-performance PCI/ISA/EISA-based device drivers.

WinDriver's architecture enables

developers to create a hardware access application without having to write a kernel-mode device driver. All hardware access is done in the application through the WinDriver interface, while maintaining kernel-mode performance.

WinDriver features a graphical Wizard for hardware diagnostics and automatic code generation. Hardware access applications created with WinDriver will run on Windows 95, 98, NT3.51, NT4.0, NT5.0, CE, Linux, and Alpha NT.

Developers can use any Win32 compiler, including Borland Delphi, MSDEV, Visual C/C++, and others.

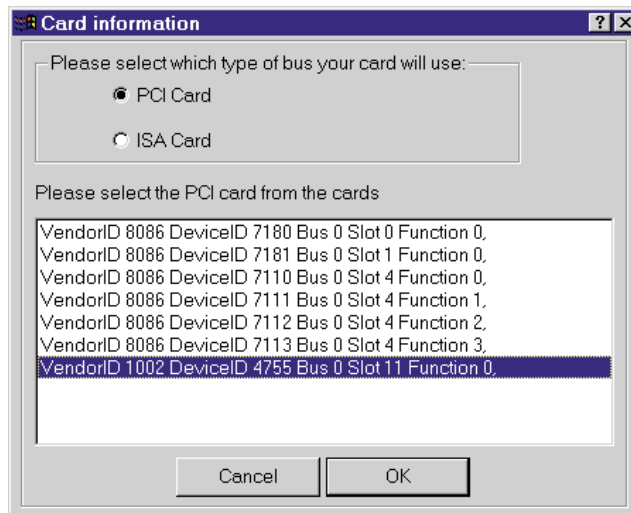
Other features include built-in support for leading PCI bridge vendors (PLX, AMCC, and V3); automatic implementation of I/O, interrupt handling, and access to memory-mapped cards; and support for DMA, Plug-and-Play, and multiple board handling.

### KRFTech Ltd.

**Price:** From US\$899 (Windows 95/98 or NT).

**Phone:** (877) 514-0537

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





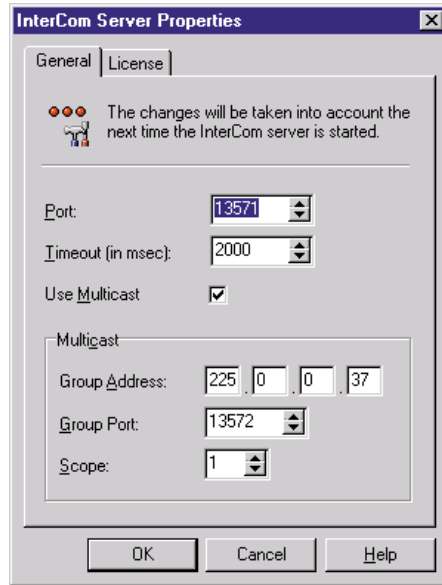
## CNS Announces Version 1.1 of The InterCom System

CNS International announced the release of *version 1.1 of The InterCom System*, the company's developer toolkit for building multi-user "aware" applications.

The InterCom System is intended for developers that create multi-user network applications, such as database and Internet applications, games, and groupware.

which is available as a service for Windows 95/98/NT.

The InterCom System consists of a client control, used by the application, and the InterCom server. The system is based on a publisher-subscriber notification architecture, where clients can subscribe to "events" that the developer defines and publish data to those events. When data is published to an event, all clients subscribed to that event are automatically notified. Clients can also send messages directly to other clients, which can be located anywhere on a LAN or WAN. The product can be used from a host of programming environments, including Delphi, Visual Basic, C++, HTML, and others.



Improvements and additions have been made in the 1.1 version to the client components to support a wider range of development tools. This includes updates to the ActiveX control, which is now fully scriptable from HTML environments, and the addition of a native VCL control for Delphi 4.

Improvements have also been made to the InterCom server,

**CNS International**  
**Price:** US\$399  
**Phone:** 31 30 2802822  
**Web Site:** <http://www.cns.nl>

## EC Releases Help & Manual 2.0

EC Software released *Help & Manual 2.0*, a Windows 95/98/NT4 application that lets you create online help files, HTML, and printed user manuals from a single source. Help & Manual can create help files for all versions of Windows, from 3.x through NT, as well as the new HTML Help files. The program also supports plain HTML, rich text, and printed formats.

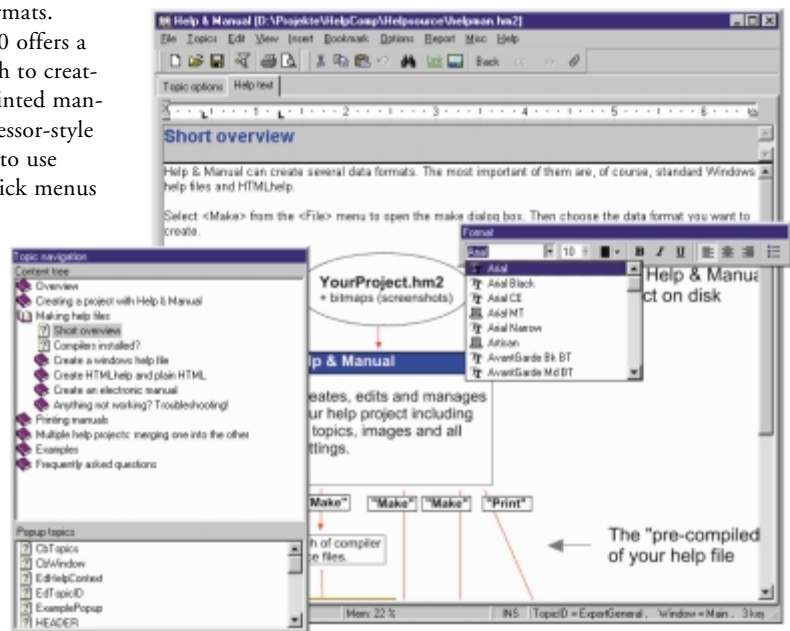
Help & Manual 2.0 offers a WYSIWYG approach to creating help files and printed manuals. The word-processor-style interface allows you to use toolbars and right-click menus to easily create topics and links, enter text, and add images. Hyperlinks are created with a simple drag and drop. The program automatically translates your left and right margins into HTML tables, and converts your Windows metaphors, bitmaps, and OLE objects

to GIF images.

With Help & Manual's ability to merge multiple help files and convert cross-references among separate parts into intra-file links, different people can work on different parts of the help file development. While designed as a development tool for programmers, Help & Manual can be used

by network administrators, business people, and anyone who wants to create printed manuals or structured HTML documents with hyperlinks.

**EC Software**  
**Price:** US\$229 for a single-user license; additional licenses cost US\$49 each.  
**Phone:** +43 6212-7838  
**Web Site:** <http://www.ec-software.com>



November 1999



## HREF Tools' WebHub Selected for Borland Web Site

*Santa Rosa, CA* — On July 1, 1999, Borland decided to use HREF's WebHub to implement some of the crucial pieces of its <http://community.borland.com> Web site. The site provides a forum for developers to obtain and exchange information regarding all aspects of Inprise/Borland products.

WebHub was used to integrate a diverse set of databases and numerous types of applications and development environments running on multiple NT and UNIX servers. WebHub detects

when users should log in and accurately returns the user to their desired entry point once that has been accomplished.

The entire site was previewed

July 21, 1999 at the Borland conference.

HREF products are available for online purchase at <http://www.href.com>.

## Inprise Announces Commitment to Linux

*San Jose, CA* — Inprise Corp. announced its commitment to support the Linux platform. The company announced the immediate availability of VisiBroker for Linux, a new version of its object request broker, and has demonstrated JBuilder for Linux, its new Java development tool.

In addition, Inprise is releasing the complete results of its Linux developer survey, in which over 24,000 respondents indicated that Linux is a critical operating system for their customers moving forward. The results also indicate that the majority of developers participating in the survey are planning application development and client/server database development projects on Linux. According to a recent report by International Data Corp., because applications drive operating system sales, having more applications available on Linux will drive Linux in Enterprise computing.

Additional findings from the Borland Linux Developer survey are available at <http://www.borland.com/linux/survey>.

## Dunn Systems Receives the Inprise 1999 Partner Solution of the Year Award

*Chicago, IL* — Dunn Systems, Inc. received the Inprise 1999 Partner Solution of the Year Award at the Inprise Annual User Conference Award Ceremony held in Philadelphia this past July. Accepting the award from Inprise President Dale Fuller and Borland.com Vice President David Intersimone was William Dunn, President, and David Piech, Vice President of Sales and Marketing for Dunn Systems, Inc.

The winning application, from 150 submitted, was a Java Enabled Tax System (JETS) that Dunn Systems developed for the State of North Carolina Department of Revenue using JBuilder and VisiBroker. The criteria Inprise used in judging the applications included technology used, complexity, and the impact of the application on the organization.

JETS is a comprehensive client/server application that allows the North Carolina Department of Revenue to manage every facet of a taxpayer's account. Users can enter information from tax returns, review the status of refunds and delinquencies, and generate reports and correspondence.

Because the state of North Carolina now employs a single system, multiple entries of the same information has largely been eliminated. JETS is also faster than the legacy system and sports more reliable backup

and archival features.

Dunn Systems, Inc. (Skokie, IL) has been providing IT consulting services to Global 2000 businesses since 1988 and has experience in business-to-business e-commerce, n-tier application development, data warehousing, quality assurance, and training. For more information on Dunn Systems, visit <http://www.dunnsys.com>, or call (847) 673-0900.

## Delphi 5 Focuses on Internet and Enterprise Development

*Scotts Valley, CA* — Today, applications need to access many sites and data sources. Leveraging the Internet and e-business opportunities must be accomplished with thin-client applications offering the same functionality through standard Web browsers. Applications must handle spikes in usership while remaining available and responsive. Delphi 5 delivers these capabilities in one integrated development environment.

Delphi 5 is built for today's Internet and enterprise developer.

Delphi 5 Professional features include WebBroker and native Internet components to develop and deploy Web applications faster; Data Module Designer, To Do Lists, and the Control Panel Wizard for higher productivity; Project Browser with Code Explorer, hyperlinks, and history lists for better project management; frames support and additional Property Editors to visually

build components; enhanced debugger with Breakpoint ToolTips, Actions, Groups, FPU/MMx View, drag-and-drop support, and more; and Import COM Servers, including a full suite of Microsoft Office Automation Controller components, plus complete support for building ActiveX components and OLE controls.

Delphi Enterprise features include XML and HTML 4 support for faster Web development; InternetExpress with Web Client Page Wizard, Borland MIDAS Page Producer, and WebBroker for building and deploying high-speed Web applications; ADOExpress to quickly access all types of information; TeamSource for higher development team productivity; and Borland Translation Suite for easier localization of applications.

For additional information, visit <http://inprise-news.com/Key=1436.Icn.B.EQBaP8>.





By *Dennis P. Butler*

## CORBA

### Creating Clients and Servers

**F**rom the stand-alone personal computer, to heterogeneous distributed clients and servers, and everywhere in between, the computing industry has evolved dramatically over the past several decades. Computer professionals are constantly required to use their skills to the utmost in their current environment, and then be able to migrate to the next level when the current framework becomes too complicated or restricting. The evolution of computing continues to point toward a common goal: simplify the work process by better planning, faster development, and sharing of resources.

The CORBA architecture — short for Common Object Request Broker Architecture — was designed to accomplish this goal. The CORBA architecture defines and implements the framework for applications to communicate across such previously unbreakable boundaries as multiple operating systems and programming languages. This capability is achieved through the use of a common interface and information passing mechanism implemented in different programming languages.

There are several factors that set CORBA apart from competitive proprietary information sharing technologies. First of all, CORBA is an open standard; that is, the specification is constantly being reviewed and updated by the OMG, or Object Management Group. This group is made up of hundreds of companies worldwide that decide how to evolve the CORBA specification. This process of evolution has been occurring since 1991, when CORBA 1.0 was released. Another feature that sets CORBA apart from other technologies is that the interface is common among languages, not the implementation of it. Other information sharing methods rely on operating-system-specific implementations to pass information. While this may be useful in LAN/WAN or intranet environments where operating systems can be standardized, true distributed applications that need to operate over any OS require a more complete solution, such as CORBA. With CORBA, the client doesn't need to know any details of how the object that it will obtain from a server was implemented.

The starting point for CORBA applications is the interface that applications share when passing

information. This common interface that defines what information is going to be passed is called IDL, short for Interface Definition Language. IDL is its own language, although the syntax is similar to that of Java and C++. As its name implies, the only purpose of this language is to define the interface for objects that will be passed between CORBA applications. The implementation and use of these objects is done in the specific target language chosen. The only stipulation here is that the target language has facilities to map to the CORBA architecture.

This is where Delphi comes in. CORBA development is typically associated with C++ or Java development. However, even non-object oriented languages such as C and COBOL have mappings to CORBA, and thus can take advantage of the open architecture. As we'll see later in this article, Delphi employs several methods to use CORBA in an application. CORBA can be implemented through the use of the Type Library editor for easily creating IDL interfaces, through MIDAS to connect to CORBA data, and soon Delphi will gain direct facilities to compile IDL code into Pascal source, which can be used to implement and use the CORBA objects. Much like the IDL2JAVA utility, this IDL2PAS utility will be available soon to Delphi developers to give complete control and flexibility in creating CORBA applications.

#### VisiBroker CORBA

VisiBroker is the ORB (Object Request Broker) used throughout this article and in the examples. VisiBroker is the Inprise implementation of the CORBA standard that adds many additional features to assist developers when creating applica-

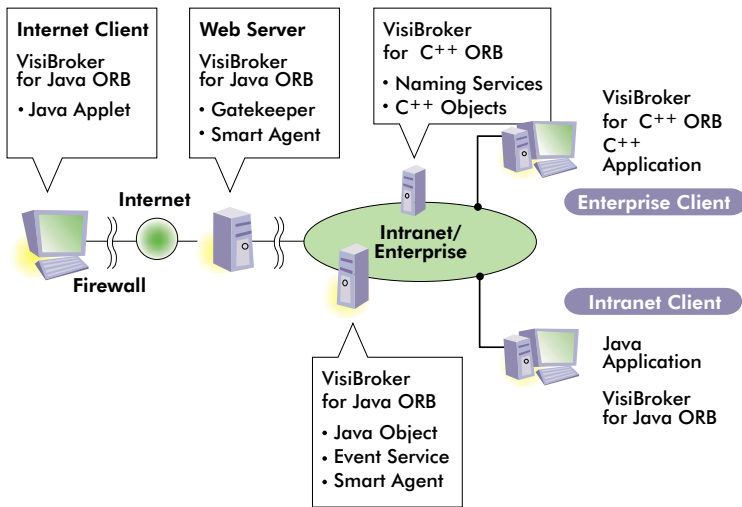


Figure 1: Sample high-level CORBA implementation.

tions. Support for thread management and connection management is included, as well as many libraries and other utilities that are created to assist the developers when developing CORBA applications. Knowledge of the intricacies of VisiBroker isn't required for this article. For the examples that are used, VisiBroker additions and standard CORBA features are used together, much as they would be for actual production situations.

Let's take a quick look at a high-level diagram of how CORBA fits into distributed applications (see Figure 1).

As you can see in this standard Inprise diagram, there can be many levels of connection across boundaries such as the Internet, an intranet, or other internal networks. This diagram introduces many VisiBroker specific items such as the Gatekeeper, Smart Agent, Naming Services, and Event Service. All we need to grasp at this point from the diagram is that one or more IDL interfaces for CORBA objects has been generated, and instances of those server object implementations are being passed to various clients. As shown in the diagram, clients can include the Internet Client running a Java applet, or the C++ application running on the corporate intranet. This article will show how Delphi can be used in this network to also take advantage of the CORBA architecture.

Before we get started with Delphi, let's take a quick look at how applications communicate with CORBA. This information will be relevant later in the article when using Delphi to implement this technology. We will start here with a very simple example of how a basic CORBA application can be started between two machines:

- First, the ORB Smart Agent (osagent) must be run on a machine on the network. The osagent will keep track of all server object implementations that have been registered with it, as well as keeping track of other osagents.
- Next, the server application must be run. This will register its object(s) with the osagent to let it know that it has object implementation(s) available for client applications.
- A client application is started. When the client application requires a server object, it will issue a UDP broadcast to find the closest osagent to search for that implementation. The osagent will find the object implementation that the client is looking for, thus allowing a connection to be established between the client and server. The client can now access the server object directly.

Now that we know the steps that take place in a basic CORBA application, we'll apply them to Delphi to see how they're accomplished. For this first example, we'll create an online auction demonstration, where the server will keep track of a particular product, and clients will bid against each other to try to buy the product. For each successful bid, the client application will notify that the bid was successful, and will update the screen to show the high bid amount. Further bids by other clients will now have to out-bid that new highest amount to win the product (which, of course, is a copy of Delphi 5 Enterprise Edition).

### Example 1: The Online Auction

Our first step in this example is to create the CORBA object for our server, and create the server that will implement this object. As I mentioned earlier, the CORBA objects are defined by IDL. Delphi developers don't need to know IDL to create their object; instead, this can be done through the use of the Type Library editor. This handy utility allows visual creation of objects and their interfaces. This utility can also be used later to export to IDL for use in other implementations of the object.

To create the server and its object, start a new Delphi application and save the form and project. You may want to shrink the dimensions of the form, as this will be your server application running on your machine. For this example, I have named the files Cserver.dpr and Cmain.pas. From the main menu of Delphi, select File | New, then select the CORBA Object item on the Multitier page. The CORBA Object Wizard will be displayed (see Figure 2).

As you can see, the object to be defined is named `OnlineAuction`, will be a shared instance, and will be single-threaded. The information required by this dialog is described in more detail below.

**Class Name.** Enter the base name of the object that implements the CORBA interface for your object. Filling in the class name will do two things; it will create a class of this name with a "T" prepended, and create an interface for the class using this name with an "I" prepended.

**Instancing.** Use the Instancing combo box to indicate how your CORBA server application creates instances of the CORBA object. There are two possible values:

- **Instance-per-client** — A new CORBA object instance is created for each client connection. The instance persists as long as the connection is open. When the client connection closes, the instance is freed.
- **Shared instance** — A single instance of the CORBA object handles all client requests. Because the single instance is shared by all clients, it must be stateless.

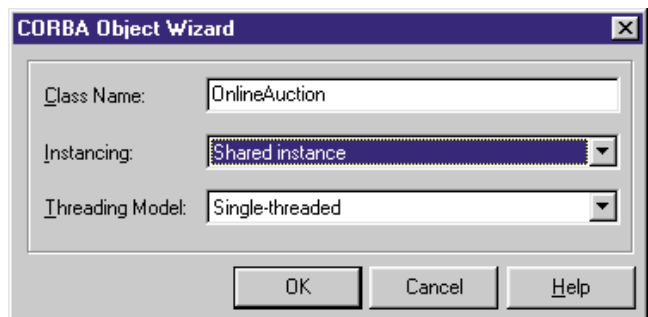
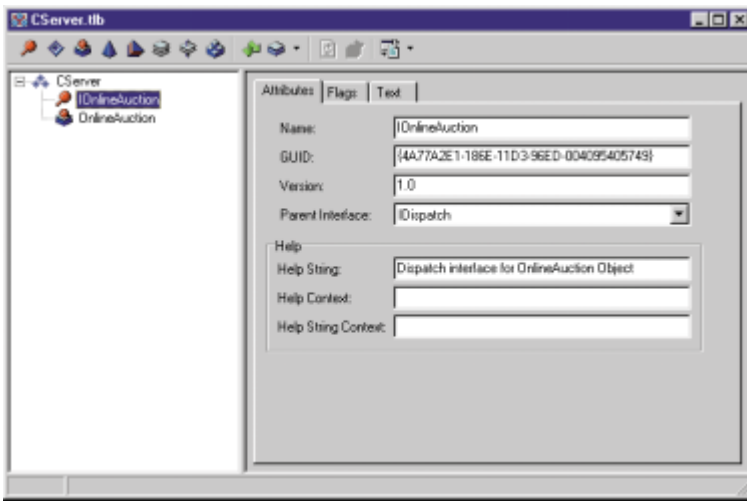


Figure 2: Creating the CORBA object interface.



**Figure 3:** Delphi's Type Library editor.

**Threading.** Use the **Threading Model** combo box to indicate how client calls invoke your remote data module's interface. Again, there are two possible values:

- **Single-threaded** — Each object instance is guaranteed to receive only one client request at a time. Instance data is safe from thread conflicts, but global memory must be explicitly protected.
- **Multithreaded** — Each client connection has its own dedicated thread. However, the object may receive multiple client calls simultaneously, each on a separate thread. Both global memory and instance data must be explicitly protected against thread conflicts.

In this example, the server object will be a shared instance because we want all clients to access the same auction object, so they can bid against each other. If we were writing an object for a banking application, an object could be created that would contain information specific to a banking account of the customer running the client application. In this case, an **Instance-per-client** setting would be more appropriate, since a separate object would be created for each client, making the contents of that object private to the client.

The object is also created for single threading; since only one client request will be processed at a time, multi-threading isn't necessary. Multi-threading is very useful when developing very large applications that require a higher level of flexibility in situations where the server must be able to handle multiple requests that may be occurring simultaneously. For this simple example we won't take advantage of this feature.

Click **OK** to create the new unit and save the file. This will create the Pascal shell for the CORBA object interface. Instead of having to type in the interface manually, Delphi allows us to visually create the object interface using the Type Library editor (see **Figure 3**). (The Type Library editor is available from the main menu by selecting **View | Type Library**.)

The Type Library editor allows us to specify all the information we need to define the interface of our CORBA object. For this example, we want to create a server object for our online auction that will hold information about the latest high bid amount and person. We will also add a property for the product that is being bid upon. We also want to add methods to place a new bid, and check information about the current bid. In true object-oriented fashion, properties cannot be modified directly, they must use accessor methods to

change their values. As we'll see, the Type Library editor takes care of this as well.

The Type Library editor is also used to define interfaces to COM objects; in fact, this was the original purpose of the Type Library editor. Because of this, it has some features that aren't used for CORBA objects. An example of this is the "Help" information shown in **Figure 3**. Also, some of the data types that are available in the Type Library editor may not be CORBA-compliant data types. Developers can easily research CORBA data types through the Delphi or VisiBroker help files.

As we can see in **Figure 3**, an Interface and CoClass have been created for our CORBA object. All we are concerned with is the Interface. The CoClass that has been created is COM-specific, and can be ignored. Note that the interface has taken our CORBA object name and prepended it with an "I", as mentioned earlier.

Methods and properties are added by right-clicking on the *IOneAuction* interface, and selecting either **Method** or **Property** from the **New** submenu. Add the following methods to the *IOneAuction* interface:

- *PlaceBid* returns an Integer, takes a Double (call it *Amount*) and WideString (call it *CustomerName*) as parameters.
- *GetCurrentPrice* returns a Double, no parameters.
- *GetCurrentUser* returns a WideString, no parameters.

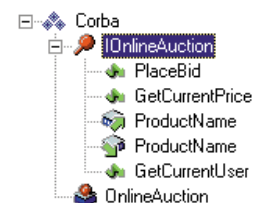
Add the *ProductName* property to the *IOneAuction* interface.

When entering the parameters for the methods in the Type Library editor, the following information is required:

- **Modifier** specifies the nature of the parameter, such as whether it should be treated as an in parameter, out parameter, etc.
- **Name** is the name of the parameter.
- **Type** is the data type of the parameter. The list contains the list of CORBA-compliant data types.
- **Default Value** specifies whether the parameter will have a default value.

For the purposes of this example, the modifier was kept blank for all parameters because no special setting was needed for this example. This defaults the parameter to the **in** setting of IDL; thus modifications made to the parameter variable once the called procedure is completed won't be reflected when control is returned. In fact, the Delphi compiler will show a hint for a parameter within its method if an attempt it made to modify the value of the parameter. There are several types of modifiers that can be used that correspond to IDL parameter types. The most commonly used parameters in IDL are **in**, **out**, and **inout**. This means that parameter information flows into the server, out from the server, or both. This is done with the Type Library editor settings of blank(**in**), out(**out**), and var(**inout**).

When completed, the tree view for the Type Library editor should look like **Figure 4**. Click on the "Refresh Interface" button (it looks like the two-arringed recycle symbol) from the Type Library editor to synchronize the source file for the CORBA object. Note: This isn't entirely WYSIWYG, as is the standard Delphi IDE; the "Refresh Interface" button must be clicked.



**Figure 4:** Completed type library tree.



Once the refresh button has been clicked, the Type Library editor can be closed, and the source file can be saved (CSrvObj.pas in this case). We now have our server object interface defined, and the Pascal code shell from which we can add functionality for the object. The Type Library editor has created some files for us, such as the \_TLB stub file that's created based on the server application project name. In this example, since the project was named CServer, it's CServer\_TLB.pas. Since this file is automatically generated, no additional work is needed on it. This file sets up stub and skeleton classes for the server object, as well as defining several other classes that may be used, such as the CORBA object factory class and the COM CoClass class. The only thing we really need to know at this point is that the CORBA shell has been created for us from how we defined the interface in the Type Library editor, and a TLB file has been created from which we can get our object reference for the server.

Our server source file csrvobj.pas has been filled in from the Type Library editor with the methods and properties that were defined. The empty shell, csrvobj.pas, is shown in [Listing One](#). Now we need to code the implementation of the object. We need to add a few private variables to hold the current high-bid price, the current high-bid customer, and the product being bid on. We also need to initialize these private variables in the constructor for the object. Finally, we need to implement the object with code to provide functionality to the methods that have been created for us. The completed source, with comments, is shown in [Listing Two](#).

All that was provided by Delphi was the code shell; the rest had to be filled in to give the object interface an implementation. We now have the server for our object. To use this server of our CORBA object, all we need to do is add the CSrvPas unit to the `uses` clause of any form of a project. When this is done, the initialization code for the object will be fired when that form is used. Thus, the server will be started, and an object will be created that is available for use.

## CORBA Clients in Delphi

Now, our server has been set up, and should provide objects as necessary for any clients looking for an instance of *TOnlineAuction*. Now it's time to create a client to access and use this object. In CORBA, there are two ways a client can get an instance of a server object. The first is known as *early binding*, or *static binding*. This means the client has knowledge of what type of CORBA object it's going to interface with. This means that another file, known as the *stub*, will be used to handle the passing of data between the client and server — processes known as *marshaling* and *unmarshaling*, respectively. The complexity of the marshaling process is taken care of for us by the stub, which makes it much easier to implement.

The other way to access the server object from the client is known as *late binding*, or *dynamic binding*. Dynamic binding is also commonly referred to as DII, or Dynamic Invocation Interface. This means the client has no prior knowledge of the server object, and thus knows nothing about the structure of objects that it can access. An observation at this point is that the client stub doesn't get used, since the client doesn't have the facility to know the structure of server objects at design time. This knowledge is what is provided by the stub to the client.

The advantage of DII is that clients can be created that may never need to be rebuilt when a server object is changed; the client code can remain constant through many changes to the server object that it uses. This is done through the use of another CORBA construct known as the Interface Repository. This holds run-time type information about what is available to the client, and allows the client to

use the available services. A drawback to this approach, as compared to early binding, is that it is more complex, slower, and requires more work for the developer.

In our Delphi example, it's still relatively simple, through the use of the type library files, and the *TAny* class. Since we know what server implementation we're looking for, we can access the methods directly using *TAny*. In actual production situations, the client may not know what server objects and methods are available, and may need to have more complicated code to accommodate this. For this article, both early and late binding clients will be created for use with the CORBA server we've created.

(There is also the capability in CORBA to provide a DSI, or Dynamic Skeleton Interface. Like DII for the server, this allows the CORBA servers to have no knowledge at compile time of what objects that will be available to them. This is also done using an Interface Repository to store information. This technique will not be covered in this article, but it's important to know that it's available.)

Before we go any further, we should provide more explanation for these new CORBA terms that we have introduced:

- **Client Stub and Server Skeleton.** These two CORBA features are used to convert information to be passed between the client or server into the CORBA packet format to be sent over the network. The stub is a layer between the client and the ORB layer, and provides a means for the client application to send information to the server. The stub is a layer between the server and the ORB layer that converts the parameters and other information sent by the client, so it can be used by the server in performing the action the client requires.
- **Marshaling and Unmarshaling.** Marshaling is the process of converting parameter values and other information so they can be sent over the network. Marshaling is accomplished by the stub to send information to a server. Unmarshaling is the opposite process, where the server skeleton converts information that has been sent over the network into the parameter values, and calls the appropriate function for the client.
- **Run-time Type Information.** RTTI is information available at run time about objects in a system. Delphi and CORBA both incorporate this feature. It's especially important when creating functionality that will wait until run time to see what types of objects are available for use, or for providing different types of functionality based on what objects are being used.

Since it's easier, the early binding client will be created first. By using the Type Library editor, we have all the files we need for our early binding client. The Type Library editor creates a stub file for us in the form of a TLB file.

When we create the client, we'll need to add this file to the `uses` section of the form, so we have a reference to the structure of the server object. We also need to add

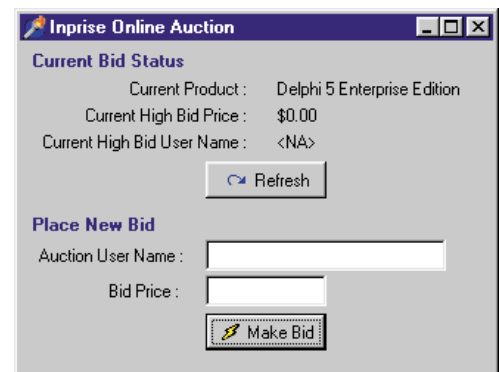


Figure 5: The Online Auction client.

CorbaObj to the **uses** section to perform the binding necessary to communicate across the ORB.

Figure 5 contains an image of our client. It has functionality to refresh the bid information, enter information for a new bid, and place the bid.

### Early Binding Client

As mentioned earlier, the early binding client uses the type library file generated by the Type Library editor to get a reference to the CORBA object that our server will create. The client code we have to access and use this object is shown in Listing Three. In this client, we implement all the methods from our server object. We're able to do this because we know the structure of the server object through the *IOnlineAuction* interface. We get the reference to the interface to the server object when the client starts, by getting an instance from the CORBA factory. We can then perform any operation on the object.

We must take several steps before running this client. The ORB Smart Agent must be running on the network somewhere — on the server machine, or on some other machine. To do this, run:

```
osagent -C
```

from the command line, or run **VisiBroker Smart Agent** from the **VisiBroker** folder installed in the Delphi folder. The **-C** at the command line designates that the **osagent** will run in the taskbar; otherwise it will not appear there, so it may not be apparent whether it's running while you're testing. Once the ORB Smart Agent is running, start the server application.

Once the server has been started, it's a good idea to ensure that the server objects are available for any clients. The **VisiBroker** utility, **osfind**, can be used to do this. Run **osfind** from the command line on the client machine to display a list of available objects within the subnet of the machine. This will verify that the client has access to the necessary server objects. For complicated implementations of CORBA, the **osagents** can be configured to look for server objects, or other **osagents** outside the current subnet. Although this isn't covered in this article, suffice it to say there are facilities to allow the client to look virtually anywhere for a server object, as long as the **osagents** have been set up correctly.

The final step is to run several clients. These should automatically get a reference to the server by including the type library file that was generated, and the client should have access to all server functions. In our example, we can launch many clients from different machines (within the same network subnet), and make successive bids against the server.

As you can see, not much is required to create an early-binding client to our server object in this simple example. DII is a little more complicated, as we'll see next.

### Late Binding Client

As described earlier, the late binding client has no knowledge of the structure of available server objects at design time, and must use a facility called the Interface Repository to find what is available. In this example, we'll implement this client and describe the requirements, benefits, and drawbacks in using this approach.

Before we get to writing the client, there are a few requirements that must be met. First, the interface for the object must be registered with an interface repository. To do this, we must first have an IDL

file. This can be created easily by returning to the Type Library editor, and selecting **Export to CORBA IDL**. This is done by dropping down the last button on the right of the toolbar (see Figure 6). In this example, the CORBA IDL setting must be selected. The MIDL export setting will not work with the interface repository functionality we are going to use.

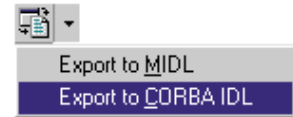


Figure 6: Selecting **Export to CORBA IDL**.

This will create the IDL file that corresponds to the server object defined previously. The filename will be `<ProjectName>.IDL` wherever the server application source has been saved. This IDL must then be registered with an interface repository. The **osagent** and the server should be started, before starting the interface repository. Then, start the interface repository. At this point, we can load the IDL for our CORBA object into the repository, so clients can see it's available on one of the available servers. The interface repository can be started by running the following statement from the command line:

```
irep <Repository Name>
```

The repository name can be anything you want, and will launch the Interface Repository application. Once it's open, you can either select **File Load** from the main menu, and select the IDL file that was exported above, or run the following statement from the command line:

```
idl2ir <IDL File Name>
```

Once this has been done, we've registered our interface with the Interface Repository. To verify that the interface has been registered with the Interface Repository, click the "Lookup" button after running the above line from the command line or loading the IDL directly. You should see something similar to Figure 7.

The only step left at this point is to create the client that will access the Interface Repository, and use an object stored there. To start with, we'll use the same form layout as in the early binding example. Start a new project in Delphi. Instead of naming it `Client.dpr` (as in the early binding example), name it `Client_DII.dpr`. Copy the client form from the early binding example, and save it as `cclient_dii.pas`. Also, change the *Caption* to designate that it's the DII version of the form. Finally, rename the form itself to *TfrmDynamicCorbaClient*. These changes aren't necessary, but serve to distinguish the different client implementations.

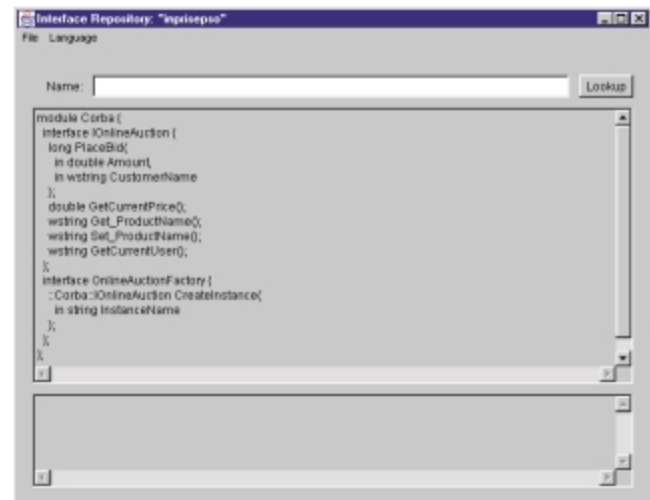


Figure 7: The Interface Repository with **CServer** IDL loaded.

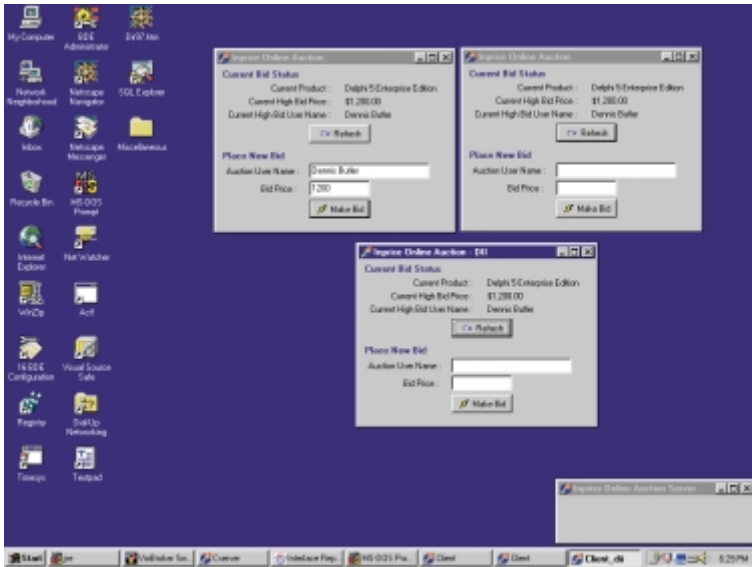


Figure 8: Desktop with server, two static clients, and one dynamic client.

Since we aren't going to use the stub that was generated for us, remove the reference to `Cserver_TLB` in the `uses` clause of the client that was carried over from the early-binding example. The code is slightly different since we don't have the client stub to give us a direct reference to the interface. We use the `TAny` class, a CORBA interface type for DII, to get a reference from the interface repository for our server object. In this case, we'll get an instance of the object factory, which will get a reference to the server object. This is done to mimic the process of the non-DII client shown earlier.

Aside from additional coding needed to get references to our server object through the object factory, the code for the late binding client remains largely identical to the early binding client. The code for the second client is shown in Listing Four. The server, and the early and late binding clients running at the same time, is shown in Figure 8.

To review, here are the steps that were taken to run the server and two types of clients on the same machine:

- Start the ORB Smart Agent (`osagent -c`)
- Start the Server (`CServer.exe`)
- Run the Interface Repository (`irep inprise9s`)
- Load the interface into the Interface Repository (`idl2ir CServer.idl`)
- Run the early binding client (`Client.exe`)
- Run the late binding client (`Client_DII.exe`)

In this example, we've shown how to implement both types of CORBA clients through Delphi. This is fine in our small example, but in real-world environments, the value of CORBA is that the clients or servers can be written in any language with the common IDL interface. The next section will review how to share this IDL information with other languages, based on what we have already created in Delphi.

### Clients in Other Languages

When we created the dynamic binding client, we needed to export the IDL for our server object so the interface repository would have a reference to what objects were available. This IDL file can also be used by any other CORBA-compliant language to provide an interface to

our server object. Tools such as JBuilder and C++Builder can be used to provide additional clients or servers based on this IDL file. In this example, we'll use JBuilder because it's a wholesale departure from the Delphi/C++Builder IDE.

In JBuilder, create a new application with a single frame. In the project manager, add the `CServer.idl` file we saved above. The file will appear in your JBuilder project's file list. Right-click on the IDL file and select **Build**. This runs the IDL file through the IDL2JAVA precompiler. The IDL2JAVA precompiler converts the IDL file to Java stub classes. The generated Java files can be used to create CORBA servers to implement these objects or CORBA clients to access the objects.

Design the frame so that it looks similar to the Delphi client that was designed earlier. Figure 9 shows what was done for our Java client.

The code for the early binding Java client will be similar to the Delphi client; we'll have variables for the object factory, and a server object that will be obtained from that factory. The code for the client is shown in Listing Five.

As you can see, we've declared the object factory and interface in our source file. In the constructor for the frame, there's a different method performed here than in Delphi to attach to the ORB and obtain an object reference. All that needs to be known is that the Java application is getting a reference to the server object through the use of automatically-generated Helper files. By doing this, an object reference is obtained, and is used in the same manner as the Delphi client. Helper files and other CORBA files are generated from the IDL2JAVA utility, which was run when the `CServer.idl` file was compiled. JBuilder uses this method to create the stub and skeleton files, as compared to using the Type Library editor in Delphi.

Once an object reference has been obtained, the code for the frame itself is similar to the Delphi application. The Delphi CORBA server has no knowledge of what language is being used for requests; Delphi and Java clients make virtually identical calls to the server object

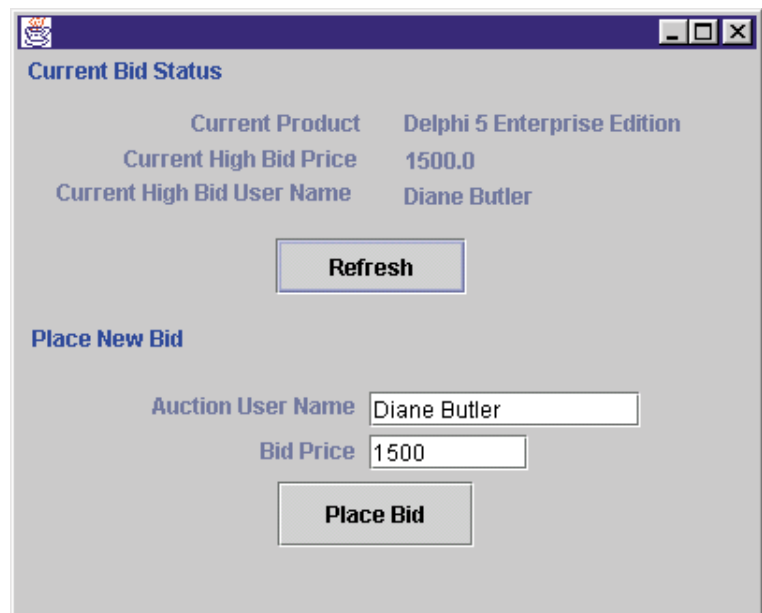


Figure 9: Java CORBA client.

through their stub files. Our Java client could have been running on a UNIX machine located on a different continent from our Delphi server. As long as the CORBA subnet or osagents were configured correctly, these separate processes could talk to each other just as easily as if they were on the same machine. This simple Java/Delphi example provides a mere glimpse of the full potential of CORBA.

## Conclusion

There's no doubt that CORBA will continue to gain momentum in enterprise computing due to its tremendous assets: flexibility, language independence, and a wide range of capabilities for virtually every distributed need. Delphi combines these assets with RAD development to make CORBA programming easier and faster for the developer, without sacrificing CORBA's capabilities. As we saw in these simple examples, Delphi is an ideal platform for setting up CORBA clients and servers for many types of applications.

Inprise developers also have advanced CORBA capabilities available through the use of the MIDAS technology. MIDAS allows users to create complicated queries easily through Delphi, and pass query results back from remote datasets using CORBA as the transportation format. This technology is especially powerful, because developers don't need to create complicated objects to hold query output; MIDAS automates this task, creating stub and skeleton classes automatically. The MIDAS technology is available in several Inprise development tools and will continue to play a key part in RAD CORBA development.

Going forward, Delphi developers can expect to see more CORBA support in new releases of Delphi. The IDL2PAS utility, when released, will give Delphi developers access to all CORBA features and will not limit implementations to the framework that Delphi has provided. This will provide the best of both worlds: RAD development for standard CORBA tasks as covered in this article, and granular CORBA development for more specific and complicated implementations through IDL2PAS.

Delphi has long been regarded as the best Windows development tool. With the merging of CORBA technology to Delphi, this reputation will only grow as Delphi's capabilities now reach across previously unbreakable boundaries, such as multiple operating systems and languages.  $\Delta$

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\NOV\DI9911DB.*

Dennis P. Butler is a Senior Consultant for Inprise Corp., based out of the Professional Services Organization office in Marlboro, MA. He has presented numerous talks at Inprise Developer Conferences in both the US and Canada, and has written a variety of articles for various technical magazines, including *CBuilderMag.com*. He can be reached at [dbutler@inprise.com](mailto:dbutler@inprise.com), or (508) 481-1400.

## Begin Listing One — csrvobj.pas shell

```
unit csrvobj;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  ComObj, StdVcl, CorbaObj, CServer_TLB;

type
  TOnlineAuction = class(TCorbaImplementation,
```

```
    TOnlineAuction)
protected
  function Get_ProductName: WideString; safecall;
  function GetCurrentPrice: Double; safecall;
  function GetCurrentUser: WideString; safecall;
  function PlaceBid(Amount: Double;
    const CustomerName: WideString): Integer; safecall;
  procedure Set_ProductName(const Value: WideString);
    safecall;
end;

implementation

uses
  CorbInit;

function TOnlineAuction.Get_ProductName: WideString;
begin
end;

function TOnlineAuction.GetCurrentPrice: Double;
begin
end;

function TOnlineAuction.GetCurrentUser: WideString;
begin
end;

function TOnlineAuction.PlaceBid(Amount: Double;
  const CustomerName: WideString): Integer;
begin
end;

procedure TOnlineAuction.Set_ProductName(
  const Value: WideString);
begin
end;

initialization
  TCorbaObjectFactory.Create('OnlineAuctionFactory',
    'OnlineAuction', 'IDL:CServer/OnlineAuctionFactory:1.0',
    IOnlineAuction, TOnlineAuction, iSingleInstance,
    tmSingleThread);
end.
```

## End Listing One

## Begin Listing Two — Implemented csrvobj.pas

```
unit csrvobj;

interface

// Note the included units for ComObj, CorbaObj,
// and Cserver_TLB.
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  ComObj, StdVcl, CorbaObj, CServer_TLB;

// Class is defined as a CORBA class that implements
// the IOnlineAuction interface.
type
  TOnlineAuction = class(TCorbaImplementation,
    IOnlineAuction)
  private
    // Private variables to hold object information
    // about auction.
    FProductName : WideString;
    FCurrentPrice : Double;
    FCurrentCustomer : WideString;
  public
    // Override create to initialize private variables.
    constructor Create(Controller: IObject;
```

```

    AFactory: TCorbaFactory); override;
protected
    // Accessor methods for FProductName property.
    function Get_ProductName: WideString; safecall;
    procedure Set_ProductName(const Value: WideString);
        safecall;
    // Function to get the current price for the
    // auction product.
    function GetCurrentPrice: Double; safecall;
    // Function to get the current customer for the
    // auction product.
    function GetCurrentUser: WideString; safecall;
    // Function to place a new bid.
    function PlaceBid(Amount: Double;
        const CustomerName: WideString): Integer; safecall;
end;

implementation

// Included with Delphi to initialize CORBA object.
uses
    CorbInit;

// Overridden create for our object.
constructor TOnlineAuction.Create(Controllor: IOBject;
    AFactory: TCorbaFactory);
begin
    inherited;
    // Initialize our private variables.
    FProductName := '<NA>';
    FCurrentCustomer := '<NA>';
    FCurrentPrice := 0;
end;

// Method to get the property value for the current
// auction product.
function TOnlineAuction.Get_ProductName: WideString;
begin
    Result := FProductName;
end;

// Method to set the property value for the current
// auction product.
procedure TOnlineAuction.Set_ProductName(
    const Value: WideString);
begin
    FProductName := Value;
end;

// Method to get the current price of the high bid.
function TOnlineAuction.GetCurrentPrice: Double;
begin
    Result := FCurrentPrice;
end;

// Method to get the current customer name of the high bid.
function TOnlineAuction.GetCurrentUser: WideString;
begin
    Result := FCurrentCustomer;
end;

// Method to place a new bid: Take parameters for amount of
// bid and customer who is placing the bid.
function TOnlineAuction.PlaceBid(Amount: Double;
    const CustomerName: WideString): Integer;
begin
    if Amount > FCurrentPrice then
        begin
            FCurrentPrice := Amount;
            FCurrentCustomer := CustomerName;
            Result := 1;
        end
    else
        Result := 0;
    end;
end;

// Code provided by Delphi to call the generated CORBA
// object factory to get an object reference for the
// server. Note parameters match what we defined in the

```

```

// CORBA object wizard. Since it's in the initialization
// section, the code will run whenever this unit is
// included in the uses section of another unit and the
// server will be started.
initialization
    TCorbaObjectFactory.Create('OnlineAuctionFactory',
        'OnlineAuction', 'IDL:CServer/OnlineAuctionFactory:1.0',
        IOnlineAuction, TOnlineAuction, iSingleInstance,
        tmSingleThread);
end.

```

## End Listing Two

### Begin Listing Three — Implemented cclient.pas

```

unit cclient;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls,
    Forms, Dialogs, CorbaObj, CServer_TLB, StdCtrls, Mask,
    Buttons;

type
    TfrmStaticCorbaClient = class(TForm)
        lblCurrentProduct: TLabel;
        lblBidPrice: TLabel;
        lblProduct: TLabel;
        lblCurrentHighBidPrice: TLabel;
        lblPrice: TLabel;
        btnRefresh: TBitBtn;
        edtBidPrice: TEdit;
        lblCustomerName: TLabel;
        edtUserName: TEdit;
        btnMakeBid: TBitBtn;
        lblCurrentHighBidUser: TLabel;
        lblUser: TLabel;
        lblBidStatus: TLabel;
        lblPlaceNewBid: TLabel;
        procedure FormCreate(Sender: TObject);
        procedure btnRefreshClick(Sender: TObject);
        procedure btnMakeBidClick(Sender: TObject);
    public
        // Our interface to the server object.
        AuctionInterface : IOnlineAuction;
    end;

var
    frmStaticCorbaClient: TfrmStaticCorbaClient;

implementation

{$R *.DFM}

// On create, we immediately establish connection to server
// using interface defined in type library stub and do
// client initializations.
procedure TfrmStaticCorbaClient.FormCreate(Sender: TObject);
begin
    // Call factory to get reference to the server object.
    AuctionInterface :=
        TOnlineAuctionCorbaFactory.CreateInstance('');
    // Set the product name; resets it for each client. This
    // wouldn't be done in production, but it's done here to
    // demonstrate use of accessor methods created by Type
    // Library editor for object properties.
    AuctionInterface.Set_ProductName(
        'Delphi 5 Enterprise Edition');
    // Refresh with server to get latest information.
    btnRefresh.Click;
end;

// Refreshes information from server. This example doesn't
// implement server callbacks, so refreshes must be
// done manually.
procedure TfrmStaticCorbaClient.btnRefreshClick(
    Sender: TObject);

```



```

begin
  // Update price and customer name information for
  // current product.
  lblPrice.Caption := FloatToStrF(
    AuctionInterface.GetCurrentPrice, ffCurrency, 18, 2);
  lblUser.Caption := AuctionInterface.GetCurrentUser;
  lblProduct.Caption := AuctionInterface.Get_ProductName;
end;

// Call object to place a new bid against the server.
procedure TfrmStaticCorbaClient.btnMakeBidClick(
  Sender: TObject);
begin
  // Do some client-side data checking to save speed.
  if edtUserName.Text = '' then
    begin
      ShowMessage('You must enter a user name first.');
```

### End Listing Three

### Begin Listing Four — Implemented cclient\_dii.pas

```

unit cclient_dii;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, CorbaObj, StdCtrls, Mask, Buttons;

type
  TfrmDynamicCorbaClient = class(TForm)
    lblCurrentProduct: TLabel;
    lblBidPrice: TLabel;
    lblProduct: TLabel;
    lblCurrentHighBidPrice: TLabel;
    lblPrice: TLabel;
    btnRefresh: TBitBtn;
    edtBidPrice: TEdit;
    lblCustomerName: TLabel;
    edtUserName: TEdit;
    btnMakeBid: TBitBtn;
    lblCurrentHighBidUser: TLabel;
    lblUser: TLabel;
    lblBidStatus: TLabel;
    lblPlaceNewBid: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure btnRefreshClick(Sender: TObject);
    procedure btnMakeBidClick(Sender: TObject);
  public
    // Servers declared as type TAny; special type for
    // CORBA interfaces using DII.
    AuctionFactory,
    AuctionServer : TAny;
  end;

var
  frmDynamicCorbaClient: TfrmDynamicCorbaClient;
```

### implementation

```

{ $R *.DFM }

procedure TfrmDynamicCorbaClient.FormCreate(
  Sender: TObject);
begin
  try
    // Bind to ORB instance for object factory.
    AuctionFactory :=
      ORB.Bind('IDL:CServer/OnlineAuctionFactory:1.0');
    // Create reference to server object from factory.
    AuctionServer := AuctionFactory.CreateInstance('');
  except
    ShowMessage('Failed to connect to server.');
```

### End Listing Four

### Begin Listing Five — CorbaClient package

```

// Title:          Corba Java Client
// Version:        1.0
// Copyright:      Copyright (c) 1999
// Author:         Dennis Butler
// Company:        Inprise Corporation
// Description:    CORBA Client for Delphi Server
package CorbaClient;

import java.util.*;
import java.awt.*;
```

```

import com.sun.java.swing.*;
import borland.jbcl.layout.*;
import java.awt.event.*;
import borland.jbcl.control.*;

public class Frame1 extends DecoratedFrame {
    public static void main(String[] args) {
        Frame1 frame1 = new Frame1();
        frame1.show();
    }

    // CORBA Object Factory and Object Interface.
    CServer.OnlineAuctionFactory pOnlineAuctionFactory;
    CServer.IOnlineAuction pOnlineAuction;

    Double r1Total = new Double(0.00);
    XYLayout xYLayout1 = new XYLayout();
    JLabel jLabel1 = new JLabel();
    JLabel jLabel2 = new JLabel();
    JLabel jLabel3 = new JLabel();
    JLabel jLabel4 = new JLabel();
    JLabel jLabel5 = new JLabel();
    JLabel jLabel6 = new JLabel();
    JLabel jLabel7 = new JLabel();
    JTextField jtfUserName = new JTextField();
    JTextField jtfBidPrice = new JTextField();
    JButton jButton1 = new JButton();

    public Frame1() {
        try {
            // Initialize the ORB.
            System.out.println("Initializing the ORB");
            org.omg.CORBA.ORB orb =
                org.omg.CORBA.ORB.init((String[]) null, null);

            // Bind to OnlineAuctionFactory object.
            System.out.println(
                "Binding to OnlineAuctionFactory object");
            pOnlineAuctionFactory =
                CServer.OnlineAuctionFactoryHelper.bind(
                    orb, "OnlineAuction");

            // Get an instance of OnlineAuction.
            System.out.println(
                "Getting an instance of OnlineAuction");
            pOnlineAuction =
                pOnlineAuctionFactory.CreateInstance(
                    "NewOnlineAuction");
        }
        catch(org.omg.CORBA.SystemException e) {
            System.err.println("System Exception");
            System.err.println(e);
        }

        try {
            jbInit();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        xYLayout1.setHeight(311);
        xYLayout1.setWidth(400);
        jLabel1.setText("Current High Bid Price");
        jLabel4.setForeground(Color.blue);
        jLabel5.setText("< NA >");
        jLabel6.setText("< NA >");
        jButton1.setText("Refresh");
        jButton1.addMouseListener(
            new java.awt.event.MouseAdapter() {
                public void mouseClicked(MouseEvent e) {

```

```

                    jButton1_mouseClicked(e);
                }
            });
        jLabel5.setForeground(Color.blue);
        jLabel6.setText("Auction User Name");
        jLabel7.setText("Bid Price");
        jButton2.setText("Place Bid");
        jButton2.setText("Place Bid");
        jButton2.addMouseListener(
            new java.awt.event.MouseAdapter() {
                public void mouseClicked(MouseEvent e) {
                    jButton2_mouseClicked(e);
                }
            });
        jLabel5.setText("Place New Bid");
        jLabelCurrentBid.setText("< NA >");
        jLabel4.setText("Current Bid Status");
        jLabel3.setText("Current High Bid User Name");
        jLabel2.setText("Current Product");
        this.setLayout(xYLayout1);
        this.add(jLabel1, new XYConstraints(57, 50, -1, -1));
        this.add(jLabel2, new XYConstraints(93, 31, -1, -1));
        this.add(jLabel3, new XYConstraints(21, 68, -1, -1));
        this.add(jLabel4, new XYConstraints(6, 3, -1, -1));
        this.add(jLabelCurrentBid,
            new XYConstraints(207, 51, 183, -1));
        this.add(jLabelCurrentProduct,
            new XYConstraints(207, 31, 182, -1));
        this.add(jLabelCurrentUser,
            new XYConstraints(207, 70, 176, -1));
        this.add(jButton1,
            new XYConstraints(138, 100, 102, 30));
        this.add(jLabel5, new XYConstraints(8, 146, -1, -1));
        this.add(jLabel6, new XYConstraints(72, 182, -1, -1));
        this.add(jLabel7, new XYConstraints(130, 206, -1, -1));
        this.add(jtfUserName,
            new XYConstraints(188, 182, 145, -1));
        this.add(jtfBidPrice,
            new XYConstraints(188, 205, 85, -1));
        this.add(jButton2,
            new XYConstraints(139, 230, 105, 35));
    }

    // Place new bid button.
    void jButton2_mouseClicked(MouseEvent e) {
        Double r1Total = new Double(jtfBidPrice.getText());

        if (pOnlineAuction.PlaceBid(
            r1Total.doubleValue(),
            jtfUserName.getText())==0) {
            Message m = new Message(this, "Sorry",
                "Bid Amount Insufficient");

            m.show();
        }
        else {
            Message m2 = new Message(this, "Success",
                "Bid Successful");

            m2.show();
        }
    }

    // Refresh button.
    void jButton1_mouseClicked(MouseEvent e) {
        Double r1Total =
            new Double(pOnlineAuction.GetCurrentPrice());
        // Update price and customer name information
        // for current product.
        jLabelCurrentBid.setText(r1Total.toString());
        jLabelCurrentUser.setText(
            pOnlineAuction.GetCurrentUser());
        jLabelCurrentProduct.setText(
            pOnlineAuction.Get_ProductName());
    }
}

```

End Listing Five





# UNDOCUMENTED

RTTI / Delphi 5

By Bill Todd



## RTTI Gets Easier

### Delphi 5 Run-time Type Information

**R**un-time Type Information (RTTI) is the information the compiler stores about the published properties of objects in your application. And it is very useful stuff. For example, RTTI is the mechanism the Object Inspector uses to determine the properties an object instance has, their data types, and current values.

RTTI has remained a mystery to many Delphi programmers for two reasons:

- RTTI has never been documented. The only information on RTTI is the source code for the TYPINFO.PAS unit, much of which is in assembler; this is the only reference in Delphi 5.
- The RTTI functions make extensive use of pointers to Pascal records and arrays of records. The Pascal language hides the use of pointers almost completely, so many Pascal/Delphi programmers aren't comfortable working with them.

A new set of functions was added to TYPINFO.PAS in Delphi 5 to make RTTI easier to use. This article will explore the new "easy" RTTI functions, and ways in which you can use them.

The single most useful thing about RTTI is that it gives you access to an object instance's properties without having to know anything about the object. For example, suppose you want to iterate through all the components on a form and change their color. Your first inclination might be to try this:

```
procedure TScanCompForm.ReadOnlyBtnClick(Sender: TObject);
var
  I: Integer;
begin
  { Scan the form's Components property. }
  for I := 0 to Pred(ComponentCount) do
    { If the component has a property named ReadOnly. }
    if IsPublishedProp(Components[I], 'ReadOnly') then
      { Set ReadOnly to True. }
      SetVariantProp(Components[I], 'ReadOnly', True);
end;
```

**Figure 1:** Setting the *ReadOnly* property of all components on a form.

```
for I := 0 to Pred(Form1.ComponentCount) do
  Form1.Components[I].Color := clRed;
```

Unfortunately, this won't work. The data type of the form's *Components* property is *TComponent*, and *TComponent* doesn't have a *Color* property. With some properties you could solve the problem by casting *Form1.Components[I]* to the ancestor class where the property was introduced. This works if every component in which you are interested inherits the property you want to set from a common ancestor. Unfortunately, this is not the case with *Color*.

The solution is to use RTTI to determine if the object has the property and to change the value of the property if it exists. The example in [Figure 1](#) uses RTTI to set the *ReadOnly* property of every component on the form to True. This code is the *OnClick* event handler for the *ReadOnly* button on the Scan Components form of the sample application that accompanies this article (see end of article for download details).

This code iterates through the form's *Components* array. The *IsPublishedProp* function is called for each component to determine if it has the *ReadOnly* property. *IsPublishedProp* takes two parameters: The first is the object instance you want to interrogate; the second is the name of the property. If the object has the property, *IsPublishedProp* returns True; otherwise it returns False.

If the component has the *ReadOnly* property, a call to *SetVariantProp* is used to set its value to True. *SetVariantProp* takes three parameters: The first is the object instance whose property you wish to set; the second is the name of the property; the third is the value.

Function	Description
<i>IsPublishedProp</i>	Returns True if the object has the property.
<i>GetPropInfo</i>	Returns a record that contains all information about the property.
<i>IsStoredProp</i>	Returns True if the property is stored in the DFM file.
<i>PropIsType</i>	Returns True if the specified property is of the specified type.
<i>PropType</i>	Returns the type of the specified property.
<i>GetObjectPropClass</i>	Returns the class of the object referred to by an object reference property.

Figure 2: The RTTI property information functions.

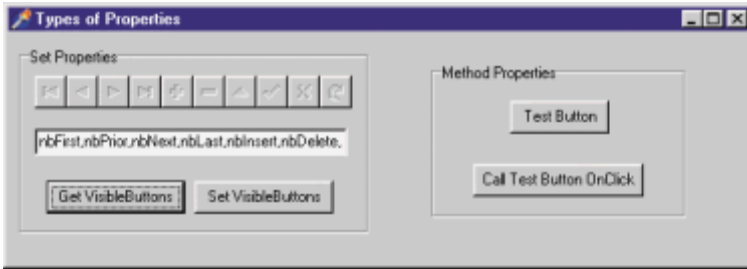


Figure 3: The edit box contains the value of the *VisibleButtons* property.

If you open the TYPINFO.PAS unit in Delphi 5's \Source\VCL folder and scan its **interface** section, you will come to this comment:

```
// Easy access methods.
```

Following this comment are the prototypes for the new RTTI functions added in Delphi 5. These functions can be separated into two categories. The first group provides information about the published members of an object instance. The second group of functions gets and sets the values of properties of varying types. Figure 2 lists the information functions.

The types used by *PropIsType* and *PropType* are not standard Pascal data types. *PropIsType* takes three parameters. The first is the object instance to interrogate, the second is the name of the property, and the third is a value of type *TTypeKind*, which specifies the type of property you are testing. *PropType* takes two parameters, an object reference and the name of a property, and returns a value of type *TTypeKind*. *TTypeKind* is an enumerated type declared in TYPINFO.PAS as:

```
type
  TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration,
    tkFloat, tkString, tkSet, tkClass, tkMethod, tkWChar,
    tkLString, tkWString, tkVariant, tkArray, tkRecord,
    tkInterface, tkInt64, tkDynArray);
```

If you declare a variable named *TKind* of type *TTypeKind*, you can call *PropType* as follows:

```
TKind := PropType(Button1, 'Caption');
```

and *TKind* will be set to *tkString*. If you need the name of the enumerated type as a string, another RTTI function, *GetEnumName*, will do the job. After calling:

```
S := GetEnumName(TypeInfo(TTypeKind), Ord(TKind));
```

*S* will be set to the string *tkString*. This works with any enumerated data type and is the way the Object Inspector loads the enumeration names in its combo boxes.

The inverse function, *GetEnumValue*, is also declared in TYPINFO.PAS so you can convert from the name of an enumeration member to its ordinal value. For example, if *I* is of type *Integer*, then:

```
I := GetEnumValue(TypeInfo(TTypeKind), S);
```

would set *I* to 5 because *tkString* is the sixth member of the enumeration. To assign the ordinal value to a variable of the enumeration type, simply cast it to the enumeration type:

```
TKind := TTypeKind(I);
```

This ability to freely convert between strings and enumerated types means you can use enumerated types internally in your program to make it faster and smaller, allow the use of **case** statements, and convert the enumeration members to strings for output.

The remaining new RTTI functions exist in pairs consisting of a method to get and a method to set the value of

each of the following property types:

- Ordinal
- Enumerated
- Set
- Object
- String
- Float
- Variant
- Method
- Int64

The syntax of all these functions is the same. The “get” functions take two parameters, an object instance and a property name, and return the value of the property. The “set” procedures take three parameters. The first is the object instance, the second is the property name, and the third is the value to assign to the property. There are also two functions, *GetPropValue* and *SetPropValue*, that work with properties of any type compatible with a variant. *GetPropValue* returns the property value as a variant and *SetPropValue* takes a variant as its third parameter, which supplies the value to be assigned to the property.

The use of most of these functions is intuitive, but there are some exceptions. First, although there are many Boolean properties in the VCL, there are no functions to get and set Boolean properties. You can handle Boolean properties using *GetVariantProp* and *SetVariantProp*, as shown in Figure 1. Another alternative is to use *GetOrdinalProp* and *SetOrdinalProp* and use 0 to represent False or -1 to represent True.

*GetSetProp* and *SetSetProp* are also a bit different. When you call *GetSetProp* to return the value of a set property, the returned value is a string. Figure 3 shows a form from the sample application that displays the value returned for the *VisibleButtons* property of a *DBNavigator* component. Note that the value shown in the edit box is a comma-separated list of the members of the set. The *GetSetProp* function also takes a third parameter named *Brackets*. If *Brackets* is

```

procedure TPropTypeForm.CallTestBtnClick(Sender: TObject);
var
    TheMethod: TMethod;
begin
    TheMethod := GetMethodProp(TestBtn, 'OnClick');
    if Assigned(TheMethod.Code) then
        TNotifyEvent(TheMethod)(Sender)
    else
        ShowMessage('No event handler.');
```

Figure 4: Calling an event handler using RTTI.

```

procedure TPropListForm.PropListBtnClick(Sender: TObject);
var
    Props: PPropList;
    I: Integer;
begin
    { Allocate memory to hold the property list
      array of records. }
    Props := AllocMem(SizeOf(Props^));
    try
        { Get the list of properties. }
        GetPropList(PropertyList.ClassInfo,
                    tkProperties, Props);
        { Loop through the list of properties and add the name
          and type of each property to the listbox. }
        I := 0;
        while (Props^[I]<>nil) and (I < High(Props^)) do begin
            PropertyList.Items.Add(Props^[I].Name +
                StringOfChar(' ', 16 - Length(Props^[I].Name)) +
                Props^[I].PropType^.Name);
            Inc(I);
        end;
    finally
        { Free the memory that holds the property list array. }
        FreeMem(Props);
    end; // try
end;
```

Figure 5: Listing all the published properties of an object.

True, the comma-separated list returned by the function will be enclosed in brackets. *SetSetProp* takes the same comma-separated list format for its *Value* parameter, and builds the set value from the comma-delimited list of set member names.

*GetMethodProp* and *SetMethodProp* work with properties whose type is a pointer to a method. All events in VCL components are properties that store method pointers, so event properties are the place to use these two functions.

The form in Figure 3 contains a button labeled **Test Button**, which has an *OnClick* event handler that shows a message to demonstrate that the event handler has been called. The code in Figure 4 is the *OnClick* event handler of the **Call Test Button OnClick** button.

The prototype for *GetMethodProp* shows that it returns a value of type *TMethod*, which is declared in the SysUtils unit as follows:

```

type
    ...
    TMethod = record
        Code, Data: Pointer;
    end;
```

*TMethod* is a Pascal record that contains a method pointer. A method pointer consists of two 32-bit memory addresses. The first, *Code*, contains the address of the code for the method. The

second, *Data*, contains the address of the object instance that contains the data.

Figure 4 shows that this value can be used to call the method by casting it to the appropriate type. In this case, because **Test Button's OnClick** event handler is of type *TNotifyEvent*, the *TMethod* type returned by *GetMethodProp* is cast to *TNotifyEvent*. A *TNotifyEvent* method requires one parameter, *Sender*, of type *TObject*. In this example the *Sender* parameter passed to the event handler in Figure 4 is passed along to the **Test Button's** event handler. Although casting the method pointer and calling the method it points to in a single statement produces this odd-looking code:

```
TNotifyEvent(TheMethod)(Sender)
```

it is a valid Pascal statement. The **if** statement checks the *Code* field of the *TMethod* record to see if it's **nil**. You can test *Code* or *Data*. Both will be **nil** if no event handler is assigned to the *OnClick* event of **Test Button**.

It's unlikely you'll ever need to use the *GetPropInfo* function. However, there is a case where it's useful to understand a little about the *TPropInfo* record it returns. This requires a look at a part of RTTI that wasn't simplified in Delphi 5.

*GetPropInfo* actually returns a value of type *PPropInfo* that is declared as follows:

```

type
    PPropInfo = ^TPropInfo;
```

This declares *PPropInfo* as a pointer to a variable of type *TPropInfo*, which is declared as:

```

type
    ...
    TPropInfo = packed record
        PropType: PTypeInfo;
        GetProc: Pointer;
        SetProc: Pointer;
        StoredProc: Pointer;
        Index: Integer;
        Default: Longint;
        NameIndex: SmallInt;
        Name: ShortString;
    end;
```

There are two members of this record you may find useful. The first is *Name*, which contains the name of the property, and the second is *PropType*. *PropType* is of type *PTypeInfo* and is declared as:

```

PTypeInfo = ^PTypeInfo;
PTypeInfo = ^TTypeInfo;
TTypeInfo = record
    Kind: TTypeKind;
    Name: ShortString;
end;
```

*PTypeInfo* is a pointer to type *PTypeInfo*. *PTypeInfo*, in turn, is a pointer to type *TTypeInfo*, which is a record with two members, *Kind* and *Name*. *Kind* is of type *TTypeKind*, an enumerated type described earlier. *Name* is the type name as a string. Understanding these types is necessary if you want to scan all the properties of an object and determine their name and type.



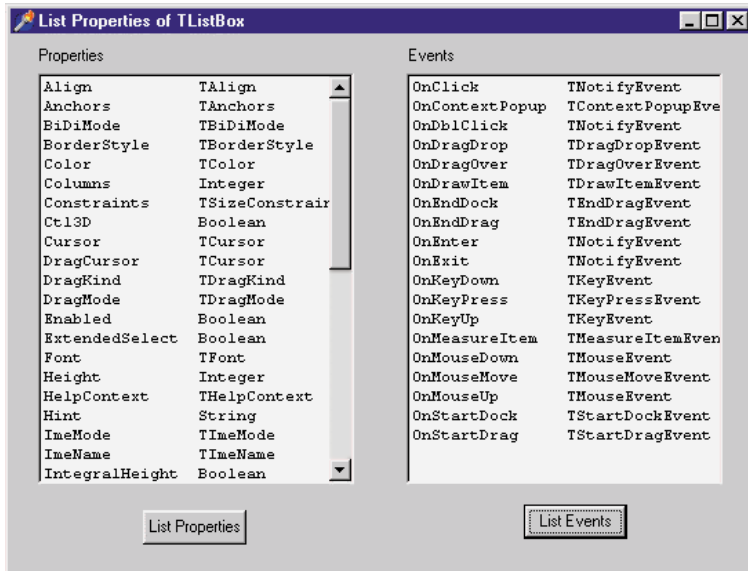


Figure 6: The List Properties form.

You can get information about all the properties of an object by calling the RTTI function *GetPropList*. *GetPropList* takes three parameters. The first is the pointer to the RTTI information returned by calling the object's *ClassInfo* method, and the second is a set parameter of type *TTypeKinds*. *TTypeKinds* is declared as a set of *TTypeKind*, the enumerated type described earlier in this article. The members of this set determine the types of properties you want returned. The third parameter is *PropList* of type *PPropList*, where *PPropList* is declared as follows:

```
PPropList = ^TPropList;
TPropList = array[0..16379] of PPropInfo;
```

Therefore, *PPropList* is a pointer to type *TPropList* and *TPropList* is a 16,380-member array of type *PPropInfo*. Figure 5 shows the *OnClick* event handler of the *List Properties* button on the *PropListForm* in the sample application.

Since the property list array is quite large, it's dynamically allocated at the beginning of the procedure and freed at the end. The first statement in Figure 5 calls *AllocMem* to allocate a block of memory for the array. Note that the amount of memory requested is *SizeOf(Props^)* where *Props* is of type *PPropList*. This can be read as, "size of whatever it is that the pointer *Props* points to." Since *Props* is of type *PPropList* and *PPropList* is a pointer to *TPropList*, *SizeOf(Props^)* is equivalent to *SizeOf(TPropList)*.

The next statement is the call to *GetPropList*. This function gets the list of properties for a listbox on the form named *PropertyList*, so the first parameter is the *ClassInfo* method of *PropertyList*. The second parameter, *tkProperties*, is a set declared in *TYPINFO.PAS* that includes all property kinds except *tkMethod* and *tkUnknown*. By using *tkProperties* we will get all the properties except the event properties. The third parameter, *Props*, is the pointer to the *TPropList* array.

The *while* loop iterates through the array and adds the name and type of each property to the listbox. The *while* loop runs until an element of the array is found whose value is *nil*, or until the highest element in the array is reached. The test for a *nil* element

works because the memory for the array was allocated by calling *AllocMem*. *AllocMem* not only allocates the requested block of memory, but also initializes each byte to *nil*.

The call to *PropertyList.Items.Add* adds the name of the property and the property type to the listbox. The name is referenced by *Props^[I].Name*. When working with pointers, the trailing caret can be read as "points to," so this notation identifies the name field of element *I* of the array to which *Props* points.

The reference to the name of the property's type, *Props^[I].PropType^.Name*, can be read as "the Name field in the record pointed to by the *PropType* field in element *I* of the array pointed to by *Props*." (For more information about using pointers, see the *Delphi Language Reference*.)

Figure 6 shows the *List Properties* form from the sample application with the properties of the listbox displayed in the left listbox and the events in the right

listbox. The code for the *List Events* button is identical to the code in Figure 5 except that *tkMethod* is used for the second parameter in the call to *GetPropList*.

### Conclusion

RTTI is a vital tool if you want to write reusable code. Using RTTI it's easy to write a procedure that takes a data module as its parameter and will set the *ReadOnly* property of all of the datasets in the data module to *True*. You can also write a security system that allows different rights to be defined for different components on a form or data module. When your application creates a form, it can call a routine that takes the form as a parameter, looks up the user's rights for the components on that form, and sets the component's *ReadOnly* or *Visible* property based on the user's rights. The uses of RTTI are endless.  $\Delta$

The files referenced in this article are available on the *Delphi Informant Works CD* located in *INFORM99\NOV\DI9911BT*.

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





# DB NAVIGATOR

Delphi 5 / Database Design

By Cary Jensen, Ph.D.



## The Data Module Designer

### Delphi 5's Gift to Database Programmers

**A** Delphi data module is a container used to share components among multiple forms. The most common type of component to share is a data set, allowing you to define database-related properties, such as constraints and event handlers, in a single location. This simplifies your application's design and maintenance. When database changes need to be made, often only the data module needs to be updated. The changes, however, affect all forms that use the data module.

Until recently, such component sharing was the primary reason for using data modules. With the release of Delphi 5, however, they're far more valuable. This is because Delphi 5 provides you with the Data Module Designer. In addition to the sharing already described, the Data Module Designer adds three new capabilities: two different visual representations of the relationships between your components, partial automation of property definitions, and the ability to comment and print your data module design for documentation purposes.

#### Overview

The Data Module Designer, shown in Figure 1, is a replacement for the form-like designer displayed

by previous versions of Delphi. The new designer is a two-pane, non-modal window. The left pane displays the Tree view, which depicts the various components that have been placed into the data module. The right pane contains a tabbed interface, providing access to two additional views of the data module's components: the Components page and the Data Diagram.

#### The Tree View

By default, the initial view of the Data Module Designer includes the Tree view and the Components page. The contents of the Tree view are similar to that displayed by the Code Explorer in Delphi 4 and 5. There are, however, some important differences. First, the Tree view is displayed while the designer is active, while the Code Explorer is associated with the Code Editor.

The second difference is that the Tree view understands the relationship between data access components. This information is used to define the contents of the various nodes of the tree. For example, the Tree view knows that each data set component is associated with a single database. Consequently, a given data set will appear as a node under its associated database. Furthermore, because databases themselves can be associated with one — and only one — session, the database node appears under the appropriate session node.

There is a third difference between the Tree view and the Code Explorer. This difference is derived in part from the fact that the Tree view is associated with a designer and not the Code Editor.

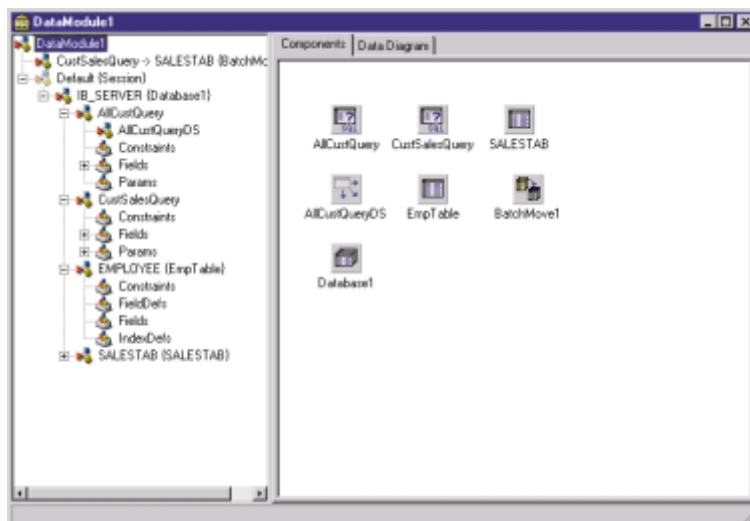


Figure 1: Delphi 5's new Data Module Designer.

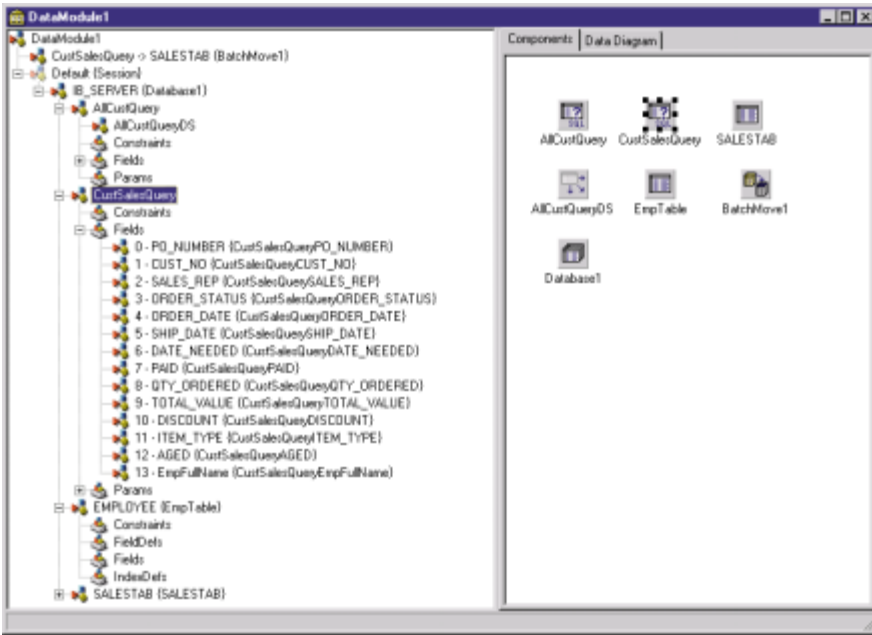


Figure 2: The expanded Fields node for a query.

Using the Tree view to configure properties isn't limited solely to dropping new components. Existing components can be dragged from one node to another, thereby causing their properties to be updated. For example, imagine that you have two database components on your data module: one whose *DatabaseName* is IBSEVER and which connects to an InterBase server, and another whose *DatabaseName* is LOCALDATA and which is used to point to a local directory. If you now place a new Query component into the LOCALDATA node, but meant to place it into the IBSEVER node, you can simply drag the query from LOCALDATA to IBSEVER, which causes the *DatabaseName* property of the query to change from the old value to that of the new node. Likewise, the *SessionName* property will also be updated, but only if the two database components use different sessions (which is unlikely).

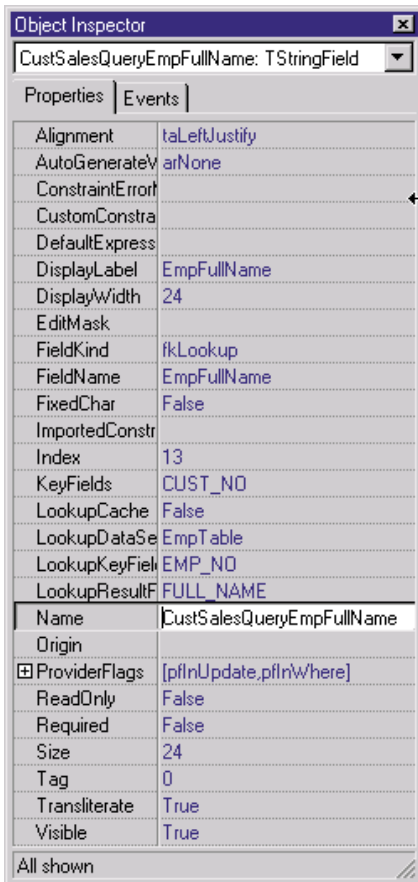


Figure 3: The Object Inspector displaying the properties of a TField selected in the Tree view.

Specifically, the Tree view is a drop site, meaning you can drop components directly into the Tree view. Components dropped into the Tree view appear in the Components page automatically.

The capability to drop components into the Tree view is more than a convenience. It's an extremely useful technique because it permits the Data Module Designer to set basic properties of the components you drop. These properties are identified based on the node onto which you drop the component. For example, if you drop a new data set onto the node for a database, the *Database* property of the data set will be automatically set to that database, and the *SessionName* property will be set to that database's session. This feature alone will not only speed the configuration of your data access components, but will also reduce the number of configuration errors.

base, the *Database* property of the data set will be automatically set to that database, and the *SessionName* property will be set to that database's session. This feature alone will not only speed the configuration of your data access components, but will also reduce the number of configuration errors.

### Data Set Nodes

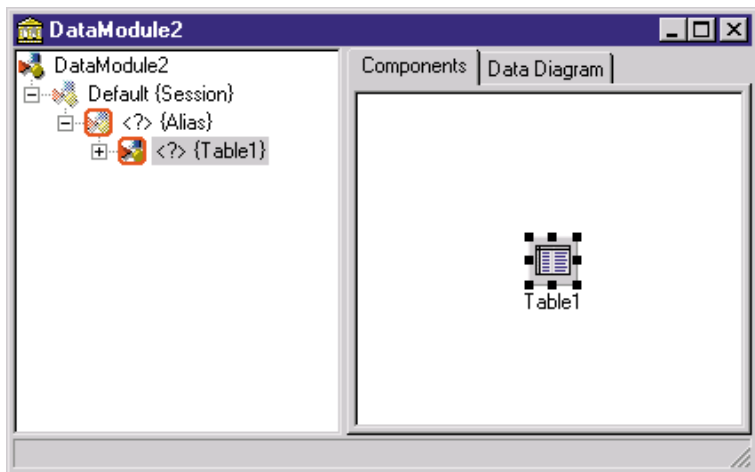
Data set nodes in the Tree view also have subnodes. Nodes for Table components have subnodes for FieldDef, TField, and IndexDef definitions. ClientDataSet nodes include all Table nodes plus nodes for Aggregates and Params. Nodes for Query and StoredProc components have TField and Params nodes. All data sets have a Constraints node.

When the appropriate property of a specific data set is defined, the corresponding node can be expanded. The expanded property node provides you with direct access to the associated object within the Object Inspector. For example, in Figure 2, the Fields node of the query named CustSalesQuery has been expanded. The expanded node provides you with access to each TField object created for this query using the Fields Editor. If you select one of these TField objects, its properties are displayed in the Object Inspector. Figure 3 shows the Object Inspector when the EmpFullName field has been selected. This field is a lookup field.

You can also right-click a data set's subnodes in the Tree view to display a limited popup menu. This menu permits you to do basic things with the selected node, such as add an item. Or, if the right-clicked item is a leaf, such as a specific Constraint, the menu allows you to easily remove the item.

The glyphs used by the nodes in the Tree view can convey information about the associated object. For example, a component whose definition is incomplete is enclosed by a red circle. This is shown in Figure 4, which is how a Tree view looks after a single Table has been dropped onto it, but before any properties, such as *DatabaseName* or *TableName* have been set.

In some cases, a glyph appears "grayed out." Although IDE users often associate this state with a disabled object, in the Tree view it means that a default object is being used. For example, if you're using the default session, the session node's name will be "Default," and its glyph will appear grayed out. You'll see this effect for both the session and the database (Alias) in Figure 4. However, if you refer back to Figure 1 or 2, you'll see that only the default session is used, in which case the glyph for the session is grayed out, but the glyph for the database is not.



**Figure 4:** A red circle around a node indicates the object's configuration is not complete.

### The Components Page

The Components page is displayed in the right pane of the Data Module Designer when the Components tab is selected. This default view provides the same capabilities available to data modules in previous versions of Delphi. Specifically, you can drop new components into the Components page. Alternatively, double-clicking the icon for a component in the Component palette causes an instance of that component to be placed in the center of the Components page.

Like the previous designer, the Components page also permits you to right-click a component. Doing so displays that component's popup menu, which will include any defined component editors. Finally, the Components page permits you to select one or more components. You can then use the Object Inspector to change the value of properties for the selected component(s).

You will normally find it more efficient to drop your components into the Tree view rather than the Components page.

However, components dropped into the Tree view will be positioned in the center of the Component Tree (the same position as components that are placed when you double-click in the Component palette), so the most common use of the Components page is to reorganize the position of components, providing a more ordered display for components dropped into the Tree view.

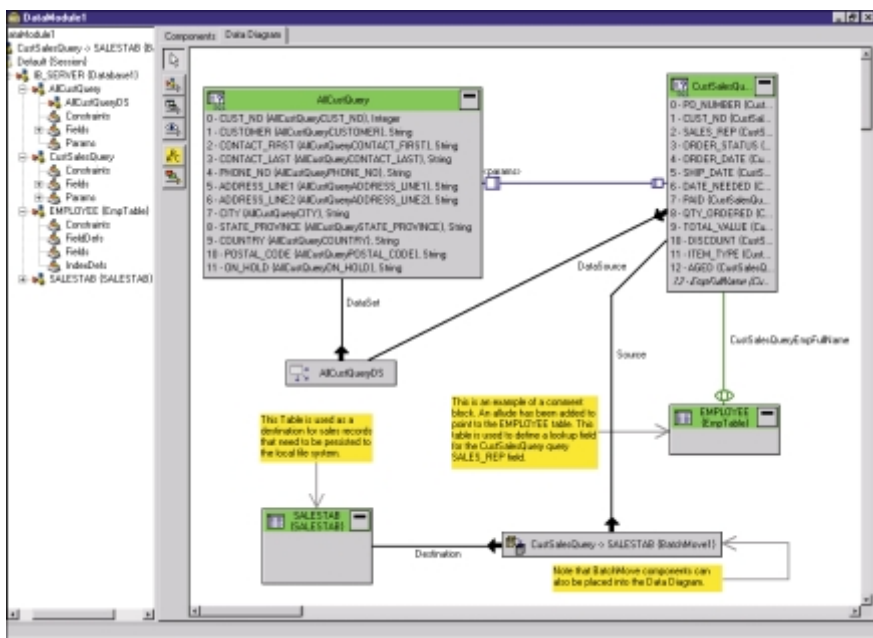
### The Data Diagram

While the Tree view provides a tremendous amount of functionality, as far as configuring data access components is concerned, it's the Data Diagram that is likely to generate the most interest from the majority of Delphi developers. The Data Diagram provides you with two complementary capabilities. First, it permits you to visually link associated data access components. For example, if you have a master-detail relationship, the association between the two data sets can be created by selecting the Master-Detail tool from the Data

Diagram palette, and then dragging a line between the detail data set and the master data set's data source.

The second capability provided by the Data Diagram is that it provides you with the option of creating a visual representation of the relationship between data access components. This visual representation can either be used to select objects in order to configure them, or it can be printed as documentation. **Figure 5** shows a Data Diagram for a data module. (The project shown in **Figure 5**, named Frames, is available for download; see end of article for details.)

As mentioned, the Data Diagram doesn't necessarily contain all the components you've placed on your data module. For example, you may want to place all data sets and data sources on the Data Diagram, but leave out database, session, dialog box, and other components. Data diagrams can quickly get very complicated. In most cases, placing only the essential components in the data diagram keeps it from getting cluttered with unnecessary information.



**Figure 5:** A Data Diagram containing a variety of data access components and comments.

You place a component in the Data Diagram by dragging its glyph from the Tree view and dropping it on the Data Diagram page of the data module. If the component you place into the data diagram has already been associated via one or more properties, with a component already in the data diagram, a line is drawn connecting the two components. If you have not yet associated a component with another component already in the data diagram, you can use the Object Inspector or the Data Diagram toolbar to create the association.

The color and shape of a line that connects two components indicates the type of association. For example, when one object references another object, this is represented by a solid black line anchored by an arrow. This arrow indicates the direction of the association. You can see this type of associ-



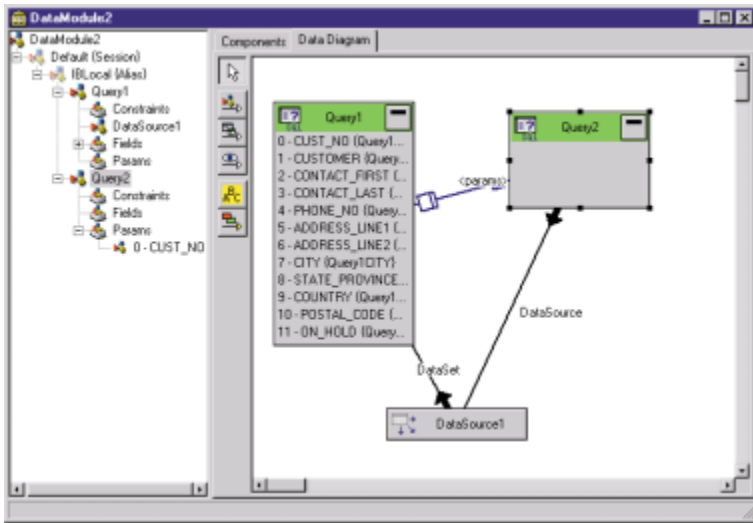


Figure 6: A simple diagram.

the simple diagram shown in Figure 6. This diagram contains only two relationships, one between a data source, and one between two queries (master-detail). (Note as well the labels that name the relationships being depicted.)

There are two types of changes you can make that will affect the reference lines that appear. The first is that you can move a component to a new location. At initial design, you will do this repeatedly, placing components in relative proximity to the components they reference. Moving a component produces an automatic re-draw of its lines.

The second change involves manipulating the connecting lines directly. By clicking your mouse somewhere on a reference line, you automatically add a new point to that line. That point can then be dragged to a new location. Although this creates a more complicated polyline (a line with multiple points), you can drag two or more of these points so the resulting line doesn't obscure or intercept other lines.

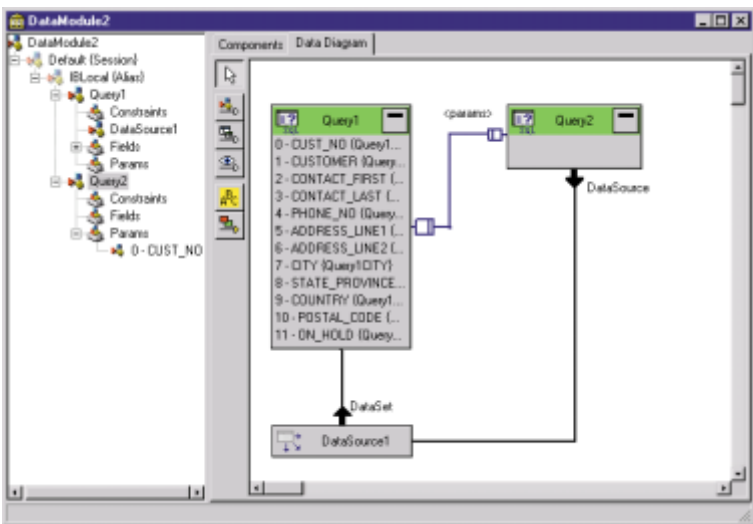


Figure 7: You can produce cleaner diagrams by adjusting line points and labels.

The labels can also be moved. To change the location of a reference label, click the label to select it, then drag it to a new location. Consider Figure 7. This is the same diagram as shown in Figure 6. However, some of the components have been moved, and both the lines and the labels have been modified. The lines have had points added, and the points have been moved to produce nice, clean lines. Likewise, the reference labels have been moved so they continue to label the line they reference.

Sometimes you might find that you want to remove a relationship between two objects. To do this, right-click the reference line and select **Remove relationship** from the displayed popup menu. The line connecting the two objects will be deleted, and the appropriate properties will be reset.



Figure 8: The Data Diagram Toolbar.

ation in Figure 5, where the data source named AllCustQueryDS references the query AllCustQuery. The name of the property, which is *DataSet* in this case, appears alongside the referencing line.

Other types of associations include lookups (a green line with an "eye" glyph), master-detail (a dark blue line with "table" glyphs), and alludes (a gray line with a terminal arrow). Alludes are only used to associate a comment block with an object. Lookup and master-detail associations are used to identify lookup field references and master-detail data set references, respectively.

As mentioned earlier, a line gets drawn to represent a relationship that is defined either by configuring properties, or by using the Data Diagram toolbar. Initially, this line is drawn pretty much directly from one object to another. However, after defining only a few such relationships, the lines can quickly form a tangled mess. Fortunately, the Data Diagram Designer permits you to easily change these lines. For example, consider

### The Data Diagram Toolbar

The Data Diagram toolbar, shown in Figure 8, contains several toolbar buttons that you use to customize and configure the data diagram. The first button is the Selection tool. You use this tool to deselect one of the other tool buttons.

The second button is the Property tool. You use it to define a property relationship between two objects. To use this tool, first select the Property button. Next, begin a drag operation from an object with a property (such as *DataSource* for a Query), and drag and then release the mouse when your pointer is positioned over an object of the appropriate property type (a *TDataSource* instance in this case).

The third button is the Master-Detail tool. Use this tool to drag a line from a detail table (or query) to the master table (or query) data source.

The fourth button is the Lookup tool. Use this tool to drag a line from one data set (the one you want to define a lookup field for) to another data set (the one that contains the field you want to look up). Complete the displayed New Lookup Field dialog box to create the lookup field.



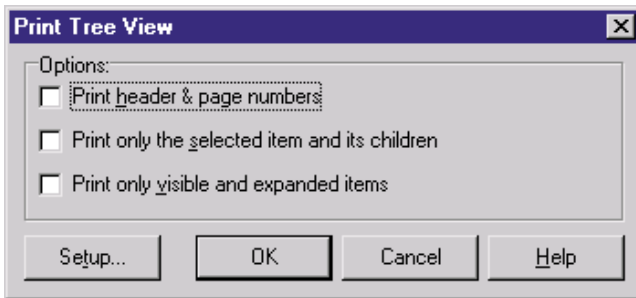


Figure 9: The Print Tree View dialog box.

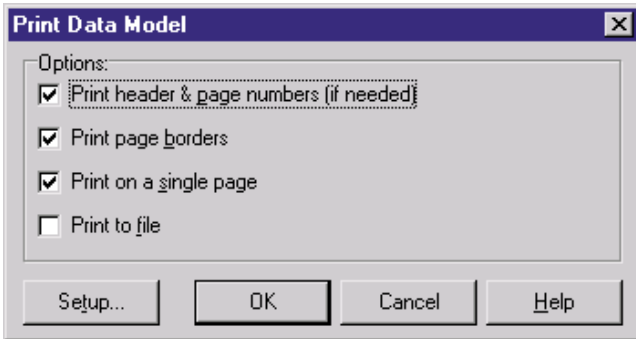


Figure 10: The Print Data Model dialog box.

The last two buttons, the Comment Block and Comment Allude tools, are used to create comments and then point them to one or more objects on the data diagram. To place a comment block, select the Comment Block tool, then drag a rectangle in the data diagram. Next, click the comment block until a cursor appears. After typing your comment, click on any other object to complete your comment. Pressing **[Esc]** while entering a comment deletes any changes.

If your comment block text describes a specific object, you'll generally want to add a comment allude to connect the comment with the object(s) it references. To place a comment allude, click the Comment Allude tool, then drag a line from the comment block to the object you want to allude. Repeat this process if you want to allude the comment block to additional objects.

## Other Features of the Data Diagram

In addition to dropping data access components into the Data Diagram, this feature provides you with convenient access to a number of tools. To access these tools, right-click one of the objects in the Data Diagram to display a popup menu. The menu contains features available for that object. For example, if you right-click a Table in the Data Diagram, you can invoke the Fields Editor, SQL Builder, etc. Likewise, right-clicking a Database gives you ready access to the Database Editor.

The right-click menu also permits you to remove an object from the diagram. To remove an item from the diagram, but not from the data module, right-click the object and select **Remove from diagram** from the displayed popup menu.

The Data Diagram also gives you a fair amount of control over the display characteristics of the objects that appear within it. For example, you can control the color of data set headers, the color and type of lines, and the color and style of fonts used for labels. Adjusting these visual features can be useful when you

want to adopt a consistent and meaningful pattern for identifying objects. For example, you may want to make all lookup tables use one color of header. To adjust the visual characteristics of an object, right-click it and select from its displayed popup menu. Data access components, reference lines, and labels can all be adjusted in this same manner.

## A Documentation Asset

The new Data Module Designer also provides you with a new tool for documenting your applications. The Tree view and the Data Diagram can both be printed. Printing the Tree view permits you to document some or all components on the data module. To print the Tree view, right-click in the Tree view area and select **Print**. The Tree view responds by displaying the Print Tree View dialog box shown in Figure 9. Notice that this dialog box permits you to print either the entire Tree view, or only the select component and its child nodes.

Printing the Data Diagram permits you to document the relationships you've defined visually (typically a subset of the components appearing in the Tree view). To print the Data Diagram, right-click anywhere within the Data Diagram and select **Print** from the displayed popup menu. This displays the Print Data Model dialog box, shown in Figure 10. Select how you want the diagram to be printed and click **OK**.

One final note about the Data Diagram is in order. In most cases, you'll want to limit the size of your data diagram. Large data diagrams can easily become congested, reducing their usefulness. For this reason, you might find it necessary to use more data modules than in previous versions of Delphi, keeping each data module relatively simple.

## Conclusion

Delphi 5's Data Module Designer provides you with a wealth of new features to make the configuration of your data access components more efficient. It also provides you with several new means of documenting your data access components. Together, these capabilities of the Data Module Designer serve as one of many compelling reasons to upgrade to Delphi 5. **Δ**

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM99\NOV\DI9911CJ.*

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



# SOUND + VISION

TAPI / Wave API / Delphi 4, 5

By Robert Keith Elias and Alan C. Moore, Ph.D.



## Extending TAPI

### Playing and Recording Sounds during Telephony Calls

Last summer, *Delphi Informant* presented a three-part series of articles demonstrating much of the basic functionality provided by TAPI, the Windows Telephony API. In this article, we'll demonstrate how to build a program in Delphi 4 using TAPI and the Multimedia API to play and record telephone messages to and from files. We'll also show how to capture digital-tone key presses after a call has been placed or received.

As a foundation, the example program we'll develop here uses the work that Major Ken Kyler and Dr Alan Moore developed in the series of articles last summer. We'll begin by briefly reviewing essential TAPI concepts. We'll then provide an overview of the TAPI and Multimedia API functions that we'll use to implement this new functionality.

After that review, we'll provide a detailed explanation of the code needed to play .WAV files to, and record them from, a phone line, and capture digital-tone key presses. We'll conclude by providing important information you need to know: additional software you'll need to use this functionality, some of the limitations and failings of TAPI, and where you can find more information.

#### TAPI Basics

As in the previous series of articles, we'll make extensive use of the TAPI.pas conversion produced by Project JEDI (<http://www.delphi-jedi.org>). We'll also use the Multimedia APIs (mainly the Wave API) included in mmsystem.pas. One could easily be intimidated by either of these large collections of structures and functions. Fortunately, we can ignore most of TAPI's 125 functions, and most of those in mmsystem.pas, and still accomplish quite a lot.

However, we do need to know what we're doing. First we need to understand the difference between lines, phones, calls, addresses, and IDs.

**Lines.** A line device generally refers to a modem or a similar piece of hardware connected to a telephone line. TAPI doesn't communicate directly with a line/modem. Instead it uses a TSP (Telephone Service Provider). A TSP is just a fancy

name for a driver, written by the modem/equipment supplier, to communicate with TAPI.

Many TAPI functions are designed to talk to TSPs that communicate with sophisticated telephone equipment. Such TSPs are needed for large offices where, for example, dozens of calls may arrive at nearly the same time on a single line. People who work with such systems must be able to manage call conferencing and call transferring, among other tasks. Unfortunately, the TAPI documentation rarely states the context within which a function is intended to operate. As we'll discuss later, TAPI has certain limitations along with the functionality we'll be using here.

**Phones.** Another TAPI device is the phone. You might assume that "phone" is synonymous with "handset." That would be a mistake. In TAPI, the phone is the speaker and microphone attached to your computer. You use phone devices to redirect caller output to the speaker and to redirect input from the user (via the microphone) to the telephone line. If you're not interested in using the speakers, you can ignore those functions that take the form phoneXxxxx.

**Calls.** The key event in the TAPI universe is the call. It begins at the precise moment when Windows sends a message to your application telling you it has picked up the line. This is done through a callback routine. Callback routines are used throughout the Windows API to provide a means for Windows to send information back to a calling application. Our application uses three callback routines: one for playing sounds, one for recording sounds, and the one used by TAPI.

Most calls are initiated with the *lineAnswer* or *lineMakeCall* function. These and similar functions are asynchronous. This means that when the function is called successfully, it immediately returns with a positive number (1, 2, etc.). However, the function isn't really complete until a confirming message is sent back via the callback routine with the same positive number in the *dwParam1* argument, and zero in the *dwParam2* argument.

**Addresses and IDs.** An address consists of a string of characters (letters, digits, and control characters) that provide a path to a phone or other device. That other device could be a modem or a computer. While addresses are often just phone numbers, they can also provide a path to a network or Internet address.

IDs are simply handles to devices. In the case of TAPI, we're usually most interested in the handle to the logical line we get by calling the *lineGetID* function. As complex as TAPI can be, at least we don't have to work directly with the COM port.

## The TAPI and Wave API Link

Before we investigate the multimedia API, we need to investigate the link between TAPI and the Multimedia API, particularly the Wave API. That link is tenuous. Despite its complexity, no existing version of TAPI includes routines for playing or recording .WAV or other sound files. The next Windows NT incarnation, Windows 2000, may include further sound playing and recording functionality within TAPI. At least we've heard such rumors. Now, however, TAPI simply provides a handle, called a device ID, toward which an application can direct the input and output of .WAV files using Wave API functions. We use TAPI's *lineGetID* function to get this device ID.

## The Wave API

In the March, 1999 issue of *Delphi Informant*, Dr Moore provided an overview of the multimedia APIs in his "File | New" column. We won't repeat that here but, rather, point out that the Wave API is a sub-API, consisting mainly of functions and related structures that begin with the prefix "Wave." To work with this API, we need to understand the structure of .WAV files.

Most sounds played in the Windows environment are produced by .WAV files, of which there are various types. For most of this discussion, we'll consider only simple uncompressed PCM (Pulse Code Modulation) .WAV files. There are at least 30 compressed formats, which we'll briefly discuss later. A simplified format of a complete PCM .WAV file is shown in **Figure 1**. .WAV files are actually a subcategory of RIFF (Resource Interchange File Format) files.

In **Figure 1**, you'll notice that the first four characters are RIFF, which identifies them as a RIFF file.

All RIFF files consist of chunks — specially structured groups of data that often contain subchunks. The first chunk in a RIFF file is called the RIFF chunk. In the case of Waveform audio RIFF files, the RIFF chunk contains two subchunks: the *fmt* chunk, which provides information about a Wave form's structure, and the data chunk, which contains the audio data itself. In **Figure 1**, *xxxx* represents DWORD/LongWord size values, and the four periods represent data in the files. Identifiers, such as RIFF and Wave appear in the file exactly as shown.

The fact chunk is optional in a simple PCM .WAV file. As shown, every .WAV file of this type begins with a header section that is typically about 100 bytes long. The chunks

Segment	Explanation
RIFFxxxx	RIFF identifier; xxxx is the size of the file minus 8 bytes.
Wavefmtxxxx	Wave identifier; <i>fmt</i> subchunk beginning; size of format section (about 50 bytes).
.....	The format section (about 50 bytes).
.....	Optional extended format information.
factxxxx	xxxx usually equals 4 bytes (number of bytes of fact data).
....	Total number of samples in the file.
dataxxxx	Size of the Wave audio data.
.....	The Wave audio data (most of the file).
.....	Optional ID garbage at the end of the file.

**Figure 1:** The structure of a Wave RIFF file.

used in the specific .WAV file type are RIFF, *fmt*, *fact*, and *data*. Other possible chunk types in multimedia files include *cue* and *playlist*. Except for RIFF and *data*, these chunks can be in any order.

Why do we need to know this? If we want to record or play a .WAV file using the low-level multimedia input/output functions, we'll need to correctly write or read the elements of the header file. The *xxxx* parts simply indicate the size of their respective chunks in numbers of bytes, making it possible to write or read the precise number of bytes in our audio data. Once we've correctly filled in the header chunks, we just feed them to the multimedia input/output functions, and the data is automatically written to, or read from, memory. We'll explore the details when we describe the code.

Let's take a closer look at some of the chunks, particularly the *fmt* and *fact* chunks. The *fact* chunk can be calculated from the *fmt* chunk, so we'll deal with that first. **Figure 2** shows the name, type, and use of each field in the *fmt* chunk.

If you've worked with electronic sound, you've probably encountered the word "sample." A sample contains an elemental unit of sound. The sample size is given in *wBitsPerSample*. For uncompressed (PCM) data, the size is usually 8, 16, 24, or 32 bits wide. The wider the size, the higher quality the sound. Because we're using a telephone line, we can use low-quality resolution. Therefore, we'll work with either 8 or 16 bits in *wBitsPerSample*. When using a compression algorithm, this value is usually smaller.

Element Name	Data Type	Represents
<i>wFormatTag</i>	Word	The format of data in a .WAV file.
<i>nChannels</i>	Word	The number of channels (i.e. mono [1], stereo [2], etc.).
<i>nSamplesPerSec</i>	DWord	The number of samples written (or read) per second.
<i>nAvgBytesPerSec</i>	DWord	The number of bytes written (or read) per second.
<i>nBlockAlign</i>	Word	The minimum size (in bytes) of a readable data block.
<i>wBitsPerSample</i>	Word	The number of bits per sample of mono data.
<i>cbSize</i>	Word	The size in bytes of extended compression info.

**Figure 2:** Elements of the *fmt* chunk.

The sample rate, which indicates how many samples are fed to the speaker each second, is given in *nSamplesPerSec*. This is typically described in Hertz (Hz); thus 8,000 Hz is 8,000 *nSamplesPerSecond*. Common audio sampling rates are 8,000, 11,025, 22,050, and 44,100 Hz. Again, because analog telephone lines function at 3,100 Hz, nothing above 8,000 Hz will produce noticeably improved performance.

Once we know *wBitsPerSample* and *nSamplesPerSec*, *nAvgBytesPerSec* and *nBlockAlign* are easy to calculate (at least for a simple PCM file):

$$nAvgBytesPerSec = nSamplesPerSec * wBitsPerSample / 8$$

$$nBlockAlign = nChannels * wBitsPerSample / 8$$

The only meaningful piece of information in the fact chunk is *dwFileSize*, a count of the samples in the .WAV file. It's easy to calculate from the above information:

$$dwFileSize = (\text{Total play time in seconds}) * nSamplesPerSec$$

Figure 3 shows actual numbers for two identical files of 10.286 seconds each using different Wave formats: the first an uncompressed PCM file, and the second a compressed IMA ADPCM file. The PCM file is 164,696 bytes, and the ADPCM file is 41,788 bytes. (The same file compressed using DSP Group TrueSpeech is 11,066 bytes.)

The first thing you'll notice is that *nAvgBytesPerSec* and *nBlockAlign* can't be calculated from the formulas we've given. In fact, we were unable to determine how to calculate these values from the published documentation.

It gets even more interesting. According to the documentation that comes with Unimodem/V (the Microsoft TSP for voice modems), the supported formats are: IMA ADPCM at 4,800 Hz, 7,200 Hz, or 8,000 Hz; and Rockwell ADPCM. However, this was not confirmed by our experience. Even though the IMA ADPCM is explicitly said to be supported, we were able to record (input from the modem) only in the uncompressed PCM format and only at 8,000 Hz with *wBitsPerSample* set to 16.

We should point out that PCM is very costly in memory and disk usage. In principle, it's possible to convert PCM files to one of the compressed formats; however, it would first be necessary to convert the *msacm.h* header file into a Pascal format (another task for Project JEDI, or an ambitious developer).

On the other hand, we were able to play (output to the modem) any .WAV file for which we had an installed CODEC, provided that

Field	PCM file	IMA ADPCM file
<i>fmt</i>	18	20
<i>wFormatTag</i>	Wave_FORMAT_PCM (1)	Wave_FORMAT_IMA_ADPCM (17)
<i>nChannels</i>	1	1
<i>nSamplesPerSec</i>	8000	8000
<i>nAvgBytesPerSec</i>	16000	4055
<i>nBlockAlign</i>	2	256
<i>wBitsPerSample</i>	16	4
<i>cbSize</i>	0	2
<i>fact</i>	4	4
<i>dwFileSize</i>	82294	82294
Actual file size	164,696 bytes	41,788 bytes

Figure 3: Comparison of PCM and IMA ADPCM files.

*nSamplesPerSec* was 8 kHz. For example, we were able to play the *Female Operator.wav* file that comes with some versions of Windows 95. This file is encoded at 8,000 Hz using the TrueSpeech compression format. IMA ADPCM and TrueSpeech are two of the five audio compression formats that come with Windows 95. To see the codecs currently installed on your system, look in the Windows registry at:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\MediaResources\acm
```

or in Control Panel via multimedia/Advanced/Audio Compression Codecs.

## Writing the Code

Having discussed the basic issues, we're almost ready to delve into the code. To test the code, you'll need a modem that can handle voice as well as data (most of the newer ones do), Unimodem/V, and TAPI 1.4 or TAPI2.x at a minimum. Note that Unimodem/V comes with Windows 98. You can perform the following test to determine whether Unimodem/V has been successfully installed on your machine. Look under Multimedia in Control Panel and ensure you see the following two lines:

- Voice Modem Wave #00 Line
- Voice Modem Wave #00 Handset

You may also want to study Microsoft's TAPI documentation, and the WaveForm documentation. As we mentioned earlier, the TAPI code for our application is based on code first published in *Delphi Informant* in the summer of 1998. We took the liberty of making certain changes to that code, and added quite a bit of new code.

First we changed the variable names to match those in the Microsoft API documentation, which makes the code easier to follow. We appended a record instance prefix — *rTAPI* — to these names to make it easy to distinguish these variables from those used for playing sounds. The latter have a prefix of either *rWvR* (record Wave) and *rWvP* (play Wave). See *Glbvars.pas* for the full declaration of these structures (available for download; see end of article for details). We've also included many comments in the code.

We've included routines for dialing out and answering calls. Being able to record conversations after dialing out may seem useless. However, it eliminates the need for someone to call you to test the play and record functions if you only have one telephone line. Because the modem cannot distinguish between digit tones generated by the local handset and a remote handset, you could implement code to play or record messages by pressing the keypad on either a remote or the local telephone.

However, the present version doesn't include this functionality. Now let's investigate the code for playing a sound.

## Using Wave API to Play a Sound

The *waveOutWrite* function plays Wave data to the phone line through the modem. However, the task is hardly trivial. Of course we could use the *PlaySound* or *sndPlaySound* functions to play .WAV files, but they don't work in this context because they always play to the speaker by default. The *waveOutWrite* function provides the format translation and redirection capabilities we need. Thus, the modem will receive the sound data.



Nine steps are needed to play a .WAV file to a modem line. We have numbered these steps in the code in the `playWav.pas` file (this file is available for download; see end of article for details):

- 1) Get the COM port handle where output will be sent.
- 2) Open the .WAV file.
- 3) Locate the format information in the .WAV file's header and copy that information into a WAVEFORMATEX structure.
- 4) Prepare and lock a memory buffer for the Wave data.
- 5) Copy the Wave data into a memory buffer.
- 6) Prepare a WAVEHDR structure to describe the memory buffer in which the Wave data is held.
- 7) Load the Wave conversion drivers and prepare a callback routine.
- 8) Play the .WAV file.
- 9) Clean up everything.

Let's discuss the details. The first step, getting the COM port handle, will be familiar to readers who followed the earlier series of articles. For this step, we use the TAPI function, `lineGetID`. If you want to send a message to the line, use this function with a format like:

```
lineGetID(,,, 'wave/out');
```

If you want to record from the line, you use:

```
lineGetID(,,, 'wave/in');
```

The Wave API will use this handle (a number like 0, 1, 2, etc.) to find the modem port to which Wave data will be sent. This information is sent in a predefined TAPI structure called a VARSTRING via `lineGetID` (see [Figure 4](#)).

In the second step, open the .WAV file. This time we'll use the `mmioOpen` function, one of the multimedia input/output functions. This special-purpose function is especially useful for accessing multimedia data. With the file open, we need to locate the format information in the header. For this we'll use the `mmioDescend` function twice, followed by `GetMem`. The latter allocates memory for the `rWvPlpWaveFormat` field based on the size of the chunk.

Then we deal with the data in the WAVE subchunk. We collect the value in the DWORD `xxxx`, which immediately follows the `WAVEfmt` marker. This gives us the size of the format section, which begins right after this `xxxx`. We place an implicit pointer here for the `mmioRead` operation in the next step. In that step we copy the format information into a WAVEFORMATEX structure. We use the `mmioRead` function to place the .WAV file format information into a predefined WAVEFORMATEX structure pointed to by our variable, `lpWaveFormat`. The WAVEFORMATEX structure is identical to the `fmt` structure just given. Now we can access the bits per sample as `lpWaveFormat^.wBitsPerSample`.

You might be wondering if we could have just copied this information straight from the header using regular Delphi streaming methods (or block read and block write), rather than using the multimedia input/output functions described here. Of course we could, but that would involve writing even more code. These multimedia functions are designed for working with this kind of data and save us some additional work.

Next we prepare and lock a memory buffer for the Wave data. For this operation we again use `mmioDescend` and `GetMem`. This time we collect the value in the DWORD `xxxx`, which immediately follows the data marker. This is the size of the data section, which constitutes most of the file. This data begins right after the `xxxx`,

where again an implicit pointer is placed for the `mmioRead` operation that we undertake in the next step.

Now we copy the Wave data into a memory buffer. We use `mmioRead` to place the .WAV file data information into a memory buffer pointed to by our variable, `lpWaveData`, which was returned by `GetMem`. Because .WAV files for telephones are usually quite small, this may be practical; however, for most operations, it's common practice to use at least two buffers (double buffering) so that `waveOutWrite` can play the contents of one buffer while data is being loaded into the other.

At this point, we can close the .WAV file because everything we need is already in memory. We prepare a WAVEHDR structure that describes the memory buffer where Wave data is held. For this we first use `GetMem` to provide a block of memory for the WAVEHDR structure and a pointer to it that we'll name `lpWaveHdr`. A WAVEHDR structure has the format shown in [Figure 5](#).

We copy our pointer to the Wave data, `lpWaveData`, into our structure at `lpWaveHdr^.lpData`, and the buffer length into `lpWaveHdr^.dwBufferLength`. We use the information in this structure in the `waveOutPrepareHeader` and `waveOutWrite` functions. Conveniently, `waveOutWrite` also updates this structure as it plays the .WAV file.

In the next step, we need to load the Wave conversion drivers and prepare the callback routine. The `waveOutOpen` function performs these two steps. Most voice modems come with drivers to convert .WAV files to voice signals. A few can even take the Wave output directly without special drivers.

To check for this functionality, `waveOutOpen` is usually called once for a test with the `WAVE_FORMAT_QUERY` flag set, and with the `WAVE_MAPPED` flag not set. If `waveOutOpen` returns an error, it's called again with `WAVE_MAPPED` set. Otherwise, `WAVE_MAPPED` isn't set. If `WAVE_MAPPED` is set, more resources are used and the operation will be slower. The `waveOutOpen` function also defines the address of a callback routine. Windows sends messages to this callback routine when playing has started, stopped, or paused. There are three possible messages the Wave API can send to this callback routine:

```
lineGetID( // Get a physical device ID.
  rTapi.hLine, // Handle to line from lineOpen.
  0, // Subaddress of rTapi.hLine.
  // From lineMakeCall (rTapi.hCall replaced by 0 here).
  myZeroHC,
  // Use information from rTapi.hLine (not rTapi.hCall).
  LINECALLSELECT_LINE,
  // Pointer to VARSTRING structure where information
  // is returned.
  lpVarString(Port),
  // "device class" (COM, Wave/out etc).
  szDeviceClass);
```

**Figure 4:** Finding a modem port to send Wave data.

```
type
  PWaveHdr = ^TWaveHdr;
  {$EXTERNALSYM wavehdr_tag}
  wavehdr_tag = record
    lpData: PChar; // Pointer to locked data buffer.
    dwBufferLength: DWORD; // Length of data buffer.
    dwBytesRecorded: DWORD; // Used for input only.
    dwUser: DWORD; // For client's use.
    dwFlags: DWORD; // Assorted flags (see defines).
    dwLoops: DWORD; // Loop control counter.
    lpNext: PWaveHdr; // Reserved for driver.
    reserved: DWORD; // Reserved for driver.
  end;
```

**Figure 5:** WAVEHDR structure (record) as defined in `mmsystem.pas`.



- 1) WOM\_OPEN is sent when *waveOutOpen* is called.
- 2) WOM\_DONE is sent when *waveOutWrite* is finished playing data, or *waveOutReset* is called.
- 3) WOM\_CLOSE is sent when *waveOutClose* has completed a close.

Finally, we're ready to play the .WAV file. To play a block of Wave data, we first call *waveOutPrepareHeader*. If we're swapping blocks to save memory (double buffering), we must call this function for each block before we call *waveOutWrite*. We use the *waveOutWrite* function to play the .WAV file to the phone line.

Finally, we need to clean things up. After our application receives the WOM\_DONE message, our code calls the various cleanup functions, freeing resources. If an error has occurred, the file is closed, *waveOutUnprepareHeader* is called, and allocated memory resources are freed depending on the flags set when the **finally** clause is reached in the **try..finally** block.

## Recording Sounds with the Wave API

As with the process of playing sound, recording sound also involves a number of steps. (Due to space constraints, the file discussed here, *recordWv.pas*, is not listed in this article. However, it is available for download; see end of article for details.) Again, to make the process easy to follow, we've included abundant comments in the code and have numbered the steps:

- 1) Get the COM port handle where modem input will be received.
- 2) Use *waveInOpen* to determine if the Wave API supports the Wave format on the selected port.
- 3) Use *waveInOpen* to open the line device for recording, and inform the Wave API that we want all messages sent to the main winproc callback routine.
- 4) Allocate the memory buffer in which the Wave data will be stored.
- 5) Use *waveInPrepareHeader* to tell the Wave API the address of the WAVEHDR structure; we can use this structure to exchange messages with the Wave API. We also put the address of the memory allocated for the Wave data in this structure.
- 6) Use *waveInAddBuffer* to prepare everything for recording.
- 7) Call *waveInStart* to begin recording from the line.
- 8) Wait for a WIM\_DATA callback message to inform us that playing has completed or has been stopped; on error, clean up everything.
- 9) Save the data to a file:
  - a) create the file where the Wave data will be saved;
  - b) use the information from the WAVEHDR structure to update the Wave header file, then write the header to the file; and
  - c) write the Wave data to the file.
- 10) Clean up everything (step 8 again, but not because of error).

As before, we begin by getting a handle to the COM port. This step is identical to the one we used for playing, except now *lineGetID* receives a *wave/in* message instead of a *wave/out* message.

Next, we need to determine if the Wave format and port are okay. You'll recall that when we were playing a .WAV file, we began by opening it and examining the contents of the *tWAVEFORMATEX* header structure. Once we have that information, we can then use:

```
waveOutOpen( , , , WAVE_FORMAT_QUERY)
```

to determine if the format was okay.

We have predefined the *tWAVEFORMATEX* structure in a record named *cnrRecordedMsgFormat*. This time, we can use *waveInOpen* to find out immediately if there is a problem with our proposed data

format. Remember, a *tWAVEFORMATEX* structure includes *nSamplesPerSec* set to 8,000 Hz and *wBitsPerSample* set to 16 bits for a standard uncompressed PCM Wave format.

Now we're ready to open the line and establish our callback routine. If *waveInOpen* returns successfully in step 2, we need to call it again to do some real work. First we need a handle to the Wave API, which we'll use for subsequent calls to other Wave API functions. We'll store this handle in *rWvR.hWaveRDevice*. We'll also provide a pointer to our *tWAVEFORMATEX* structure for other purposes.

We also supply the CALLBACK\_WINDOW flag to tell the Wave API that we want all messages sent to the program's main callback routine, located in the *DefaultHandler* procedure at the beginning of the main unit. The three possible messages that the Wave API can send to this callback routine are:

- 1) WIM\_OPEN, sent when *waveInOpen* is called;
- 2) WIM\_DATA, sent when the buffer is full or *waveInReset* is called; or
- 3) WIM\_CLOSE, sent when *waveInClose* is called to close the operation.

The one that is most useful for us is WIM\_DATA. When the *DefaultHandler* receives a WIM\_DATA message, we call routines to save the .WAV file and perform the required clean up.

It's worth mentioning that during early versions of this program, we attempted to use a specialized callback routine (like the one used for the TAPI functions) by specifying the CALLBACK\_FUNCTION flag. However, the program tends to hang if any but a limited number of functions, such as *PostMessage*, are called from within this function, so we decided to use the main callback routine to simplify things.

In the next step, we allocate memory for the Wave data. Having verified that the modem port and the Wave format are okay, we now allocate 60 seconds of memory using the following formula:

```
cn60SecMemSize:
  DWORD = DWORD(60) // 60 seconds msg storage time.
  * DWORD(8000) // nSamplesPerSec (8kHz).
  * DWORD(16) // wBitsPerSample.
  div DWORD(8); // bits in word.
```

This adds up to 960,000 bytes, hardly an insignificant amount. If memory is an issue, it's possible to use the technique of double buffering, switching back and forth between two much smaller blocks of memory. While the Waveform audio device is processing one memory buffer, the application can process the other.

Next, we need to set up the message-exchanging structure with the Wave API. If we were switching back and forth between blocks of memory, we would need a WAVEHDR structure for each buffer and use it for much of the communication between the Wave API and our application (see [Figure 6](#)).

For our simplified application, we'll be putting just the address of the memory location into which the Wave data will be written. We put the address into *lpData*, and the total size of the available memory in *dwBufferLength*. After we're finished recording, we'll retrieve the amount of data actually recorded from *dwBytesRecorded*. This is necessary because recording can end early under a variety of circumstances. The function, *waveInPrepareHeader*, takes care of informing the Wave API about our WAVEHDR structure after we've put data into it.

```

cnrRWaveHdr: WAVEHDR = ( // Wave header for data/file.
lpData      : nil; // Pointer to locked data buffer.
dwBufferLength : 0; // Length of data buffer.
dwBytesRecorded : 0; // # of bytes recorded.
dwUser       : 0; // Put anything you want in here.
dwFlags      : 0; // Flags describing buffer state.
dwLoops      : 0; // Loop control counter.
lpNext      : nil; // Reserved for driver.
reserved     : 0); // Reserved for driver.

```

**Figure 6:** A WAVEHDR structure for each buffer to use for much of the communication between the Wave API and our application.

## Recording

Before recording, we must make final preparations. Once the *waveInPrepareHeader* function has informed the Wave API about the WAVEHDR structure, the *waveInAddBuffer* function must actually set it up in preparation for playing. If we're using double buffering, we must prepare separate WAVEHDR structures for each memory buffer and set up each with *waveInPrepareHeader*. When a particular buffer has been filled, its associated *dwFlags* is set to WHDR\_DONE so that the buffer can be saved and then set up again.

The *waveInPrepareHeader* and *waveInAddBuffer* functions are always used together; the former to prepare the data buffer headers, and the latter to actually place the data buffer on the input queue to be recorded. For large files, these data blocks can be re-used once the application has recorded the material in them. However, a detailed discussion of this process is beyond the scope of this article.

Finally, we are ready to begin recording from the line. The *waveInStart* function begins recording sound data coming from the phone line to a memory buffer. At this point, we must wait for a WIM\_DATA completion message. Once the memory buffer is full, a WIM\_DATA message is sent to the application's main callback routine, the *DefaultHandler*. In our case, we use this message as a trigger to call the *ShutDownRecording* function. If there are no errors, the first thing it does is call the *SaveMessageTooFile* routine.

Now we're ready to save the data to a file, a process of three steps. First we create a file with the name *msg#.wav*. If we record more than one message while the application is open, the first one is named *msg1.wav* and the second *msg2.wav*. The resulting Waves can easily be played simply by creating shortcuts, and then clicking on them. If you rename one or the other to *Greeting.wav*, it can also be played back over the telephone line.

Next, we update our custom Wave header record named *cnrRiffWaveHeader* (the one with the RIFF, fmt, fact, and data chunks), filling it with information returned to us from the WAVEHDR structure in *dwBytesRecorded*. This quantity is essential to calculating the correct values for the Wave header.

Once these values have been calculated, the header is saved to the open file. Finally, we write the Wave data to the file. Here again, *dwBytesRecorded* comes in handy. We close the file and proceed to clean everything up. The primary cleanup steps (apart from freeing memory) consist of executing two functions, *waveInUnprepareHeader* and *waveInClose*. You need to call *waveInUnprepareHeader* before you free memory for its associated buffer. The *waveInClose* function simply closes the Waveform input device and marks all pending buffers as done.

## TAPI Problems and Limitations

As convenient and helpful as TAPI can be, it does have its problems and limitations. As already mentioned, TAPI uses TSPs (drivers) to


communicate with modems. The most generally annoying characteristic of TAPI is that on inexpensive standard modems (anything under US\$300), a call's progress cannot be monitored. Specifically, you can't tell when someone has answered an outgoing call, or when the line is hung up on the other end. It may be possible to do this by recording line activity with Wave functions and examining the noise level. However, we don't think that would be a trivial task.

Another problem that's commonly reported is that digit detection seems to be disabled after a .WAV file finishes playing on certain modems. The program we have created may not have this problem because we reinitialize TAPI every time a call is completed (instead of reusing the line handles and TAPI instances). While this might eliminate certain problems, we consider it extremely inelegant. Also, it might cause problems if we're working with multiple lines or multiple applications sharing the same line.

A further limitation is that with standard modems, you can't collect information from the line before a call is open. Therefore, there is no obvious way of detecting that someone is dialing out on a line connected to the modem unless a user opens a call first from within the application. There is also no method for knowing if an incoming call is a voice, fax, or data call before the call is picked up. In fact, even determining this after the call has been picked up presents an interesting challenge. Finally, because Unimodem/V isn't supported on Windows NT below version 5.0 (Windows 2000), there is no support for voice in that environment.

## Conclusion

We are at the end of a rather long journey that has enabled us to add sound playing and recording capabilities to our TAPI applications. Along the way, we have learned more about the various Windows APIs and how they work together. As far as exploring the Wave API, we have only scratched the surface — just enough to get the job done. In an upcoming article, Dr Moore will explore this API in more detail, and will explain other facets of it. He will also devote an upcoming "File | New" column to multimedia and communications resources.

We used a number of online sources while preparing this article. One such site contains much information on programming with sound; visit <http://www.wotsit.org>, or the Usenet newsgroup at [news:comp.os.ms-windows.programmer.multimedia](mailto:news:comp.os.ms-windows.programmer.multimedia). 

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM99\NOV\DI9911RE.*

Robert Keith Elias is an independent developer, consultant, and practicing Broccolist based in Quebec, Canada. He specializes in Windows programming with Borland Delphi and Web site development. He can be reached at [keliass@CLIC.NET](mailto:keliass@CLIC.NET).

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



## COLUMNS & ROWS

AS/400 / SQL / Delphi Client/Server, Enterprise Editions

*By G. Bradley MacDonald*

# An AS/400 Skeleton Key

## Approaching Client/Server Applications on the AS/400

**A**n important part of developing any database application is database optimization. This is especially true of client/server applications. For example, you may know what SQL statement you passed to the server, but you don't necessarily know what the server did with the statement. Did it use an index? Did it have to rebuild the access plan? Did it have to do any extra work you're not aware of? If so, why? These are the sorts of questions developers must be able to answer to produce applications that will work well in their environment.

The AS/400 is being used increasingly as a back-end database server for client/server applications. However, while many Delphi and PC application developers have experience using SQL with databases such as Oracle, Sybase, and Microsoft SQL Server, they don't usually have experience developing with the AS/400. At the same time, a lot of AS/400 shops have terminal-based systems they use to run their business. These systems use the native file system on the AS/400, and don't usually use SQL to access their files, so they don't have a lot of experience accessing the AS/400 files via SQL. While this situation may appear to be a stumbling block to developing client/server applications with the AS/400, it doesn't need to be.

The AS/400 native file system and DB2/400 are built into the operating system. In fact, they're nearly one and the same. This allows you to use operating system commands to view and analyze native files and SQL tables and views alike. This means you aren't solely dependent on SQL statements to deal with your data, and gives developers another set of tools to debug and analyze their applications. It also means that developers using client/server tools such as Delphi can work with AS/400 developers — who are familiar with the various debugging methods on the AS/400 — to create better applications.

To begin, we'll review some OS/400 commands that will be useful to Delphi developers working with the AS/400. Then, we'll discuss how you can see exactly what the AS/400 is doing with the SQL commands sent to it by your application. Finally, we'll show you how to retrieve the full descriptions of the SQL codes the AS/400 returns to your application. (Note: Some screen captures in this article were edited to hide system information about the test machine for this article.)



Command	Description
<b>FormatDSPPFM</b> (Display Physical File Information)	Displays the contents of a file in a fixed column format. This is an easy, quick way to view the data in a file. Format: DSPPFM FILE(library/file)
<b>DSPFD</b> (Display File Description)	Displays information about the file itself. It contains all the information about the file, such as initial size, extents, etc. When used to view a table or view that was created using SQL statements, it shows the full SQL statement used to create the file. It contains no field information. Format: DSPFD FILE(library/file)
<b>DSPFFD</b> (Display File Field Description)	Displays all the field information, such as field name, field type, field size, start position, end position, etc. Format: DSPFFD FILE(library/file)
<b>STRDBG</b> (Start Debug)	Starts a debugging session on an OS/400 job. This can be either the current job or a service job. While this command has parameters, when using it with a client/server application and the STRSRVJOB command, they're not usually required.
<b>STRSRVJOB</b> (Start Service Job)	Starts an AS/400 service job. From this service job you can monitor and debug other OS/400 jobs, such as client/server connections from Delphi. Sometimes used to debug AS/400 batch jobs. Format: STRSRVJOB JOB(job/user/number)
<b>DSPJOBLOG</b> (Display Job Log)	Shows the user what has transpired during the running of the job. It logs all commands, errors, and informational messages. It's a handy source of information about what your job has done. It can be as brief or as detailed as you want, depending on how you set up the AS/400 job. Format: DSPJOBLOG JOB(job/user/number)
<b>WRKACTJOB</b> (Work with Active Jobs)	Shows the user what jobs are active on the system. It's a good way to get the job number for your client/server connection to the AS/400.
<b>STRSQL</b> (Start SQL)	Starts the interactive SQL command processor. It provides a full screen for entering SQL statements. This utility is useful for trying SQL statements directly on the AS/400 before trying to run them from your client/server application. A nice feature of this screen is the ability to prompt the SQL statement. By typing in a SQL statement, such as SELECT, then pressing <b>F4</b> , you can build your statement by filling in the fields provided, making it especially handy if you can't remember the syntax for a complicated SQL statement.

Figure 1: Selected OS/400 commands.

### Common Commands

The commands shown in Figure 1 can be entered on any OS/400 command line. These commands are primarily concerned with debugging, and displaying file information, file content, and job information. We won't get into the details of the commands, but we'll review them so you have a starting point from which to learn more. We've simplified the example calls to the commands, as most have many parameters. The examples/formats I use should work in most cases, but may need to be customized for your particular situation.

### Tracing an AS/400 Client/Server Connection

One of the issues facing any client/server development environment is to know what is happening on the server side when the client side issues a SQL statement or system command. Because DB2/400 is part of the operating system, you are able to use OS/400 commands to watch exactly what the client/server job is doing on the AS/400. The one catch is that you must do it from an AS/400 terminal session.

To use the following technique, you must have the authority to run four OS/400 commands:

- STRSRVJOB (Start Service Job)
- ENDSRVJOB (End Service Job)

```

Work with Active Jobs
CPU %: 22.1 Elapsed time: 01:07:54 Active jobs: 10/31/98 12:58:28
                                         154
Type options, press Enter.
 2=Change  3=Hold  4=End  5=Work with  6=Release  7=Display message
 8=Work with spooled files 13=Disconnect ...

Opt Subsystem/Job User Type CPU % Function Status
---
 1 BATCH QSYS SBS .0 DEQW
 2 C0400TCP QSYS SBS .0 DEQW
 3 P016002014 CONBM BCH .0 PGM-C0400TCP TIMW
 4 SERVSOCKET QUSER ASJ .0 PGM-SERVSOCKET TIMW
 5 EASYCOM QSYS SBS .0 DEQW
 6 QSYS SBS .0 DEQW
 7 NETMGR BCH .0 DEQW
 8 QSYS SBS .0 DEQW
 9 QSYS SBS .0 DEQW
More...

Parameters or command
==>
F3=Exit F5=Refresh F10=Restart statistics F11=Display elapsed data
F12=Cancel F23=More options F24=More keys
    
```

Figure 2: The Work with Active Jobs screen.

- STRDBG (Start Debug)
- ENDDDBG (End Debug)

If you don't have authority to run these commands, contact your AS/400 administrator and ask them to grant you \*USE authority for them. Also, if the job on the AS/400 is run under a specific user profile, you may require \*USE authority to that profile. Although this is possible, for security reasons it would better to change the job on the AS/400 to run under the user profile of



```

Work with Job
Job:   P016002014   User:   CONBM   Number:  774728
Select one of the following:
1. Display job status attributes
2. Display job definition attributes
3. Display job run attributes, if active
4. Work with spooled files
    
```

Figure 3: An AS/400 job number.

```

Change Job (CHGJOB)
Type choices, press Enter.
Message logging:
Level . . . . . 4          G-4, *SAME
Severity . . . . . 18       G-99, *SAME
Text . . . . . *seclvl     *SAME, *MSG, *SECLVL, *NOLIST
Log CL program commands . . . . . *NO      *SAME, *YES, *NO
Inquiry message reply . . . . . *RD      *SAME, *RD, *DFT, *SYSRPLY
Break message handling . . . . . *NORMAL  *SAME, *NORMAL, *NOTIFY...
    
```

Figure 4: Message logging parameters.

```

Start Service Job (STRSRVJOB)
Type choices, press Enter.
Job name . . . . .          Name
User . . . . .             Name
Number . . . . .           000000-999999
    
```

Figure 5: The STRSRVJOB command.

```

Display All Messages
Job . . . : P016002014   User . . . : CONBM   Number . . . : 774728   System:
2 rows fetched from cursor CURS1.
2 rows fetched from cursor CURS1.
2 rows fetched from cursor CURS1.
ODP deleted.
Cursor CURS1 closed.
PREPARE of statement STAT1 completed.
DESCRIBE of prepared statement STAT1 completed.
All access paths were considered for file MASSUM.
Access path of file MASSU1 was used by query.
ODP created.
Blocking used for query.
Cursor CURS1 opened.
2 rows fetched from cursor CURS1.
2 rows fetched from cursor CURS1.
Press Enter to continue.
More...
    
```

Figure 6: An example of the information provided about a query.

```

Additional Message Information
Message ID . . . . . : CPI432C   Severity . . . . . : 00
Message type . . . . . : Information
Date sent . . . . . : 10/31/98   Time sent . . . . . : 13:02:07
Message . . . . . : All access paths were considered for file MASSUM.
Cause . . . . . : The OS/400 Query optimizer considered all access paths
built over member MASSUM of file MASSUM in library TEST.
The list below shows the access paths considered. If file MASSUM in
library TEST is a logical file then the access paths specified are
actually built over member MASSUM of physical file MASSUM in library
TEST.
Following each access path name in the list is a reason code which
explains why the access path was not used. A reason code of 0 indicates
that the access path was used to implement the query.
TEST/ MASSU6 5, TEST/ MASSU5 4, TEST/ MASSU4 5,
TEST/ MASSU3 5, TEST/ MASSU2 5, TEST/ MASSU1 0.
Press Enter to continue.
More...
    
```

Figure 7: Detail information from the item in Figure 6.

```

Additional Message Information
Message ID . . . . . : CPI432B   Severity . . . . . : 00
Message type . . . . . : Information
Date sent . . . . . : 10/31/98   Time sent . . . . . : 13:02:07
Message . . . . . : Access path of file MASSU1 was used by query.
Cause . . . . . : Access path for member MASSU1 of file MASSU1 in
library TEST was used to access records from member MASSUM of file
MASSUM in library TEST for reason code 2. The reason codes and their
meanings follow:
1 - Record selection.
2 - Ordering/grouping criteria.
3 - Record selection and ordering/grouping criteria.
If file MASSUM in library TEST is a logical file then member
MASSUM of physical file MASSUM in library TEST is the actual file
being accessed.
Index only access was used for this query: *NO.
Press Enter to continue.
More...
    
```

Figure 8: Additional detail information from the item in Figure 6.

the person running the client/server job. Delphi/400 and EasyComm/400 — two of the more popular ways of connecting Delphi to the AS/400 — are capable of doing this.

### Determining the Job Number

Every connection to the AS/400 starts a job on the AS/400. Whether the connection is a terminal emulation program or a client/server application, they all require a job on the AS/400 to communicate with it. This is the job we'll trace. The basic process is to start the Delphi job and sign on to the AS/400. At this point, the job has been initiated on the AS/400, and you should start the trace process on the AS/400.

This process takes a few moments to set up, so your Delphi client/server application must not be allowed to complete before you finish the setup of the trace. If the Delphi job ends before you are able to display the trace, you'll lose the information you're trying to gather. The easiest way to do this is to place one break point just after the logon is completed, and another just before the program terminates. This will give you all the time you need to set up the trace and run the program. The break point at the end of the program ensures the job won't terminate before you get a chance to log the information you require.

To trace a job on the AS/400, you must know its job number. The job number consists of three parts: Job, User, and Number. To find the job number, use the WRKACTJOB command (see Figure 2).

You may need some assistance from your AS/400 system administrator to determine the subsystem in which your job is running. By default, the Delphi/400 jobs run in the CO400TCP or CO400SNA subsystems, depending on which communication protocol is being used (TCP/IP or SNA) to communicate with the AS/400. The EasyComm/400 jobs all run in the EASYCOM subsystem. Then enter a 5 beside the job and hit **[Enter]**; you'll see the full job number at the top of the resulting screen (see Figure 3).

To see the information you want, you may have to change the job itself. On the Work with Active Jobs screen, you can enter a 2 in the options field beside the job you want to change, then hit **[Enter]**. You'll also want to change the Message Logging fields, as shown in Figure 4. These fields tell OS/400 how much detail to keep track of for that job.

### Start the Service Job

You now have the job number of the job you want to trace. However, because you are using a terminal session to monitor the client/server job, it has a job of its own. This means that you must use a service job to monitor the other job. The command to do this is STRSRVJOB. Enter STRSRVJOB on the command line, select **[F4]** to prompt the command, then enter the job information in the fields provided (see Figure 5).

### Putting the Job in Debug Mode

Once the STRSRVJOB command has completed, you'll be



```

Display Message Description (DSPMSGD)

Type choices, press Enter.

Range of message identifiers:
  Lower value . . . . . > SQL0100      Name, *ALL, *FIRST
  Upper value . . . . . *ONLY         Name, *ONLY, *LAST
Message file . . . . . > QSQLMSG      Name
Library . . . . . *LIBL              Name, *LIBL, *CURLIB...
Detail . . . . . *FULL               *BASIC, *FULL
Format message text . . . . . *YES    *YES, *NO
Output . . . . . *                    *, *PRINT
    
```

Figure 9: The DSPMSGD command prompted with [F4].

```

Display Formatted Message Text
System:

Message ID . . . . . : SQL0100
Message file . . . . . : QSQLMSG
Library . . . . . : QSYS

Message . . . . . : Row not found for 81.
Cause . . . . . : One of the following conditions has occurred:
-- If this is a FETCH statement, no more rows satisfy the selection values
(end of file). The name of the cursor is 81.
-- If this is a FETCH statement for a scrollable cursor, a record was not
found. If NEXT was specified, end of file was reached. If PRIOR was
specified, the beginning of the file was reached. If RELATIVE was
specified, either the beginning of file or the end of file was reached,
depending on the value specified. If FIRST or LAST was specified, then no
records satisfy the selection criteria. The name of the cursor is 81.
-- If this is an embedded SELECT statement, no rows satisfy the selection
values.
-- If this is an UPDATE, INSERT, or DELETE statement, no rows satisfy the
values.
More...
Press Enter to continue.
    
```

Figure 10: Detailed information provided by DSPMSGD.

back at the command line. At this point, you must enter the STRDBG command and hit [Enter] to place the service job in debug mode. The debug command will cause the AS/400 to log more information to the job log than it normally would. It's in this extra information that we'll see the trace information for the SQL work done by the DB2/400 server on the AS/400.

### Displaying the Job Log

At this point, you are able to enter the DSPJOBLOG (Display Job Log) command to view the job details. This command must be prompted, using [F4], and the job number must be entered in the fields provided. You are also able to choose whether to display the information on the screen or print it in a report. I almost always send the information to a spool file so I can review it later and perform searches on it.

When looking at the information on the screen, the first thing you should do is to hit [F10]. This will display all messages that have occurred since the beginning of the job. Note that in Figure 6 the screen capture shows that the job fetches some records from the server, closes the cursor, prepares a new SQL statement, opens a new cursor, and then fetches more records. It also tells you that it considered all access paths (Indexes) and that it chose the access path of file MASSU1. This is important information. What if you thought the query would be using a different index? This could severely impact the performance of your SQL statement.

The next question you want to ask is why it chose that index over others. To see the details of that particular item, move the cursor down to that line and hit [F1]. This will display a screen similar to that shown in Figure 7. The details tell you why each index was not used to complete the query. The second part of the screen, which cannot be seen in the screen capture, explains what each of the reason codes mean.

Figure 8 is the detail of the next message from Figure 6, and it tells you why that particular index was chosen for the query. This type of detail is available where appropriate, and can be of great help in telling exactly what the AS/400 has done with the SQL statement your application sent to it.

### Ending the Tracing Process

After you have the trace, you need to end the tracing process. First you must enter ENDDBG; then enter ENDSRVJOB. In other words, the debug mode and the service job must be ended.

### Displaying DB2/400 SQL Error Messages

When your program generates a SQL statement with which the DB2/400 engine on the AS/400 has a problem, the AS/400 passes back a SQL code to your program. An example would be SQL -100. Delphi programmers are left wondering what SQL -100 means. While many of the SQL messages the AS/400 produces are standard, there are a few that are particular to the AS/400. The trick is determining what the SQL return codes mean.

The easiest way to do this is to use an AS/400 terminal session to run the DSPMSGD command. This command will display the message description of any message on the AS/400 that resides in a message file. This is the common method for AS/400 developers to look up error messages generated by their programs.

All SQL messages reside in their own message file called QSQLMSG. Enter DSPMSGD on a command line and hit [F4] to prompt it (see Figure 9). You must fill in the first field (Lower Value) with the SQL code returned to your program. A lot of the return codes are only three characters long, but this field requires the characters "SQL" plus a four-digit number. To make up the fourth character, you must put a zero (0) in front of the three-digit code. Also, all negative return codes should be entered as positive, as there is no way to enter negative numbers in the field.

Figure 10 shows the details about the SQL -100 code that was passed to the application.

### Conclusion

The AS/400 provides some powerful trace utilities for client/server applications. Unfortunately, they aren't well known to either the Delphi or AS/400 communities. The goal of this article is to provide client/server developers with some basic tools and a starting point for them to learn more about developing applications for the AS/400. To keep the article brief, I have not gone into a lot of detail. However, I hope it has given you enough information to get you started tracing and debugging your client/server applications that use the AS/400 as the database server. ▲

G. Bradley MacDonald is a consultant who specializes in connecting Delphi to the AS/400. He has written various articles dealing with Delphi and the AS/400, as well as a technical White Paper for Inprise on Delphi/400. Mr MacDonald can be reached via e-mail at [BradleyMacDonald@BC.Sympatico.CA](mailto:BradleyMacDonald@BC.Sympatico.CA).





## NEW & USED

By Ron Loewy

# Raize Components 2.1

## 80 Well-thought-out UI Controls

Normally, I'm not a big fan of large component sets. I prefer third-party solutions with a small number of components to master. You could say I like my solutions focused. Although I did purchase some of the large component collections for my work in the past, I did it for a specific control or two and found I didn't use the suite for anything other than the component that initially caught my attention.

Raize Components 2.1 (RC2) from Raize Software Solutions, Inc. is a large set of user-interface components. I'm a big fan of another Raize product, CodeSite, which suits my preference for simple operation and a minimal learning curve. My favorable experience with that tool prompted me to try the new component collection, despite the fact that more than 80 new glyphs were added to my Component palette.

### Installation

Raize went the extra mile with the installation program for this product. It prompts you for a user name and company, the serial number supplied by Raize, and the Delphi version(s). My computer has Delphi 3 and Delphi 4 installed, but I opted to install the components for Delphi 4 only.

The installation program copies the required packages to disk, registers the components with Delphi, updates your library path (I wish more third-party tools would do that), and even combines its Help

file with Delphi's Help system. It's a rare occasion when a component installation process requires nothing more than activating the setup program.

Four new tabs were added to the Component palette:

- The Raize tab includes a set of user-interface components I consider as Raize Software's replacement for the Standard, Additional, and Win32 tabs that ship with Delphi. These are the components we're always using, only with some interesting features, which I'll briefly cover later.
- The Raize List tab includes a collection of list boxes, combo boxes, tree components, and list views with enhanced functionality or specific uses, such as color and font list boxes, and a tree that can have checkboxes associated with its nodes.
- The Raize Data tab includes a collection of data-aware, user-interface controls (akin to the standard Data Controls tab of Delphi).
- The Raize Misc tab is a collection of gadgets that don't fit anywhere else. These include spinners, gradient forms, balloon hints, and more.

### The "Standard" Visual Controls

The controls provided on the Raize tab are a direct replacement for many of the standard components that ship with Delphi. The *TRzLabel* component, for example, is the replacement for the standard *TLabel*. The Raize version provides additional display properties, such as the text rotation angle, custom borders, and text styles.

When you place an RC2 component on a form, you can usually activate a custom component editor by using the context menu. In the *TRzLabel* case, this

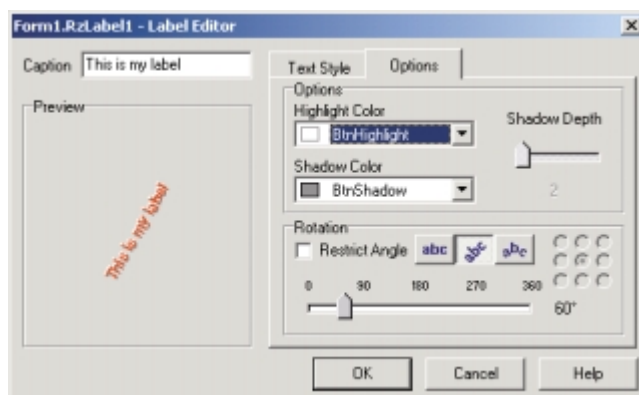


Figure 1: The *TRzLabel* component editor.

editor is available as **Edit Label** at the top of the popup menu. This opens a component editor that provides immediate feedback for the different custom settings available for the component (see [Figure 1](#)).

In addition to this nice feature, other commonly used options for a component are usually also available from the popup menu and save you the time to search for the required property in the long list of properties that are displayed in the property editor. For example, the *TRzPanel* component has an **Align Client** option in the popup menu — definitely one of the more common things I do with panels.

It's obvious that Raize Software took the time to fix some of the common annoyances of Delphi's standard components: How often do you place a *TPanel* on a form and immediately proceed to clear its *Caption* property? The *TRzPanel* component comes with a blank caption by default — not a huge improvement by itself, but definitely an indication of the attention to detail that is part of this library.

Most of the controls provide custom frame options. If you want your applications to look different from the standard Windows buttons and edit boxes on a gray background, these framing options make it easy to create forms that look like regular printed forms, checks, or other real-life data entry paper sources we're used to. You can define many frame styles or define which of the frame sides will be displayed (if only the bottom side of the frame is visible, for example, a paper form look is easily imitated).

If your form contains many edit controls that need to share a custom frame option, it's a hassle to change them at design time (and even worse at run time). Fortunately, the *TRzFrameController* component can be used to iterate through all the components on the form and set the frame attributes property. Thus, you can easily change between a classic Windows look and your custom frame look, based on user preferences.

The standard components include button components that can be linked to a drop-down menu, list and combo boxes with the frame options (and other display properties), progress bar, track bar, and other replacements to the standard Delphi components that I won't mention here because of space limitations.

Other notable components are the *TRzSplitter*, which creates splits by defining a container with two sections (vertical or horizontal), instead of the standard Delphi method of dropping panels with a splitter between them. This solution makes it easy to create custom splitter areas, and each part of the two parts in the splitter container is used to host other components. You can easily drop another splitter on one of the splitter parts and cascade your splitter to create complicated frames.

The *TRzToolbar* component makes the design of toolbars easy and fun. A set of common glyphs is available from the component editor, so you don't have to hunt for images for your buttons. The *WrapControls* property can be used to wrap the toolbar buttons automatically as the user changes its size. A set of standard toolbar controls (button, button with drop-down menu, or spacer) are included to make toolbar construction easy. Another advantage is that this control doesn't require a Microsoft DLL, as the standard *TToolbar* component does.

The status bar replacement (*TRzStatusBar*) provides standard status bar panes that are always a pain to implement. The available resources, the state of the keys, a progress bar, and other options are available. Again, this is not the kind of component that will

make or break the collection, but it is a nice replacement for the standard Delphi-supplied component.

## List Controls

The Raize List tab includes several general-purpose list, combo box, and tree view controls that provide additional visual properties, such as checkboxes associated with the items (*TRzCheckList*, *TRzCheckTree*), or the ability to have multiple columns for every item (*TRzTabbedListBox*). *TRzEditListBox* allows the editing of items by the user at run time.

Descendants of the *TreeView* and *Listview* controls implement the custom frame options that allow you to integrate these components in your form with its special look. In addition to these general-purpose controls, some special common-case controls are available, e.g. color and font combo boxes, a file system combo box, list boxes, and a directory selection dialog box.

## Data Controls

The data components included with RC2 are either simple extensions to the regular Delphi DB Controls with the Raize frame and display options (e.g. *TRzDBLabel*, *TRzDBEdit*, *TRzDBMemo*, etc.) or database versions of controls included in the "standard" Raize tab, such as *TRzDBStatusPane*, which can be used as a data-aware pane in the status bar component described earlier.

As most of the components in this package, these can get their frame settings from the frame controller mentioned previously, and — when appropriate — provide easy-to-use visual component editors.

## The "Other" Controls

The Component palette tab labeled Raize Misc hosts all the components that didn't seem to fit in the other groups. This is a collection of mostly visual gadgets that range from fun and (for most applications) useless, to professional, essential code that can save you a lot of time.

These are some of the components that caught my eye:

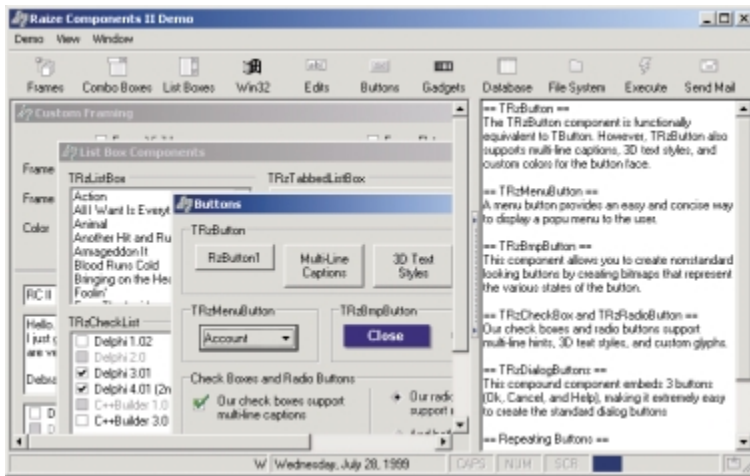
- *TRzLauncher* launches an external application. This is basically a replacement to the *WinExec* Windows API call, but with the option to wait for the spawned process to finish, which isn't a simple thing to do properly (you must dig into the Windows API).
- *TRzButtonEdit* is a combination of an edit box with a built-in button, like the one that appears in the Object Inspector for properties that activate a property editor. I wish I'd had this component when I was implementing a property inspector for one of my applications.
- *TRzDialogButtons* is a panel with standard dialog box buttons (OK, Cancel, etc.) that automatically aligns to the bottom of your dialog box and allows you to add/remove standard buttons as Boolean properties of the component. It's not an Earth-shattering component, but it is a real time saver. The only problem with this component is that it doesn't offer the Borland glyphs for *TBitBtn*, *bkOK*, *bkCancel*, and *bkHelp* as an option.

**INFORMANT**  
**FACT FILE**

Sometimes small improvements in things we use every day are more important than a revolutionary change in something we use infrequently. Raize Components 2.1 definitely qualifies as such a tool. It has no single, must-have component, but features many components that I use all the time. I just wish I'd had it when I started my current development project.

**Raize Software Solutions, Inc.**  
2111 Templar Drive  
Naperville, IL 60565

**Phone:** (630) 717-7217  
**Web Site:** <http://www.raize.com>  
**Price:** US\$249



**Figure 2:** The demonstration program in action.

- *TRzBMPButton* is a button created from a set of bitmaps you provide for states such as Up, Up and Focused, Down, Disabled, etc.
- *TRzAnimator* animates a group of images in an image list — great for small animations that don't require the use of an AVI or MPEG file.
- *TRzMeter* displays a set of values in several formats — yet another example of a component that's not going to make or break your application, but the cool graphic options make it nice to have.

## Documentation, Samples, Support

Attention to detail is one of the things that differentiates this package from others on the market. The Help file is complete, providing all the reference information you would want about the components.

The better products provide good samples that demonstrate the capabilities of the components and provide you with a head start for development. RC2 doesn't disappoint in this regard. Its sample

application (all the source is included) is an impressive and educational showcase of the various components (see Figure 2).

If you can't figure out how to use the components from the provided component editors, Help files, and sample program, you might be in the wrong profession. But when questions do arise, Raize Software provides Internet newsgroups, and questions are answered quickly.

## So What's Missing?

Raize Software should consider adding grid components to future releases (standard and data-aware). I would not expect a do-it-all, cook-and-sing grid like some of the tools available on the market, but a simple descendant of the Delphi grid with some of the frame and display capabilities that are RC2's claim to

fame would be nice. If the resident wizard would also allow you to use the Raize components as the in-place editors, I would be in UI heaven.

## Conclusion

If you can't guess that I like this package, I obviously failed in the task of writing this review. It's different from most third-party tools I use; it has no component that — for me — makes it a must-have, but it features many components that I use all the time. I just wish I had it when I started my current development project. ▲

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



## Inprise/Borland Conference 1999

This was my fourth Inprise/Borland Conference, and it was the most enjoyable so far. Among other things, a conference can be a barometer of the company sponsoring it — of its direction, priorities, and health. After I highlight some of the more memorable events at the conference, I will attempt to “read the tea leaves” and discuss possible indicators of the future for Inprise/Borland.

Again this year, I had an opportunity to meet and exchange ideas with readers and friends. Spending a lot of time in the vendor area, I was able to learn about advancements in some of my favorite component libraries, including Digital Metaphors' ReportBuilder, SkyLine Tools' ImageLib Corporate Suite, and the unsurpassed Raize Components. Of course, I spent a good deal of time at the Eagle Software booth, receiving my annual CodeRush tutorial. In the process, I discovered the many keyboard templates I should have been using the past year. At the TurboPower booth, I learned the venerable company was finally getting ready to release the profiler I had been bugging them to develop for years.

At the core of each year's conference are excellent tutorials, technological presentations, vendor showcases, and birds-of-a-feather sessions. One highpoint for me this year was a birds-of-a-feather session that Mark Miller asked me to moderate entitled “Can Borland's Third-Party Vendors Survive?” This standing-room-only session was as provocative as we had hoped it would be, and gave me the opportunity to return to issues related to developer ethics I've discussed previously in this column.

The panel members were a group of Delphi all-stars: Mark Miller, President of Eagle-Software; Ray Konopka, President of Raize Software Solutions; Julian Bucknall, who manages all of the developers at TurboPower; and Marco Cantù, best-selling author of several outstanding Delphi books. As moderator I was prepared to broach three topics related to vendor survivability: software piracy, project management, and human resource management. Because we were under severe time constraints, we did not get beyond the first topic. Nevertheless, attendees had the opportunity to gain a much greater understanding of some of the problems that face the makers of the tools we use — problems that could potentially drive some vendors out of the business.

**New leadership.** Usually I am not too interested in the various industry keynotes. However, one of the delightful experiences this year was Larry Constantine's “On Becoming a Leader: Advice for Tomorrow's Development Managers.” This masterful presentation included proven pointers for anyone intent on making a positive impact on their working environment.

Interestingly, Inprise's new President, Dale Fuller, seems to understand and embody many, if not most, of the enlightened management approaches that Constantine recommends. The latter's recommendations include listening and questioning, instead of talking, as an important key to effective leadership. He also suggests letting others lead meetings and learning from within the group.

The opening keynote was more like the “opening scam.” Dale hid himself in the middle of the audience, and when David I. acted con-

fused about the CEO's whereabouts, he “decided” to take questions from the audience. Continuing the scam, Dale stood up with a question that would have been more typical from a disgruntled Borland customer than from its CEO.

His involvement in the remainder of the conference was also marked more by a listening-and-learning approach, than by a behind-the-scenes directing approach. He was everywhere, as accessible as anyone else from Inprise/Borland. I had an opportunity to speak with him personally in my favorite haunt, the vendor area, and found him to be very open and receptive to my suggestions.

Borland developers with whom I spoke indicated that his approach back at the company is similar. The ivory tower has been replaced by a cubicle in the midst of hectic research and development. In the opinion of this writer, at least, Borland has returned to its roots and is not only acknowledging the importance of its developer base, but is taking increasingly meaningful steps to support that developer base. This is how Borland became the great development company it was in the 1980s. I remember those days. Like most of you, I hope that they are returning.

If my perceptions are accurate, and if this company continues on this path of rediscovery, then I predict its future will be bright and we have much to look forward to. Of course, all of us have a role to play as well. As Ben Riga stated in his inspiring showcase of Delphi 5, a most powerful tool in marketing Delphi is word of mouth. I am aware of many of you out there who are Delphi evangelists (like my friend from Project JEDI, Shiv Kumar). Developers like Shiv are willing to take time off from their busy schedules and lead their ignorant brethren from the thick dark forest of inferior development tools, to the sunny valley of Delphi. We need more of them. In this coming year, let's work together to make the resurrection of this company a reality. Together — in our user groups, with our business partners, with Project JEDI, and in our Internet discussions — we can do it.  $\Delta$

— Alan C. Moore, Ph.D.

*Note: This is an abridged version of this column. The complete column is available on the Delphi Informant Web site at <http://www.DelphiMag.com>.*

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