May 1997, Volume 3, Number 5

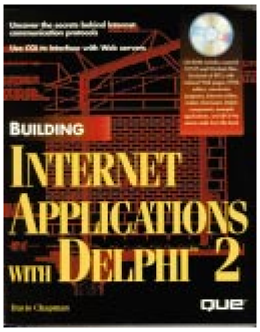
Cover Art By: *Tom McKeith*

Delphi 3 The ActiveX Foundry

From COM to DCOM, Delphi 3 Means Business

New Delphi Book

Building Internet Applications with Delphi 2
Davis Chapman
QUE Corp.



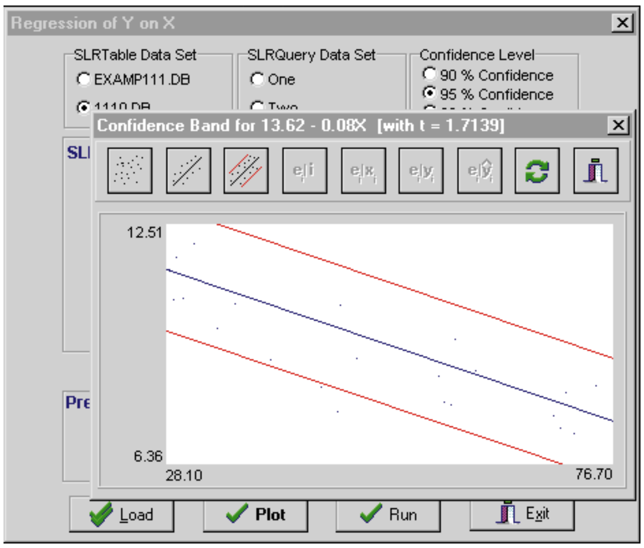
ISBN: 0-7897-0732-2
Price: US$49.99
(624 pages, CD-ROM)
Phone: (800) 428-5331

Applied Analytic Systems Ships New Components

Applied Analytic Systems, Inc. of Carnegie, PA has released its statistical analysis components for Delphi 1 and 2. The *TRegress*, *TAnova*, and *TForecast* components perform randomized block analysis of variance, and exponential smoothing forecasting on data in tables accessed through the BDE.

The statistical components can be used in place of the *TTable* component, allowing developers to incorporate database-aware statistical capabilities into their software.

The components can analyze the data (or subset of the data) in any numeric column of the table to which they are attached. Developers using the components supply the database and table name as they would with *TTable* or *TQuery*. The resulting software can produce robust inferential outputs, even though the programmers may have little or no training in statistical methodologies.



Price: Each component, US$79 for a single-developer; three-component set, US$199. Site licenses are available by arrangement for multiple developers at a single location. All components include unlimited royalty-free run-time licenses.

Contact: Applied Analytic Systems, Inc., 600 North Bell Ave., Bldg. 1, Carnegie, PA 15106
Phone: (412) 278-2360
Fax: (412) 788-4205
E-Mail: info@aasdt.com
Web Site: http://www.aasdt.com

NuMega Announces BoundsChecker 5.0 Delphi Edition

NuMega Technologies, Inc. of Nashua, NH has announced *BoundsChecker 5.0 Delphi Edition*, the newest release of its error detection system for developers of Windows, Internet, and enterprise applications.

BoundsChecker 5.0 Delphi Edition performs active API validation on over 3,000 calls to the latest Windows APIs and OLE methods, including Win32, ActiveX, DirectX, ODBC, and more. BoundsChecker's active validation checks each API call for valid parameters, valid return codes, proper number of parameters, out-of-range parameters, invalid flags, conflicting flags, uninitialized fields, and bad pointers. BoundsChecker's API validation facility is open and extensible. Users can extend BoundsChecker's error detection power to include their proprietary and third-party APIs.

Smart Debugging is included. It allows developers to detect and fix errors as a normal by-product of the development process. Smart Debugging works inside the Delphi IDE and is compatible with Delphi 2 and 3. It monitors all events, searches for bugs as the user steps through code, and displays any errors found.

BoundsChecker displays multiple call stacks, identifying the source of memory overwrites and leaks, and the location in the program where memory is allocated or deallocated. Users have immediate, contextual information about why an error occurred and where. Errors and events are viewed in real time, providing immediate and comprehensive information on program execution.

BoundsChecker 5.0 introduces ActiveCheck and FinalCheck, error detection technologies that deliver performance increases of 500 to 700 percent and higher. It analyzes a program at run time and pinpoints errors without requiring instrumentation, re-compiling, or re-linking. ActiveCheck performs thorough run-time checking of each API call to ODBC 3.0, Internet APIs, ActiveX, and DirectX, as well as the Win32 interface to the Windows operating system. FinalCheck is effective at finding difficult memory and pointer errors.

Price: US$399
Contact: NuMega Technologies, Inc., #9 Townsend West, Nashua, NH 03063
Phone: (800) 468-6342 or (603) 578-8400
Fax: (603) 889-1135
E-Mail: info@numega.com
Web Site: http://www.numega.com

Borland Developers Conference

The 8th Annual Borland Developers Conference will take place July 12-16th in Nashville, TN.

Attendees will be able to design their own agendas, selecting from beginning, intermediate, or advanced developer sessions in the following categories: components; design and methodology; enterprise computing; InterBase; Internet/intranet; management issues; programming, tools, and techniques; and solutions.

For more information, visit http://www.borland.com.

SuccessWare International Ships Apollo 3

SuccessWare International of Temecula, CA has shipped *Apollo 3*, an update to its database engine for Delphi.

Apollo 3 features its proprietary technology, Roll-Your-Own Indexes and Filters; ascending/descending index orders; increased support for Delphi's integrated components; improved data access and optimized-query speeds; a smaller deployment footprint; and more.

Price: US$179
Contact: SuccessWare International, 27349 Jefferson Ave., Ste. 110, Temecula, CA 92590
Phone: (800) 683-1657 or (909) 699-9657
Fax: (909) 695-5679
E-Mail: sales@gosware.com
Web Site: http://www.gosware.com

Page Technology Marketing, Inc. Releases PCLTool SDK Version 4.4

Page Technology Marketing, Inc. of San Diego, CA has released *PCLTool SDK Version 4.4*. This .DLL library views, indexes, archives, searches, retrieves, overlays, and converts PCL 4/5 print data files for COLD, faxes, browsers, and other applications. Developers can integrate PCLTool's .DLLs to add PCL5 file-handling capabilities into their high volume e-form applications.

PCLTool SDK can index mortgage-loan print datastreams, retrieve individual pages by index or full text search of the PCL, view it, overlay data, or convert it to a .TIF for imaging, storage, or faxing.

It can also convert PCL form overlays into a Windows Metafile Format (.WMF) for porting PCL forms to device-independent Windows applications.

Version 4.4 includes: location indexing; a PCL text search; drag-and-drop operations; and a 'Net browser helper. In addition, it's command-line driven.

Price: US$495 for a 100-user license.
Contact: Page Technology Marketing, Inc., 10671 Roselle St., Ste. 100, San Diego, CA 92121
Phone: (800) 748-3668 or (619) 658-0191
Fax: (619) 658-0194
E-Mail: pagetech@tfb.com
Web Site: http://www.tfb.com/pagetech

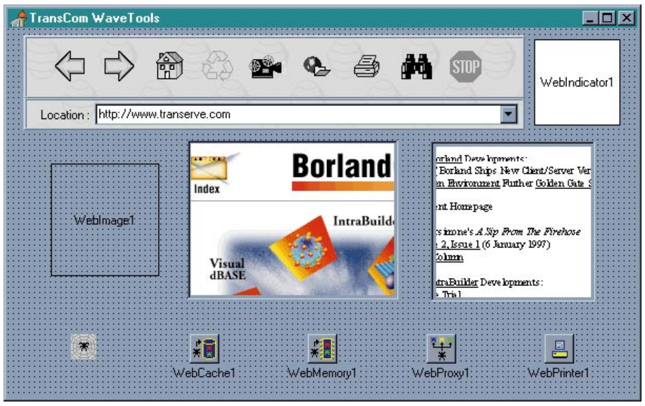
TransCom Software Inc. Releases TCP/IP Development Tool

TransCom Software Inc. of Castletown, UK has released *WaveTools*, a RAD TCP/IP development tool. WaveTools allows developers to build Internet/intranet-based applications with little or no knowledge of the underlying protocols and document formats.

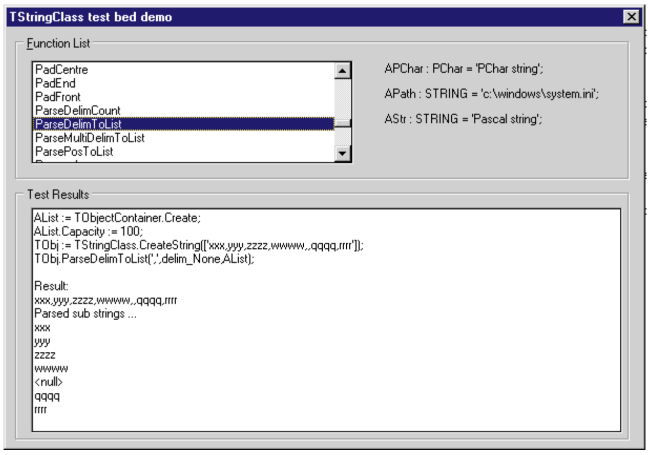
Created for Internet and intranet software developers using Delphi, WaveTools comes in native Delphi .DCU format, which means there are no other external support files. Developers can build and ship an application in a compact single file knowing it will run without any further user installation or intervention, and won't experience any version conflict with other previously installed software. Users can process the information from the HTML document they are retrieving, either locally or from the network, instead of simply viewing it.

WaveTools includes: HTTP connections over TCP/IP-enabled networks; facilities to deal with remote HTML documents as if they were disk-based files; and position- and pattern-based text-extraction engines. It also supports the saving of HTML documents with all images; floating images, even within tables; GIF87, GIF89a, and JFIF/JPEG image decompression with no external .DLLs; progressive display of decompressing images; as well as disk- and memory-based cache with user-definable size. In addition, WaveTools includes HTTP-based, multiple- file downloading capability; proxy support; print preview and print engine; and a large number of methods to control almost all aspects of all components.

Price: US$399
Contact: TransCom Software Inc., 11 Malew St., Castletown, Isle of Man IM9 1AB, UK
Phone: 41 (0) 22 849 8358
Fax: 41 (0) 1624 825384
E-Mail: sales@transerve.com
Web Site: http://www.transerve.com



ICFM Software Launches *TStringClass* for Delphi Developers



TStringClass test bed demo

Function List

```
PadCentre
PadEnd
PadFront
ParseDelimCount
ParseDelimToList
ParseMultiDelimToList
ParsePosToList
```

```
APChar : PChar = 'PChar string';
APath : STRING = 'c:\windows\system.ini';
AStr : STRING = 'Pascal string';
```

Test Results

```
AList := TObjectContainer.Create;
AList.Capacity := 100;
TObj := TStringClass.CreateString(['xxx,yyy,zzzz,wwww,,qqqq,rrrr']);
TObj.ParseDelimToList(',',delim_None,AList);

Result:
xxx,yyy,zzzz,wwww,,qqqq,rrrr
Parsed sub strings ...
xxx
yyy
zzzz
wwww
<null>
qqqq
rrrr
```

ICFM Software of London, England has released *TStringClass*, a large string management component.

The *TStringClass* class is designed to manage large string variables by encapsulating a core PChar type text buffer within a controlled class wrapper.

The *TStringClass* object controls its own internal buffer for holding the PChar variable and, before performing any assignments or concatenations, checks to see that sufficient room is available. If there is insufficient space, it re-sizes the buffer to fit the required action.

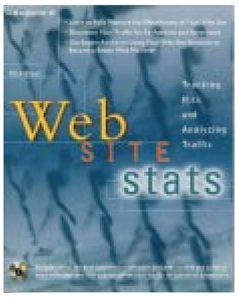
TStringClass has been expanded to cover almost anything that can be done with a string type variable, including: multiple constructors for initiating the object for different situations; a set of high-level methods for working with other *TStringClass* type variables; a range of different *Assign* methods for moving different types of text variables into the object data value; and a range of different *Append* methods for adding different types of text variables onto the end of any existing object data value.

TStringClass is designed to handle the problems of passing nil or zero length PChar parameters. It can also manage the problem of passing parameters which are uninitialized or remain in use after being disposed.

TStringClass includes a test bed application that illustrates main methods. A fully functional shareware product, it includes .DCU files for Delphi 1 and 2. *TStringClass* must be registered to obtain source code.

Price: US$45
Contact: ICFM Software, 12 Lightfoot Rd., London, England N8 7JN, UK
E-Mail: davout@dial.pipex.com
Web Site: http://dspace.dial.pipex.com/town/estate/ns21/icfmdc.htm

Web Site Stats
Rick Stout
Osborne/McGraw-Hill



ISBN: 0-07-882236-X
Price: US$29.95
(298 pages, CD-ROM)
Phone: (800) 262-4729

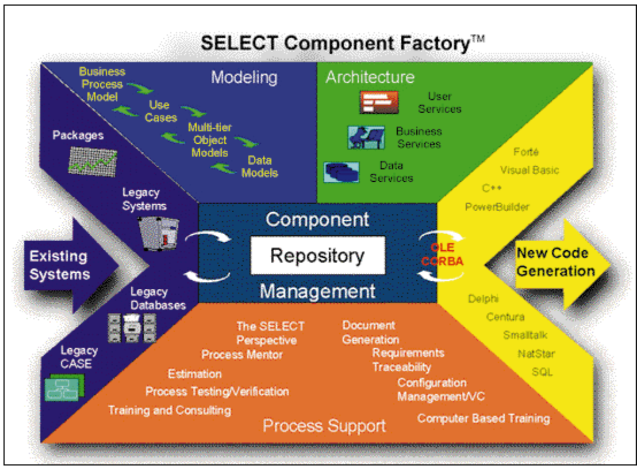
SELECT Software Tools Ships SELECT Component Factory

SELECT Software Tools, Inc. of Irvine, CA has released a line of products that support the wrapping of existing systems to create components that can be re-used.

SELECT has integrated the wrapping tools with its existing products, and named its overall product line the **SELECT Component Factory** (SCF). The wrapping tools provide the ability to create components from existing systems such as COBOL applications, databases, and application packages, and can integrate the components with new applications being developed with modeling tools such as SELECT Enterprise. Within SCF are analysis and design tools, wrapping tools, management tools, and a repository.

Analysis and design tools include: SELECT Enterprise, an object-oriented modeling toolset supporting integrated BPM and the Unified Modeling Language; SELECT Enterprise Generators, a round-trip code generation tool for Delphi, Visual Basic, PowerBuilder, and others; and SELECT SE, an industrial-strength application-development toolset providing integrated data modeling for SELECT Enterprise users.

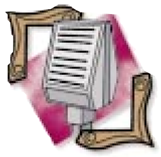
The wrapping tools include: SELECT CASE Wrapper, SELECT Database Wrapper, SELECT Legacy Wrapper, and SELECT Package Wrapper.



Price: Not available at press time.
Contact: SELECT Software Tools, Inc., 19600 Fairchild, Ste. 350, Irvine, CA 92612
Phone: (714) 477-4100
Fax: (714) 477-3232
E-Mail: ellenh@selectst.com
Web Site: http://www.selectst.com

News LINE

May 1997



Borland and Symantec Settle Trade Secret Litigations

Borland and Symantec Corp. have settled the trade secret lawsuit that Borland brought against Symantec, its President and Chief Executive Officer Gordon Eubanks, and former Executive Vice President Eugene Wang in September 1992.

The parties agreed to a dismissal of the lawsuit and related counterclaims, and have entered into mutual releases of all claims relating to the lawsuit.

Borland Appoints Birmingham as Vice President and General Counsel

Hobart Birmingham has been named Borland's vice president and general counsel. Birmingham, who reports to Borland's chairman and chief executive officer, Delbert W. Yocam, will manage legal services for Borland's world-wide operations. Birmingham replaces Robert Kohn who left the company in October 1996.

Borland Announces Restructuring Plan, Reduces Staff by 30 Percent

Scotts Valley, CA — Borland announced a world-wide restructuring plan aimed at returning the company to profitability in its 1998 fiscal year, which began April 1, 1997. The restructuring plan involves reductions in operational expenses, including a reduction of approximately 300 employees and contractors.

In addition, programs aimed at increasing the company's revenues are planned. Only selective restructuring was conducted in Borland's international operations, which continue to show positive results.

After the reductions in staffing, Borland will employ approximately 700 full-time employees world-wide.

This restructuring plan is anticipated to produce annual operating cost savings in excess of US$30 million, and product restructuring savings in excess of US$30 million.

Borland's restructuring program also features a renewed interest in new product development. Research and development programs at Borland plan to deliver at least one major release each quarter. New client/server and Internet/intranet products scheduled include C++Builder, Delphi 3, and JBuilder.

In anticipation of expanding the company's enterprise business, Borland is adding fee-based technical support programs.

Borland also downsized its Open Environment division, located in Boston. Key technical personnel were relocated to Scotts Valley, CA.

This restructuring plan serves as the foundation for Borland's new business organization.

A strategic business plan will be presented by Delbert W. Yocam, Borland's Chairman and CEO, at the Borland Developers Conference in Nashville, TN.

Three Editions of Borland's C++Builder Begin Shipping

Scotts Valley, CA — Borland has released the C++Builder product line, which includes the C++Builder Client/-Server Suite, C++Builder Professional, and C++Builder Standard editions.

C++Builder is a 32-bit Windows development environment that combines C++ with Borland's visual tools and database architecture.

C++Builder Client/Server Suite is for consultants, VARs, and C++ developers creating client/server solutions and connecting to data using a visual RAD tool for C++. It includes native SQL drivers for Oracle, Sybase, Microsoft SQL Server, Informix, DB2, and InterBase, as well as a suite of SQL tools. C++Builder features SQL Explorer, SQL Monitor, Visual Query Builder, Data Migration Wizard, Cached Updates, a four-user InterBase server for prototyping and testing multi-user SQL applications, and companion products.

C++Builder Professional is designed for programmers, ISVs, VARs, and consultants. Its toolset includes the VCL source code, advanced data-aware components, a scalable data dictionary, the Internet Solutions Pack, and other programming tools.

C++Builder Standard includes a tutorial, *Teach Yourself Borland C++Builder in 14 Days*, in-depth code examples, sample applications, and more.

In addition, Borland will update its current Borland C++ 5.0 product line.

C++Builder Client/Server Suite is US$1,999; C++Builder Professional is US$799 (current owners of C++, Delphi, Microsoft Visual Basic, Microsoft Visual C++, PowerBuilder, Optima++, Watcom C++, or Symantec C++ products are eligible to purchase Borland C++Builder Professional for a special upgrade price of US$299.95). C++Builder Standard is US$99.95.

Borland plans to ship a fourth version, Learn to Program with Borland C++Builder, for beginning programmers and students. Pricing was not available at press time.

For details, visit http://www.borland.com.

New Arabic Language Support for Delphi

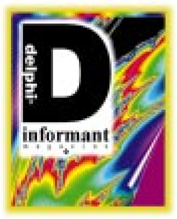
Scotts Valley, CA — Borland has released an Arabic language enablement for its Delphi 2 line of RAD tools.

By using this Arabic support package, any existing installation of Delphi 2 running on the Arabic version of Windows 95 can be upgraded to develop new Arabic applications.

Delphi now supports 14 languages: English, German, French, Japanese, Danish, Dutch, Italian, Portuguese, Spanish, Swedish, Korean, Chinese, Thai, and Arabic.

For details about Delphi Arabic Enablement, or to place orders, call (800) 233-2444. International customers should contact their local Borland office or distributor.

table of CONTENTS



By Danny Thorpe

Delphi 3 The ActiveX Foundry

From COM to DCOM, Delphi 3 Means Business

It's spring again, a time when many eyes look to Borland for a new generation of Delphi development tools. This spring brings us the third incarnation of Delphi, a lush garden of new tools and technologies designed to expedite business-critical data handling and analysis.

The most exotic fruits of Borland's year-long labors include the ActiveX Component Foundry, Business Object Broker, multi-tier Remote Data Brokers, Distributed COM, Web Deployment, and a suite of IDE features known collectively as Code Insight.

There are so many new technical bits in Delphi 3 that it's difficult to grasp the scope of the whole product simply by looking at its technical pieces. Let's first take a high-level tour, focusing on how Delphi 3's new tools enable new ways to solve tough development and deployment problems.

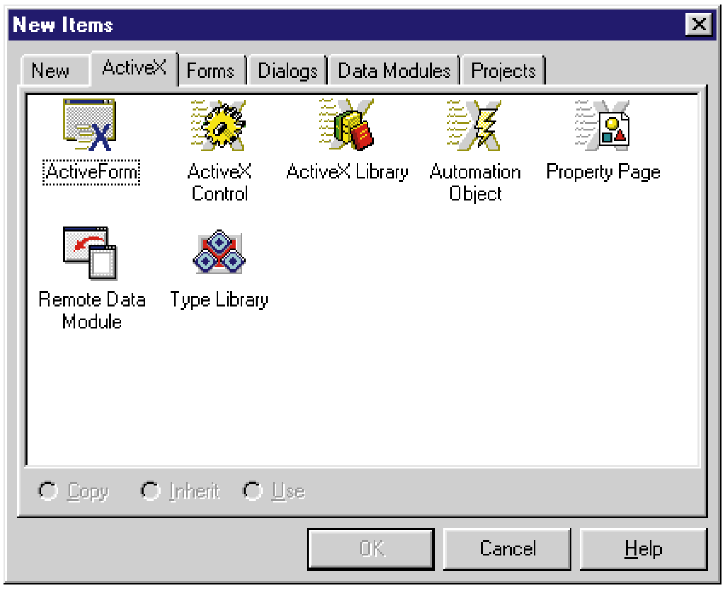


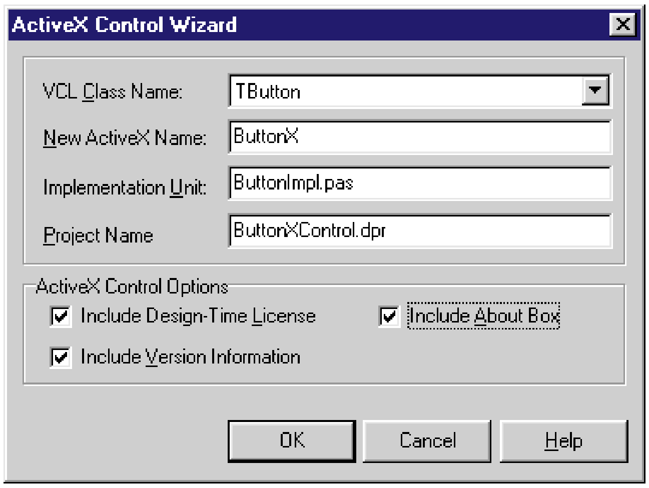
Figure 1: One aspect of the ActiveX Component Foundry, a page of wizards makes it short work to create an ActiveX control. Delphi 3 makes creation, debugging, deployment, and maintenance of ActiveX controls, COM servers, and COM interfaces simple and reliable.

ActiveX Component Foundry

Delphi 3 goes completely overboard to support, adopt, and internalize Microsoft's ActiveX technology initiative. Delphi 3 is to ActiveX creation as Delphi 1 was to Windows application creation. Through a combination of new language extensions, new classes, and new design-time wizards and tools (see Figure 1), Delphi 3 cuts through the Microsoft rhetoric to deliver what Microsoft has been trying to do for ages: a development environment that makes creation, debugging, deployment, and maintenance of ActiveX controls, COM servers, and COM interfaces simple and reliable.

Revisionist Terminology

Microsoft's Component Object Model specification, COM, is the standard to which all OLE objects are implemented. COM is the low-level stuff; OLE is a service built on COM. Depending on who you talk to at Microsoft, ActiveX is the new name for OLE Controls (OCXs, the 32-bit replacement for VBXs), the new name for all things formerly known as OLE, or the new name for all things new. Pessimists are already assuming the latter definitions. In any case, ActiveX is also a group of services and standards built on COM interfaces.



ActiveX Control Wizard

VCL Class Name: TButton

New ActiveX Name: ButtonX

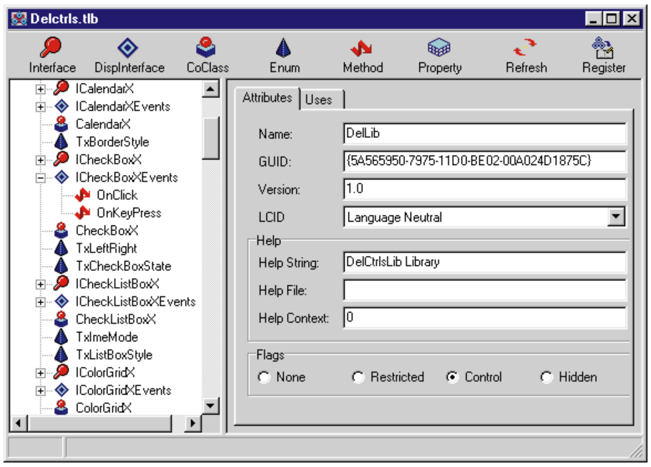
Implementation Unit: ButtonImpl.pas

Project Name: ButtonXControl.dpr

ActiveX Control Options

- [✓] Include Design-Time License
- [✓] Include About Box
- [✓] Include Version Information

Figure 2: Delphi 3 has a new wizard that generates an ActiveX control class wrapper around the VCL component you specify.



Delctrls.tlb

```
⊞ ICalendarX
⊞ ICalendarXEvents
  CalendarX
  TxBorderStyle
⊞ ICheckBoxX
⊟ ICheckBoxXEvents
    OnClick
    OnKeyPress
  CheckBoxX
  TxLeftRight
  TxCheckBoxState
⊞ ICheckListBoxX
⊞ ICheckListBoxXEvents
  CheckListBoxX
  TxImeMode
  TxListBoxStyle
⊞ IColorGridX
⊞ IColorGridXEvents
  ColorGridX
```

Attributes | Uses

Name: DelLib

GUID: {5A565950-7975-11D0-BE02-00A024D1875C}

Version: 1.0

LCID: Language Neutral

Help

Help String: DelCtrlsLib Library

Help File:

Help Context: 0

Flags: None [✓ Control] Restricted Hidden

Figure 3: A new OLE type library editor makes short work of defining a new COM interface definition, saving it into an OLE-standard typelib file, and generating a source code unit with the interface type declaration.

Create ActiveX Controls from VCL Components

Delphi 3 has a new wizard that generates an ActiveX control class wrapper around the VCL component you specify (see Figure 2). It's as simple as that. If you write VCL components for a living, you're now just a few button clicks away from selling those components as ActiveX controls to the Visual Basic and C++ markets. If you work in a mixed-tool environment, you can develop your core business objects as VCL components, then spit out ActiveX versions of your work for folks hopelessly shackled to other tools.

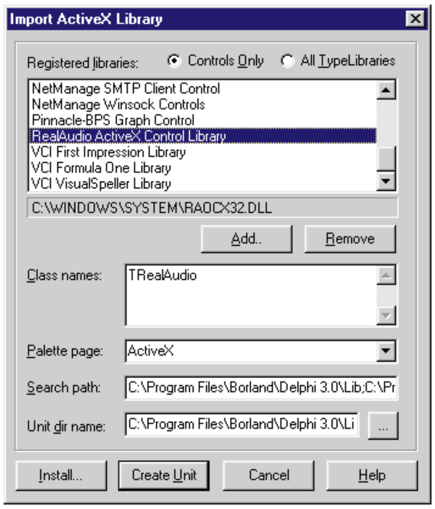
Create COM Servers and Automation Servers from Scratch

A new OLE type library (typelib) editor (see Figure 3) makes short work of defining a new COM interface definition, saving it into an OLE-standard typelib file, and generating a source code unit with the interface type declaration. You create new COM interfaces whenever you build a new COM server, be it a visual ActiveX control or a non-visual data processing server.

OLE typelibs are symbol files that tell other applications what methods are available in your COM server, and how to call them. Just as the IDE form designer and source code editor are linked two-way tools (i.e. modifications to one are reflected immediately in the other), the new typelib editor is also a two-way tool — modifications made to the Pascal interface **type** declaration source code are reflected in the typelib editor, and vice-versa.

Generate Pascal Declarations from Typelibs

As a side effect of the extensive typelib editor work, you can also generate Object Pascal source-code constants and interface type declarations from any OLE typelib. Think of this as an expansion of the OCX wrapper class generation in Delphi 2. If you can obtain a typelib file for a COM object you want to use in Delphi (ActiveX controls are required to have a typelib), creating Pascal interface declarations to use that COM object are a snap, and considerably more accurate than trying to mechanically convert ambiguous C header files into Pascal declarations. Unfortunately, some COM objects do exist without typelibs; Microsoft's DirectX is probably the biggest offender in this category.



Import ActiveX Library

Registered libraries: [✓ Controls Only] All TypeLibraries

- NetManage SMTP Client Control
- NetManage Winsock Controls
- Pinnacle-BPS Graph Control
- RealAudio ActiveX Control Library
- VCI First Impression Library
- VCI Formula One Library
- VCI VisualSpeller Library

C:\WINDOWS\SYSTEM\RAOCX32.DLL

Class names: TRealAudio

Palette page: ActiveX

Search path: C:\Program Files\Borland\Delphi 3.0\Lib;C:\Pr

Unit dir name: C:\Program Files\Borland\Delphi 3.0\Li

Figure 4: You can now directly access interfaces provided by ActiveX controls and take advantage of other control features that were previously only available through variant variables.

Create VCL Components from ActiveX Controls

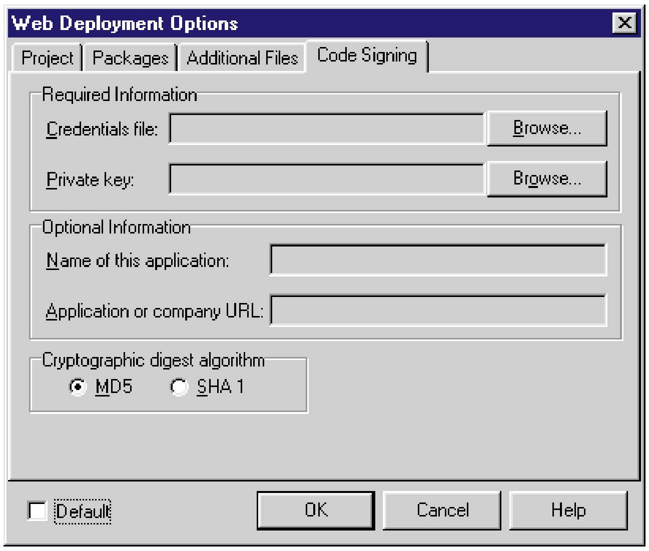
The ActiveX wrapper class generation found in Delphi 2 has been expanded to take advantage of new language features like interface types, and new buzzwords like ActiveX. You can now directly access interfaces provided by ActiveX controls and take advantage of other control features that were previously only available through variant variables (see Figure 4).

ActiveForms

Another nifty spin-off of the core ActiveX development work is the creation of an ActiveX control to encapsulate an entire Delphi form (again, see Figure 1). This is necessary to support ActiveX property pages, but it's also handy for creating mini-application modules that can be automatically downloaded over the Internet and displayed inside a Web browser such as Microsoft Internet Explorer 3.0. The Web browser sees the thing as an ActiveX control, but you can pack an entire application into it.

Web Deployment

To support Web-deployed ActiveX controls, the Delphi 3 IDE includes tools to digitally sign and seal your .DLL or .EXE file with your Software Publisher digital certificate, and



Web Deployment Options

Project | Packages | Additional Files | Code Signing

Required Information

Credentials file: Browse...

Private key: Browse...

Optional Information

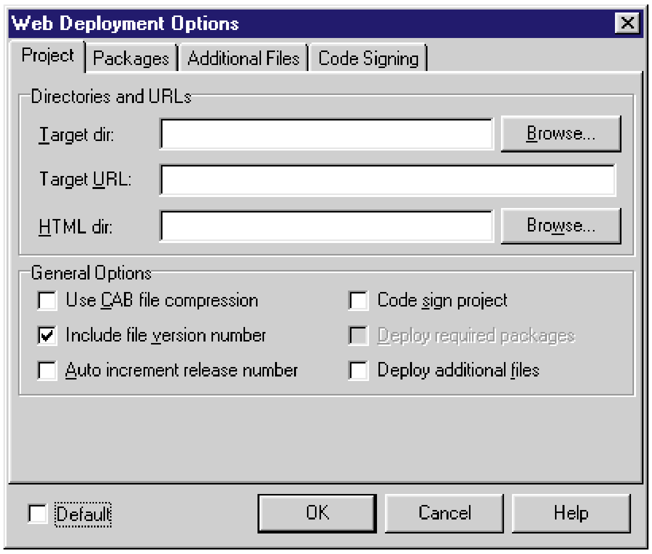
Name of this application:

Application or company URL:

Cryptographic digest algorithm

[✓ MD5] SHA 1

☐ Default — OK — Cancel — Help



Web Deployment Options

Project | Packages | Additional Files | Code Signing

Directories and URLs

Target dir: Browse...

Target URL:

HTML dir: Browse...

General Options

☐ Use CAB file compression
☑ Include file version number
☐ Auto increment release number
☐ Code sign project
☐ Deploy required packages
☐ Deploy additional files

☐ Default — OK — Cancel — Help

Figure 5 (Top): The new Web Deployment Options dialog box. To support Web-deployed ActiveX controls, Delphi 3 can digitally sign and seal your .DLL or .EXE file with your Software Publisher digital certificate, and deliver it to a directory you select.
Figure 6 (Bottom): The Web deployment wizard can also bundle and compress multiple files into the Microsoft .CAB file format, generate .INF files needed for a downloadable component to refer to required modules downloadable separately, and much more.

deliver it to a directory of your choosing (see Figure 5). This signature is checked by the Microsoft Internet Explorer 3.0 Web browser after downloading the ActiveX control as part of an HTML document to verify that the file is from who it says it's from, and that the file has not been tampered with or corrupted in transfer.

The IDE deployment wizard can also bundle and compress multiple files into the Microsoft .CAB file format, generate .INF files needed for a downloadable component to refer to required modules downloadable separately, and generate an HTML object tag for you to paste into your Web page, to refer to your downloadable component (see Figure 6).

Distributed COM

Delphi 3 supports Distributed COM, Microsoft's newest implementation of COM that enables an application on one machine to talk to an application on another across the network wire. Basically, DCOM is the heir-apparent to Remote Procedure Calls (RPC). Delphi 3's remote datasets use DCOM to make the hop from the client machine to the middle-tier data broker. You can implement your own middle-tier business logic by creating a COM server in Delphi 3, and call its methods using interfaces in the client application. DCOM takes care of the network transport; you just have to ask for it.

New Interface Type

OLE objects are always accessed through COM interfaces — abstract, virtual, base classes that define a group of related functions, but not their implementation. All COM interfaces are derived from the *IUnknown* standard interface, which defines simple reference-counting methods and a *QueryInterface* method to gain access to other interfaces supported by that object.

The most common programming error when using OLE objects is forgetting to increment or decrement the reference count of interfaces onto which you're holding. If you forget to call *AddRef* on an interface, the object behind that interface may delete itself if some other action causes the object's reference count to drop to zero. Subsequent use of the interface pointer you held onto will cause an access violation. If you forget to call *Release* on an interface when you're finished with it, the object behind that interface will remain in memory, because its reference count is artificially inflated.

Delphi 3 eliminates this debugging nightmare by adding a new standard type to the Object Pascal language definition: the *interface* type. An interface **type** declaration looks much like a class **type** declaration, but an interface type is only a declaration — it has no implementation of its own. An interface is like a standardized subset of methods that an object can implement. Regardless of what else the implementing object implements, you know that if it implements the *XYZ* interface, you can use the *XYZ* methods on it. Furthermore, you can obtain the *XYZ* interface from the implementing object, and use it without knowing anything about the implementor's class type.

In use, interface variables are initialized, reference-counted (through standard *IUnknown* methods), and released automatically by compiler-generated code, just as long strings and variants are dynamically allocated, reference-counted, and released in Delphi 2. In Delphi 3, transferring values between interface variables or passing them as parameters is as simple and reliable as transferring integer or string values. In many respects, passing interface values is safer than passing object instances, because interface reference-counting eliminates the question of who is responsible for freeing the object.

MI = Multiple Interfaces, *Not* Multiple Inheritance

The flip side of interfaces is how they are implemented by an object. An OLE object may support many different interfaces, such as for streaming, printing, or drawing on the screen. Delphi 3 extends the declaration syntax of the class type, so you can declare a class as an implementor of one or more interface specifications. The Delphi compiler takes care of binding declared methods in the interface types to implemented methods in the class type. No tables of macros of pointers to functions to crosswire with a typo — the compiler does it all. When something isn't quite right, the Delphi compiler tells you where and what you've missed in your declarations. For example, an interface-implementing class must implement all methods declared in the interface type. If you forget one of the interface methods, the compiler will remind you, just as it reminds you when you declare a method in a class type, but forget to give it a method body.

When a class implements an interface, instances of that class type are assignment-compatible with variables of the interface type. You can take an object instance and assign it to an interface variable, and the compiler will do the magic of extracting the correct interface pointer from the object instance automatically. You can also do late-bound (run-time) interface extraction using the **as** typecast operator, which calls the implementor's *QueryInterface* method to obtain the desired interface at run time.

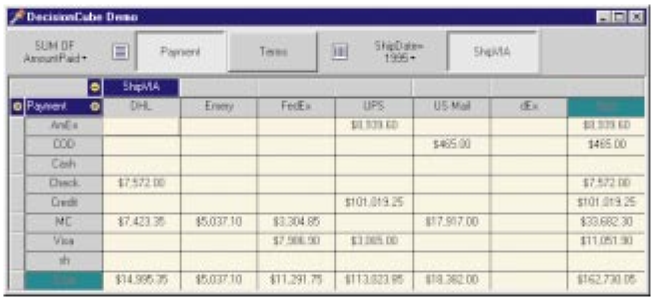
Note that while an object may implement multiple interfaces, this is not the same as multiple inheritance; you are not inheriting any implementation details from the multiple interfaces. What this means is that implementing OLE objects (such as ActiveX controls and custom COM servers) is now almost trivial. Delphi 3 requires none of the unintelligible tables of macros of pointers that Microsoft's ActiveX SDK heaps on itself. Where Microsoft implements ActiveX as a system of macros and C++ template classes on top of the C/C++ language, Delphi implements ActiveX by incorporating the essential enabling technologies into the Object Pascal language and VCL classes. Why bother with macros and well-meaning source code conventions when you can have the compiler do the dirty work for you?

Data Visualization

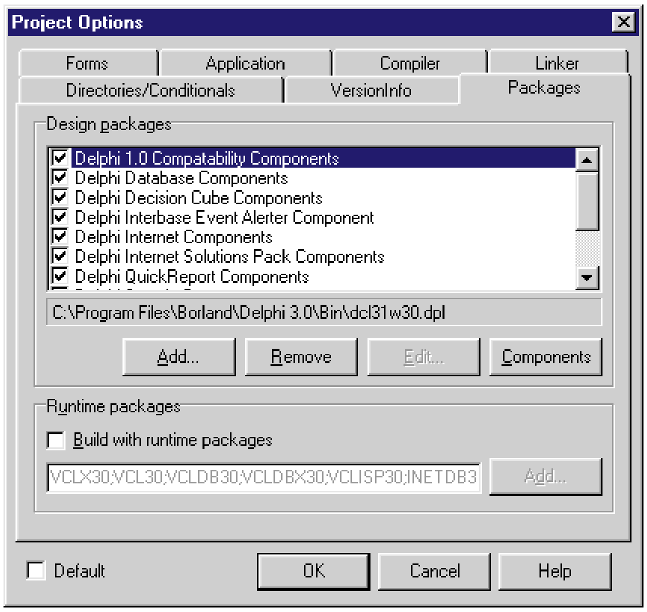
Delphi 3's Component palette includes three new heavy-hitters: an all-new version of QuickReport, powerful charting capabilities in TeeChart, and the DecisionCube interactive crosstab (see Figure 7). You can embed TeeCharts in QuickReport reports, as well as link a TeeChart to the DecisionCube to graphically display the crosstab data on-the-fly.

Application Deployment: Web or Otherwise

If you've ever installed multiple Delphi applications on the same machine, you've probably wondered if there was some way to share the VCL component code between the multiple applications. Well, now there is: a Delphi package. A *package* is a special .DLL which contains and exports one or more units for applications or other packages to share. A package is different from a .DLL in that it's Delphi-specific



ShipVIA / Payment	DHL	Emery	FedEx	UPS	US Mail	dEx	[illegible]
AmEx				$8,939.60			$8,939.60
COD					$465.00		$465.00
Cash							
Check	$7,572.00						$7,572.00
Credit				$101,019.25			$101,019.25
MC	$7,423.35	$5,037.10	$3,304.85		$17,917.00		$33,682.30
Visa			$7,986.90	$3,065.00			$11,051.90
sh							
[illegible]	$14,995.35	$5,037.10	$11,291.75	$113,023.85	$18,382.00		$162,730.05



Design packages
- ☑ Delphi 1.0 Compatability Components
- ☑ Delphi Database Components
- ☑ Delphi Decision Cube Components
- ☑ Delphi Interbase Event Alerter Component
- ☑ Delphi Internet Components
- ☑ Delphi Internet Solutions Pack Components
- ☑ Delphi QuickReport Components

C:\Program Files\Borland\Delphi 3.0\Bin\dcl31w30.dpl

Runtime packages
- ☐ Build with runtime packages

VCLX30;VCL30;VCLDB30;VCLDBX30;VCLISP30;INETDB3

☐ Default

Figure 7 (Top): Delphi 3's Component palette includes three new heavy-hitters, including the powerful DecisionCube.
Figure 8 (Bottom): The new Packages page of the Project Options dialog box. A package is a special .DLL which contains and exports one or more units for applications or other packages to share.

(non-Delphi applications should not try to link directly to a package .DLL) and you don't have to change any Delphi source code to use it.

In compiler parlance, packaging is a code-generation option (see Figure 8), which means it should have no effect on the semantics of your source code. When your Delphi application is compiled to use the VCL core package, for example, the compiler generates code to reference the Forms unit in the VCL package .DLL instead of placing the Forms unit code in your .EXE. The result is that your .EXE size drops from around 200KB to less than 20KB. With packages, the .EXE file contains only your application logic and form resources. This also makes ActiveX control .DLLs extremely small — about 25KB — far smaller than Microsoft's 50KB minimum ActiveX template-based control library, or 800KB minimum, MFC-based ActiveX control library. The package .DLL must contain every bit of code and data that its member units define in their interface sections. This makes the core VCL package weigh in at just over 1MB. (Because most units in the core VCL package are used by the simplest blank form application, and that minimal application produces a 150KB .EXE file, that should tell you something about the value of smart linking.)

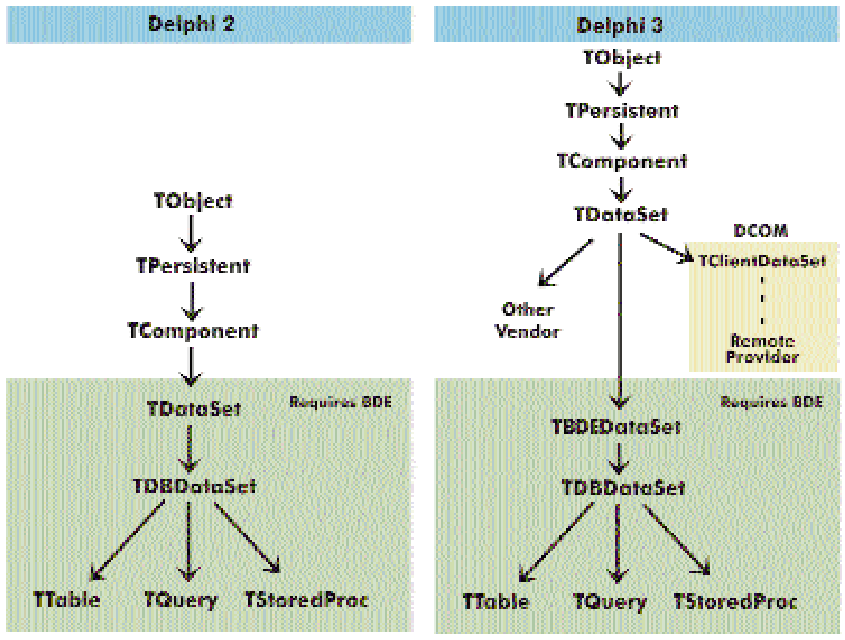


Figure 9: Class hierarchy of *TDataSet* for Delphi 2 and 3. In Delphi 3, *TDataSet* is now abstract and has a new ancestor, *TBDEDataSet*. The Delphi 3 BDE now features direct links to Access and DB2, as well as dBASE, Paradox, ODBC, Informix, InterBase, Microsoft SQL Server, Oracle, and Sybase.

Packages are a great way to reduce the overall size of a suite of applications, and open up interesting options for such bandwidth-sensitive applications as ActiveX controls deployed over the Internet (as objects embedded in HTML documents) or network-deployed shareware. The common packages could be bundled separately from the main application file set, so that folks who already have the packages don't have to download them again. Better yet, you could refer to the Borland Web site as the source for the Delphi core packages instead of bundling them yourself, and consuming disk space on your file server.

Code Insight

How many times have you started to write a function call statement, but forgotten what parameters that function call requires? Wouldn't it be great if you could type a function name and hit a hotkey to show the function's parameter declaration, right there in the editor? Wouldn't it be great if it showed the functions you created, as well as the Borland-documented stuff? Wouldn't it be wild if it would help you fill in the parameters too?

Delphi's Code Insight provides all this, and more. Its Code Parameters feature uses the compiler to determine what function you're trying to use, what its parameters are, and the types of those parameters. Moreover, it's nearly instantaneous and non-intrusive. (Beware of similar-sounding features in other products, which only give you help on functions defined by the tool vendor. Delphi uses the compiler symbols to give you help on all functions in your project — Borland's, yours, and all third-party units used by your project.)

Using code templates, you can define standard code blocks (**if/then/else**, **begin/end**, **for** and **while** loops, etc.) with short-cut names to insert in the editor with just a keystroke or two.

The Code Completion feature helps you enter field names and parameter values by displaying a pop-up list of identifiers that are type-compatible with the source code expression to the left of the editor cursor. For example, typing:

```
"Caption := IntToStr(ProgressBar1."
```

and pressing a hotkey will show all the integer properties and functions available on the form's ProgressBar1 component. The helper knows that ProgressBar1 is a component, and that *IntToStr* requires an integer type parameter, so it shows you the things in ProgressBar1 that can provide an integer-compatible value. For debugging, the ToolTip Expression Evaluation feature shows the values of variables in a hint balloon as your mouse moves over the source code symbols in the Code Editor. I may never use the Evaluate/Modify dialog box again!

To make it easier to debug ActiveX control .DLLs and COM servers, particularly when they are used by non-Delphi applications, you can tell the Delphi IDE debugger to run a particular .EXE to debug the current .DLL project. So, you can compile your ActiveX control .DLL project, set a few breakpoints in the source code, tell the debugger that the host .EXE for your .DLL is VB.EXE, select **Run | Run**, and Visual Basic is displayed. Tell VB to load your ActiveX control .DLL and execution stops at a breakpoint in the Delphi debugger. You can step, evaluate, watch (and so forth) items in your .DLL while it's being used by VB.

Virtualized Datasets

To enable BDE-less remote datasets (and to respond to a common customer request), Delphi 3 virtualizes all database activity through a — now abstract — *TDataSet* class. BDE awareness is introduced in a new *TDataSet* descendant, *TBDEDataSet*, which serves as the ancestor of *TDBDataSet* and the familiar *TTable*, *TQuery*, and *TStoredProc* classes (see Figure 9).

Delphi 3 also implements support for "thin-client" remote datasets as a descendant of the base *TDataSet* class, independent of the BDE. This abstraction of the dataset will also enable third parties to implement Delphi dataset support for other data providers and file formats, without resorting to fate-tempting BDE .DLL hacks.

Breaking Up Client/Server

Seasoned SQL database folks can rattle off all sorts of weaknesses and liabilities of the industry-standard two-tier SQL client/server application model. For example, the client machine and application are often intimately bound to the SQL server's network name and SQL dialect or vendor. BDE aliases allow you to re-vector server references on a client machine without recompiling the client application, but those aliases are still *on the client machine*. If your SQL server goes down and you have to prop up your business with a backup machine, how do you make all your clients automatically talk to the backup machine instead?

Another problem with two-tier is related to centralization of business rules and data policies. In the standard two-tier SQL model, the rules that determine data relationships and links within the database must be implemented on either the server or the client. SQL has proven itself to be an adequate tool for describing and managing data, but is terrible for implementing the programming logic required for complex business rules, such as non-tabular tax calculations or least-cost resource allocation. This means enterprise-wide business rules tend to be implemented in the client application instead of on the centralized server, inflating the size of the client application and creating a maintenance liability.

Multi-Tier Remote Data Brokers

The solution to these and many other weaknesses of the traditional two-tier SQL model is to break the direct connection between the client application and the server. Multi-tier data models make the client application talk to an intermediate machine or service (a broker), which can then process or forward the information to an appropriate server. The client never talks directly to the final SQL server that owns the actual data, so the client application doesn't need to know how to talk SQL — the client application can speak simply and frankly to the intermediate data broker, and the broker can carry the burden of speaking SQL to the data servers, and fret with maintaining connections to multiple data servers — SQL and otherwise. Because business rules and other data-handling logic can live on a middle tier broker, a significant portion of what you've been calling your client application can be moved off the client machine and onto centrally managed servers. What's more, the middle-tier broker can be implemented — and debugged — using real programming tools (such as Delphi, of course) instead of primitive SQL stored procedures (see Figures 10 and 11).

Delphi 3 opens the floodgates to multi-tier distributed application development with the introduction of the remote dataset. A remote dataset looks and acts like any other dataset (*TTable*, *TQuery*), serving rows of data to data-aware controls. The difference is that a remote dataset doesn't require the presence of a full database engine on the client machine — the dataset talks to a second machine (the middle-tier data broker) that contains the database engine, complete with querying, filtering, and SQL connectivity intelligence. With no BDE to install or configure on the client machine, remote datasets enable you to reduce the size and complexity of your client application's file set by an order of magnitude.

With only a handful of middle-tier machines connecting to your SQL servers, Delphi 3's remote datasets could save you an enormous amount of money in SQL server connection licenses alone. Disconnecting the client from the server also opens many options for failover and server load balancing, simply by causing the middle-tier broker to forward client requests to the least busy machine in a server farm.

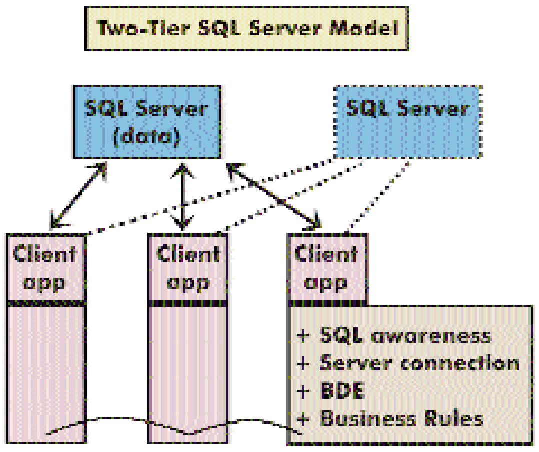


Figure 10: The two-tier SQL server model.

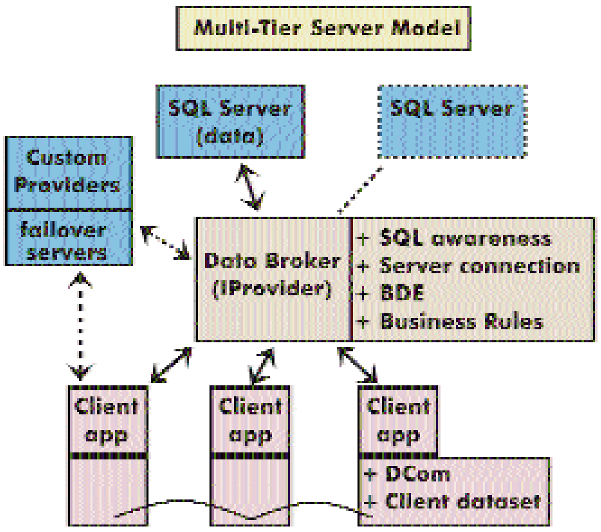


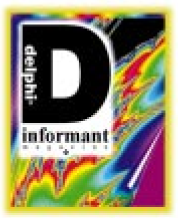
Figure 11: The multi-tier server model.

Conclusion

There are too many exciting new features in Delphi 3 to cover in one article, or even to try to absorb in one sitting. I'd love to rattle on about the Web-server dispatch and database components for building Netscape and Microsoft Internet Information Server extension .DLLs, or the extensive support for DIB image formats and direct pixel memory pointers in *TBitmap*, the new *TJPEGImage* class, or the new "globalization" of the Delphi RTL and VCL classes to support multi-byte character sets in Asian locales, or the all-new documentation set and online Help, but for now a tease will have to do. So many ideas, so little time. ∆

***Important note:** This article is based on a prerelease version of Delphi 3. Features may differ or be absent in the shipping version.*

Danny Thorpe is a Delphi R&D engineer at Borland. He has also served as technical editor and advisor for dozens of Delphi programming books, and recently wrote *Delphi Component Design* [Addison-Wesley, 1997] on advanced topics in Delphi programming. When he happens upon some spare time, he rewrites his to-do list manager to ensure that it doesn't happen again.



By Robert Vivrette

Easier Yet

A Look at the Delphi 3 Code Editor

When car shopping, you're first drawn by outward appearance. Then you get close enough to check out the details of the trim, upholstery, and dashboard. Then you open the door and ease into the driver's seat, maybe push a few buttons, or check out the stereo. Car manufacturers strive to make your driving experience more enjoyable, so they're always trying to match the car's functionality to consumer desires.

It's the same for software. Users demand a better "look," an easier-to-use interface, and powerful new features that make their lives simpler. This is certainly the case with the enhancements to the Code Editor of Delphi 3 — and Borland has delivered.

A New Flat World

Even before you open the Code Editor, one of the first things you'll notice when launching the Delphi 3 IDE is the flat, "buttonless" appearance of the SpeedBar and Component palette (see Figure 1). This look is becoming popular in new Microsoft applications, such as Internet Explorer and Office 97.

At first glance, it appears the buttons don't have edges. However, when the mouse moves over them, their edges appear to "pop up." Each button has its normal fly-over hint as well. Some may question the value of borderless buttons. The more I work with them, however, the more I appreciate the clean, uncluttered appearance. To make this look available to the Delphi applications you create, the SpeedButton component now has a Boolean property named *Flat*.

In the Gutter

The first difference you'll notice when you open the Code Editor is the inclusion of a "gutter" on its left side. We've all experienced trying to select a line of text by clicking near the front of the line, only to set a breakpoint instead. By adding this gutter, the new IDE provides a clickable area for setting breakpoints, bookmarks, and the like, as well as providing an area to show various status glyphs.

You'll also notice one or more glyphs next to each line of code (see Figure 2). The small blue dots appear after you compile or build, on each line for which code is generated. This is a particularly useful aid in setting breakpoints.

As a test, I placed:

```
if False then
```

just above the *ShowMessage* statement shown in Figure 2. When it compiled, the linker knew this code would never be called, so it didn't link it in. As a result, no blue dots appeared on

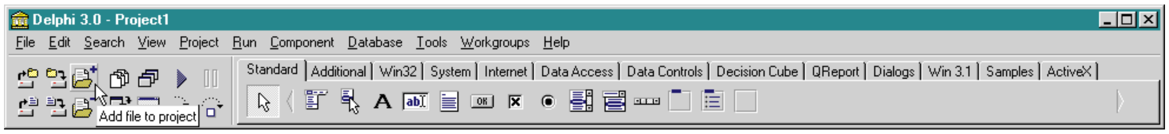
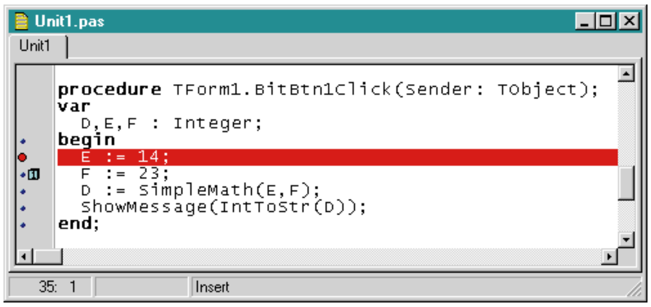
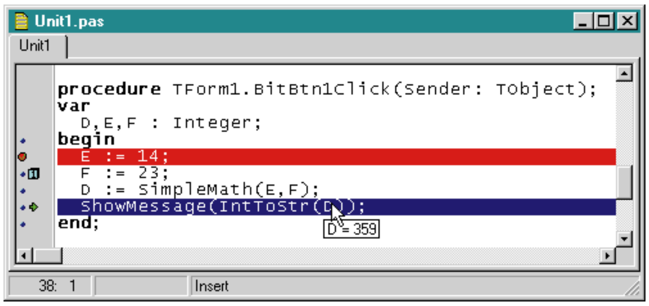

Delphi 3.0 - Project1
File Edit Search View Project Run Component Database Tools Workgroups Help
Standard Additional Win32 System Internet Data Access Data Controls Decision Cube QReport Dialogs Win 3.1 Samples ActiveX
Add file to project

Figure 1: The new look of Delphi 3 features the flat buttons made popular by Microsoft Office 97.



```
procedure TForm1.BitBtn1Click(Sender: TObject);
var
  D,E,F : Integer;
begin
  E := 14;
  F := 23;
  D := SimpleMath(E,F);
  ShowMessage(IntToStr(D));
end;
```

Figure 2: The Code Editor now features a "gutter" with breakpoints, bookmarks, linked status, etc. indicated with glyphs.



```
procedure TForm1.BitBtn1Click(Sender: TObject);
var
  D,E,F : Integer;
begin
  E := 14;
  F := 23;
  D := SimpleMath(E,F);
  ShowMessage(IntToStr(D));
end;
```

Figure 3: The new ToolTip Expression Evaluation displays the values of variables and properties at run time.

either line. If I changed the `False` to `True`, the *ShowMessage* line was linked back in, but the

```
if True then
```

line remained linked out (i.e. no blue dot). This is because the *ShowMessage* line will always occur, so there's no point in compiling the logic to jump around it.

Then I set bookmark 1 on the statement:

```
F := 23;
```

which made an appropriate notation in the gutter.

Bookmarks are used to save your place in a piece of code (by pressing Ctrl Shift and a single number from 0 to 9), then allow you to jump back to these saved locations later (by pressing Ctrl and the same single number). Although bookmarks aren't new with Delphi 3 (they've been there since Delphi 1), the gutter provides a better place for bookmark glyphs, keeping them out of the way of source code in the main editing pane.

The gutter is a property of the Code Editor. It's on by default, but can be turned off if you like. You can also adjust its width.

Drag-and-Drop

Another new feature in the Code Editor is support for drag-and-drop text editing (*à la* Word). You can now select a section of text, grab it with the mouse, and drag it somewhere else. If you hold down Ctrl while doing this, the text is copied rather than moved.

Code Insight

One of the great time savers in the new IDE is a group of features named Code Insight. This term refers to four specific enhancements made to the Delphi 3 Code Editor: ToolTip Expression Evaluation, Code Templates, Code Completion, and Code Parameters. Let's look at each enhancement.

ToolTip Expression Evaluation

When you're debugging your code at run time, you might step through the program logic using F7 or F8. The Delphi 3 IDE adds a handy new feature called ToolTip Expression Evaluation (fly-over evaluation) hints. At run time, you can position the mouse over a variable, and it will display that variable's current value (see Figure 3). This also works on constants and parameters passed to a function or procedure. If no value appears, the value hasn't been defined, or optimization was turned on and the variable isn't available.

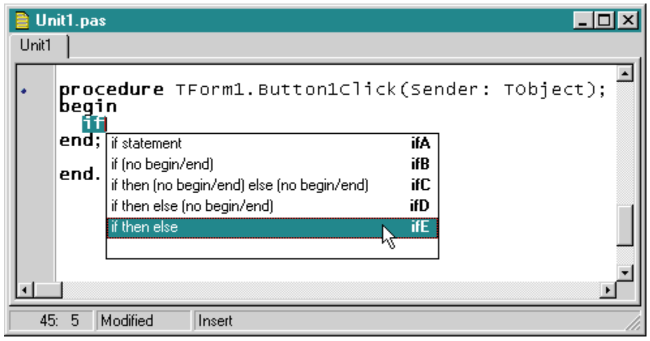
Evaluation hints can also show the properties of an object. For example, if you had a line of code with a reference to `BitBtn1.Caption` and you placed the cursor over the `Caption` portion, it would give you a hint showing the caption's current setting. However, if you position it over the `BitBtn1` part, it would give you the reference to the visible properties of `BitBtn1`. Essentially, you get the same information in a fly-over evaluation hint that you would normally see in the standard Watch window.

Code Templates

Another nifty time-saver of the new IDE are Code Templates. A Code Template is a defined structure of code that you can access through a menu, or by means of a shortcut mnemonic.

The best way to understand how it works is to see it in use. In Figure 4, I began typing code for a *ButtonClick* event. I want to use an **if** statement, so I enter `if`, and call the Code Template by pressing Ctrl J. Delphi then presents a list box of all **if** statements I've defined, allowing me to pick one. When I choose one, it inserts the text associated with that template (see Figure 5).

You can easily define new templates, or modify existing ones on the Code Insight page of the Environment Options dialog box (see Figure 6). For each template, you

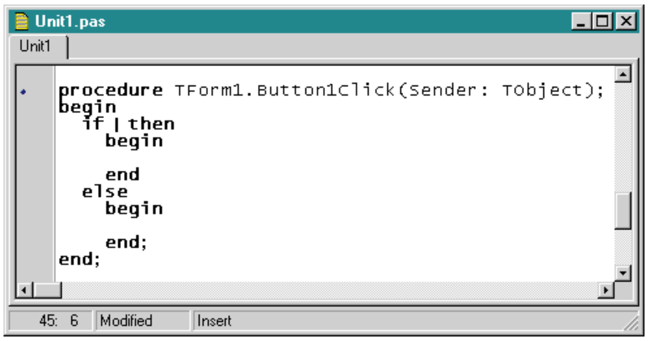


```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if
end;

end.
```

if statement	ifA
if (no begin/end)	ifB
if then (no begin/end) else (no begin/end)	ifC
if then else (no begin/end)	ifD
if then else	ifE

Figure 4: The new Code Template feature relieves the tedium of oft-typed code sequences.

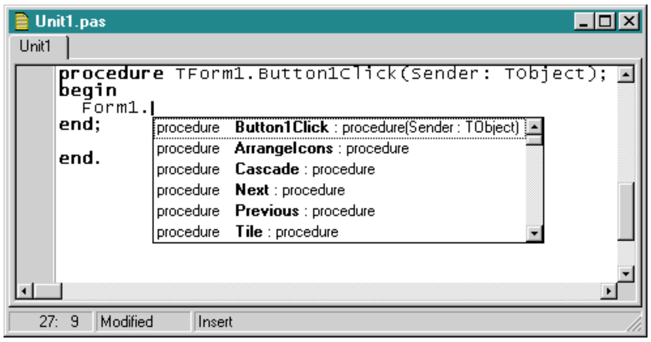


```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if | then
    begin

    end
  else
    begin

    end;
end;
```

Figure 5: The result of choosing a Code Template.



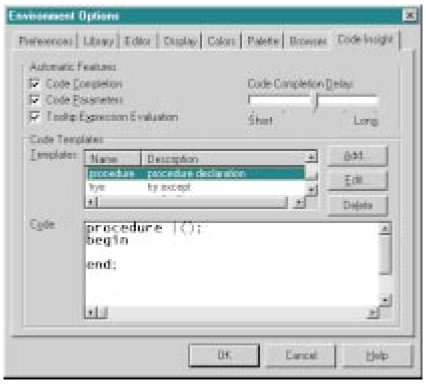
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.|
end;

end.
```

```
procedure   Button1Click : procedure(Sender : TObject)
procedure   ArrangeIcons : procedure
procedure   Cascade : procedure
procedure   Next : procedure
procedure   Previous : procedure
procedure   Tile : procedure
```

Figure 7: The new Code Completion feature offers valid procedures, functions, properties, and variables while you type.

define its description text (e.g. *`if then else (no begin/end)`*), the code you want inserted, and a mnemonic. The mnemonic is used by the IDE to filter the list of available templates before presenting them to you. For example, because I only typed the letters `if` in the previous example, it presented templates that had mnemonics starting with those letters. If I had known that I wanted the template associated with the "ifE" mnemonic, I could have typed `ifE` and it would have simply inserted the code, without presenting the list (the list would have had only one choice anyway).



☑ Code Completion
☑ Code Parameters
☑ Tooltip Expression Evaluation

Name	Description
procedure	procedure declaration
trye	try except

```
procedure |();
begin

end;
```

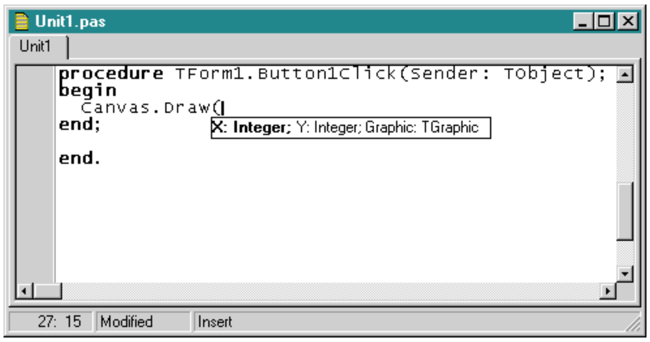
Figure 6: You can create, modify, or delete Code Templates.

You don't need to type anything before calling the Code Templates list. In that case, it will show all available templates. Delphi 3 includes a wide range of pre-defined Code Templates, from **if** statements, to **for** loops, to procedure and function headers — even class declarations. And nothing stops you from changing them; you can modify existing templates, delete ones you don't want, and of course, add new ones. If you don't like the mnemonic codes Borland uses, you can create your own. For each template, you can even define where the cursor will be left after inserting the code — a very nice touch.

Code Completion

The third portion of Code Insight is the Code Completion feature. Whenever you enter a class name followed by a period, you will be presented with a list of properties, methods, and events appropriate to the class. You can then select an item, and it will be entered into the code. Figure 7 shows Code Completion presenting a list of the methods, properties, and events appropriate for a Form object.

Furthermore, when you enter an assignment statement and press Ctrl Space, a list of valid variables is displayed.



```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Canvas.Draw(|
end;        X: Integer; Y: Integer; Graphic: TGraphic

end.
```

Figure 8: The new Code Parameters feature helps you remember valid parameters without consulting the Help system.

Just select the variable you want; it will be entered into your code automatically.

Code Parameters

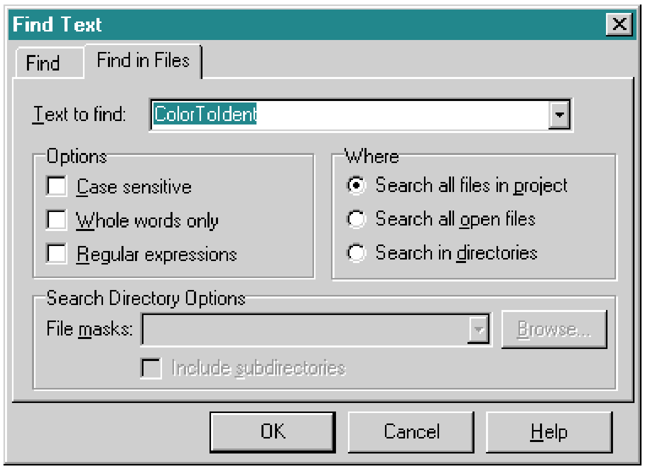
The fourth piece of Code Insight is the Code Parameters feature, which allows you to view the required arguments for a method as you type it. Code Parameters is triggered when you enter a method name followed by an opening parenthesis (see Figure 8).

In Figure 8, Code Parameters is activated for the *Draw* method of the form's *Canvas*. There are three parameters for this method: the X and Y coordinates, and the graphic to be drawn. Code Parameters even highlights the particular parameter you're working with. As you move the cursor around in the Code Editor, the Code Parameters feature keeps up with you, highlighting the appropriate parameter.

Even more interesting, Code Parameters knows the difference between Windows API calls and Delphi wrapper methods that happen to have the same name. For example, if you were to type `Canvas.Rectangle`, it knows that you are talking about Delphi's *Rectangle* method, which only requires the coordinates of the four corners of the rectangle. However, if you just enter `Rectangle`, it knows you're talking about the **Rectangle** function in the Windows API, which requires completely different parameters.

Conducting a Thorough Search

The last enhancement I want to mention was that made to its find facility. In previous versions, you can find text in one file at a time. In Delphi 3, however, the Find Text dialog box has a


Find Text

Find | Find in Files

Text to find: ColorToIdent

Options
- [ ] Case sensitive
- [ ] Whole words only
- [ ] Regular expressions

Where
- [✓ Search all files in project]
- Search all open files
- Search in directories

Search Directory Options

File masks: Browse...

- [ ] Include subdirectories

OK | Cancel | Help

Figure 9: With Delphi 3's enhanced text search capabilities, you'll no longer have to turn to Windows 95 to find all instances of a variable in your project.

Find in Files page which allows you to perform a search in all project files, all open files, or even in a list of specified directories (see Figure 9). And if the cursor is on a word, that word is automatically placed in the Text to find edit box — another slick feature that makes a programmer's life just a little easier.

Conclusion

The enhancements to the IDE in Delphi 3 will save a lot of time and effort for developers. The new Code Editor gutter provides visual clues about breakpoints, bookmarks, lines that generate code, the program execution point, and more. The Code Insight system is a real time-saver that allows you to quickly insert common code structures into the Editor, and helps with parameters to method calls. The ToolTip Expression Evaluation hints help you examine variables at run time in a manner more convenient than the standard Watch window.

On top of an already powerful language and compiler, these IDE improvements really will make Delphi 3 a more functional and productive development system. ∆

Important note: *This article is based on a prerelease version of Delphi 3. Features may differ or be absent in the shipping version.*

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

Delphi 2 / Object Pascal / Word for Windows

By Ian Davies

Automated Word

Creating OLE Automation Clients: Part 1

This article will cover some general uses of OLE automation, building on Cary Jensen's article "OLE Automation" in the October 1996 issue of *Delphi Informant*. Here, we'll concentrate on creating OLE automation clients using Delphi 2 that access services and functions provided by existing Microsoft Office OLE servers, such as Microsoft Word, Excel, and Access.

Why Microsoft Office? According to Microsoft, Office has over 80 percent of the office suite market — that's why.

Why Use OLE Automation?

OLE automation offers the ability to use code already developed in an application-specific language, without the expense of porting it to Delphi. In some cases, the conversion process may simply be time-consuming. In others, such as when dealing with complex financial calculations in an Excel spreadsheet, it may be extremely difficult.

Although I refer specifically to Word, Excel, and Access, the principles of OLE automation discussed here can be applied to any automation server, including one you create.

The Importance of Scope

Chapter 15 of the Delphi 2 *User's Guide* explains how to develop a simple application that creates a new document in Word, and inserts some text:

```
uses OLEAuto;

procedure TForm1.Button1Click(Sender:
                              TObject);
var
  V: Variant;
begin
  V := CreateOLEObject('Word.Basic');
  V.Insert('Hello from Delphi');
end;
```

The first line of the procedure attempts to load Word into memory, create the link with the Word.Basic object exposed by Word, and store its instance data in the Variant V. However, when the variable holding the instance data loses scope (in this case, immediately after the **Insert** statement is executed), the OLE automation server is removed from memory. Because V is a local variable that exists only within the *Button1Click* procedure, Word, as a server, will exist only for the length of time this procedure is being executed.

```
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, OLEAuto;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    V: Variant;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  V := CreateOLEObject('Word.Basic');
  { Execute the Word AppShow method to un-hide itself. }
  V.AppShow;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  V.Insert('Hello from Delphi');
end;

end.
```

Figure 1: Keeping Word in scope by declaring the Variant variable at the form level.

Each time this procedure is called (i.e. each time the button is clicked), Word will be loaded, some text will be inserted in the normal document, and Word will be closed again. This is an important issue, because the Word executable file (not including any supporting .DLLs) is almost 4MB (for version 7).

In contrast, if V was declared as a global variable or as a member of the form, it would exist for the life of the form (i.e. normally for the life of the application). If the *CreateOLEObject* method was then placed in the *OnCreate* event of the form, the server would be loaded once when your form is created, and lose scope only when the form is destroyed, normally as the application is closed (see Figure 1).

Note the use of the **AppShow** statement in the *FormCreate* procedure. OLE automation servers are normally hidden from view; this is the statement issued to Word to make it visible. Now, when the project is executed, Button1 can be clicked repeatedly, and the text appears almost immediately. **AppShow** and **Insert** are OLE server statements.

In the case of Microsoft Office (and most other servers), they would be used if you programmed those applications directly. Word versions 6 and 7 use WordBasic, Access 2 uses Access Basic, and Word 97, Excel (versions 5, 7, and 97), and Access version 7 all use Visual Basic for Applications (VBA).

There are a number of issues to bear in mind when using OLE automation with different versions of OLE servers. The code previously shown and in Figure 1 works fine when using Word 6 and 7 as a server, but fails when used with Word 97. The reason for this is that Word 97 doesn't start with an initial document loaded; so, the **Insert** statement is attempting to enter text into a document that doesn't yet exist. This is easily overcome by placing the following statement immediately before the call to the **Insert** statement:

```
V.FileNew;  // This creates a new, blank document.
```

The WordBasic statements may seem strange to a Delphi programmer, but comprehensive online help is available in Word. You can get a head-start in producing the code by recording your actions using the Record Macro feature in Word and Excel, and viewing the resulting code. This is normally a good place to start, because the important functions can be identified, then called from your Delphi application.

As stated earlier, Word 97 uses VBA. This provides your OLE client application with far more power, flexibility, and control over the OLE server. For backward compatibility, the Word.Basic object available in previous versions of Word remains available in Word 97. Although not as powerful as VBA, Word.Basic does provide a usable and effective OLE automation interface.

This article will concentrate on the use of the Word.Basic object, because this will enable your applications to be used on all recent versions of Word, rather than just Word 97. Any differences between Word.Basic in Word 97 and earlier versions are identified at the appropriate points.

It should be said that using VBA with Word 97 is likely to offer a better solution in the future. In the interest of compatibility, however, we'll look at that another day.

Using Word as a Reporting Tool

With only a few lines of code, Word can be used as a sophisticated tool for reporting any information available to Delphi. Word offers precise control over the layout of documents on paper sizes other than the standard A4 (UK) or legal (US), as well as the ability to control from which trays the paper is taken. Coupled with the use of document templates, this makes it useful as a form-generation system for standard letters, invoices, and other small, but often-used documents.

Word's Bookmark facility provides a simple way to control the placement of text in a document. A bookmark is simply a named location in a document, which can be used for placement of the cursor before text entry. It's accessed through OLE automation, by using the **EditGoto** statement. In Word 97, the **EditGoto** statement has been replaced by the Bookmark object, but it remains available, provided it's prefixed with the version of Word being used (i.e. **EditGoto** in Word 7 becomes **WW7_EditGoto,** where "WW7" denotes Word for Windows version 7).

```
procedure TMainForm.GenerateInvoiceBtnClick(Sender:
                                              TObject);
begin
  Screen.Cursor := crHourglass;
  Application.ProcessMessages;

  { Create new invoice document. You may have to amend the
    path, depending upon where the template is stored. }
  MSWord.FileNew(Template :=
    'C:\MSOffice\Templates\Other Documents\INVOICE.DOT');
  { Remove protection on this document to
    allow insertion of text. }
  MSWord.ToolsUnprotectDocument;

  { Call the procedure that jumps to correct bookmark and
    inserts the current date. }
  GotoBookmark('Date');
  MSWord.Insert(FormatDateTime('dd/mm/yy', Now));
  { Call the procedure that jumps to the correct bookmark
    and inserts the Invoice Number, etc. }
  GotoBookmark('Invoice_Number');
  MSWord.Insert('123456');
  GotoBookmark('Quantity_1');
  MSWord.Insert('1');


  GotoBookmark('Description_1');
  MSWord.Insert('Marine Magnetometer');
  GotoBookmark('Price_1');
  MSWord.Insert('545.58');
  GotoBookmark('Amount_1');
  MSWord.Insert('545.58');
  GotoBookmark('Subtotal');
  MSWord.UpdateFields;
  GotoBookmark('Total');
  MSWord.UpdateFields;

  { Save the document with the filename TEMP.DOC
    and close the file without prompting the user. }
  MSWord.FileSaveAs('c:\temp.doc');
  MSWord.FileClose(2);

  { Link the OLE Container to the file just created. }
  PreviewForm.OLEContainer1.CreateLinkToFile('c:\temp.doc',
                                             False);
  Screen.Cursor:=crDefault;
  Application.ProcessMessages;

  { Show the form containing the OLE Container. }
  PreviewForm.ShowModal;
end;
```

Figure 2: Using Word as an OLE server to create a "report previewer."

To create a functional example, we'll use the Invoice template provided with Word 7, insert the appropriate information, and offer the user a preview of the invoice and the ability to print it.

The invoice is created by calling the WordBasic **FileNew** statement. To enable some text to be programmatically inserted, the file protection on this document must be removed using the **ToolsUnprotectDocument** method. The sub-total and total fields on the Invoice template are calculated fields, which simply need to be updated to show the correct values. This is done using the **UpdateFields** statement. Finally, the document can be printed by using the **FilePrint** statement.

Function	Use
`FileOpen(`*`FileName`*`)`	Opens file named *FileName* in Word.
`CountBookmarks()`	Returns number of bookmarks in current document.
`BookmarkName$(`*`Index`*`)`	Returns name of bookmark referenced by *Index*.
`ToolsMacro(`*`MacroName`*`)`	Executes Word macro named *MacroName*.
`FilePrint(NumCopies := `*`x`*`)`	Prints *x* copies of the current document.
`FileClose(`*`x`*`)`	Closes the current document: 0 - Prompts user to save unsaved document; 1 - Closes and saves current document; 2 - Closes current document without saving it.

Figure 3: Selected WordBasic functions.

Adding a Print Preview Facility

A preview of the document before printing can be achieved using an OLE container, which can be linked to the file created in the previous steps. Instead of printing the docu-ment, the WordBasic **FileSaveAs** statement will create a file on the disk, which can be viewed using the OLE container and its *CreateLinkToFile* method. The example uses an OLE container located on another form named `PreviewForm` (see Figure 2). Note that, for clarity's sake, I've changed the name of the Variant holding the server instance details from `V` to `MSWord`.

An advantage of using a word processor as a reporting tool is that it allows your customers to create and modify their reports using a familiar tool. The interface between the templates and your application could be achieved by providing a facility to allow the users to map the bookmarks in a document to fields in a database table or any other information available to your control program. The **CountBookmarks** and **BookmarkName$** functions will provide access to all bookmarks defined by the author of the template (see Figure 3).

Using the Word Spell Checker from Delphi

Not only can you create and control documents within Word, but you can also gain access to its proofing tools, such as the spell checker. This requires a little more effort, because you must create a macro in Word that performs the spell check, and passes the results back to your Delphi application.

Information can be passed to and from Word through the use of Word's *document variable* feature. These can be treated just like variables common to both client and server, and are accessed using the WordBasic functions **GetDocumentVar$** and **SetDocumentVar** (or, in this case, by Delphi calling the functions using OLE automation).

On the Word Side

First, you need to create a new macro in Word. Open Word and close any open documents. Then choose **Tools |**

Macro and enter `GetSpelling` as the **Macro Name**. Next, click on the **Create** button and insert the WordBasic code to describe the macro.

If using a version of Word earlier than Word 97, use:

```
Sub MAIN
  Dim WordList$(100)
  WordToCheck$ = GetDocumentVar$("WordToCheck")
  NumWords = ToolsGetSpelling(WordList$(), WordToCheck$)
  Suggestions$ = WordList$(0)
  If NumWords > 0 Then
    For index = 1 To NumWords - 1
      Suggestions$ = Suggestions$ + Chr$(13) +
        WordList$(index)
    Next index
  End If
  SetDocumentVar "Suggestions", Suggestions$
End Sub
```

For Word 97 use:

```
Sub GetSpelling()
  ReDim WordList$(100)
  WordToCheck$ = WordBasic.[GetDocumentVar$]("WordToCheck")
  NumWords = WordBasic.ToolsGetSpelling(WordList$(),
    WordToCheck$)
  Suggestions$ = WordList$(0)
  If NumWords > 0 Then
    For index = 1 To NumWords - 1
      Suggestions$ = Suggestions$ + Chr(13) +
        WordList$(index)
    Next index
  End If
  WordBasic.SetDocumentVar "Suggestions", Suggestions$
End Sub
```

When you're done, close the macro document, and respond **Yes** to the question asking if you want to keep the changes to the macro.

The macro simply retrieves the word to spell check from the document variable named `WordToCheck`, and passes it to the WordBasic function **ToolsGetSpelling**. This function returns a list of suggested replacements in the `WordList$` array.

Because arrays cannot be passed using OLE automation, it's converted into a string separated by carriage returns, which is then passed back to the calling procedure in Delphi using the document variable `Suggestions`.

On the Delphi Side

Next, we need to create the application in Delphi that will call this macro. Create a new project and add Edit, Button, ListBox, and Text components, as shown in Figure 4.

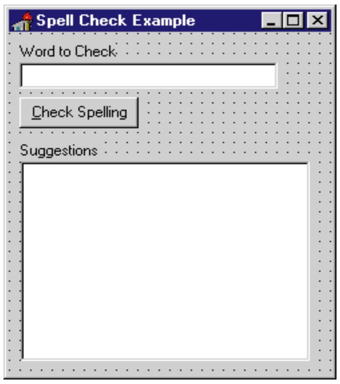


Figure 4: The Spell Check Example demonstration program.

We need to assign the work to check the document variable `WordToCheck`; this is achieved using the **SetDocumentVar** function. Next, call the `GetSpelling` macro in Word by using the **ToolsMacro(GetSpelling)** statement. The results are passed back to the calling routine using the `Suggestions` document variable, which is then split into its component parts and added to the list box (see Figure 5).

```
procedure TSpellChkForm.CheckSpellingBtnClick(Sender: TObject);
var
  suggestions : string;
begin
  { Assign the word to be checked to the
    WordToCheck document variable. }
  if MSWord.SetDocumentVar('WordToCheck',
                           Edit1.Text) = True then
    begin
      { Call the user-defined GetSpelling macro in Word. }
      MSWord.ToolsMacro('GetSpelling', 1);
      { Retrieve the list of suggestions from
        the Suggestions document variable. }
      suggestions := MSWord.GetDocumentVar('Suggestions');
      { Call the procedure to break returned string into
        its component parts, and add them to a list box. }
      ParseAndAdd(ListBox1, suggestions);
    end
  else
    MessageDlg('There was an error with the Word macro',
               mtError, [mbOK], 0);
end;
```

```
function TSpellChk.ParseAndAdd(Lst: TListBox;
                               Items: string) : Boolean;
var
  posn: Word;
begin
  Lst.Clear;
  if (Length(Items) > 0) then
    begin
      { Parse the results and add them to a listbox. }
      posn := pos(#13, Items);
      while (posn <> 0) do begin
        Lst.Items.Add(Copy(Items, 1, posn-1));
        Delete(Items, 1, posn);
        posn:= pos(#13, Items);
      end;
      Lst.Items.Add(Items);
    end
  else
    Lst.Items.Add('No suggestions or spelled correctly');
end;
```

Figure 5 (Top): This procedure calls a Word macro to perform a spell check. **Figure 6 (Bottom):** The *ParseAndAdd* function.

Then the Object Pascal function shown in Figure 6 is used to search the string for each occurrence of a carriage return (ASCII code 13) that delimits the returned words. The function then adds those words to a list box.

Keeping Word Hidden

By default, as with the vast majority of OLE automation servers, Word remains hidden while in use. However, certain functions — such as those that cause a dialog box to be displayed by the server — will automatically "unhide" the server. Fortunately, you can take precautions to ensure the server remains hidden.

First, any operation that would normally display a dialog box should be treated with care. For example, closing a document that may have been modified since it was last saved will result in a prompt to save changes. Either ensure that the document is saved before closing, or that the document is closed with a parameter indicating that you do not want to save any changes, i.e. `FileClose(2)`.

Also, sending the **FileSave** statement when a document hasn't been saved will display the Save As dialog box. This can be avoided by using the **FileSaveAs** statement with a filename passed as a parameter.

Finally, Word has a facility that prompts the user to enter summary information that will appear when the document properties are obtained (using Explorer, for example). To prevent this, either enter summary information using the **FileSummaryInfo** statement, or turn this facility off with the following statement:

```
ToolsOptionsSave(SummaryPrompt := 0, GlobalDotPrompt := 0)
```

Conclusion

This article has provided some background information on creating OLE automation clients, and introduced some practical uses of using OLE automation with Microsoft Word. Next month, we'll discuss how Microsoft Excel can be put to good use from a Delphi application.

It's well worth coming to grips with OLE automation — or simply "Automation" as it's becoming known. Microsoft has chosen this technology (over the likes of DDE) as a key feature of future versions of Windows. Taking the concept one step further, Windows NT 4 (server and workstation) now supports DCOM (Distributed Component Object Model), which means the OLE server doesn't need to reside on the same computer as the client, but can be accessed over a LAN, an intranet, or the Internet. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\MAY\DI9705ID.

Ian Davies is a developer of 16- and 32-bit applications for the Inland Revenue in the UK. He began Windows programming using Visual Basic about four years ago, but has seen the light and is now a devout Delphi addict. Current interests include Internet and intranet development, inter-application communication, and sometimes a combination of the two. Ian can be contacted via e-mail at 106003.3317 @compuserve.com.

By Dan Ehrmann

The Paradox Files: Part II

Properties, Field Names, and Record and Table Sizes

Many Delphi developers use the Paradox file format every day, yet the Delphi documentation offers almost no information about it. In this article, we'll explore the available field types in Paradox tables, and examine some important properties of each type. You'll learn how fields are stored in the .DB file, and how to calculate record and table size for any table. Also, I'll supply a sample application that performs these calculations for any specified table.

Working the Fields

A record may contain from 1 to 255 fields. The Paradox file format supports 17 field types, described in Figure 1. As noted in Part I of this series, Paradox tables use a fixed length for each field, even if the field contains no data.

A Number field always uses eight bytes, and an Alpha field always uses *nn* characters, even if the contents of the field are shorter than *nn*. In Figure 1, the Size column shows how many bytes the field uses in a record. (Terminology note: Unlike most database systems, which use the terms *columns* and *rows*, Paradox uses the terms *fields* and *records*, respectively.)

The Level column (in Figure 1) shows the table level at which that field type was introduced. When you save the definition of a new table, the Borland Database Engine (BDE) will use the highest level necessary to support the newest field type included in the table. (Index type and other factors also affect the table level.)

The Delphi Object column lists the object used to represent each field type. Each of these is a descendant of the *TField* object. If you don't create persistent objects for each field at design time, Delphi will create these objects at run time, allowing you to review and/or modify their properties, or to invoke methods for the object type.

The Standard Component column lists the data-bound control normally used to edit this object. Note that other controls (e.g. *TDBLookupListBox*) can be used, especially if the field is a foreign key to another table. The final column includes comments about most field types.

The Memo Type

For the Memo type, the BDE will store the first *nn* characters in the .DB file, and the whole memo in the .MB file, unless the memo for a particular record is less than or equal to *nn*. The main reason for this dual-storage feature is the table view in the Database Desktop (DBD).

In table view, the DBD displays only the first *nn* characters, and doesn't read the complete memo until you enter field view by pressing a hotkey, or clicking on a toolbar icon.

However, this isn't how Delphi works. When memos are displayed, the whole memo is read for each displayed field. The first *nn* characters in each memo are stored twice — once in the .DB file, and again in the .MB file. If *nn* is large, this duplication might use a lot of disk space.

The Paradox file format provides no way to limit how much data can be stored in a memo field. Fortunately, Delphi performs this function through the *MaxLength* property of the *TDBMemo* component.

Here's a tip: If the memos you're working with are frequently very short and only occasionally need to be much longer, sizing *nn* as a larger number allows you to fit the whole memo in the .DB file, with nothing in the .MB file except the longer memos. In this approach, designating M*nn* instead of A*nn* allows your text to overflow in the rare instances when it's necessary.

Another tip: If you often search on the first few characters of the memo, use that number of characters for *nn*. The BDE will then need to search only the .DB file.

Other Field Types

For the Formatted Memo, Graphic, OLE, and Binary field types, the BDE allows you to optionally specify the number of bytes to be stored in the table. But a fixed number

Type	Abb.	Size (bytes)	Level	Delphi Object	Standard Component	Notes
Alpha	A*nn*	*nn* bytes; 1 <= *nn* <= 255	Pre-4	*TStringField*	*TDBEdit*	
Number	N	8	Pre-4	*TFloatField*	*TDBEdit*	15 significant digits in the range of $\pm 10^{-307}$ to $\pm 10^{308}$.
Money	$	8	Pre-4	*TCurrencyField*	*TDBEdit*	Same as Number, but displayed by default at two decimal places.
Short	S	2	Pre-4	*TSmallIntField*	*TDBEdit*	16-bit signed integer; $\pm 2^{15}$ with a bit for the sign; -32,767 to +32,767; $00 holds the "null" value.
Long Integer	I	4	5	*TIntegerField*	*TDBEdit*	32-bit signed integer; $\pm 2^{31}$ with a bit for the sign; -2,147,483,647 to +2,147,483,647; $0000 holds the "null" value.
Date	D	4	Pre-4	*TDateField*	*TDBEdit*	A sequential number for each day, starting from 1/1/100, and allowing for leap years.
Time	T	4	5	*TTimeField*	*TDBEdit*	Milliseconds since midnight; 24 hours max.
Timestamp	@	8	5	*TDateTimeField*	*TDBEdit*	A combination of a date and a time, each using four bytes.
Memo	M*nn*	10 + *nn* in the .DB; 1 <= *nn* <= 240	4	*TMemoField*	*TDBMemo*	*nn* must be specified. Max size of 4MB per memo.
Formatted Memo	F(*nn*)	10 + *nn* in the .DB; 0 <= *nn* <= 240	5	*TBlobField*	*TDBRichEdit*	*nn* is optional. Max size of 64MB per memo, including formatting.
Graphic	G(*nn*)	10 + *nn* in the .DB; 0 <= *nn* <= 240	5	*TGraphicField*	*TDBGraphic*	*nn* is optional. Max size of 64MB per image.
OLE	O(*nn*)	10 + *nn* in the .DB; 0 <= *nn* <= 240	5	*TBlobField*	*TDBImage* (see note)	*nn* is optional. Max size of 64MB per OLE file.
Binary	B(*nn*)	10 + *nn* in the .DB; 0 <= *nn* <= 240	4	*TBlobField*	*TDBImage* (see note)	*nn* is optional. Max size of 64MB per object.
Byte	Y*nn*	*nn* bytes; 1 <= *nn* <= 255	5	*TBytesField*	*TDBEdit*	
Logical	L	1	5	*TBooleanField*	*TDBCheckBox*	Only True, False, and Blank are allowed. Truw displays as checked, False as unchecked, and Blank as a grayed box.
BCD	#.*nn*	17 1 <= *nn* <= 3	5	*TBCDField*	*TDBEdit*	Binary Coded Decimal; used to avoid precision and rounding errors in calculations. *nn* represents the number of digits after the decimal.
Autoincrement	+	4	5	*TAutoIncField*	*TDBEdit*	Internally, this is a Long Integer Max of one per table; usually the primary key.

Figure 1: The Paradox file format supports 17 field types.

Field Name	Type	Size (bytes)
OrderNo	+*	4
CustNo	A8	8
SaleDate	D	4
SaleTime	T	4
ShipDate	D	4
EmpNo	A5	5
ShipToContact	A20	20
ShipToAddr1	A30	30
ShipToAddr2	A30	30
ShipToCity	A15	15
ShipToState	A2	2
ShipToZip	A10	10
ShipToPhone	A20	20
ShipVIA	A5	5
PO	A12	12
Terms	A6	6
PaymentMethod	A7	7
ItemsCount	S	2
ItemsTotal	$	8
SalesTax	N	8
Freight	$	8
TotalInvoice	$	8
AmountPaid	$	8
Government	L	1
OrderNotes	M10	20
Record Size		**249**

Figure 2: This example has a total size of 249 bytes.

of bytes doesn't represent directly viewable data, as with Memo fields. Therefore, it's best to omit the number, or specify 0 for the *nn* value.

Delphi doesn't provide native components to display OLE, Binary, and Byte fields on forms. And for good reason. The contents of an OLE field are normally surfaced using the OLE-enabled application that created the object. Binary and Byte fields are used for data streams that require special processing, and aren't directly displayed on a form; you're responsible for writing this code. For example, you might store readings from scientific instruments in a Binary field, then write special procedures to process and interpret this data.

The next Autoincrement value is stored in the table's header, where there's room for only a single value; hence the limit of one Autoincrement field per table.

There is no easy way to change this value, although with an existing table, you can change a Small Integer or Long Integer field to an Autoincrement field; the next number will then be set to the highest found in the table.

Tip: Delphi doesn't automatically designate a displayed Autoincrement field as read-only. You should do this manually, so that users don't receive an exception if they try to change this field.

Field Names

Paradox field names follow these rules:

- They can be from 1 to 25 characters in length.
- They must be unique within a table.
- They can contain letters, numbers, and any printable character except double quotes ("), left or right brackets, left or right braces, left or right parentheses, the # sign, or the combination ->. The restriction on some of these characters is a holdover from the old days of Paradox for DOS; for compatibility reasons, these limits have remained.
- The period, comma, vertical bar, and exclamation mark are allowed, but not recommended. These characters create parsing and compiler problems with syntax options in Delphi and other environments.
- Field names can contain spaces, but cannot begin with a space. Consider also that most other database products, especially database servers (e.g. Oracle and InterBase) don't allow spaces in field names. If you plan to move your data to another format, or upsize it to a database server, don't use spaces in field names.

Calculating Record and Table Size

Suppose you want to calculate the minimum size of a table, assuming a particular record structure and number of records. As we've seen, this is relatively easy to do.

Consider the Orders table shown in Figure 2. The total record size for this table is 249 bytes. Because the table is keyed, the BDE will use the smallest size (excluding 1KB) that holds a minimum of three records. In this instance, that's 2KB, or 2048 bytes. Six bytes are used by the file format at the beginning of each record, leaving 2042 bytes available for data.

Records	Table Size (bytes)
500	131,072
5,000	1,282,048
50,000	12,802,048
500,000	128,002,048

Figure 3: Table size calculations.

The BDE can fit eight records into this space, using 1992 bytes. Fifty bytes will be wasted at the end of each block. However, this also means that you could add six bytes to each of the eight records without increasing the table size, because this data will fit nicely into the wasted space.

But consider what happens if you add seven bytes to a record. This makes the total record size 256 bytes; the BDE will be able to fit only seven records into each block, using 1792 bytes. At the end of each block, 240 bytes are wasted, and overall table size increases sharply.

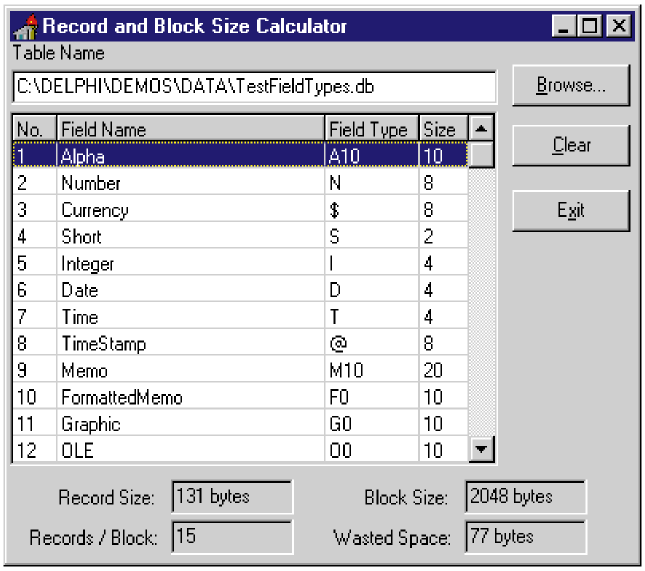

Record and Block Size Calculator

Table Name: C:\DELPHI\DEMOS\DATA\TestFieldTypes.db

No.	Field Name	Field Type	Size
1	Alpha	A10	10
2	Number	N	8
3	Currency	$	8
4	Short	S	2
5	Integer	I	4
6	Date	D	4
7	Time	T	4
8	TimeStamp	@	8
9	Memo	M10	20
10	FormattedMemo	F0	10
11	Graphic	G0	10
12	OLE	O0	10

Record Size: 131 bytes
Records / Block: 15
Block Size: 2048 bytes
Wasted Space: 77 bytes

Figure 4: The Paradox Record and Block Size Calculator.

When you restructure and pack a table, the BDE will fill each block completely. Using this information, it's easy to calculate the minimum size of the table.

Simply divide the number of records by eight, and round up to get the number of blocks required; then multiply the result by 2KB to get the total size of data blocks. Add 2KB for the header — because this is a small table — and you have your answer. Figure 3 shows this calculation for different numbers of records.

When individual record sizes grow especially large (for example, between 500 and 1000 bytes) you can use these calculations to carefully analyze wasted space.

Sometimes you'll find that a small reduction in one or two fields (usually the longer Alpha fields) means that you can fit four records into a pre-defined block size instead of three, resulting in a significant reduction in total disk space. Figure 4 shows a small Delphi application that lets you select a Paradox table, then calculates record size, block size, records per block, and wasted space per block, using the algorithms described earlier.

Until Next Time

The next article in this series will explore indices, including the index on the primary key, as well as the options for secondary indices. It will explain how indices are structured internally, and will discuss the performance and file-size implications of keeping the primary key as short as possible. ∆

The sample application referenced in this article is available on the Delphi Informant Works CD located in INFORM\97\MAY\DI9705DE.

Dan Ehrmann is the founder and President of Kallista, Inc. a database and Internet consulting firm based in Chicago. He is the author of two books on Paradox and is a member of Team Borland and Corel's CTech. Dan was the Chairman of the Advisory Board for Borland's first Paradox conference, which evolved into the current BDC. He has worked with the Paradox file format for more than 10 years. He can be reached via e-mail at dan@kallista.com.

By *Bill Todd*

Deployment: Part I

Deploying Delphi 2 Applications with InstallShield Express and InstallShield Express Professional

One of the third-party tools that ships with Delphi 2 is InstallShield Express from InstallShield Corp. InstallShield Express enables you to create a single installation disk set that will install your application and its files, the Borland Database Engine (BDE), and ReportSmith Runtime. This, the first of two articles, will take you through the process of building a setup for a typical Delphi application using InstallShield Express. You will see not only how to use InstallShield Express, but also how to deal with some of its limitations. In the second article, we'll tour InstallShield Express Professional and look at the additional features it offers.

InstallShield Corp. makes a third setup program, InstallShield3. InstallShield3 is an extremely powerful script-based installation program. Any time you install a piece of commercial software on your PC, chances are the installation program was built using InstallShield3. It's the most popular installation program among commercial software companies, because it's extremely powerful and flexible. However, the price you pay for power and flexibility, as with most software, is that it's complex and not easy to learn. By the time you read this, InstallShield5 Professional should be shipping. InstallShield5 features a visual integrated development environment that will make it easier to use than InstallShield3. Because InstallShield3 is beyond the needs of most Delphi programmers, it won't be covered in these articles.

The Setup

This article will take you through creating a setup for a typical Delphi application. It needs the BDE, but it doesn't use ReportSmith. The program will be installed on stand-alone PCs at some locations, while at others it will be installed on a LAN. The application uses local Paradox tables as its database.

Installing a Delphi database program on a LAN presents some special problems. First, you have to provide two setup options. One setup will install the program on a workstation, and install the initial set of data tables on the file server. You must also provide a second setup option to install the program on a new workstation after the system is in use. This second option is required because it must not install the database tables. If it did, it would overwrite the database with a new set of empty tables.

BDE Installation Options

There are several potential options for installing a BDE application on a LAN:

1) You can put the program files (.EXEs and .DLLs) on the server or on each workstation.
2) You can install the BDE on each workstation, or on the server.
3) You can install the BDE on each workstation, but have all workstations share a common BDE configuration file on the server for ease of administration.

Unfortunately, InstallShield Express only supports installing the BDE and its configuration file on each local machine, and installing the program files and database on the server. Even in this case, a problem exists with Paradox tables, because there is no safe way to set the Net Dir (network control directory path) parameter of the Paradox driver as part of the installation process. (A more detailed explanation of network installation problems will be presented later in this article.)

When you install the BDE on a workstation, the installation not only installs the BDE files on the workstation hard disk, but also makes many entries in the Windows registry. Unfortunately, there's no installation program I'm aware of that supports a server-only BDE installation, i.e. an installation that puts the BDE files on the server and makes only the necessary registry entries on the workstation. Using a local BDE installation with a shared configuration file on the server isn't easy either. This is because the BDE uses a registry entry to determine where the BDE configuration file is located. Certainly the most common BDE architecture is to install the BDE files on each workstation. This provides much better performance if, like most networks, the data transfer rate across the network is ten times slower than the data transfer rate from the workstation's local hard disk.

"Do I have to install the entire BDE?"

This frequently-asked question is usually motivated by the size of the BDE. Borland's answer is "No," but mine is "Yes." It is possible using InstallShield Express to set up a partial BDE installation. If you do this, however, the BDE will be installed in the same directory that contains your program's .EXE file. Because the BDE is designed to be a shared component that can be used by many different programs, a partial installation isn't safe.

To understand why, consider a PC that already has a program installed that uses the BDE (WordPerfect, Paradox, Quattro Pro, dBASE, Delphi, Borland C++, or a custom Delphi application). When a Borland certified BDE installation program installs the BDE, the installation program checks to see if the BDE is already installed. If it finds an existing copy of the BDE, the installation program checks the version and only installs the BDE if the version to be installed is newer than the one already installed. If the installation software would let you install a partial copy of the BDE in the same directory that already contains a copy of the BDE, you might update some BDE files and not others. This will almost certainly leave the user with a copy of the BDE that will not run.

To avoid this problem, InstallShield Express and any other Borland certified installation system that supports partial BDE installations, will only install a partial copy of the BDE in the directory that contains the program's .EXE file. While this prevents your installation from damaging an existing BDE installation, it also means that you may use a lot more disk space in the long run by forcing each of your Delphi programs to have its own copy of the BDE.

My opinion is that the best way to install the BDE is the way it was designed to be installed: as a shared component that will support any and all BDE-based programs that the user may install.

Using InstallShield Express

InstallShield Express is not automatically installed when you install Delphi 2. To install InstallShield Express, go to the IsExpress\Disk1 subdirectory on your Delphi 2 CD and run SETUP.EXE. This will install InstallShield Express in the \Program Files\InstallShield\IS Express Delphi directory. As is true of the other third-party products that ship with Delphi 2, there is no manual for InstallShield Express. The only documentation is in the online Help.

When you select **File | New** from the InstallShield Express menu, the New Project dialog box appears (see Figure 1). Enter a name for your project. By default, this name will be used for the project directory. Optionally, you can also enter the path to a new subdirectory where your project will be stored. If you leave this field blank, your project directory will be created in the \Program Files\InstallShield\IS Express Delphi directory. Be sure to check the **Include a custom setup type** check box if you want to offer Typical, Compact, and Custom setup types as you do in this example.

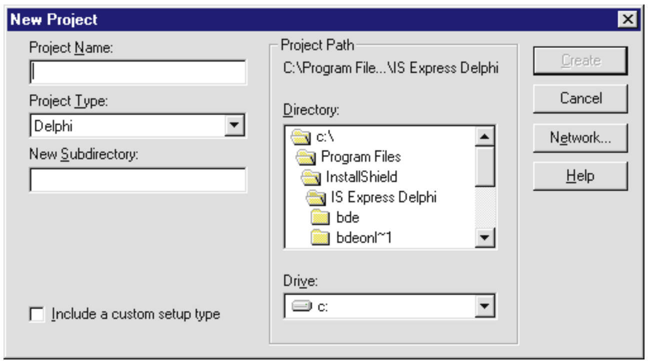
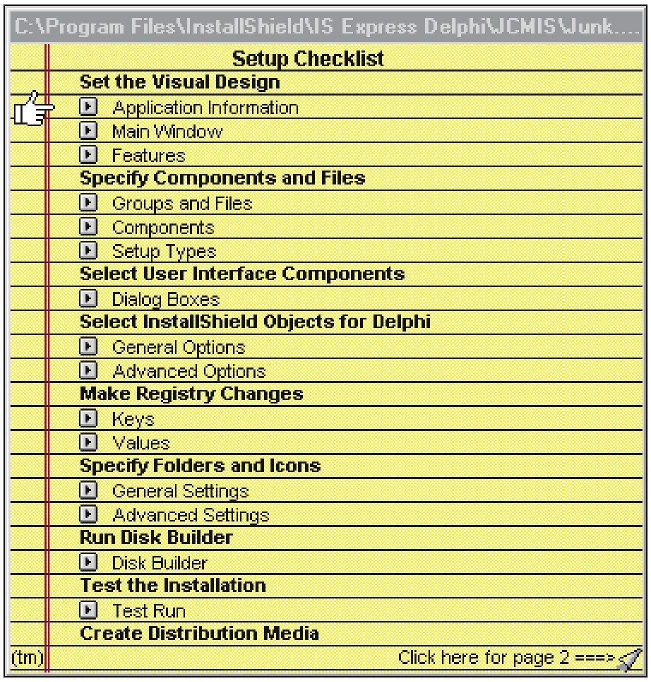


Figure 1: InstallShield's New Project dialog box.

After you have created a project, the InstallShield Express main window looks like Figure 2. If you look closely, you will notice a mildly annoying problem: Its main form doesn't fit on the screen if you are running at 640 x 480 — even with the Windows 95 task bar hidden. The obvious solution is to resize the window, but that doesn't work. If you make the window shorter, no vertical scroll bar appears, so there is no way to reach the choices at the bottom of the yellow notepad. The only solution is to use the **View** menu to turn off either the toolbar or the status bar at the bottom of the window. For the rest of this article, you will not see the toolbar.

The notepad metaphor is actually quite nice. It's supposed to resemble a checklist with the hand at the left pointing to the next item you need to do. As you click each button to work your way down the list, a red check mark appears next to each button so you can tell at a glance what you've done. The only problem is that you can't go through the items in the order listed. You will see why as you go through the steps.

Start by clicking the **Application Information** button to display the Set the Visual Design dialog box (see Figure 3).



C:\Program Files\InstallShield\IS Express Delphi\JCMIS\Junk....

Setup Checklist

Set the Visual Design
- Application Information
- Main Window
- Features

Specify Components and Files
- Groups and Files
- Components
- Setup Types

Select User Interface Components
- Dialog Boxes

Select InstallShield Objects for Delphi
- General Options
- Advanced Options

Make Registry Changes
- Keys
- Values

Specify Folders and Icons
- General Settings
- Advanced Settings

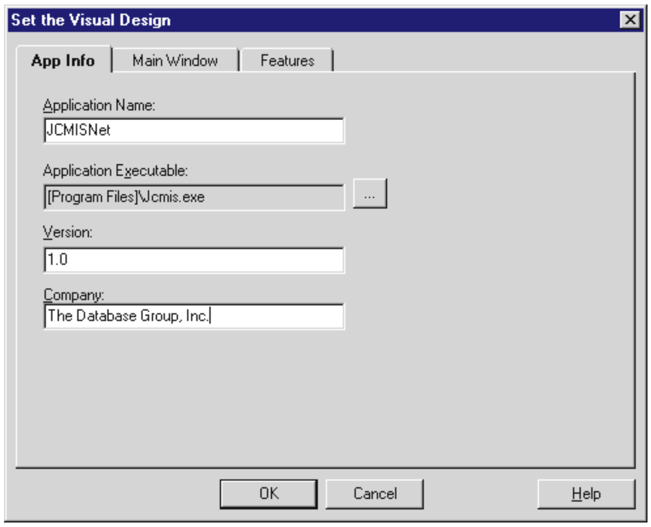
Run Disk Builder
- Disk Builder

Test the Installation
- Test Run

Create Distribution Media

(tm) Click here for page 2 ===>

Figure 2: The main window after creating a new project — InstallShield's Setup Checklist.



Set the Visual Design

App Info | Main Window | Features

Application Name: JCMISNet

Application Executable: [Program Files]\Jcmis.exe

Version: 1.0

Company: The Database Group, Inc.

Figure 3: The Set the Visual Design dialog box.

Enter the application name. This is the name that the user will see in the background when installing the application. Click the ... (browse) button to the right of the **Application Executable** edit box to find the main .EXE file for your application. If the .EXE contains a version resource, the version number will be displayed automatically in the **Version** field. If not, you can enter any version number you wish. This is an alphanumeric field that can accommodate complex version numbers that include letters. Finally, enter your company name or abbreviation. The string you enter here will be the name of the top-level directory that InstallShield Express will create in the Program Files directory to hold your programs.

The Set the Visual Design dialog box has two additional pages, Main Window and Features, that correspond to the buttons of the same name on the main form. At the Main Window page you can opt to display the application name as text on the background during installation, a custom bitmap for the application's title, and a logo bitmap as your company's logo. Finally, you can choose the background color displayed during installation.

The Features page contains a single option, **Automatic Uninstaller**. This check box is checked by default so that an uninstall option will be provided for your application. I can't imagine why anyone would not want this feature.

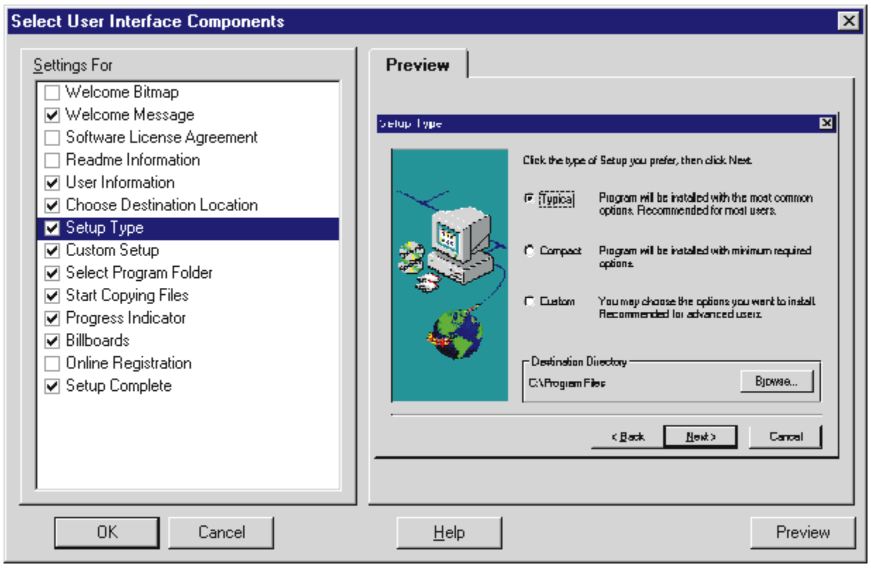
Creating Components Groups and Files

The next step is to select the files that must be installed as part of your application. InstallShield Express organizes your files into groups; all files in a group will be installed in the same directory. The groups are organized into components. InstallShield Express allows you to provide three setup types for your users: Typical, Compact, and Custom. You organize your file groups into components, and use the components to determine which files are installed for each of the three setup types.

As an alternative, you can have a single setup type, in which case, you will only need one component and it will be created for you automatically. If you do want to offer Typical, Compact, and Custom setups and didn't check the **Include a custom setup type** check box when you created your project, you can still do so. Simply skip down to the **Dialog Boxes** button under Select User Interface Components. Clicking this button displays the dialog box shown in Figure 4. The **Setup Type** and **Custom Setup** check boxes are linked so that both are checked, or neither are checked. That means, for example, that you can't offer the user just two setup types, Typical and Compact. You also cannot change the name of the setup types.

At this point, you have a major decision to make. Because our sample program uses Paradox tables and you must be able to install it on either a stand-alone system or a network, you need two installation types. One that installs the program, the BDE, and the data tables, and a second that installs only the program and the BDE so that the user can install the system on additional network workstations without overwriting the data tables and destroying the existing data. There are two ways to do this.

The first approach is to build a single installation disk set that offers the user three setup options: Typical, Compact, and Custom. This is the option you'll use. You'll design the installation so that if the user chooses a Typical installation, the program, the BDE, and the data tables will be installed. If the user chooses a Compact or Custom installation, only the program files and the BDE will be installed. The big problem with this approach is the names of the three types of installations; it would be much less confusing if you could change the names to File Server, Network Workstation, and Stand Alone.



Select User Interface Components

Settings For

- ☐ Welcome Bitmap
- ✓ Welcome Message
- ☐ Software License Agreement
- ☐ Readme Information
- ✓ User Information
- ✓ Choose Destination Location
- ✓ Setup Type
- ✓ Custom Setup
- ✓ Select Program Folder
- ✓ Start Copying Files
- ✓ Progress Indicator
- ✓ Billboards
- ☐ Online Registration
- ✓ Setup Complete

Preview

Setup Type

Click the type of Setup you prefer, then click Next.

[✓ Typical] Program will be installed with the most common options. Recommended for most users.

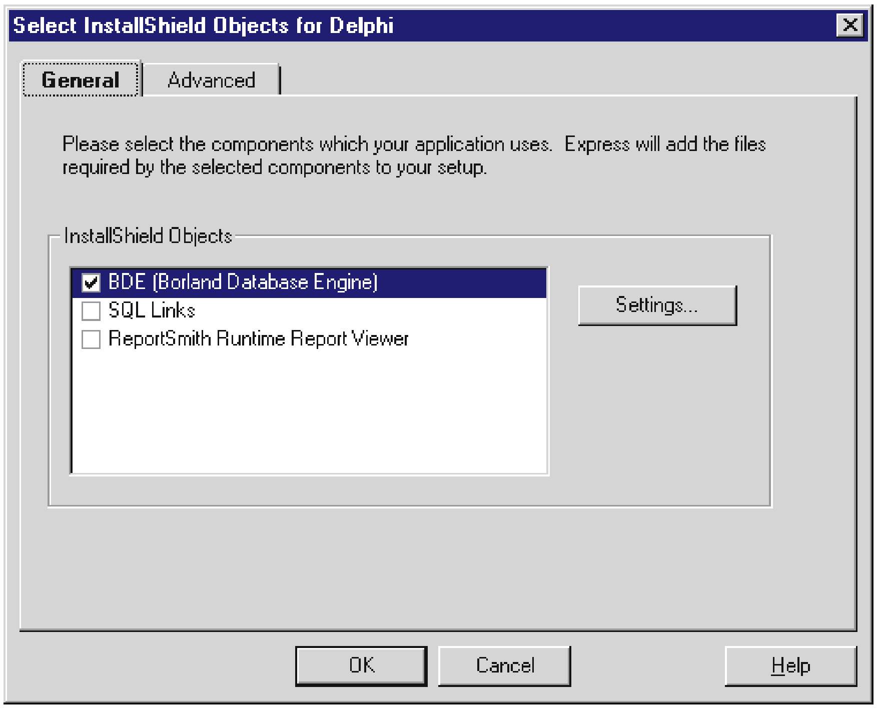
Compact Program will be installed with minimum required options.

Custom You may choose the options you want to install. Recommended for advanced users.

Destination Directory

C:\Program Files

Figure 4: The Select User Interface Components dialog box.



Select InstallShield Objects for Delphi

Please select the components which your application uses. Express will add the files required by the selected components to your setup.

InstallShield Objects

- ✓ BDE (Borland Database Engine)
- ☐ SQL Links
- ☐ ReportSmith Runtime Report Viewer

Figure 5: The Select InstallShield Objects for Delphi dialog box.

The second alternative is to build two sets of installation disks and send both sets to the user. The disadvantage of this technique is that you have to create twice as many disks, and the user has to keep the sets straight.

Because you're creating one disk set with multiple setup types, you will check the **Setup Type** check box (see Figure 4), and return to the **Groups and Files** button (again, see Figure 2). One of the groups of files that you have to install is the BDE, and you can't create that group from here. You need to take a detour in the InstallShield Express checklist down to the **General Options** button under Select InstallShield Objects for Delphi. (This problem with the order of the checklist has been corrected in InstallShield Express Professional.)

The dialog box in Figure 5 enables you to choose to install the BDE, SQL Links, or ReportSmith Runtime. After checking the BDE check box, you can use the **Settings** button to define any aliases you want added to the BDE configuration file. If you are creating an alias to a database server, you can also enter any parameters you want to set for the alias in the window at the bottom of the form. Parameters must be entered one-per-line in the format:

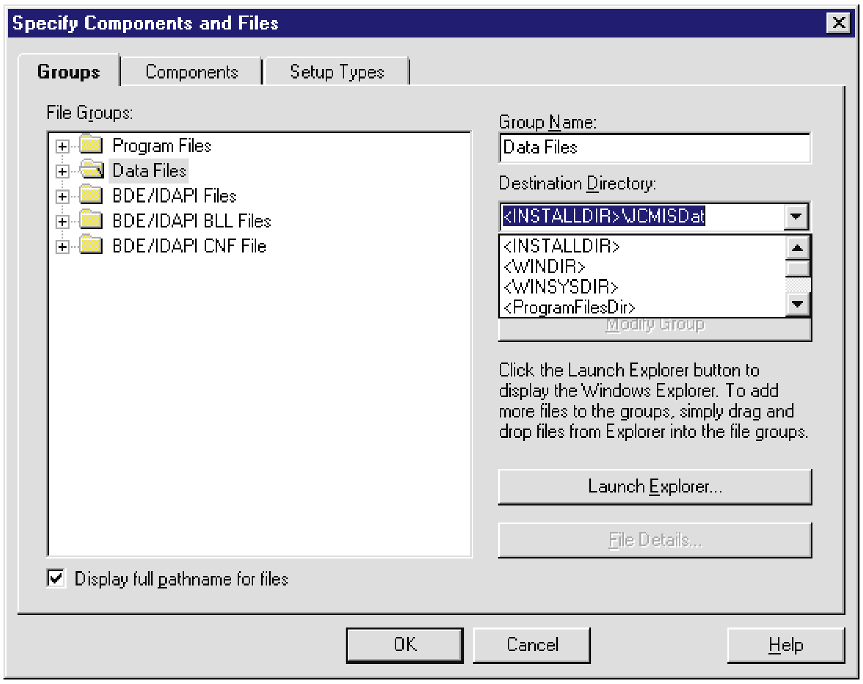
```
ParameterName=Value
```

Now you're ready to return to the **Groups and Files** button that displays the Specify Components and Files dialog box (see Figure 6). The BDE file groups were created automatically by InstallShield Express when the **BDE** check box in Figure 5 was checked.

To create a new group, enter the group name in the **Group Name** field and the directory where you want the files placed in the **Destination Directory** field, then click the **Add Group** button. There is no way to prompt the user for a directory path. You are limited to the destination directory specifiers in the drop-down list shown in Figure 6. The complete list of directory specifiers is shown in Figure 7. Note that you can use sub-directories under any of these directory specifiers. In the example in Figure 6, the data tables are placed in the <INSTALLDIR>\JCMISDat sub-directory. InstallShield Express will automatically create any directories that don't exist.

After the group has been created, you can add files to it by dragging the files from Explorer and dropping them on the group. In this example, I have created two groups: Program Files, which contain the application's .EXE file; and Data Files, which contain all the Paradox tables the application uses. If you need to create an empty subdirectory as part of your installation process, simply add a group, but don't add any files to it. You will get a warning when you build your setup files, but the user sees nothing unusual and the directory will be created.

Next, click on the Components tab and create two components. The first, Application Files, is created by InstallShield Express. You need to change it so that it contains the Program Files group and all three BDE groups. To add a group to a component, select the component in the **Application**



Specify Components and Files

File Groups:
- Program Files
- Data Files
- BDE/IDAPI Files
- BDE/IDAPI BLL Files
- BDE/IDAPI CNF File

Group Name: Data Files

Destination Directory: <INSTALLDIR>\JCMISDat

<INSTALLDIR>
<WINDIR>
<WINSYSDIR>
<ProgramFilesDir>

Click the Launch Explorer button to display the Windows Explorer. To add more files to the groups, simply drag and drop files from Explorer into the file groups.

[✓] Display full pathname for files

Figure 6: The Specify Components and Files dialog box.

Specifier	Description
<INSTALLDIR>	The main installation directory specified by the user.
<WINDIR>	The Windows directory.
<WINSYSDIR>	The Windows\System directory.
<WINDISK>	The drive that the Windows directory resides on, for example, C:.
<WINSYSDISK>	The drive that the Windows system directory resides on.
<WINSYS16DIR>	On an NT system this is the 16-bit Windows directory.
<PROGRAMFILESDIR>	The Program Files directory.
<COMMONFILESDIR>	The Program Files\Common Files directory.

Figure 7: InstallShield Express directory specifiers.

Components list, select the group in the **File Groups** list, and click the **Add to Application Component** button.

On the Setup Types page you will find the three setup types, Custom, Typical, and Compact, listed on the left, and the components listed on the right. To add a component to a setup type, click the setup type, then the component, then the **Add to Setup Type** button. Adding the Application Files component to all three setup types and the Data Tables component to the Typical setup provides the configuration you need. Typical will install the entire application including the BDE and data tables on either a file server or stand-alone system. Compact will install only the BDE and application for a network workstation. Note that when users perform a

Compact installation to a network workstation, they must specify the directory on the server that contains the program files as the installation directory. This will configure their workstation to run the copy of the program's .EXE file on the server, and install the BDE on their local hard drive.

Selecting User Interface Components

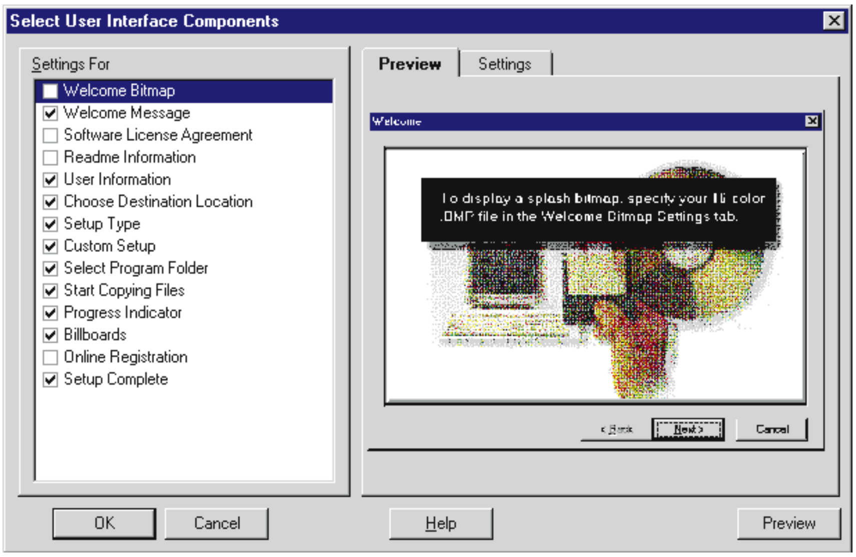
Next, return to the **Dialog Boxes** button, and click it to display the dialog box shown in Figure 8. The check boxes let you control what dialog boxes are displayed during the installation process. For example, **Software License Agreement** and **Readme Information** allow the user to see and accept your license agreement and view your readme file. To specify the file that contains your license agreement, click on **Software License Agreement** to select it, then click the Settings tab and enter the path to the text file. The **Readme Information** option works the same way.

The Settings page for the **Select Program Folder** check box enables you to enter the name of the program folder that will be created for your program. If you don't enter a value, your InstallShield Express project name will be used. You even have the option to sign up for Pipeline Communications' online registration service to allow your users to register their software via modem during installation.

Creating Registry Entries

If you need to create or change any registry entries as part of your installation, begin by clicking the **Keys** button under Make Registry Changes (back on the Setup Checklist shown in Figure 2) to display the dialog box in Figure 9. To add or change an entry in the registry, select the top-level key, then click the **Add Key** button and enter the complete path to the key you want to create or change. Do not type a leading backslash (\) at the beginning of your list of keys. If you do, InstallShield Express will create a null key in the registry between the top-level key you selected and your first key. To add a value under any key, select the key, then click the Registry - Values tab, and use the **Add Value** button to add as many values as you wish under the selected key.

For the sample installation, you will change one of the keys for the BDE to ensure the Local Share setting in the BDE configuration is set to True. Setting Local Share to True is required to make table and record locking work correctly on a peer-to-peer network. In addition, it disables write caching, to protect against data loss and table corruption in case of a system crash. Setting Local Share requires adding the key Software\Borland\Database Engine\Settings\System\Init under HKEY_LOCAL_MACHINE, and adding the value Local Share = True.

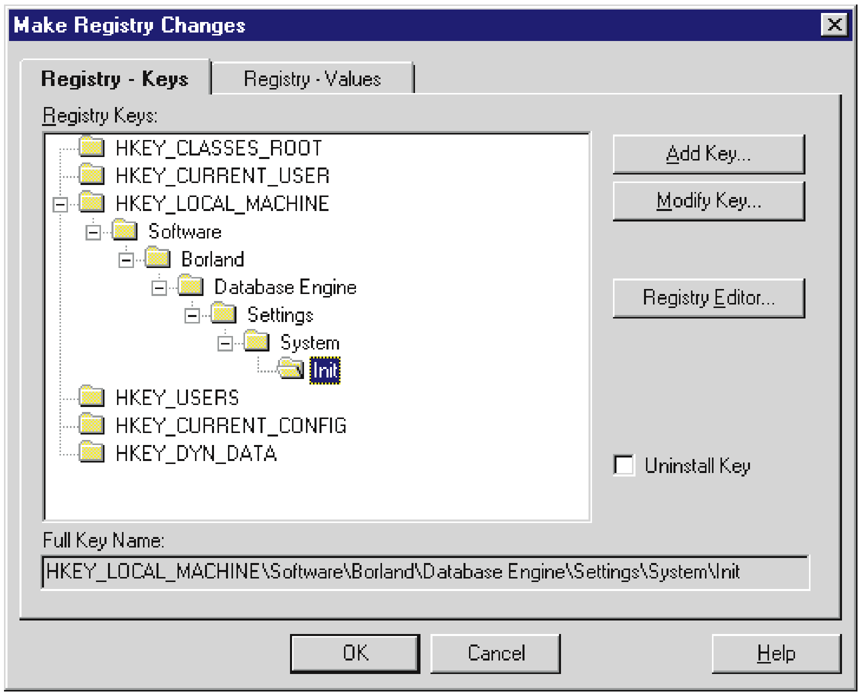


Settings For

- [ ] Welcome Bitmap
- [✓] Welcome Message
- [ ] Software License Agreement
- [ ] Readme Information
- [✓] User Information
- [✓] Choose Destination Location
- [✓] Setup Type
- [✓] Custom Setup
- [✓] Select Program Folder
- [✓] Start Copying Files
- [✓] Progress Indicator
- [✓] Billboards
- [ ] Online Registration
- [✓] Setup Complete

To display a splash bitmap, specify your 16 color .BMP file in the Welcome Bitmap Settings tab.

Figure 8: The Select User Interface Components dialog box.



Registry Keys:

```
HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
    Software
        Borland
            Database Engine
                Settings
                    System
                        Init
HKEY_USERS
HKEY_CURRENT_CONFIG
HKEY_DYN_DATA
```

Full Key Name:

HKEY_LOCAL_MACHINE\Software\Borland\Database Engine\Settings\System\Init

Figure 9: The Make Registry Changes dialog box.

Configuring the BDE

When you install a program that uses the BDE and Paradox tables, you must be concerned with four things:

1) If the user installs your program on a peer-to-peer network such as a Microsoft network, you want to set Local Share to True to disable write caching and ensure that file sharing works correctly.
2) You must create any aliases your program requires.
3) You must set the NET DIR property of the Paradox driver to the shared network directory where you want the network control files created.
4) You must set any other options in the BDE configuration file.

As you have already seen, you can set Local Share by setting the value of its registry key. Creating aliases is also easy: Click the **Settings** button in the Select InstallShield Objects for Delphi dialog box (again, see Figure 5). That takes care of bullet items 1 and 2.

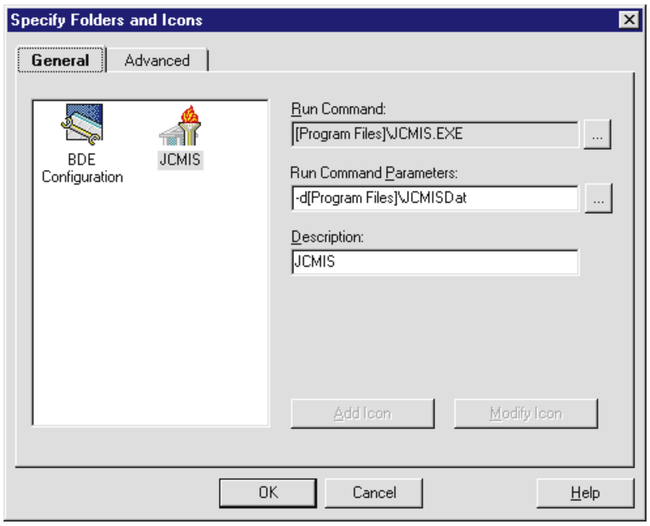
The fourth bullet item — other BDE configuration file options — is a bit of a problem. InstallShield Express uses the IDAPI32.CNF file in the IS Express Delphi\Redist subdirectory as the default BDE configuration file, and you can edit this file using the BDE configuration program. If the BDE doesn't exist on the target machine, InstallShield Express will install IDAPI32.CNF and rename it to IDAPI32.CFG. If the BDE has already been installed on the target machine, InstallShield Express will merge the settings in IDAPI32.CNF into the existing BDE configuration file. However, it will not overwrite existing settings. If a setting in the existing BDE configuration file differs from a setting in the IDAPI32.CNF file, the setting in the existing file will prevail. There is no way to change this behavior.

Bullet item 3, setting the NET DIR path, is simply impossible. Although this setting is stored in the registry, it's also stored in the BDE configuration file, and the value in the configuration file is the one the BDE configuration program reads. Even if you could change the registry entry, you may not want to. If the user already has the BDE installed and configured to use Paradox tables on a network, changing the network control directory will cause a serious problem, unless it's changed for every user. If not, users that have different

Specifying Folders and Icons

The next step in creating an installation is defining the contents of the application folder you want created as part of the installation. Figure 10 shows the Specify Folders and Icons dialog box. To add an icon to the folder, use the ... (browse) button to the right of the **Run Command** edit box to select a file from one of your groups. Next, enter any command-line parameters and the description you want to appear below the icon, and click the **Add Icon** button.

For the sample installation, an icon for the program's .EXE file has been added, as well as one for the BDE configuration program. It's a good idea to provide access to the BDE configuration program, just in case any settings have to be changed manually.


Specify Folders and Icons

Run Command: [Program Files]\JCMIS.EXE

Run Command Parameters: -d[Program Files]\JCMISDat

Description: JCMIS

Figure 10: The Specify Folders and Icons dialog box.

NET DIR settings will not be able to access the same database at the same time. You will either have to write a separate program that checks to see if the path to the network control directory is already set, and if not, set it using BDE API calls. Or you will have to give the user instructions for changing this setting using the BDE configuration program.

If you are adding one or more new aliases to the target system, you must be careful to pick names that will be unique. If an alias on the target system has the same name as an alias you are trying to add, the Type and Path parameters for the alias will be updated to the values you supplied, but none of the other parameters will be changed. This could leave the target system in a state where the alias exists, but your program does not work as expected.

Conclusion

After you have created your installation, run the Disk Builder to create the diskette images for your installation diskettes.

You can test the installation on your computer from within InstallShield Express by clicking the **Test Installation** button. When everything works the way you want it to, use the **Copy to Floppy** button to make your diskettes.

InstallShield Express is a big step up from the days of Delphi 1, where you got a self-installing version of the BDE, but you were on your own as far as installing your program and its associated files. It has a simple, clear user interface and good online Help. With the exception of a more elegant way to set the Local Share option and a way to set the network control directory path for Paradox tables, InstallShield Express provides all the features you need to install basic database applications.

The only thing that makes InstallShield Express a bit difficult to learn to use is that you can't go through the checklist in the order it's presented, and there is no tutorial in the online Help to take you through the process in order. I think most users could learn to use the program faster if the Help file contained a topic that listed each of the steps in creating a setup, and the topic to search for that described that step in detail. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\MAY\DI9705BT.

Bill Todd is President of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is co-author of *Delphi: A Developer's Guide* [M&T Books, 1995], *Delphi 2: A Developer's Guide* [M&T Books, 1996], and *Creating Paradox for Windows Applications* [New Riders Publishing, 1994], and is a member of Team Borland providing technical support on CompuServe. He is also a nationally known trainer and has been a speaker at every Borland Developers Conference and the Borland Conference in London. He can be reached on CompuServe at 71333,2146, on the Internet at 71333.2146@compuserve.com, or at (602) 802-0178.

By Cary Jensen, Ph.D.

Cached Updates: Part I

Diversify Your Record-Editing Options

Delphi 2 introduced a powerful new database-related capability: cached updates. These allow changes made to one or more DataSets (Tables, Queries, or StoredProcs) to be stored locally, and applied as a group. The benefits of this technique include increased performance, reduced network traffic, additional user interface options, and enhanced programmatic control over data updates.

This is the first of a three-part series that takes an in-depth look at cached updates. It begins with an overview of what cached updates are, and what they are not. It continues with a step-by-step discussion of how to implement cached updates in your database applications. Unless otherwise specified, this material applies to Delphi 2 and 3.

Overview

Unless you program otherwise, changes to records being edited by the user are posted (applied) to the underlying database as soon as the user has performed an action that posts the record, e.g. moving off the record, or clicking the **Post changes** button of a linked DBNavigator component. Likewise, you can programmatically post edits by initiating a move off the record, or by explicitly calling a DataSet's *Post* method.

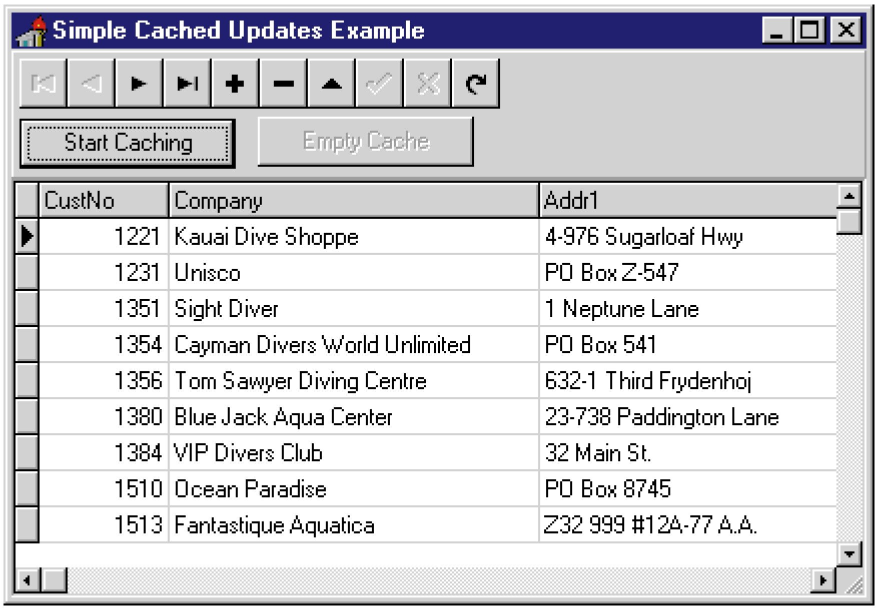
When cached updates are enabled, the Borland Database Engine (BDE) tracks all data changes (modifications, insertions, and deletions) locally. After the necessary changes are made, they're applied to the underlying database as a group. The advantages of using cached updates include:

- Users can edit one or more records, then cancel all changes — without ever affecting the underlying data.
- Network traffic may decline, because edits needn't be transmitted across the network individually.
- You can allow users to preview edits before the cached changes are applied to the underlying database. Even "deleted" records can be previewed.
- A user can selectively revert individual edits to their pre-edited state without affecting other edits still in the cache. Even "deleted" records can be undeleted.
- Edits being cached can be updated within a transaction, and a transaction can be rolled back if all the edits can't be applied.
- You can permit a user to edit read-only DataSets. When applied correctly, cached updates can, for example, permit a user to edit the result set of a join query.
- Cached updates provide complete programmatic control over the posting of cached changes. Specifically, you can create an event handler that's called as each changed record is posted. From this code, you can choose to modify the record before posting, ignore the change, and continue to the next record — or abort the posting process.
- You can use cached updates as a means of creating an audit trail. As each record is posted, you can programmatically determine the type of change made to that record (e.g. insertion, deletion, or modification). Also, you can easily determine which fields in a modified record were edited, and write both the old and new values for those fields to your audit trail.

- You can create an event handler that executes when an attempt to post a cached change fails. Again, your code can modify the record and retry, ignore just that one record and continue applying the other edits in the cache, or abort the posting entirely.

As you can see, this list argues powerfully for the use of cached updates.

The following demonstration uses a Paradox table, for simplicity's sake. While cached updates are especially useful in client/server applications, characteristics of individual servers affect which techniques you'll use to apply your updates. By using a Paradox table, those server-specific issues can be avoided.



CustNo	Company	Addr1
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy
1231	Unisco	PO Box Z-547
1351	Sight Diver	1 Neptune Lane
1354	Cayman Divers World Unlimited	PO Box 541
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj
1380	Blue Jack Aqua Center	23-738 Paddington Lane
1384	VIP Divers Club	32 Main St.
1510	Ocean Paradise	PO Box 8745
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.

Figure 1: The main form of the CACHE1 project.

(An alternative to cached updates — transaction processing — is available under all versions of Delphi. You can initiate a transaction before the edits begin, and commit or roll back the transaction when the edits are complete. This technique is quite different from cached updates. During a transaction, the database is informed of each edit as it occurs, although the changes are not made permanent until the transaction is committed. Also, transaction processing is handled by the database server, not at the application level.)

Cached Updates: The Basics

Despite the power that cached updates provide, they're remarkably easy to use in their simplest form. All DataSets have a *CachedUpdates* property. When you set this property to *True*, all edits to the corresponding DataSet are cached. *CachedUpdates* is a published property, and consequently can be set to *True* at run time or design time. You can set *CachedUpdates* to *True* on more than one DataSet simultaneously.

As long as *CachedUpdates* is *True*, the BDE continues to cache updates. Setting *CachedUpdates* to *False* causes the BDE to cancel any pending edits in the cache, then stop caching. Likewise, if you close the DataSet before posting pending edits in the cache, the edits are canceled. To post the cached changes to a DataSet, you must call the *ApplyUpdates* method of either the DataSet or its Database. The use of a Database's *ApplyUpdates* method is described later in this article.

The DataSet *ApplyUpdates* method attempts to post all edits pending in the cache. If it's unsuccessful, then an exception is raised, the cache's contents remain pending, and the DataSet continues caching. If the method succeeds, it's then necessary to exit the Cached Updates mode by setting the DataSet's *CachedUpdates* property to *False*, or to clear the cache by calling *CommitUpdates*, then continue caching. If you want the BDE to continue caching after posting pending edits, it's essential to call *CommitUpdates*.

Consider the following event handler associated with a button. Assuming the DataSet named Table1 is set to cache updates, clicking this button will attempt to apply the updates, then clear the cache. If *ApplyUpdates* generates an exception, then *CommitUpdates* doesn't execute; consequently, the cache remains intact:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.ApplyUpdates;
  Table1.CommitUpdates;
end;
```

If you close this DataSet before applying the updates, cached changes are lost. However, a more elegant technique is to call the DataSet *CancelUpdates* method, which has the effect of clearing the cache. However, like *ApplyUpdates*, calling *CancelUpdates* doesn't affect the *CachedUpdates* property.

These basic techniques are demonstrated in the project named CACHE1.DPR, shown in Figure 1. This project makes use of a table named CUST1.DB, which is an exact duplicate of the CUSTOMER.DB table that ships with Delphi. The CUST1.DB table is copied from the CUSTOMER.DB table each time you run the downloadable example projects for this article. This is because cached updates can be adequately demonstrated only by editing a table.

The code performing this task (see Figure 2) is attached to the *OnCreate* event handler for this project's main form, which includes two buttons. The button labeled **Start Caching** is associated with the *OnClick* event handler shown in Figure 3.

When clicked, it tests the *CachedUpdates* property of Table1 to determine if it's caching updates. If not, it places Table1 into the Cached Update mode, sets the DataSource's *AutoEdit* property to *True* (to permit the user to edit the

```
procedure TForm1.FormCreate(Sender: TObject);
var
  OldTable: TTable;
begin

  OldTable := TTable.Create(Self);
  try
    OldTable.DatabaseName := 'DBDEMOS';
    OldTable.TableName    := 'CUSTOMER.DB';
    Table1.TableName      := 'CUST1.DB';
    Table1.BatchMove(OldTable,batCopy);
    Table1.AddIndex('','CustNo',[ixPrimary, ixUnique]);
    Table1.Open;
  finally
    OldTable.Free;
  end;

end;
```

Figure 2: This code copies the CUST1.DB table from the CUSTOMER.DB table.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if not Table1.CachedUpdates then
    begin
      Table1.CachedUpdates := True;
      DataSource1.AutoEdit := True;
      Button1.Caption      := 'Apply Updates';
      Button2.Enabled      := True;
    end
  else
    begin
      if Table1.State in [dsInsert, dsEdit] then
        Table1.Post;
      if Table1.UpdatesPending then
        begin
          Table1.ApplyUpdates;
          Table1.CommitUpdates;
        end;
      DataSource1.AutoEdit := False;
      Table1.CachedUpdates := False;
      Button1.Caption      := 'Start Caching';
      Button2.Enabled      := False;
    end;
end;
```

Figure 3: The *OnClick* event handler for the **Start Caching** button of the CACHE1 project.

table), updates Button1's caption, and enables the **Empty Cache** button. If Table1 is already caching updates, clicking Button1 results in posting any current edits to the cache, applying the updates, clearing the cache, turning cached updates off, toggling the button's caption, and disabling the **Empty Cache** button (Button2).

The **Empty Cache** button simply removes any changes stored in the cache. The *OnClick* event handler for this button calls Table1's *CancelUpdates* method to empty the cache. Next, *ShowMessage* reminds the user that the table is still caching updates. This is the *OnClick* event handler for Button2:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Table1.CancelUpdates;
  ShowMessage('Cache emptied. Still caching updates');
end;
```

As you inspect the code for Button1's *OnClick* event handler, you'll notice that this project doesn't permit users to edit the table unless they begin caching. Though this isn't an essential feature, you might still want to employ it. However, the primary technique to prevent the user from editing the table — that of setting the DataSource's *AutoEdit* property to *False* — is sometimes only partially successful. When you view a table through a DBGrid (or use a DBNavigator, as is being done here), a user can still insert and delete records — even when *AutoEdit* is set to *False*. Consequently, this project contains two additional event handlers. One is on Table1's *BeforeDelete* event property, while the other is on Table1's *BeforeInsert*. From this code, the table's *CachedUpdates* property is inspected; an exception is raised if the property is not set to *True*. This technique is demonstrated in the following event handler, which is assigned to Table1's *BeforeDelete* event property:

```
procedure TForm1.Table1BeforeDelete(DataSet: TDataSet);
begin
  if not Table1.CachedUpdates then
    raise EDenyEdit.Create(
      'Click Start Caching to delete a record');
end;
```

Both the *BeforeDelete* and *BeforeInsert* event handlers raise a custom exception named *EDenyEdit*. Alternatively, it would be just as effective to raise one of the predefined exceptions, such as *Exception*, rather than a custom exception. However, it's generally considered good programming practice to leave the raising of predefined exceptions to Delphi, and define a custom exception when you need to explicitly raise one within your code. The following is the **type** declaration of *EDenyEdit*:

```
type
  EDenyEdit = class(Exception);
```

Finally, there's the issue of closing the table without applying cached updates, if some are pending. As you learned earlier, all pending cached updates are lost if the table is closed before the updates are applied. You can determine if any cached updates are pending by inspecting the table's *UpdatesPending* property. This property is *True* if the table is caching updates, and if unapplied updates remain in the cache; otherwise, it's *False*. Note, however, that inspecting the value of *UpdatesPending* when the corresponding DataSet's *CachedUpdates* property is set to *False* generates an exception. Consequently, you should test *UpdatesPending* only after confirming that *CachedUpdates* is *True*.

The CACHE1 project demonstrates the use of this property in the form's *OnCloseQuery* event handler, as shown in Figure 4. When the form is being closed, any pending edits are first posted to the table. Next, if updates are being cached, *UpdatesPending* is tested to determine whether or not the cache is empty. If not, the user is asked to indicate whether to cancel pending edits, or post them. If the user confirms posting of the edits, but *ApplyUpdates* fails, an exception is raised, and the form doesn't close.

```
procedure TForm1.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if Table1.State in [dsInsert, dsEdit] then
    Table1.Post;
  if Table1.CachedUpdates and Table1.UpdatesPending then
    if MessageDlg('Updates pending. ' +
                  'Select OK to apply, Cancel to lose',
                  mtInformation,
                  [mbOK,mbCancel],0) <> mrOK then
      Table1.CancelUpdates
    else
      begin
        Table1.ApplyUpdates;
        Table1.CommitUpdates;
      end;
end;
```

Figure 4: The *OnCloseQuery* event handler.

Cached Updates and Transactions

The use of *ApplyUpdates* in the preceding example has some limitations. As mentioned, when *ApplyUpdates* fails, an exception is raised. A potential problem arises if some of the pending edits were actually applied prior to the exception. In most cases, it's better if the pending edits are applied in an "all or none" fashion. This requires transaction processing.

I offer two solutions to providing transaction control with cached updates. One is to provide a transaction wrapper around your call to a DataSet's *ApplyUpdates* method. The second is to use the *ApplyUpdates* method of the DataSet's Database.

Adding a transaction wrapper is easy. First, begin a transaction. Next, enter a **try..except** block. Within the **try** block, apply the updates, commit the transaction, then commit the updates. From within the **except** block, roll back the transaction and raise the exception again. (Repeating the exception is optional, but provides a simple means of letting the user know that updates couldn't be applied.)

Transaction processing is provided by the *TDatabase* class. In most cases, this means adding a Database component to your form, and using it as the source of data access for your DataSets. For example, suppose you have a Database named Database1, and Table1 uses this Database. The following code permits you to apply updates within a transaction:

```
Database1.StartTransaction;
try
  Table1.ApplyUpdates;
  Database1.Commit;
  Table1.CommitUpdates;
except
  Database1.Rollback;
  raise;
end;
```

But can you use this technique without explicitly adding a Database component to your form? Yes, you can. To use a DataSet, a database is required; but if you haven't added one to your form explicitly, Delphi will create one for you at run time. There are two ways to access a Database component created automatically by Delphi. One way is by using the *Databases* property of the *TSession* class. Because Delphi automatically creates an instance (named *Session*) of this class, any database application has ready access to databases created automatically. *TSession.Databases* is an array property. Because it's a zero-based array — and assuming only one automatic database is created, and no explicitly defined databases appear on the form — you can use `Session.Databases[0]` to access the automatic database. Another way is through the *Database* property of a DataSet. This read-only property contains a pointer to the DataSet's database — whether it's automatically created, or one you explicitly placed on the form.

The following code demonstrates how to wrap a transaction around a DataSet using its *Database* property to access the automatically created database:

```
Table1.Database.StartTransaction;
try
  Table1.ApplyUpdates;
  Table1.Database.Commit;
  Table1.CommitUpdates;
except
  Table1.Database.Rollback;
  raise;
end;
```

The use of a transactional wrapper for cached updates is demonstrated in the CACHE2 project. (This project is identical to CACHE1, with the exception of the transaction.) One additional adjustment is necessary in this project, however, because a Paradox table is being used. Whenever you use transactions with local tables (dBASE, Paradox, and so forth), Delphi requires that the database's *TransIsolation* property be set to `tiDirtyRead`. In essence, *tiDirtyRead* treats those records that have been modified by another pending transaction (one that has not yet been committed) as if they have been committed, even though they may eventually get rolled back by that transaction. Because the *TransIsolation* property of the default Database is not *tiDirtyRead*, this must be set within your code.

The code in Figure 5 is associated with the *OnClick* event handler for the **Start Caching** button in the CACHE2 project.

Using *Database.ApplyUpdates*

The *TDatabase* class also supports an *ApplyUpdates* method. This method has the following syntax:

```
procedure ApplyUpdates(const DataSets: array of TDataSet);
```

As you can see, when calling *ApplyUpdates*, you must pass an array of DataSets. The Database component will then call *ApplyUpdates* and *CommitUpdates* for each of the DataSets in this array.

In addition, it makes the calls to *ApplyUpdates* from within a transaction, which it will roll back if any of the updates can't be applied. This is obvious from the source code of the *TDatabase* class (see Figure 6), which can be found in the DB.PAS unit.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if not Table1.CachedUpdates then
    begin
      Table1.CachedUpdates := True;
      DataSource1.AutoEdit := True;
      Button1.Caption      := 'Apply Updates';
      Button2.Enabled      := True;
    end
  else
    begin
      if Table1.State in [dsInsert, dsEdit] then
        Table1.Post;
      if Table1.UpdatesPending then
        // Need to update only if cached edits are pending.
        begin
          Table1.Database.TransIsolation := tiDirtyRead;
          Table1.Database.StartTransaction;
          try
            Table1.ApplyUpdates;
            Table1.Database.Commit;
            Table1.CommitUpdates;
          except
            Table1.Database.Rollback;
            raise;
          end;
        end;
      DataSource1.AutoEdit := False;
      Table1.CachedUpdates := False;
      Button1.Caption      := 'Start Caching';
      Button2.Enabled      := False;
    end;
end;
```

Figure 5: The *OnClick* event handler for the **Start Caching** button of the CACHE2 project.

```
procedure TDatabase.ApplyUpdates(
  const DataSets: array of TDBDataSet);
var
  I: Integer;
  DS: TDBDataSet;
begin

  StartTransaction;
  try
    for I := 0 to High(DataSets) do begin
      DS  := DataSets[I];
      if DS.Database <> Self then
        DatabaseError(FmtLoadStr(
          SUpdateWrongDB,[DS.Name, Name]));
      DataSets[I].ApplyUpdates;
    end;
    Commit;
  except
    Rollback;
    raise;
  end;

  for I := 0 to High(DataSets) do
    DataSets[I].CommitUpdates;
end;
```

Figure 6: The source code of the *TDatabase.ApplyUpdates* method.

The code shows that the Database component begins the transaction, then calls the *ApplyUpdates* method for each DataSet in the passed array of DataSets. Only after all DataSets have been updated does the transaction get committed. If an exception is raised by any call to *ApplyUpdates*, or the call to *Commit*, the transaction is rolled back. Because the **except** clause raises the exception again, the subsequent calls to *CommitUpdates* are made only if no exception is encountered. (A side note: The call to *CommitUpdates* cannot fail.)

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if not Table1.CachedUpdates then
    begin
      Table1.CachedUpdates := True;
      DataSource1.AutoEdit := True;
      Button1.Caption      := 'Apply Updates';
      Button2.Enabled      := True;
    end
  else
    begin
      if Table1.State in [dsInsert, dsEdit] then
        Table1.Post;
      if Table1.UpdatesPending then
      begin
        Table1.Database.TransIsolation := tiDirtyRead;
        Table1.Database.ApplyUpdates([Table1]);
      end;
      DataSource1.AutoEdit := False;
      Table1.CachedUpdates := False;
      Button1.Caption      := 'Start Caching';
      Button2.Enabled      := False;
    end;
end;
```

Figure 7: The Database *ApplyUpdates* method could have been substituted for *TTable.ApplyUpdates* in the CACHE2 project.

Because of its strong support for both transaction processing and the inclusion of the call to *CommitUpdates*, you should definitely consider using *TDatabase.ApplyUpdates* in lieu of *TDataSet.ApplyUpdates*. The code in Figure 7 demonstrates how the Database *ApplyUpdates* method could have been substituted for the *TTable.ApplyUpdates* method in the CACHE2 project.

Conclusion

Cached updates greatly increase your record-editing options. In addition, the technique can generally improve your application's performance. Fortunately, in its simplest case — that of editing a single table — cached updates are easy to employ.

Next month's installment will continue the discussion of cached updates by considering how to work with individual records in the cache, and how to use the UpdateSQL component to permit editing of read-only DataSets. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\MAY\DI9705CJ.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including *Delphi In Depth* [Osborne/McGraw-Hill, 1996]. Cary is also a Contributing Editor of *Delphi Informant*, as well as a member of the Delphi Advisory Board for the 1997 Borland Developers Conference. For information concerning Jensen Data Systems' Delphi consulting and training services, visit the Jensen Data Systems Web site at http://gramercy.ios.com/~jdsi. You can also reach Jensen Data Systems at (281) 359-3311, or via e-mail at cjensen@compuserve.com.

Delphi 2 / Object Pascal / Internet / Intranets

By John Penman

NetCheck: Part I

A 32-Bit Tool for Debugging Networks

With the Internet and intranets playing important roles in the way we work and communicate, any network application we use must be robust. An application must always check the connectivity between itself (the client) and the server.

Developers can easily add this connectivity "check" to any network application; however, developing applications for the Internet and intranets can be a tricky, frustrating exercise. A developer must contend with two possible sources of apparent program failure: poor network connectivity and internal program errors (i.e. bugs). A network problem is less common than an application bug, but can happen at unexpected times. Network problems can be caused by an improperly installed network card, a malfunctioning router that can cause a break in the network, etc. Sometimes it's difficult to differentiate between the two, so it's important for a developer to have a network debugger that helps track down such problems.

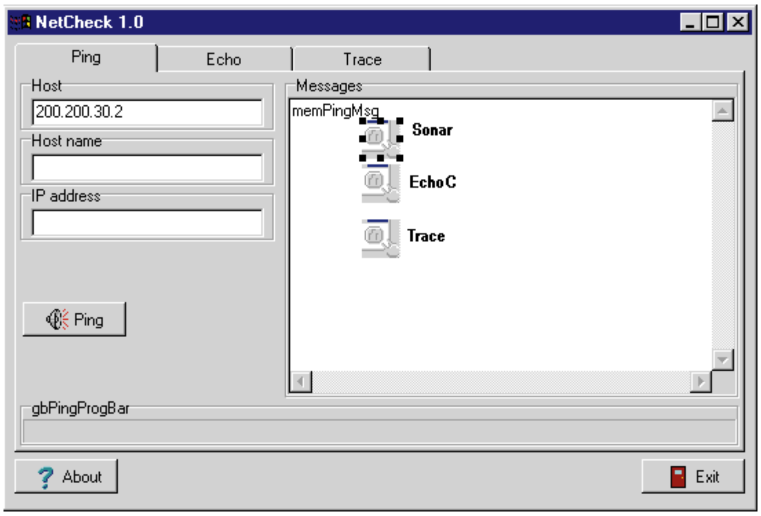


Figure 1: The complete NetCheck program with its Sonar, EchoC, and Trace components.

NetCheck: A Simple Network Debugging Tool

In this two-part series, we'll examine NetCheck, a simple network debugging tool that uses three non-visual Delphi components (see Figure 1). Each component encapsulates a well-known debugging service:

- Sonar is a wrapper for the ping application,
- EchoC is an echo client wrap per for the Echo service, and
- Trace encapsulates the TraceRoute application.

Each component uses the Windows Sockets (Winsock) .DLL to interface with the Internet. To follow this series, you must have some knowledge of the Winsock API. (It's too large a topic to cover here in detail; for more information on the Winsock API, see the references at the end of this article.)

Ping

To test connectivity between two peer machines on a network, we'll use ping, a popular diagnostic tool with its roots in the UNIX world, now widely available on diverse platforms, including Windows. The ping application uses the Internet Control Message Protocol (ICMP) to send an echo request packet to the server. The server, if running, responds automatically by returning an echo reply packet.

The nature of ICMP dictates such a response, meaning no server program is required on the server-to-service echo request packets. In contrast, an echo service (as we'll discuss in Part II) requires a server program on the target host to service echo requests. Using ping is an excellent way to check the connectivity between peers at the Internet Protocol (IP) layer. Additionally, ping yields interesting information such as the round-trip time between the sender and target host, which we can analyze for clues to packet loss.

The downside to ping is that it's fallible. For example, the target host may hide behind a firewall server that filters out echo request packets, preventing the target host from replying.

In UNIX and some Windows platforms, a ping application uses raw sockets to work with ICMP packets. Some implementations of Winsock, such as Microsoft's Winsock 1.1 for Windows 95, do not support raw sockets. Microsoft has included the ICMP.DLL file with its current release of Windows 95 to rectify this lack of raw sockets support. The Sonar component uses this .DLL.

(Note that Windows NT 4.0 uses Winsock 2.0, which does support raw sockets, and Winsock 2.0 for Windows 95 is expected to be released later this year. In a future article, I'll present an enhanced version of the Sonar component to work with raw sockets.)

I named this component Sonar because it "sounds" a target host similar to how a ship's sonar sounds an underwater object. The client sends an echo request packet to a server, which in turn must respond automatically by echoing an echo reply packet. Hopefully, the name Sonar will distinguish it from other ping applications.

Inside Sonar

Sonar is based on the *TSonar* class (see Listing One on page 40), a direct descendant of *TComponent*. (I used Martien Verbruggen's demonstration ping program to develop Sonar. See the reference listed at the end of this article.) Like all components, the *TSonar* class has a constructor method, *TSonar.Create*, that initializes some properties and calls two functions to perform essential tasks.

The first function, *CheckWS*, initializes WINSOCK.DLL. If Winsock isn't functioning or is missing, Sonar displays a Warning dialog box and closes the calling application. Sonar does this because the Sonar, EchoC, and Trace components rely heavily on WINSOCK.DLL. Thus, it doesn't make sense to continue with the application when Winsock isn't available.

The second function, *CheckICMP*, tries to create the *hIcmpModule* handle for ICMP.DLL. If *LoadLibrary* fails to load the .DLL, it sets *hIcmpModule* to **nil**. *CheckWS* assigns a Boolean value to a read-only property, *PingAvail*, which NetCheck starts up. When *PingAvail* is *False*, NetCheck disables the **Ping** button, and displays a warning message in the *memPingMsg* Memo control. In contrast to *CheckWS*, when *CheckICMP* fails to load ICMP.DLL, it doesn't abort the calling application because the EchoC component does not require ICMP.DLL. Additionally, the *TSonar.GetHost* method calls several functions from the Winsock API to resolve the name of the target host to an address Sonar uses to ping.

As mentioned, Microsoft's Winsock 1.1 does not support raw sockets, so Sonar uses ICMP.DLL to send ping packets instead. Before Sonar can ping a host, it must initialize two important structures, *TIPOptions* and *TICMPEchoReply*, with appropriate values (see Listing Two beginning on page 40). The *IcmpSendEcho* function, which sends and transmits echo packets, requires these filled structures before use.

The *TIPOptions* record specifies the options in the IP header. Such options include the Time To Live (TTL), Type Of Service (TOS), and OptionsData data fields. The TTL is set nominally to 128 (although you can change this from 32 to 255 in the Object Inspector). This value tells the packet how many hops it can go. That is, Sonar can send a packet that lives for a number of hops — set by the TTL field — before expiring. For our purposes, a hop is a "link" between hosts.

The *TICMPEchoReply* structure contains the data *IcmpSendEcho* returns in response to an echo reply request. For example, *pEchoReply*, a pointer to *TICMPEchoReply*, contains an Address field holding the target host's replying address. The Status field is important because the component uses it to check the state of the returned packet. The RTT field tells Sonar the total round-trip time for a packet in milliseconds.

Because Sonar calls the *IcmpCreateFile*, *IcmpCloseHandle,* and *IcmpSendEcho* functions from a .DLL, it must call *GetProcAddress* to establish each function's address. Sonar does this before calling *IcmpSendEcho* in the *DoSonar* method (see Listing Three beginning on page 41). The *IcmpCreateFile* function creates a context handle for *IcmpSendEcho*, and *IcmpCloseHandle* closes the same context handle when Sonar is done with *IcmpSendEcho*.

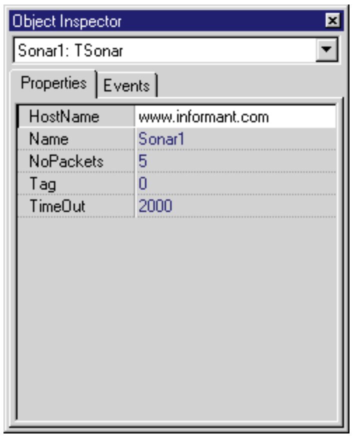
To monitor the progress of these echo request packets sent by Sonar, the echo reply packets sent by the host, and other messages, Sonar uses these *TNotifyEvent* procedures:

1) *OnRecvDataEvent* posts all messages including statistics to NetCheck,
2) *OnResolveEvent* posts the host name and IP address of the target host, and
3) *OnProgressEvent* updates the progress bar to chart progress (or lack thereof) of packets sent to the target host.

Are You There?

Let's look at Sonar in action. In NetCheck, before we can click on the **Ping** button, we must enter the name, or IP address, of the target host. We pass this string to Sonar's *GetHost* method via the *Sonar1.Ping* method. *GetHost* determines the target host's IP address, which *IcmpSendEcho* uses to ping that host.

IcmpSendEcho is synchronous, meaning it will "block" when it's waiting for a reply from the target host before timing out. In other words, NetCheck will be unresponsive, and appear dead. This isn't satisfactory. We can avoid this by creating a thread for *IcmpSendEcho*. (This feature is not implemented in this version of NetCheck; we'll address this thorny issue next month.)



Object Inspector

Sonar1: TSonar

Properties | Events

HostName	www.informant.com
Name	Sonar1
NoPackets	5
Tag	0
TimeOut	2000

Figure 2: The published properties of the Sonar component in the Object Inspector.

The *IcmpSendEcho* function sends a number of pings as determined by *FNoPackets*. *IcmpSendEcho* uses *FAddress* that Sonar obtained from *GetHost* to locate the target host. *FTimeOut* is a time-out value of Sonar's *TimeOut* property, set in the Object Inspector. *TimeOut*'s default value is 2000. You can increase this for hard-to-reach hosts on slow and difficult connections (see Figure 2).

After sending a packet, *IcmpSendEcho* returns an echo reply. A non-zero value indicates a request packet reached the target host, but this doesn't necessarily mean the host was successfully "pinged." A value of zero indicates a failure, possibly due to one or more factors such as a time out. We check the Status field of the *pEchoReply* record, a pointer to the *pIcmpEchoReply* record, to determine the state of the returned packet. The *pEchoReply^.Status* function returns a value of *IP_SUCCESS* to indicate a successful reply; any other value indicates an error.

As Sonar receives a successful reply packet, it updates variables such as *Fmin*, *Fmax*, *FNoEchoes*, and *FRTTSum* needed to calculate simple statistics (see Figure 3). Sonar then calls the *OnRecvDataEvent* procedure to pass a status message to NetCheck's *memPingMsg* Memo control.

Also, *OnProgressEvent* is called to update the *pbPing TProgressBar* control in NetCheck. Figure 4 shows NetCheck during a pinging session. At the end of the session, NetCheck displays the statistics as shown in Figure 5.

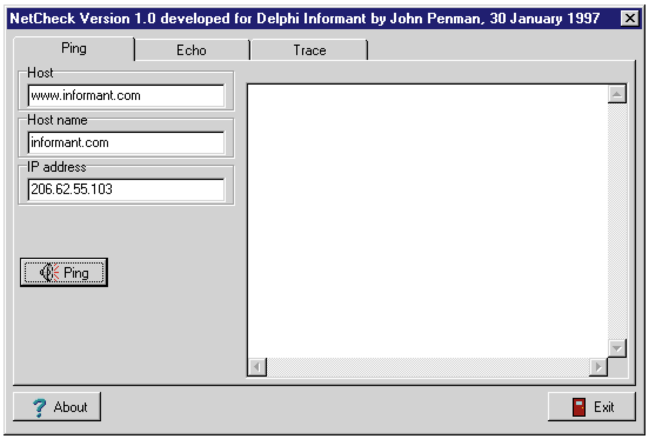
To test another host, you must enter a new string in the *edPingHost* edit control. You could, of course, ease the pain of retyping a host name by adding a *TListBox* to hold the list of hosts in NetCheck. I leave this improvement (and others) to you.

Installing NetCheck

To make NetCheck work on your machine, first install the Sonar component. Use the SONAR.DCU unit to install Sonar onto the Winsock page of the Component palette. For more information on component installation, refer to Delphi's online Help.

```
// Calculate statistics for a batch of packets.
procedure TSonar.Stats;
begin
  try
    FAve := Round(FRTTSum/FNoEchoes);
    FMsg := Concat('Sent ',IntToStr(FNoPackets),' packets');
    OnRecvDataEvent;
    FMsg := Concat('Received ',IntToStr(FNoEchoes),
                   ' packets');
    OnRecvDataEvent;
    FMsg := Concat('Min/Avg/Max= ',IntToStr(FMin),
                   '/',IntToStr(FAve),'/',IntToStr(FMax));
    OnRecvDataEvent;
  except
    on EDivByZero do
      begin
        FMsg := 'Cannot calculate stats';
        OnRecvDataEvent;
      end;
  end;
end;
```

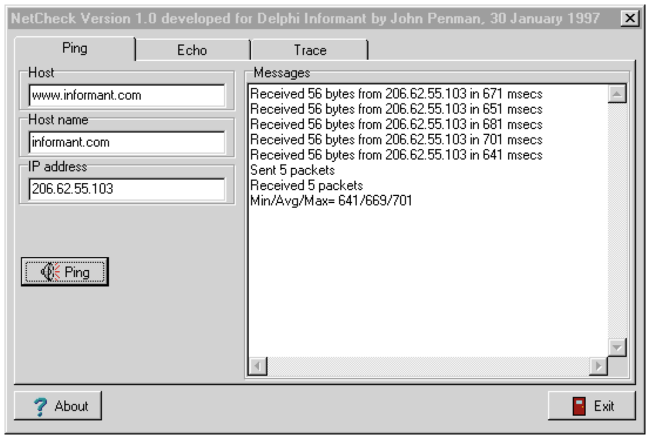
Figure 3: Calculating simple statistics.



NetCheck Version 1.0 developed for Delphi Informant by John Penman, 30 January 1997

Host: www.informant.com
Host name: informant.com
IP address: 206.62.55.103

Figure 4: NetCheck during a pinging session.



NetCheck Version 1.0 developed for Delphi Informant by John Penman, 30 January 1997

Host: www.informant.com
Host name: informant.com
IP address: 206.62.55.103

Messages
```
Received 56 bytes from 206.62.55.103 in 671 msecs
Received 56 bytes from 206.62.55.103 in 651 msecs
Received 56 bytes from 206.62.55.103 in 681 msecs
Received 56 bytes from 206.62.55.103 in 701 msecs
Received 56 bytes from 206.62.55.103 in 641 msecs
Sent 5 packets
Received 5 packets
Min/Avg/Max= 641/669/701
```

Figure 5: NetCheck after successfully pinging a host.

Conclusion

A growing need exists for networking applications to embed some connectivity checks for network integrity. Therefore, Delphi developers should use a simple network debugger to test their Internet applications and ensure their robustness. In Part II of this series, we'll extend NetCheck's capabilities to include the complementary components to Sonar — EchoC and Trace. See you then. ∆

References

Chapman, Davis, *Building Internet Applications with Delphi 2* [QUE, 1996].
Dumas, Arthur, *Programming Winsock* [SAMS, 1995].
Quinn, Bob and Shute, David, *Windows Sockets Network Programming* [Addison-Wesley, 1996].
Stevens, W. Richard, *UNIX Network Programming* [PTR Prentice Hall, 1990].
Taylor, Don, et al., *KickAss Delphi Programming*, Chapters 4 and 5 [Coriolis Group, 1996].
Verbruggen, Martien. Source code for the demonstration ping program is available on the Web from http://www.tcp.chem.tue.nl/~tgtcmv and http://www.delphi32.com/apps.

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\MAY\DI9705JP.

John Penman is the owner of Craiglockhart Software, a company specializing in providing Internet and intranet software solutions. John can be reached on the Internet at jcp@iafrica.com.

Begin Listing One — The *TSonar* Class

```
type
  TSonar = class(TComponent)
  private
    { Private declarations }
    FParent          : TComponent;
    FStatusWS        : Boolean;
    FNoSent, FNoRecv, FNoEchoes, FMin, FMax,
    FAve, FRTTSum    : Word;
    FHostName        : string;
    FOnRecvData, FOnResolveData, FOnProgress : TNotifyEvent;
    FSocket          : TSocket;
    FHwnd            : THandle;
    FwMsg            : Word;
    FSockAddr        : TSockAddr;
    FSockAddrIn      : TSockAddrIn;
    FVersion, FVersionDate,
    FComponentName, FDeveloper : string;
  protected
    { Protected declarations }
    FICMPAvail, FICMPDone : Boolean;
    FStatus          : Integer;
    FHostIP, FMsg    : string;
    FProgress        : LongInt;
    FAddress, FTimeOut : DWord;
    FOkay, FTrace    : Boolean;
    FPktSize, NoPackets : Word;
    FTTL             : Byte;
    FIPOptions     : TIPOptions;
    FPIPE          : pIcmpEchoReply;
    FHost          : PHostent;
    FProtocol      : PProtoEnt;
    procedure   OnRecvDataEvent;
    procedure   OnResolveEvent;
    procedure   OnProgressEvent;
    function    GetTTL : Byte;
    procedure   SetTTL(ReqdTTL : Byte);
    function    GetPktSize : Word;
    procedure   SetPktSize(ReqdPktSize : Word);
    constructor Create(AOwner : TComponent); override;
    destructor  Destroy; override;
    function    CheckWS : Boolean;
    function    CheckICMP : Boolean;
    procedure   GetHost;
    procedure   DoSonar;virtual;
    procedure   Stats;virtual;
  public
    { Public declarations }
    property  PingAvail : Boolean read FICMPAvail;
    property  Msg : string  read FMsg;
    property  HostIP : string read FHostIP write FHostIP;
    property  Progress : LongInt
      read FProgress write FProgress default 0;
    property  PktSize : Word
      read GetPktSize write SetPktSize default PacketSize;
    property  Version     : string  read FVersion;
    property  VersionDate : string  read FVersionDate;
    property  Name        : string  read FComponentName;
    property  Developer   : string  read FDeveloper;
    procedure Ping;
  published
    { Published declarations }
    property HostName : string
      read FHostName write FHostName;
    property NoPackets : Word
      read FNoPackets write FNoPackets default 5;
    property TimeOut : DWord
      read FTimeOut write FTimeOut default 2000;
    property OnRecvData : TNotifyEvent
      read FOnRecvData write FOnRecvData;
    property OnResolveData : TNotifyEvent
      read FOnResolveData write FOnResolveData;
    property OnProgress : TNotifyEvent
      read FOnProgress write FOnProgress;
  end;
```

End Listing One

Begin Listing Two — The ICMP Unit

```
unit ICMP;

interface

uses
  Windows, WinSock;

type
{ The following structures are required
  for calling the ICMP.DLL }
  pIPOptions = ^TIPOptions;
  TIPOptions = record
                 TTL          : Byte;
                 TOS          : Byte;
                 Flags        : Byte;
                 OptionsSize  : Byte;
                 OptionsData  : PChar;
               end;

  pIcmpEchoReply = ^TICMPEchoReply;
  TICMPEchoReply = record
                     Address  : DWord;
                     Status   : DWord;
                     RTT      : DWord;
                     DataSize : Word;
                     Reserved : Word;
                      Data    : Pointer;
                     Options  : TIPOptions;
                   end;
  TIcmpCreateFile = function : THandle; stdcall;
  TIcmpCloseHandle  = function (ICmpHandle : THandle) :
    Boolean; stdcall;
  TIcmpSendEcho = function (ICmpHandle : THandle;
    DestAddress : DWord; RequestData : Pointer;
    RequestSize : Word; RequestOptions : pIPOptions;
    ReplyBuffer : Pointer; ReplySize : DWord;
    TimeOut : DWord) : DWord; stdcall;

const
  IP_ERROR_BASE              = 11000;
  IP_SUCCESS                 = 0;
  IP_BUF_TOO_SMALL           = IP_ERROR_BASE;
  IP_DEST_NET_UNREACHABLE    = IP_ERROR_BASE + 2;
  IP_DEST_HOST_UNREACHABLE   = IP_ERROR_BASE + 3;
  IP_DEST_PROT_UNREACHABLE   = IP_ERROR_BASE + 4;
  IP_DEST_PORT_UNREACHABLE   = IP_ERROR_BASE + 5;
  IP_NO_RESOURCES            = IP_ERROR_BASE + 6;
  IP_BAD_OPTION              = IP_ERROR_BASE + 7;
  IP_HW_ERROR                = IP_ERROR_BASE + 8;
  IP_PACKET_TOO_BIG          = IP_ERROR_BASE + 9;
  IP_REQ_TIMED_OUT           = IP_ERROR_BASE + 10;
  IP_BAD_REQ                 = IP_ERROR_BASE + 11;
  IP_BAD_ROUTE               = IP_ERROR_BASE + 12;
  IP_TTL_EXPIRED_TRANSIT     = IP_ERROR_BASE + 13;
  IP_TTL_EXPIRED_REASSEM     = IP_ERROR_BASE + 14;
  IP_PARAM_PROBLEM           = IP_ERROR_BASE + 15;
  IP_SOURCE_QUENCH           = IP_ERROR_BASE + 16;
  IP_OPTION_TOO_BIG          = IP_ERROR_BASE + 17;
  IP_BAD_DESTINATION         = IP_ERROR_BASE + 18;
  // The next group is status codes passed up on status
  // indications to transport layer protocols.
  IP_ADDR_DELETED      = IP_ERROR_BASE + 19;
  IP_SPEC_MTU_CHANGE   = IP_ERROR_BASE + 20;
  IP_MTU_CHANGE        = IP_ERROR_BASE + 21;
  IP_UNLOAD            = IP_ERROR_BASE + 22;
  IP_GENERAL_FAILURE   = IP_ERROR_BASE + 50;
  MAX_IP_STATUS        = IP_GENERAL_FAILURE;
  IP_PENDING           = IP_ERROR_BASE + 255;

  // Values used in the IP header Flags field.
  IP_FLAG_DF    = $2;    // Don't fragment this packet.
  // Supported IP Option Types.
  // These types define the options that may be used
  // in the OptionsData field of the
  // ip_option_information structure.
  // See RFC 791 for a complete description of each.
  IP_OPT_EOL        = 0;     // End of list option
  IP_OPT_NOP        = 1;     // No operation
  IP_OPT_SECURITY   = $82;  // Security option
  IP_OPT_LSRR       = $83;  // Loose source route
  IP_OPT_SSRR       = $89;  // Strict source route
  IP_OPT_RR         = $7;   // Record route
  IP_OPT_TS         = $44;  // Timestamp
  IP_OPT_SID        = $88;  // Stream ID (obsolete)

  // Maximum length of IP options in bytes.
  MAX_OPT_SIZE   = 40;
  IP_TRACE             = IP_TTL_EXPIRED_TRANSIT;
  IP_REACHED           = IP_SUCCESS;
  IP_TIMEOUT           = IP_REQ_TIMED_OUT;
  ICMP_STATUS_BASE     = $80000100;
  ICMP_INVALID         = ICMP_STATUS_BASE + 0;
  ICMP_NO_ADDRESS      = ICMP_STATUS_BASE + 1;
  ICMP_CANCEL          = ICMP_STATUS_BASE + 3;
  ICMP_BUSY            = ICMP_STATUS_BASE + 4;
  MAX_TTL              = 255;
  MIN_PACKET_SIZE      = 8;
var
  IcmpSendEcho       : TIcmpSendEcho;
  wsaData            : TWSAData;
  LibCount           : Integer;
  hIcmpModule        : HModule;
  IcmpCreateFile     : TIcmpCreateFile;
  IcmpCloseHandle    : TIcmpCloseHandle;
  hIcmp              : THandle;
```

```
implementation

end.
```

End Listing Two

Begin Listing Three — The *TSonar.DoSonar* Procedure

```
procedure TSonar.DoSonar;
var
  BufferSize, nPkts  : Integer;
  pReqData, pData    : Pointer;
  pEchoReply         : pIcmpEchoReply;
  Count              : Integer;
begin
  { Creating handles to ICMP.DLL's functions. }
  if hIcmpModule <> 0 then
    begin
      @IcmpCreateFile  :=
        GetProcAddress(hIcmpModule, 'IcmpCreateFile');
      @IcmpCloseHandle  :=
        GetProcAddress(hIcmpModule, 'IcmpCloseHandle');
      @IcmpSendEcho     :=
        GetProcAddress(hIcmpModule, 'IcmpSendEcho');
      if (@IcmpCreateFile  = nil) or
         (@IcmpCloseHandle = nil) or
         (@IcmpSendEcho    = nil) then
        begin
          MessageDlg(
            'Error: can't get a handle to ICMP.DLL.',
            mtError, [mbOk], 0);
          Exit;
        end;
      hIcmp := IcmpCreateFile;
      if hIcmp = INVALID_HANDLE_VALUE then
        begin
          MessageDlg('Error: can't open file handle.',
                     mtError,[mbOk], 0);
          Exit;
        end;
      BufferSize := SizeOf(TICMPEchoReply) + FPktSize;
      { Prevent warning messages from the compiler. }
      pEchoReply := nil;
      pReqData   := nil;
      pData      := nil;
      try
        GetMem(pReqData, FPktSize);
        GetMem(pData, FPktSize);
        GetMem(pEchoReply, BufferSize);
        FillChar(pReqData^, FPktSize, $AA);
        pEchoReply^.Data := pData;
        FillChar(FIPOptions, SizeOf(FIPOptions), 0);
        FIPOptions.TTL  := FTTL;
        FMin       := 9999;
        FMax       := 0;
        FAve       := 0;
        FNoEchoes  := 0;
        FRTTSum    := 0;
        FOkay      := True;
        { We loop and ping. Blocking can occur here. }
        for Count := 1 to FNoPackets do begin
          MessageBeep(MB_OK);
          nPkts := IcmpSendEcho(hIcmp, FAddress, pReqData,
                     FPktSize, @FIPOptions, pEchoReply,
                     BufferSize, FTimeOut);
          FOkay := nPkts <> 0;
          try
            FProgress := (Count * 100) div FNoPackets;
            OnProgressEvent;
          except
            on EDivByZero do FProgress := 0;
          end; { try..except }

        with pEchoReply^ do begin
          case Status of
            IP_SUCCESS :
              begin
                MessageBeep(MB_OK);
```

```
                  FMsg := Concat('Received ',
                    IntToStr(DataSize),' bytes from
                  ',FHostIP,
                    ' in ',IntToStr(RTT),' msecs');
                  Inc(FNoEchoes);
                  if RTT > FMax then
                    FMax := RTT;
                  if RTT < FMin then
                    FMin := RTT;
                  FRTTSum := FRTTSum + RTT;
                end;
              IP_BUF_TOO_SMALL           :
                FMsg := 'Buffer too small';
              IP_DEST_NET_UNREACHABLE    :
                FMsg := 'Network unreachable';
              IP_DEST_HOST_UNREACHABLE   :
                FMsg := 'Host unreachable';
              IP_DEST_PROT_UNREACHABLE   :
                FMsg := 'Protocol unreachable';
              IP_DEST_PORT_UNREACHABLE   :
                FMsg := 'Port unreachable';
              IP_NO_RESOURCES            :
                FMsg := 'No resources';
              IP_BAD_OPTION              :
                FMsg := 'Bad option';
              IP_HW_ERROR                :
                FMsg := 'Hardware error';
              IP_PACKET_TOO_BIG          :
                FMsg := ‘Packet too large’;
              IP_REQ_TIMED_OUT           :
                FMsg := 'Request timed out';
              IP_BAD_REQ                 :
                FMsg := 'Bad request';
              IP_BAD_ROUTE               :
                FMsg := 'Bad route';
              IP_TTL_EXPIRED_TRANSIT     :
                FMsg := 'TTL expired in transit';
              IP_TTL_EXPIRED_REASSEM     :
                FMsg := 'TTL expired in reassembly';
              IP_PARAM_PROBLEM           :
                FMsg := 'Parameter problem';
              IP_SOURCE_QUENCH           :
                FMsg := 'Source quench';
              IP_OPTION_TOO_BIG          :
                FMsg := 'Option too big';
              IP_BAD_DESTINATION         :
                FMsg := 'Bad destination';
              IP_ADDR_DELETED            :
                FMsg := 'Address deleted';
              IP_SPEC_MTU_CHANGE         :
                FMsg := 'Specified MTU changed';
              IP_MTU_CHANGE              :
                FMsg := 'MTU changed';
              IP_UNLOAD                  :
                FMsg := 'Unload';
              IP_GENERAL_FAILURE         :
                FMsg := 'General failure';
              IP_PENDING                 :
                FMsg := 'Pending';
            end;  { case }
          end;  { with pEchoReply }
          OnRecvDataEvent;
        end;{ for }
        IcmpCloseHandle(hIcmp);
      finally
        FreeMem(pReqData, FPktSize);
        FreeMem(pData, FPktSize);
        FreeMem(pEchoReply, BufferSize);
      end;
    end;
end;
```

End Listing Three

Delphi / Object Pascal

By *Robert Vivrette*

Shell Games, etc.

Delphi Tips and Techniques

Adding a File to the Windows 95 Documents Menu

With the ever-growing popularity of Windows 95 and Windows NT, many Delphi developers are looking for better ways to integrate their applications into the operating system. Delphi makes this easy with the ShellAPI and ShlObj units.

One of the simplest things you can do is add a recently-accessed file to the **Documents** menu that is accessible from the **Start** menu in Windows 95 (see Figure 1). All that's needed to add an entry to this menu is a simple call to the **SHAddToRecentDocs** function in the Win32 API. To use this procedure, you must of course add the ShlObj unit to your **uses** clause. A sample of how the routine is called is:

```
procedure TForm1.Button1Click(Sender:
                              TObject);
begin
  SHAddToRecentDocs(
    SHARD_PATH,PChar(
      'c:\My Documents\Resume.doc'));
end;
```

The first parameter, SHARD_PATH, indicates that the second parameter specifies a path name to the recently-used document. In this example, the second parameter is a pointer to a buffer that contains the filename. After executing this call, there will be a reference to the Microsoft Word document Resume.doc in the **Documents** menu.

This technique is most useful in situations where the file has a registered association with your Delphi application. That way, selecting the item from the **Documents** menu will launch your particular Delphi application.

If you include NIL as the second parameter, all documents are cleared from the list.

Making Your Code Processor-Speed Independent

Remember the early days of PC games? The game ran great on your 8MHz IBM XT, but later, when you got a faster machine, the game ran so quickly it was unplayable.

Normally, we really don't care if a program runs more quickly. Faster is better, right? In some cases, however, you may need to limit how quickly a program or segment of code executes: an arcade-style game, for example, where things moving faster is not necessarily better.

Although there are a number of ways to do this (some more complex than others), one

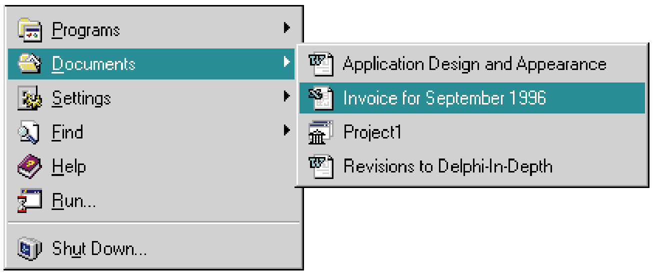

Programs
Documents
Settings
Find
Help
Run...
Shut Down...
Application Design and Appearance
Invoice for September 1996
Project1
Revisions to Delphi-In-Depth

Figure 1: The Windows 95 **Documents** menu.

simple way is to have the code take a little breather at regular intervals. The following code shows one technique for doing this:

```
PacingCount := GetTickCount;
repeat
  Application.ProcessMessages;
until (GetTickCount - PacingCounter) > 50;
```

First, a variable is created (*PacingCount*) to hold the result of a call to *GetTickCount*. This function retrieves the number of milliseconds that have elapsed since Windows was started.

Now, we go into a **repeat..until** loop until a new call to *GetTickCount* minus the saved value in *PacingCount* is greater than some predetermined amount of milliseconds. In this case, the application will stay in a loop until 50 milliseconds have passed.

Inside the loop we put an *Application.ProcessMessages* call to ensure other applications get some processor time. This call is very important in a non-preemptive operating system such as Windows 3.x, where the application must surrender time for other applications to run.

In a preemptive environment, the operating system takes control away from the application, so the *Application.ProcessMessages* call is not as critical.

With this code, any machine will be guaranteed to stay in the loop at least 50 milliseconds (or whatever value you determine). This will have the effect of causing faster machines to appear to execute a section of code no faster than that of a slower machine.

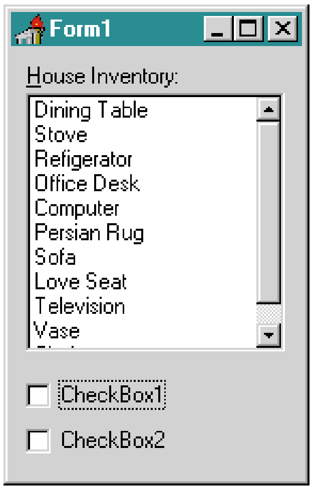


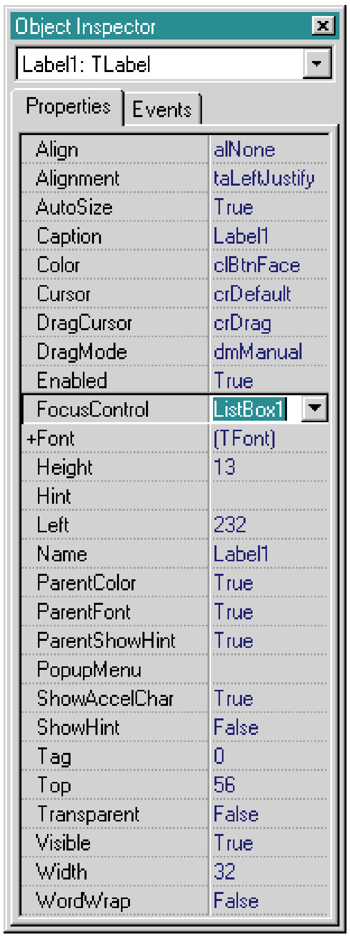
Figure 2: An accelerator key in a Label.

Tying Labels to List Boxes

One of the lesser-known properties of a Label component is the *FocusControl* property. This property links the Label's control with another control on the form. If the Label's caption includes an accelerator key, the control specified in its *FocusControl* property gets the focus when the user hits that accelerator key.

For example, Figure 2 shows a ListBox component with a Label component above it; the Label's *Caption* is `&House Inventory`. By placing an ampersand character in *Caption*, the character following the ampersand becomes the label's accelerator key, and is indicated as such on the form by being underlined.

Because a ListBox doesn't have its own *Caption* property, the Label serves that function. By specifying *ListBox1* in the Label's *FocusControl* property (see Figure 3), the focus will shift to the list box when the user hits Alt H (the "H" in House Inventory).



Align	alNone
Alignment	taLeftJustify
AutoSize	True
Caption	Label1
Color	clBtnFace
Cursor	crDefault
DragCursor	crDrag
DragMode	dmManual
Enabled	True
FocusControl	ListBox1
+Font	(TFont)
Height	13
Hint	
Left	232
Name	Label1
ParentColor	True
ParentFont	True
ParentShowHint	True
PopupMenu	
ShowAccelChar	True
ShowHint	False
Tag	0
Top	56
Transparent	False
Visible	True
Width	32
WordWrap	False

Figure 3: Using the *FocusControl* property of a Label component.

In the event the list box needs to be disabled, it generally gives a clearer signal to the user if you disable the label as well. That way, there is no confusion about whether the list box can obtain focus.

Remember that any *TWinControl* on a form can be chosen for a Label's *FocusControl* property, not only list boxes! ∆

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

By *The EDD Development Team*

California's CalJOBS Project

Developing Cost-Effective Information Solutions Using the Web

With an opportunity to revamp its job-matching system, the State of California Employment Development Department (EDD) turned to Borland's Delphi, HREF Tools' WebHub, Informix databases, and O'Reilly's WebSite Professional, to create an interactive, database-driven Web site, known as CalJOBS (California Job Openings Browse System).

The CalJOBS Web site offers employers and job seekers self-help options, and includes the ability to maintain the user's identity throughout a session, provide complex drill-down queries, and increase information-delivery speed with extreme modular scalability.

What the CalJOBS Project Sought to Provide

Initially, the site was established to offer an integrated employer/job seeker job service to anyone with Internet access. The site needed to allow employers to place and maintain job listings, and search the EDD's applicant pool for qualified applicants. Job seekers also needed access to list their applications or résumés with the EDD for employers to view. They also needed the ability to directly apply for jobs.

The CalJOBS Web site required more than simple static information, because customers accessing the site (job seekers or employers) would need to submit or retrieve specific information, tailored to their needs. CalJOBS needed to be connected to a searchable, amendable database, and it needed to generate dynamic, user-specific pages to match users' requests. For example, if you wanted a list of available construction jobs, you could make a request for that information (drawn from a database), which would then be presented in a browser. If you wanted to further define the search by entering your county, the original information would then be drilled down to include only those jobs offered in that county.

What Made This Possible?

After looking at many different tools and languages, the development team chose two technologies that enabled them to create a structured development framework. This framework consisted primarily of Delphi and WebHub.

With these tools, the team developed the initial site in approximately eight weeks. The team united the employer and employee job search features, connected the Web site to an independent client-server relational database, and developed the applications necessary to perform each of the basic job search and posting tasks. Delphi and WebHub gave them the solutions needed to create a Web site, with the capacity to expand.

WebHub addressed many issues, including:

- **Server-side Surfer Tracking**. The EDD's first prototype attempts used client-side tracking of user data, but it was limited. Then, in late 1995, the team found

Application Profile

The CalJOBS site unites employers and job seekers to exchange information about job openings and qualifications. The system also handles the administrative tasks of submitting completed job applications to interested employers.

Third-Party Products: Informix, O'Reilly's WebSite Professional, and HREF Tool's WebHub.

Informix Software	**O'Reilly & Associates, Inc.**	**HREF Tools Corp.**
4100 Bohannon Dr.	Software Products	300 B St.
Menlo Park, CA 94025	101 Morris St.	Santa Rosa, CA 95401
Phone: (415) 926-6300	Sebastopol, CA 95472	**Phone:** (707) 542-0844
Web Site: http://www.informix.com	**Phone:** (707) 829-0515	**Web Site:**
	Web Site:	http://www.href.com
	http://www.ora.com	

WebHub — which (at that time) was unique in providing a built-in, server-side saving state. This prevents the user from providing the same information multiple times.

- **Distributed Processing**. WebHub's architecture provided the team with a scalable solution that reduced required computing resources, and improved the speed at which information was delivered to the customer. The Hub distributes page requests to the least busy instance of the program. CalJOBS uses a "cluster" of two Web servers, which further distributes processing to multiple instances of the application for optimal performance of the transactions to a database server.
- **Multiple Open Queries**. WebHub database publishing components simplified the process of providing quick response via queries over the Web. Users are given a specified number of records in response to a query; WebHub keeps the answer available, enabling quick forward and backward scrolling.
- **Database Independence**. Database security and integrity is maintained by allowing it to operate independently from the Internet application, and implementing a captive browser using ActiveX technology. Delphi allows the application to communicate with key client-server relational databases (CalJOBS uses Informix).
- **Division of Labor**. WebHub provides an ideal development environment, because it allows separating the HTML from the SQL code. Development tasks could be divided amongst staff, and, unlike other Web application tools that embed SQL code within the HTML document, WebHub keeps the HTML in separate files, which are collected at run time.

Pilot Period (January 1997 to October 1997). The pilot program will provide automated employment services for job seekers and employers by introducing a full range of employment opportunities. EDD offices in eight counties will participate in the pilot: Butte, Kern, Los Angeles, Placer, Sacramento, San Bernardino, San Mateo and Ventura.

Access to CalJOBS will be restricted to employers and job seekers in the eight pilot counties. The EDD will only allow access in the eight counties to partner organizations such as Private Industry Council offices, Service Delivery Area offices, community colleges, and the county welfare department offices.

The Significance of This Project for California. The CalJOBS project not only marks the successful implementation of cutting-edge technology, it also sets the standard for high-quality, cost-effective government service programs. Using Delphi and WebHub, the EDD merged desktop computing, allowing people to access database information easier than they could from their personal computers. ∆

The EDD is a California State agency dedicated to providing California residents with job services at all levels, while promoting the development of new fields and encouraging employment opportunities throughout the state. For more information about the EDD CalJOBS project, contact Jesse Odell at jesse@fordodell.com or (707) 575-4543.

By Robin Karlin

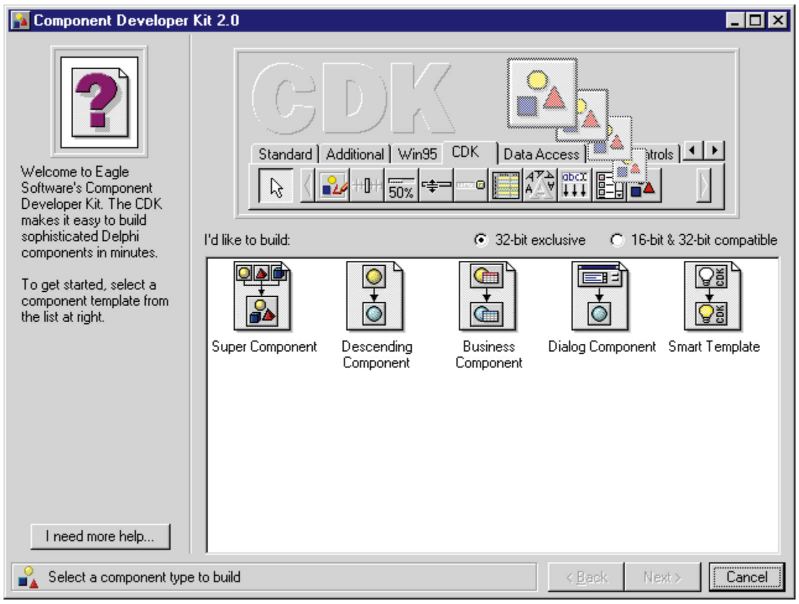
Component Developer Kit 2.0

A Dream Tool for Component Builders

Do you go to sleep at night thinking about all the wonderful Delphi components you could build with some extra time and expertise? Perhaps you'd build a compound component consisting of an edit box, a list box, and an **Add** button — you could type a new entry into the edit box, press **Add**, and your entry would be added to the list box. Or how about a currency control that displays negative numbers in red? Maybe a *TStringGrid* that sorts itself on any column when you click on that column's header ...

Maybe you've created components, but still have many questions about the process. Have you wondered when you must override the *Loaded* procedure? How to test if your code is called at run or design time? What the *Notification* method is used for? Or if there's a way to hide an inherited published property in a descendant component?

The Component Developer Kit version 2.0 (CDK) from Eagle Software can help you build the components described above, and learn the ins and outs of component construction. This "wizard-driven" tool provides step-by-step instruction through the component development process, and generates Delphi code with comments showing you where changes are needed.


Component Developer Kit 2.0

Welcome to Eagle Software's Component Developer Kit. The CDK makes it easy to build sophisticated Delphi components in minutes.

To get started, select a component template from the list at right.

I'd like to build: (● 32-bit exclusive) (○ 16-bit & 32-bit compatible)

Super Component | Descending Component | Business Component | Dialog Component | Smart Template

Select a component type to build

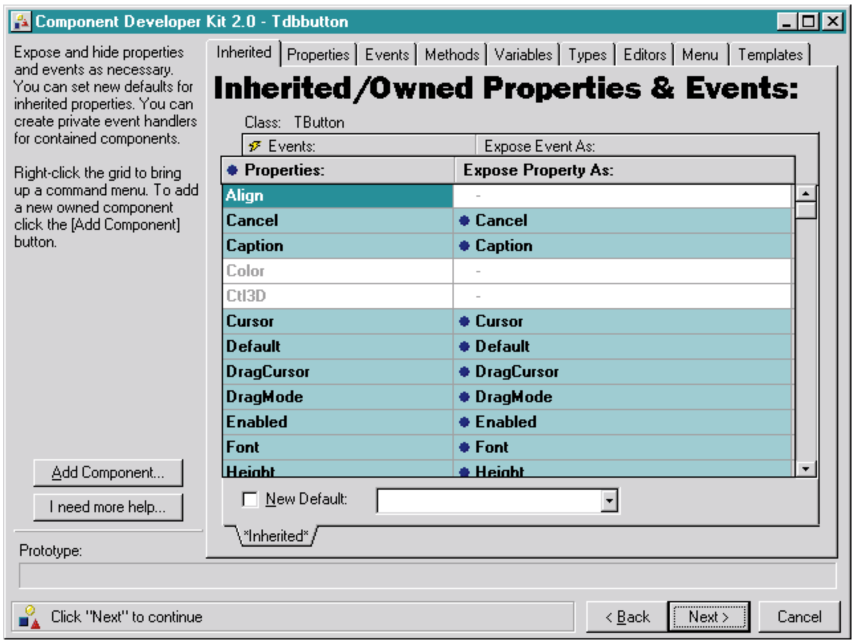
Figure 1: Welcome to the CDK.

Using CDK

The CDK installs easily, and seamlessly integrates with Delphi. It adds the following options to the **Component** menu: **Modify**, **Quick Install**, **Browser**, **New Test Program**, **Code Style**, and **Directories**. After installation, the **Component | New** command (**File | New | Component** in Delphi 1) invokes the main CDK interface, replacing Delphi's Component Expert.

When you select **Component | New**, the CDK displays five options: **Super Component**, **Descending Component**, **Business Component**, **Dialog Component**, and **Smart Template** (described in this article, these aren't a component type). Figure 1 shows the first screen in the CDK.

While some dialog boxes are specific to each of these options, the main CDK interface is a tabbed notebook (see Figure 2). Using information you enter in the notebook, the CDK generates declarations and skeleton code. Of course, you must understand


Component Developer Kit 2.0 - Tdbbutton

Expose and hide properties and events as necessary. You can set new defaults for inherited properties. You can create private event handlers for contained components.

Right-click the grid to bring up a command menu. To add a new owned component click the [Add Component] button.

Inherited | Properties | Events | Methods | Variables | Types | Editors | Menu | Templates

Inherited/Owned Properties & Events:

Class: TButton

Events:	Expose Event As:

● Properties:	Expose Property As:
Align	-
Cancel	● Cancel
Caption	● Caption
Color	-
Ctl3D	-
Cursor	● Cursor
Default	● Default
DragCursor	● DragCursor
DragMode	● DragMode
Enabled	● Enabled
Font	● Font
Height	● Height

Figure 2: The CDK's main interface is a tabbed notebook.

enough about component creation to know which properties, methods, events, editors, etc. you'll need.

Fortunately, the CDK's interface, the generously commented code it produces, and its clear manual do an excellent job of guiding you through the component creation process. After you have completed the information in the notebook and the Palette Image editor, the CDK previews it in a read-only window. To change something, you can back up as far as you want. Finally, you can choose whether to have the CDK build a test program, then press the **Finish** button.

Super Components. Without the CDK, building compound components is arduous. With the CDK, this task becomes much easier, although you must still keep track of many details. To help you learn about creating compound components, the CDK manual has step-by-step instructions for creating a "Super" component. It consists of three buttons (**OK**, **Cancel**, and **Help**) that align to the top right corner of your form.

The first step in creating a super component is to visually design it in Delphi. Start by selecting the CDK_Container component from the CDK tab on the Component palette and dropping it on a blank form. Then, place your sub-components in the CDK_Container. Make adjustments until you're satisfied with the super component's visual design.

Next, select the container component and copy it to the Clipboard. To include non-visual components, place them on the form outside the CDK_Container. To copy visual and non-visual components to the Clipboard, click on the container to select it, then Shift-click to select the non-visual components. Now you can select **Component | New** to start the CDK, and choose **Super Component**. After specifying name and class information, paste your super component into the waiting screen.

In the Inherited page of the tabbed notebook (again, see Figure 2), you must specify which properties and events of the sub-components and container component to publish in your super component. Be careful you don't expose two properties with the same name. Fortunately, the CDK makes this tedious job much easier. It can generate unique names for you, and allows you to rename individual properties.

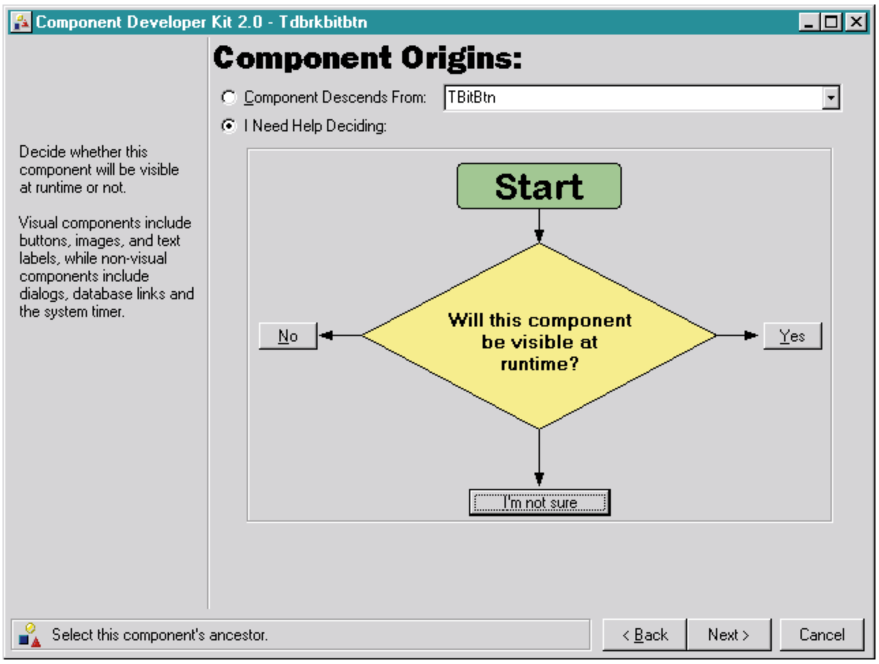
When the CDK generates the super component code, the CDK_Container becomes a *TCompoundComponentPanel*, the parent and owner of the sub-components. The CDK code initializes private properties, and provides for communication between components when handling events.

Descending Components. To create a component that adds functionality to an existing one, select the **Descending Component** option. The descending component will inherit the properties and methods of an existing component, and add new properties and methods, or override virtual methods in an ancestor class.

If you know which component you want to descend from, you can select it from a combo box listing all the components in your VCL. If you're unsure, click on the **I Need Help Deciding** radio button. The CDK presents a colorful flowchart (see Figure 3) that walks you through the available classes by asking questions such as, "Will this component be visible at runtime?" and "Is there a similar VCL Component that already exists?"

Business Components. Delphi's RAD model does not encourage developers to build three-tier applications. The RAD "path of least resistance" is to place business rules into the form unit. The unfortunate result of doing this is that business code becomes difficult to maintain and impossible to reuse.

Delphi 2 adds data modules that are supposed to be used as a middle tier. Business components, together with data modules, provide an even better solution to this problem. (The business components can and probably should be located in a data module in your project.) They provide a business rules layer that's separate from the UI and data-access layers. The advantage of this separation is that the business component can be reused independently of the UI and the data access for any particular form or application. Because business components are data-aware, you can still take advantage of the native Delphi data controls. The business components generated by the CDK descend


Component Developer Kit 2.0 - Tdbrkbitbtn
Component Origins:
Component Descends From: TBitBtn
I Need Help Deciding:
Start
Will this component be visible at runtime?
No
Yes
I'm not sure
< Back
Next >
Cancel

Decide whether this component will be visible at runtime or not.

Visual components include buttons, images, and text labels, while non-visual components include dialogs, database links and the system timer.

Select this component's ancestor.

Figure 3: This flowchart helps you pick a descendant.

from a pre-defined business component class developed by Ray Konopka. They are described in chapter 13 of his book, *Developing Custom Delphi Components* [Coriolis Group Books, 1996].

The CDK automates the process of creating business components. After you select the Business Component option, you're presented with the following:

- The Business Component - Field Selector screen allows you to specify a database, a table, and the fields to include in your business component.
- The Business Component - New Fields screen enables you to specify any new calculated fields to create. (It would be nice if you could also specify new lookup fields, but you must add that programmatically.)
- Finally, the Business Component - Field Options screen allows you to select from the following options (where applicable) for each field: Alignment, Min Value, Max Value, Edit Mask, Currency, Invisible, Read-Only, Required, and Validate.

When you're finished creating your business component, you can drop it on a form (or better yet, a data module), and you have an OOP middle tier that uses a *TTable* or *TQuery* to access the back end.

Eagle Software and Ray Konopka have just released a new version of the business component that's available (free to existing customers) on Eagle's Web site. When installed, the Business Component Wizard automatically integrates into the CDK. Some features of this new version are the ability to embed business components within other business components, and to link business components. The new version will also work with descendants of *TTable* and *TQuery*, such as InfoPower's *TwwTable* or *TwwQuery*.

Dialog and Data-Aware Components. The Dialog Component option enables you to convert any Delphi form into a non-visual component. The Dialog Encapsulation Options screen requests the name of the form and the name of the execution method (the default is *Execute*).

The CDK provides two Method templates for building read/write or read-only data-aware components. Let's say you want to create a data-aware button with its *Enabled* property controlled by the value in a string field in your database. Figure 4 shows the *DataChange* method generated by the read-only Data-Aware template. *DataChange* is triggered when data changes in the data source. Figure 5 shows code to set your button's *Enabled* property to *True,* if the data field contains the word "charge." A more useful version of the button would have a string property indicating which value (in the database) enables it.

Embedded Components. Using the main tabbed notebook, you can embed components in any component you're building. The difference between super and embedded components is that the latter do not require a CDK_Container. Embedded components are owned by your original component, and will be children of it, if they have a *Parent* property. This is especially useful for embedding non-visual components in a visual component. The example in the CDK manual is a rotating Label that's converted into a Clock by adding

```
procedure TDBButton.DataChange(Sender: TObject);
{ Triggered when data changes in DataSource. }
begin
  if FFieldDataLink.Field = nil then
    begin
      { CDK: Update your control to show there is no
        data link (optional). For example, if your
        control has a Caption property, you might use
        the following line:

        Caption := '*No Data Link*';
      }
      exit;
    end;
{ CDK: Update your control to reflect data change.
  For example, if this control were a descendant of
  TCalendar, you could use the following line:

    CalendarDate := FFieldDataLink.Field.AsDateTime;

  Other ways to look at data:
    AsBoolean
    AsDateTime
    AsFloat
    ...
}
end;
```

Figure 4: CDK generates plenty of helpful comments.

```
procedure TDBButton.DataChange(Sender: TObject);
{ Triggered when data changes in DataSource. }
begin
  if FFieldDataLink.Field = nil then
    begin
      { If there's no data link we won't do anything. }
      exit;
    end;
    { This will enable our button when the current string
      data field contains the word charge. The button
      will bring up a page of charge information. }
    Enabled := FFieldDataLink.Field.AsString = 'Charge';
end;
```

Figure 5: Altering the *Enabled* property of a Button object.

a Timer component to the original Label class. The *Timer* event is used to update the clock label.

Method Templates

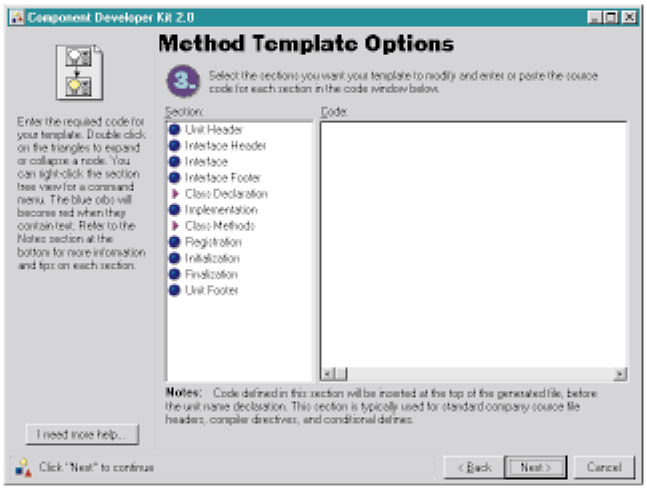
The CDK ships with many Method templates that provide common functionality for your components or other classes. Templates can be used with any class, including forms. Thus, they can be used to add common functionality to any application code.

The CDK's Keypress Filter, Component Link Handler, Runtime Drag, and Method Override templates are Smart templates. A Smart template displays dialog boxes allowing you to customize the template code before it's generated. For example, when adding the Keypress Filter template to your component, a dialog box requests information about what numbers, letters, and punctuation to allow. This reduces the amount of custom code you must write.

Get Smart

If you are not yet completely impressed by the CDK, read on. Today's software standards demand that a first-class tool be extensible. The CDK meets that criterion with its Smart template generation facility.

When you select the **Smart Template** option, a CDK "wizard" leads you through the steps for generating a Smart template (see Figure 6). The SDK's manual shows how to generate a Smart


Component Developer Kit 2.0

Method Template Options

3. Select the sections you want your template to modify and enter or paste the source code for each section in the code window below.

Enter the required code for your template. Double click on the triangles to expand or collapse a node. You can right-click the section tree view for a command menu. The blue orbs will become red when they contain text. Refer to the Notes section at the bottom for more information and tips on each section.

Section:
- Unit Header
- Interface Header
- Interface
- Interface Footer
- Class Declaration
- Implementation
- Class Methods
- Registration
- Initialization
- Finalization
- Unit Footer

Code:

Notes: Code defined in this section will be inserted at the top of the generated file, before the unit name declaration. This section is typically used for standard company source file headers, compiler directives, and conditional defines.

Figure 6: The Method Template Options dialog box.

template that adds code to set minimum and maximum sizes for a resizeable form. This code prevents the user from resizing a form below the minimum size or above the maximum size.

A Smart template is a .DLL, and the CDK generates all the code to create it. You programmatically connect any values that the user enters to variables you define. Then, you can use those values in the code that you cause the CDK to generate.

Component and Property Editors

Component editors provide a design-time interface for your components when you right- or double-click on them. This interface can be used to add functionality beyond what's available in the Object Inspector, or to make existing functionality more accessible. Delphi books usually consider component editors an advanced topic. The CDK makes it very easy to add your own component editors to the components you create.

Property editors provide a design-time interface for editing component property values. This is another advanced task that the CDK makes easier. However, you must understand how property editors work to successfully create one with the CDK. You will have to carefully study the generated code to see what you must change. As usual, the CDK provides plenty of help, as well as "Mr. CDK" Advice dialog boxes.

Modifying Existing Components

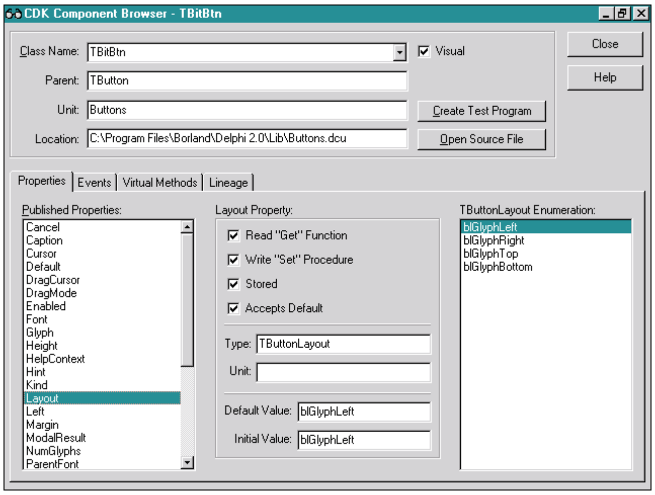
This is the CDK's one weak area. The CDK allows you to modify existing components (as opposed to creating a descending component class). This is useful when you have created a component, then realize you want to add or subtract functionality. Component creation, such as application coding, works best as an iterative process. The **Component | Modify** command allows you to expose or hide properties and events that you have not previously modified, and to add new components on-the-fly.

The weakness arises because the CDK doesn't display your previous work, or allow you to modify it. Properties and events you have exposed or hidden don't appear in the Modify dialog box. In a warning, Mr. CDK explains: "The CDK does not perform a deep scan of the code you are modifying and therefore doesn't know which sub-component events and properties you've already published. This also means that if you try to re-expose a property or event that is already exposed in the code, the CDK will let you." The suggested solution is to be familiar with your code to avoid mistakes.

The CDK Component Browser

The CDK Component Browser is a tabbed notebook that helps you investigate Delphi's class hierarchy and the structure of individual classes (see Figure 7). The Properties page provides information about the implementation of each property, including whether it uses procedural access specifiers, has a default value, etc. The Events page displays the declaration of each event type.

The Virtual Methods page shows the inherited virtual methods for components that are part of the Delphi 2 VCL. It


CDK Component Browser - TBitBtn

Class Name: TBitBtn ☑ Visual
Parent: TButton
Unit: Buttons
Location: C:\Program Files\Borland\Delphi 2.0\Lib\Buttons.dcu

Close | Help | Create Test Program | Open Source File

Properties | Events | Virtual Methods | Lineage

Published Properties: Cancel, Caption, Cursor, Default, DragCursor, DragMode, Enabled, Font, Glyph, Height, HelpContext, Hint, Kind, Layout, Left, Margin, ModalResult, NumGlyphs, ParentFont

Layout Property:
☑ Read "Get" Function
☑ Write "Set" Procedure
☑ Stored
☑ Accepts Default

Type: TButtonLayout
Unit:
Default Value: blGlyphLeft
Initial Value: blGlyphLeft

TButtonLayout Enumeration: blGlyphLeft, blGlyphRight, blGlyphTop, blGlyphBottom

Figure 7: The CDK Component Browser showing properties information for the *TBitBtn* control.

displays the declaration of each virtual method and a reference to the component that first declares the method. The Lineage page depicts the entire ancestry of all the components in the VCL, including new ones (a handy feature that's missing from Delphi's Browser).

The interface of the CDK Component Browser is more straightforward than Delphi's, and it provides some information that you must otherwise search the VCL source files to find. However, it does not replace the functionality of the Delphi Browser. This is because CDK's Component Browser doesn't include information from the private and protected sections of your components, or new virtual methods declared in your new components.

But Wait, There's More

The CDK ships with an evaluation copy of reAct for Delphi 2.0, a component testing tool. This limited version tests up to 10 properties and five events. reAct is a useful tool for testing your new components, standard Delphi components, and third-party components. reAct is worthwhile even for the experienced developer, because it transforms testing from drudgery to an easy, even enjoyable, task.

In addition to the CDK_Container mentioned, the CDK ships with 24 other components, some of which are quite useful. The highlights are: the CDKAnimation component, which animates a series of bitmaps specified by the user; the CDKLight, which displays an "LED" light in different colors, with an "on" or "off" state; and the CDKCheckList, a list box with check boxes next to each item.

Conclusion

The Component Developer Kit version 2.0 makes component-building faster and less tedious for novice and experienced developers. It's also a valuable tool for learning about the nuts and bolts of component building. The CDK manual is clearly written, with plenty of illustrations and tutorials; especially useful are tips identified with a small Mr. CDK graphic.

Other Eagle Software products, including a professional programmer's editor and a free upgrade of the CDK (to existing customers) to compile under Delphi 3, are in the works. Try the CDK. It will help your late-night component "fantasies" become reality. ∆

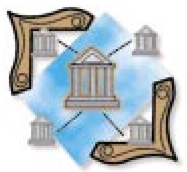
INFORMANT FACT FILE

The Component Developer Kit version 2.0 from Eagle Software assists developers in building Delphi components. This product installs easily and integrates seamlessly into Delphi. Those using the CDK must have rudimentary knowledge of component creation; however, the CDK interface, the generously commented code it produces, and the well-written manual are excellent guides through the component creation process. The CDK fully supports 16- and 32-bit components within Delphi, and comes with an evaluation copy of reAct for Delphi 2.0, a component tester and debugger. The CDK also comes with 25 VCL components. The CDK will be an excellent addition to your suite of third-party Delphi development tools.

Eagle Software
12021 Wilshire Blvd., Ste. 655
Los Angeles, CA 90025

Phone: (310) 441-4096
Fax: (310) 441-1996
Web Site: http://www.eagle-software.com
Price: US$279, with a 60-day, money-back guarantee.

Robin Karlin is a Senior Software Developer at PCSI, a leading client/server and Internet/intranet consulting and development firm. She is a Borland Certified Developer, and specializes in component design and object-oriented programming solutions. She can be reached at (201) 816-8002 or by e-mail at rkarlin@pcsiusa.com.



By Chris McNeil

Full Report Control

Building a Custom ReportSmith Component

Are we there yet? With the advent of modern travel, virtually all destinations are within reach. However, as with most things, no single mode of transportation is perfect. In air travel, for example, it seems most flights require a time-consuming layover. Printing a ReportSmith report from a Delphi application can also involve a delay. In fact, quite the rigmarole is required to print to a specific destination — other than the Windows default — using a Report object. You must minimize your application and select the required print driver as the Windows default printer.

Typically, an application with printing capabilities will also provide a means of setting the print destination. However, when using a ReportSmith report within a Delphi application, the ReportSmith Runtime Viewer formats the report and ultimately sends the report to the printer, while the Delphi application handles the user interaction.

ReportSmith Implementation under Delphi

Borland's implementation of ReportSmith under Delphi is the Report component (its properties are listed in Figure 1). It encapsulates the DDE communication between the Delphi application and ReportSmith, so you can:

- load ReportSmith,
- open an existing report (ReportSmith reports have an .RPT extension), and
- preview the report on screen (the value of the *Preview* property is *True*), or
- print the report while ReportSmith remains minimized (the value of *Preview* is *False*).

In preview mode, the Report object starts the ReportSmith Runtime Viewer in a restored (non-minimized) state and loads the report. Then, you can either navigate through the report or print it. Printing can be directed to specific printers by selecting **File | Print Setup** from the menu. This command displays the Print Setup dialog box and allows you to select any Windows-installed printer. Note that this is possible because the Runtime Viewer is the active application and is issuing the print commands to Windows.

If, however, the *Preview* property is *False* and you execute the *Run* (or *Print*) method of the *Report* object, ReportSmith starts minimized, loads and formats the requested report, then prints the report automatically. Notice, however, that you never get an opportunity to establish the print destination. Therefore, ReportSmith has no alternative but to print to the Windows default printer.

Is There an Alternative?

Inspection of the *TReport* class source code (the file name is REPORT.PAS) shows that a method named *Print* is executed when *Preview* is *False*. This assumes that

Property	Description
AutoUnload	Determines whether ReportSmith Runtime unloads from memory when you have finished running a report.
EndPage	Specifies the last page of the report. The default value is 9999; if the report is fewer than 9999 pages and you don't change the value of *EndPage*, your entire report is printed.
InitialValues	List of report variable strings the specified report requires to run. By specifying these initial values, your application can bypass the dialog boxes that prompt you for these values when the report runs.
Preview	Determines whether a report should be viewed on screen or printed. If *Preview* is *True*, the report appears on screen when the report is run. If *Preview* is *False*, the report is printed.
PrintCopies	Determines how many copies of the report will print when you run a report. The default value is 1.
ReportDir	The directory where ReportSmith expects to find saved reports. By specifying a report directory, you won't have to include a path when specifying a report name.
ReportName	Determines which report you want to run. You can include a full path name as part of the report name if you have not specified a *ReportDir* property value or want to run a report that is stored elsewhere. If you have specified a *ReportDir* value, omit the path name and simply specify the name of the report.
StartPage	Determines the page from which you want the report to start printing. The default value is 1, indicating the first page. You can change that value to begin printing the report on some other page.

Figure 1: Common properties associated with a *TReport* object.

ReportSmith and the requested report have been successfully loaded. Depending on the version of Delphi you have, the *Print* method interacts with ReportSmith in different ways.

Delphi 1 Implementation. Under Delphi 1, the *TReport.Print* method issues the following ReportBasic macro to ReportSmith:

```
PrintReport StartPage$, EndPage$, "Printer$",
            "Port$", "Driver$", Copies$
```

Notice that this command does support the *Printer$*, *Port$*, and *Driver$* parameters. However, the ReportSmith Help file states: "To use the default printer, use null strings for the Printer$, Port$, and Driver$ arguments." Because a *Report* object doesn't have properties to support printer information, the *Print* method simply issues:

```
PrintReport 1, 9999, "", "", "", 1
```

Delphi 2 Implementation. Under Delphi 2, Borland significantly updated the interface to ReportSmith by writing a new ReportSmith API that dodges some pitfalls of the DDE. Borland has replaced most of the DDE communication with calls to their new ReportSmith API. One of the functions of this API is *RS_PrintReport*; its parameters match the PrintReport macro exactly. Therefore, the *TReport.Print* method executes this function:

```
{ Assuming default property values }
RS_PrintReport(1, 9999, nil, nil, nil, 1)
```

Notice that Borland continues to leave the printer information **nil**, which still forces ReportSmith to print to the Windows default printer.

We can change all that!

The *TRptSmith* Component

Armed with this information, I set out to create a *TReport* descendent that would print to any printer defined to Windows. The only method that must be overridden in *TReport* is *Print*. It's a **public** method; however, it's not declared as **dynamic** or **virtual**, so it cannot be overridden.

This leaves us with two choices for replacing *Print*'s functionality. One: We can duplicate the function name in our descendent object, effectively hiding the ancestor implementation. This would also require us to duplicate every other static method that calls *Print* and every static method that calls those methods, and so on. Conceivably, we could end up with a complete copy of *TReport* under a new name. This solution isn't possible, because it provides too much source code that isn't in the public domain.

Two: The alternative is to modify the existing *TReport* and make *Print* a virtual method by adding the **virtual** directive:

```
function Print: Boolean; virtual;  { Delphi 1 }

function Print: Integer; virtual;  { Delphi 2 }
```

If you don't have the original source code, the modified REPORT.DCU files for Delphi 1 and 2 are available for download (see end of article for details). Before proceeding, however, be aware that this implementation requires a minor change to the VCL.

There is a further, special consideration for Delphi 2. The ReportSmith API is implemented using a Windows .DLL that is loaded explicitly using the Windows API function, **LoadLibrary**. Internally to *TReport*, Borland has defined the ReportSmith API as local variable references to functions in the .DLL. This means that descendent objects have no way to call these functions directly.

To circumvent this shortcoming, I created a new protected method named *PrintTo*. Protected methods have the benefit of being visible to descendent objects. The *PrintTo* method is declared as follows:

```
function PrintTo(ADevice, APort, ADriver: string): Integer;
```

Figure 2 shows the implementation of *PrintTo*. It's functionally equivalent to *Print,* except it allows printer information to be passed in. In addition, it can call *RS_PrintReport*. It would have been possible to call the

```
function TReport.PrintTo(
  ADevice, APort, ADriver: string): Integer;
begin
  if not Busy then
    Result := RS_PrintReport(
               StartPage, EndPage,
               PChar(ADevice),    { Requires a PChar }
               PChar(APort),      { Requires a PChar }
               PChar(ADriver),    { Requires a PChar }
               PrintCopies)
  else
    Result := RS_BUSY;
end;
```

Figure 2: *TReport.PrintTo* is the new protected method of Delphi 2 required to call RS_PrintReport.

RunMacro method using the PrintReport macro (similar to the Delphi 1 implementation). However, I speculate that Borland updated the interface for a reason, so I chose to stay with their approach.

Using the modified *TReport*, we can declare our new component as follows:

```
type
  TRptSmith = class(TReport)
    public
      constructor Create(AOwner: TComponent); override;
      destructor Destroy; override;
      {$ifdef WIN32}
        function Print: Integer; override;  { Delphi 2 }
      {$else}
        function Print: Boolean; override;  { Delphi 1 }
      {$endif}
  end;
```

The Printer Object

Before getting into the details of the *TRptSmith Print* method, let's look at how Delphi manages printer information. Delphi defines a global object named *Printer* that is of type *TPrinter*. This object is accessible by including the Printers unit in a **uses** statement. The *TPrinter* object contains information about all the printers currently defined to Windows. It gleans this information by using WIN.INI or the Registry. The table in Figure 3 lists commonly used properties of *TPrinter*; the table in Figure 4 lists its commonly used methods.

Normally, *TPrinter* is used to manually print information directly to a specific printer using the *BeginDoc* and *EndDoc* methods and the *Canvas* property. In addition, by dropping a *TPrintSetupDialog* component on your form, you enable your application to have instant access to a facility to change the currently selected printer for your application. Any print commands issued through *TPrinter* would then print to the new destination. This frees us from a lot of hassle over print control.

Even if you don't need to print directly from Delphi, the *TPrinter* object can provide valuable information about the currently selected printer. The method that we're interested in is named *GetPrinter*. It returns the following information about the currently selected printer:

Property	Description
Canvas	Represents the surface of the currently printing page.
Orientation	Determines if the print job prints vertically or horizontally on a page. The possible values are: *poPortrait* and *poLandscape*.
PrinterIndex	Specifies which printer listed in the *Printers* property is the currently selected printer. To select the default printer, set the value of *PrinterIndex* to -1.
Printers	List of all printers installed in Windows.
Title	Determines the text that appears in the Print Manager and on network header pages for the current print job.

Figure 3: Common properties associated with a *TPrinter* object.

Method	Description
BeginDoc	Sends a print job to the printer. If the print job is sent successfully, the application should call *EndDoc* to end the print job. Printing won't start until *EndDoc* is called.
EndDoc	Ends the current print job and closes the text file variable. After the application calls *EndDoc*, the printer begins printing. Use *EndDoc* after successfully sending a print job to the printer.
GetPrinter	Retrieves the current printer.
NewPage	Forces current print job to begin printing on a new page.
SetPrinter	Specifies the current printer.

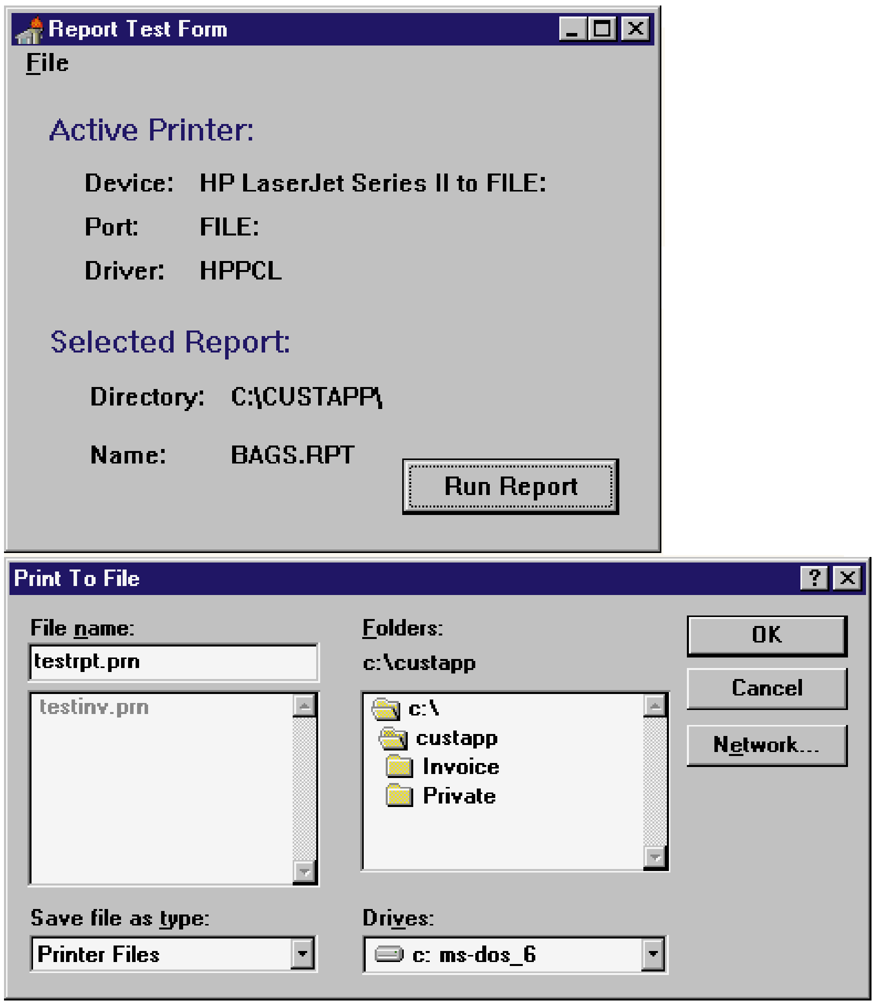
Figure 4: Commonly-used *TPrinter* methods.

- device name
- port
- driver
- printer handle

Do any of these look familiar? We can directly use the first three of these parameters in our component. The code for the *TRptSmith* component is available for download (see end of article for details). Note that there are two versions conditionally compiled into the VCL, depending on your version of Delphi. In both instances, it calls the *TPrinter.GetPrinter* method to retrieve information about the currently selected printer, and passes this information to ReportSmith.

I have provided a sample Delphi application that allows you to select any report on your system, as well as any printer defined to your system. When you press the **Run Report** button, RS_RunTime is started, and the report is printed to the requested destination. The rest of the *TReport* methods and properties are unchanged and work as expected. Figure 5 shows RS_RunTime displaying the Print To File dialog box. Figure 6 shows RS_RunTime about to fax a report (by selecting a WinFax print driver). Of course, you can select a physical printer as well.

One interesting note about Delphi 2 is that the print driver is not returned by *GetPrinter*. It seems that the Win32 API doesn't



Active Printer:

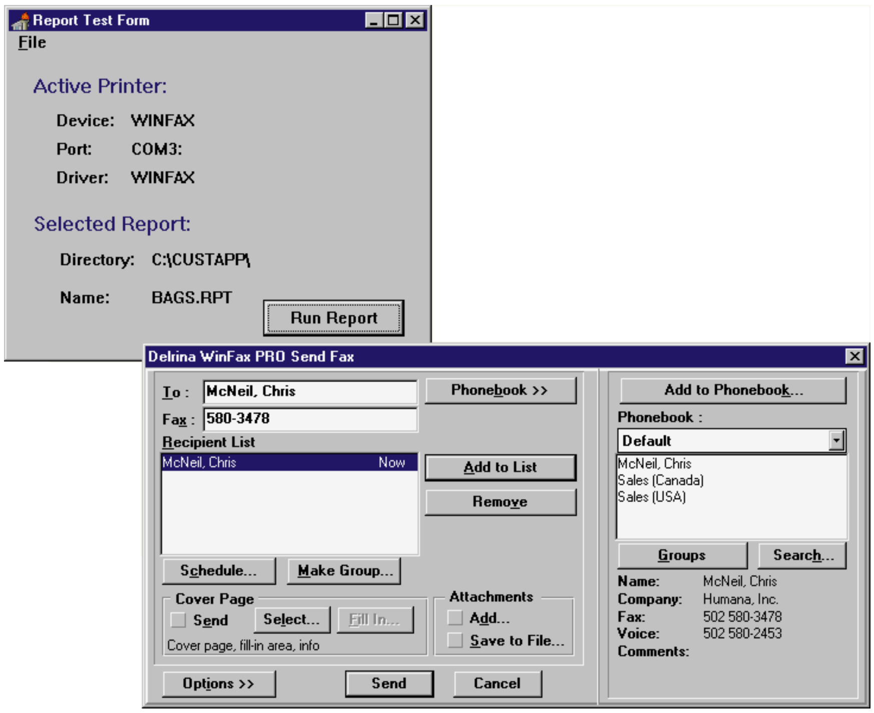
Device: HP LaserJet Series II to FILE:
Port: FILE:
Driver: HPPCL

Selected Report:

Directory: C:\CUSTAPP\
Name: BAGS.RPT

File name: testrpt.prn
testinv.prn
Folders: c:\custapp
c:\
custapp
Invoice
Private
Save file as type: Printer Files
Drives: c: ms-dos_6

Figure 5: The Report Test Form with the **Port** set to `FILE:`. Below is the result generated by ReportSmith.



Active Printer:

Device: WINFAX
Port: COM3:
Driver: WINFAX

Selected Report:

Directory: C:\CUSTAPP\
Name: BAGS.RPT

To : McNeil, Chris
Fax : 580-3478
Recipient List
McNeil, Chris Now
Cover Page
Send
Cover page, fill-in area, info
Attachments
Phonebook : Default
McNeil, Chris
Sales (Canada)
Sales (USA)
Name: McNeil, Chris
Company: Humana, Inc.
Fax: 502 580-3478
Voice: 502 580-2453
Comments:

Figure 6: The Report Test form has the **Port** set to COM3. Below is the result generated by ReportSmith.

require this parameter. In fact, by passing **nil** as the print driver, Windows 95 uses the device name and port to derive which print driver to use.

Conclusion

Are we there yet? Our implementation isn't perfect, because it relies on an external object, *TPrinter*, to provide several significant pieces of information. However, given the nature of the *TPrinter* object, it's doubtful that its external interface will change significantly. It also modifies the VCL source which can be problematic. With a little care, however, this technique provides a simple and functional implementation of print destination capabilities for the *Report* component. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\MAY\DI9705CM.

Chris A. McNeil is an independent software developer in Louisville, KY specializing in Delphi and Paradox for Windows database solutions. You can reach Chris on CompuServe at 72734,2270.

Delphi Programming Problem Solver

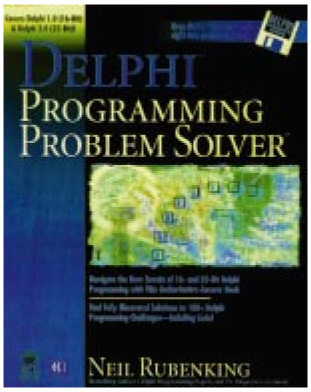
Neil Rubenking's *Delphi Programming Problem Solver* has created quite a buzz since it appeared, but I think not nearly enough. While it's possible to look at this book as just a collection of solutions to common problems, I tend to view it more as a short course in Windows programming. These are my favorite books — the ones that pop the hood and tinker around with the engine.

A lot of people call themselves Windows programmers. Since the first appearance of Microsoft Visual Basic, people with no clue about Windows' underlying structure have been writing Windows programs — some of them decent, some of them not — and being well paid for their efforts. Delphi helped accelerate that trend. Usually, I think this is a great advancement, until I install some of their software. The real problem isn't with programmers who write things within the confines of the systems they use. The problem is with developers who try to do things more intricate than their development environments allow.

They need this book. There are better books on the IDE, and certainly better books on database programming; only one chapter discusses database programming — display issues more than anything else. However, there's no better book on using the Windows API and messaging system from Delphi.

Although only a 28-page chapter on messaging and a 39-page chapter on the API are listed in the table of contents, they're dealt with in every chapter. The reader is grabbed early and thrown into the volcano. Not three pages into *Solver*, a procedure is presented to respond to *WM_NCHITTEST*! Mostly concerning himself with problems that can't be resolved within the confines of the VCL, Rubenking either grabs a Windows message, or derives a new component that responds to messages its ancestor doesn't. He uses API calls constantly. In short, *Solver* is the most compact tutorial on Windows programming I've seen.

Rubenking assumes his readers are competent programmers with years of Windows experience. What a refreshing approach. Some of the program samples took me a little longer to understand than others, but after presenting them, Rubenking comes back with short



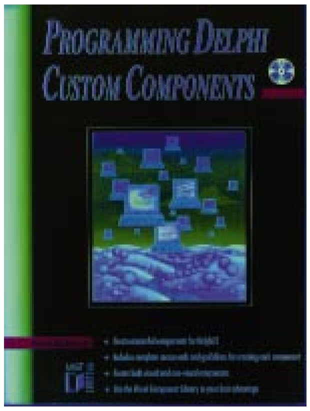
"Delphi Programming Problem Solver" continued on page 57

Programming Delphi Custom Components

While some early Delphi books contained an obligatory chapter on component writing, it took quite a while for the first book dedicated to this important topic to appear. That book, Ray Konopka's *Developing Custom Delphi Components* [Coriolis Group Books, 1996], became an instant hit, and is now considered a classic by many (see Richard Wagner's insightful review in the June, 1996 *Delphi Informant*).

Fred Bulback's *Programming Delphi Custom Components* was published not long afterwards. While my impression of *Programming* is largely positive, there are certain aspects I take issue with. First, the publisher categorizes the target reader's level as "Intermediate/Advanced." Bulback accurately articulates the prerequisites in the introduction as being for those "somewhat familiar with the Pascal programming language" and having "at least a basic understanding of the Delphi environment." What level does that sound like to you?

Also at issue is the nature of the introduction. Entitled "Introduction to Delphi," this 52-page chapter spends too much time covering fundamental topics, such as the nature of properties and Delphi's relationship to other programming languages. In fairness, the author discusses these topics in an interesting and insightful manner.



Furthermore, he includes topics that are often omitted, such as the function of **initialization** and **finalization** sections, and using array properties. Still, I prefer a more focused discourse.

Following this introduction, Bulback gets to the business

"Programming Delphi Custom Components" continued on page 57

Delphi Programming Problem Solver (cont.)

explanations of some of the trickier elements. Like a good teacher, he knows what to emphasize and what to let us figure out on our own. In particular, in the last chapter — a great .DLL primer — we are frequently reminded to use types common to most languages for .DLL function returns.

Solver is broken into five sections, and the chapters in those sections are filled with program examples. Nearly every programming problem includes a full code solution in the text. I know that when companion disks contain complete source code (as is the case here) many people prefer to see only code fragments in the text, but I've always preferred it in front of me. Often, a solution consists of one long code listing with two or three short, yet telling, remarks. Some readers might disagree, but this style is perfect for me.

Computer books are never perfect, and *Solver* is definitely a computer book. Other than one figure duplicated from chapter two to chapter six, however, the mistakes aren't worth mentioning. What is worth mentioning repeatedly is how much you need this book. Whether you're a casual programmer who needs to figure out a few problems, or an experienced developer who needs a quick refresher on Windows messaging, *Delphi Programming Problem Solver* is for you.

— Richard A. Porter

Delphi Programming Problem Solver by Neil Rubenking. IDG Books, 919 E. Hillsdale Blvd., Ste. 400, Foster City, CA 94404, (800) 762-2974.

ISBN: 1-56884-795-5
Price: US$34.00
574 pages, Diskette

Programming Delphi Custom Components (cont.)

of creating new components. Here the value of *Programming* becomes apparent. Beginning with the simplest of non-visual components, the author introduces increasingly more complex components. Not counting the trivial component with which he begins, Bulback introduces seven components: a WIN.INI file watcher, a serial communications component, a custom About box, an LED gauge, a check grid, a color-selection combo box, and a printer component.

TWinIni is a simple, non-visual component that watches for and reports changes to WIN.INI using the *WM_WININICHANGE* message. It demonstrates how to create and use Delphi events, specifically *TNotifyEvent*.

Another interesting component, *TComm*, provides access to basic serial communications functions, such as opening and closing the COM port, setting configuration options (baud rate, parity checking, etc.), and transmitting data. *TComm* is basic; you'll need to add functionality and perhaps additional components for most actual communications applications. To his credit, Bulback provides an excellent conceptual framework and foundation on which to build, and even points out where the shortcomings lie.

I found the printing component particularly interesting. The print preview capability of this component is attractive and useful. With this and other components, Bulback demonstrates his knowledge and skill in working with Windows graphics. If you're new to this, you'll gain valuable experience working with *TCanvas*, brushes, pens, and other graphics elements.

The book concludes with a chapter on property and component editors. However, if you are an experienced component writer and are looking for more information on these last two topics, I recommend Ray Lischner's *Secrets of Delphi 2* [Waite Group Press, 1996] because of its greater depth and detail.

Which Delphi developers are likely to benefit from *Programming*? I can't recommend it to advanced Delphi developers unless they have a special need in one of the component areas discussed. However, I think this is an excellent introduction to the topic for the Delphi programmer new to component writing. The detailed, basic information included throughout *Programming Delphi Custom Components* will be particularly helpful and appropriate for the less-experienced programmer.

— Alan C. Moore, Ph.D.

Programming Delphi Custom Components by Fred Bulback. M&T Books, 4375 West 1980 South, Salt Lake City, UT 84104, (800) 488-5233.

ISBN: 1-55851-457-0
Price: US$39.95
420 pages, CD-ROM

Goliath Lives

The release of Delphi 3 gives us an opportunity to look at the brief history of Borland's flagship product and see how it stacks up in the marketplace. When Borland introduced Delphi 1 in early 1995, I heard many Scotts Valley supporters tout it as the "giant killer," fully expecting Delphi to dominate the Windows client/server development tool market. That didn't happen.

The good news for Delphi developers in 1997 is that the tool has legitimized itself and gained strong market acceptance. If we're honest, however, we must also admit that it's not about to rule the world. With its first two versions, Delphi has penetrated the Visual Basic/-PowerBuilder fortress, but it hasn't broken the bulwark wide open. And, while Delphi 3 may be a "must" upgrade for developers already using Delphi, its new features are not sexy enough to win over massive droves of VB, PowerBuilder, or C++ developers.

No VB Killer. It's now apparent that Delphi will never significantly penetrate the VB developer marketplace. When Delphi was introduced, its technical superiority was compelling, enough so that it was able to win the hearts of many VB programmers. But this number was never the legions that Borland had hoped for. The problem for Borland in 1997 is that any window of opportunity for converting VB developers has largely closed. Those who have remained steadfast with VB will not likely find anything in Delphi 3 to win them over. Essentially, Visual Basic 5.0 is *good enough* for most VB developers.

C++ and Java. Delphi is not the "C++ Killer" either. I know of many C++ programmers who switched to Delphi, but any C++ developer who hasn't already converted has little reason to do so given the release of Delphi's sister product, C++Builder. In fact, I suspect many C++ developers who switched to Delphi will likely spend an increasing amount of time with C++Builder.

The relentless Java hype has also taken its toll on Delphi. Just when Delphi 2 was gathering steam, Java altered the development tool landscape, bringing more attention to multi-platform Web development rather than the technical merits of Windows desktop development tools.

Looking Ahead. Given this reality, what must Borland do to ensure Delphi remains a viable development tool? Allow me to offer two suggestions.

First, Borland must continue to focus their attention on keeping Delphi as the premier Windows desktop and client/server development environment. While RAD software vendors have been running scared given the Internet hype, a recent study by Sentry Research Services suggests that Internet-based technologies will not replace client/server as the dominant computing model. Therefore, while Delphi 3 has some interesting Web deployment technology that will serve some of you, Borland cannot let the Web radically alter future development efforts. Borland should focus on Delphi's core strength: making Windows programming easier. Second, Borland must continue to enhance Delphi's database access. I've long thought of the BDE as the Achilles' heel of Delphi. By and large, developers tend to tolerate the BDE, not embrace it. The new BDE of Delphi 3 is a step in the right direction, providing much easier ODBC access. Before you hear me rave about Delphi's database prowess, however, I want to see the elimination of the quirks often associated with SQL connectivity which have given many Delphi SQL developers headaches since version 1.

No, Delphi is not a David preparing to defeat the mighty Goliath up north in Redmond, but it does have two key factors in its favor: technological prowess and a sizable, loyal developer base. So long as Borland (or another suitor) continues to enhance Delphi's core strengths, you can continue to feel comfortable adding semicolons to the end of your code statements. ∆

— Richard Wagner

Richard Wagner is Chief Technology Officer of Acadia Software in the Boston, MA area, and is Contributing Editor to Delphi Informant. *He welcomes your comments at rwagner@acadians.com.*