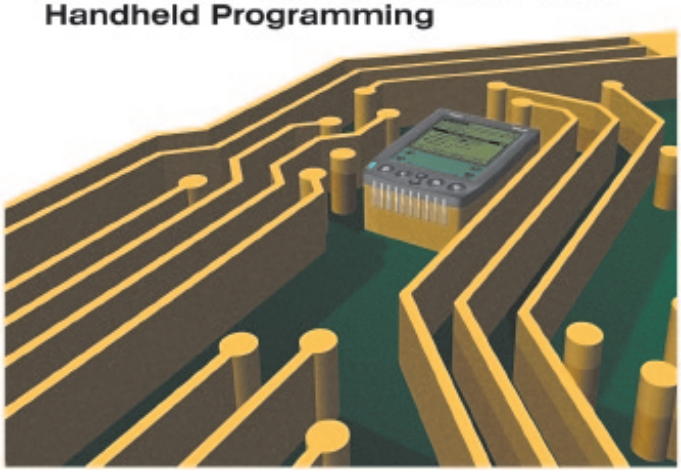
June 2000, Volume 6, Number 6

Palm Conduits

Handheld Programming



Cover Art By: *Arthur Dugoni*

ON THE COVER

FEATURES

REVIEWS

DEPARTMENTS

Delphi TOOLS

New Products and Solutions

TurboPower Announces SysTools 3

TurboPower Software Co. announced *SysTools 3*, a new version of its toolkit of system-level routines for professional Delphi and C++Builder developers. SysTools 3 improves programmer productivity by providing over 1,000 optimized, time-tested routines that can be incorporated into any Delphi or C++Builder project.

The library's routines are logically grouped into units so programmers can include just what they need. Units are provided for string manipulation, date and time math, generating bar codes, manipulating the Microsoft Windows shell, high-precision math, financial and statistical analysis, high-speed sorting, astronomical calculations, and expression evaluation.

SysTools 3 also includes a selection of reusable container classes, such as stacks, queues, deques, and trees. Among the new features in this version are the Explorer Components, Network Management routines, support for Regular Expression Search and Replace, POSTNET bar coding, an enhanced string manipulation unit, Application Control components, and support for VCL and COM development.

Explorer Components are included for embedding the capabilities of Windows Explorer inside Delphi and C++Builder applications. The new controls, Explorer TreeView, Explorer ListView, and Explorer ComboBox, can be used separately or together to mimic Windows Explorer functionality. They are also useful for creating custom versions of the Windows common dialog boxes with enhanced functionality.

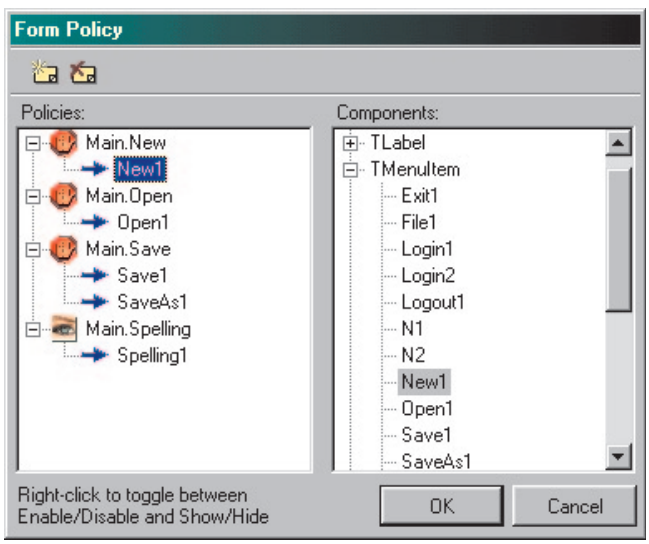
Support is included for managing LAN Manager-compatible networks running Microsoft Windows.

TurboPower Software Co.
Price: US$249
Phone: (800) 333-4160
Web Site: http://www.turbopower.com

UIL Releases Security System 2.06

Unlimited Intelligence Limited announced the release of *UIL Security System 2.06*, a method to add end-user security to your Delphi 3, 4, or 5 applications. Applications using this system can control what users and groups have access to. This latest update adds keyboard-only support and improved look and feel.

UIL Security System supports any database accessible by Delphi, including all third-party *TDataSet* descendants. In addition, no DLLs, ActiveX controls, or external programs are required; everything you need is included in the executable — a login dialog box, a user and group management dialog box, and a form policy, which allows you to specify what controls are disabled or hidden from people who do not have access.



In addition, UIL includes powerful design-time tools, including a form policy designer that allows you to easily group controls into access rights using drag-and-drop. Users can edit users and groups at design time if needed.

Unlimited Intelligence Limited
Price: US$199 (limited-time offer).
E-Mail: info@uil.net
Web Site: http://www.uil.net

IP*Works! Java Edition Now Shipping

devSoft Inc. announced *IP*Works! Java Edition*, a pure Java version of its IP*Works! Internet Toolkit. The package consists of 20 JavaBeans for Internet programming, bringing ease of use to developers of Internet-enabled applications.

The product eliminates much of the complexity of developing connected applications, and is available in several editions, including native Delphi and C++Builder VCLs, ActiveX controls, C++ classes, C libraries, and JavaBeans.

IP*Works! contains simple high-level programmable components, such as interfaces to Internet Mail and Usenet News, as well as powerful TCP/IP programming tools used to build generic clients and servers. All components have interfaces that shield developers from the complexity of TCP/IP programming. Corporate developers will find the features they need to enable their applications to participate in a TCP/IP network without steep learning curves. IP*Works! Java Edition implements standard protocols specified in Internet RFCs, and is written in Java for portability across platforms.

devSoft Inc.
Price: US$295
Phone: (919) 493-5805
Web Site: http://www.dev-soft.com

Brainbench Offers Certification Exam Online

Brainbench is offering certification examinations online, allowing Delphi developers and other technical professionals to test their skills. As a technical professional, you can use the test results to get a better understanding of your strengths and weaknesses or to earn a certification that helps you get a better job.

Upon registration, you will immediately receive a free test access code, which allows you to take the multiple-choice exam any time within the next 30 days. You can register at http://destinationsite.com/c?c=62939.15491.0.142.0.

If you pass the exam, Brainbench will certify your skill and mail you a certificate free of charge. Also, you can make your certification available online if you choose.

Brainbench has 60 different exams from which to choose.

Delphi TOOLS

New Products and Solutions

Seagate Introduces Crystal Reports 8

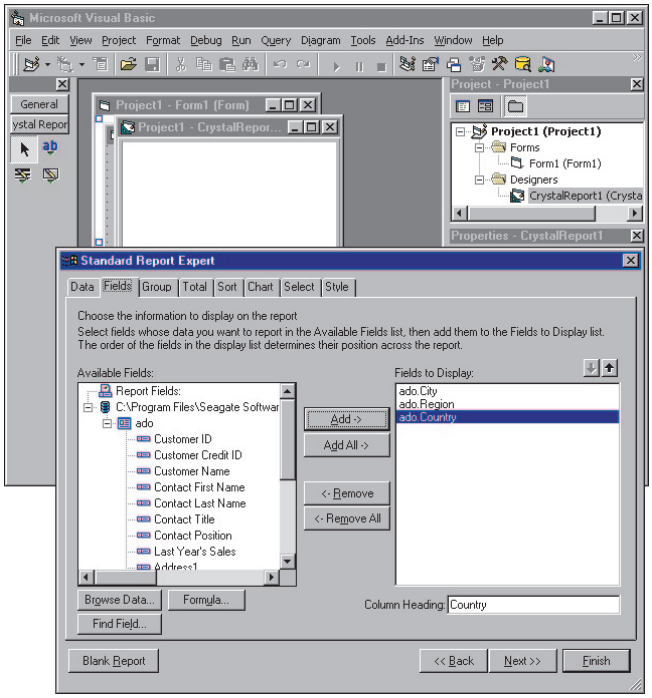
Seagate Software announced the release of ***Seagate Crystal Reports 8***, the company's Web and e-reporting solution. Seagate Crystal Reports is the foundation for Seagate Software's line of business intelligence solutions, all of which expand and extend Crystal Reports' e-reporting functionality. Crystal Reports 8 offers integration with Microsoft Office and reporting tools for Web and Windows developers.

The enhanced Web report server provides greater scalability and improved access to common Web infrastructures (Lotus, Microsoft, Netscape, and CGI-based servers). Export support for DHTML, new report viewers, and hyperlink capabilities offer increased flexibility for organizations looking to deploy an all-encompassing reporting system over the Web.

Crystal Reports 8's integration with Microsoft Office allows users to create reports within the Microsoft Excel or Microsoft Access environments using Crystal Reports Add-ins. By exporting reports to Word, Excel, and RTF, any Crystal report can be integrated into any Office document.

Seagate Crystal Reports 8 gives application developers the reporting tools needed for Microsoft Visual Basic, Visual InterDev, Domino Designer, and other development environments.

The Report Designer Component (RDC), which includes more than 850 properties, methods, and events for control over an application's reporting engine, offers key enhancements, including creation of reports without existing database connections; report creation in multi-tier applications using Microsoft Transaction Server; and simple integration of reports into Active Server Pages applications.



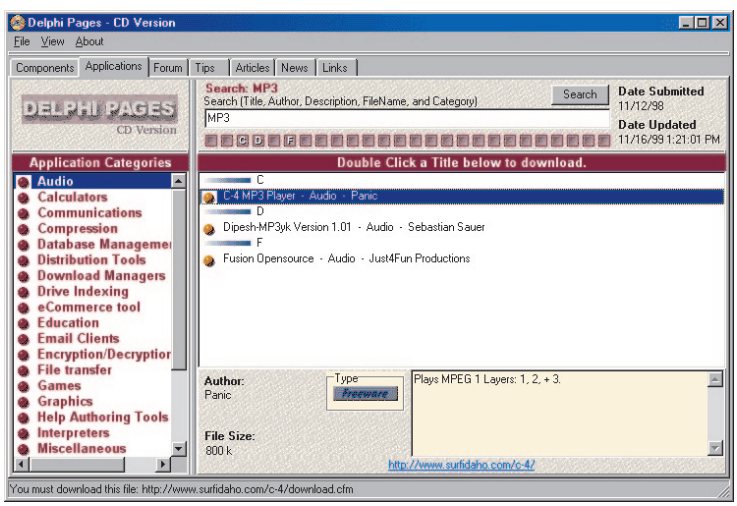
Seagate Software, Inc.
Price: Standard Edition, US$149; Professional Edition, US$395; Developer Edition, US$495.
Phone: (800) 877-2340
Web Site: http://www.seagatesoftware.com

Delphi Pages Offers Delphi Pages CD Version

Delphi Pages is offering ***Delphi Pages CD Version***, a CD that contains over 511MB of information, over 1,100 components (CD contains actual files), over 100 links to applications developed with Delphi, a complete forum section, tips, articles, news, and links to other sites.

The CD features a fully searchable engine, quick access to files, description, version information, source information, file sizes, buy-now options, screenshots, date submitted and updated, and author information.

Every three months, a new CD will be developed for purchase. It will include previous components and information, as well as all components added during the quarter. You can also take advantage of a reduced fee by subscribing on an annual or bi-annual basis.



Delphi Pages
Price: US$29.95 (purchase online at http://www.delphipages.com/cd/buycd.cfm).
Web Site: http://www.delphipages.com/cd/

Delphi TOOLS

New Products and Solutions

TechSmith Announces SnagIt 5.0

TechSmith Corp. announced the ***SnagIt 5.0*** screen capture tool for Windows. SnagIt enables users to capture desktop images, text, and video to a file or to the printer with a single mouse click. The newest release includes SnagIt Studio, an object-based vector drawing tool, so users can quickly mark up any screen capture or image. Using one tool to grab and annotate screen shots saves users time and eliminates the need for additional applications.

Another new feature is "capture profiles" to configure frequently used settings. Webmasters and Web developers can use the new "Web-cam" feature to automatically send captures to the Web from the desktop.

SnagIt 5.0 works for Windows 95/98/NT/2000.

TechSmith Corp.
Price: US$39.95; quantity discounts and site licenses are available.
Phone: (517) 333-2100
Web Site: http://www.techsmith.com/download.asp

InstallShield Software Ships InstallShield Pro 2000 Second Edition

InstallShield Software Corp. announced ***InstallShield Professional 2000 Second Edition***, the latest version of its installation-authoring solution for ISVs and corporate developers. InstallShield Professional 2000 Second Edition offers expanded support for code reuse and the latest version of the Microsoft Windows Installer-based service.

InstallShield Professional 2000 Second Edition features new releases of two products, InstallShield Professional 6.1, the latest version of the setup-authoring tool, and InstallShield for Windows Installer 1.1, a comprehensive solution for creating Microsoft Windows 2000 logo-compliant installations. The two products combine to enhance productivity, usability, and control for professional Windows developers distributing applications with sophisticated installation requirements.

Second Edition's expanded features include an Object Development Kit (ODK) in InstallShield Professional 6.1, which allows setup authors to create and distribute reusable pieces of installation projects. InstallShield for Windows Installer 1.1 includes new InstallScript support, a Spy Repackager, and enhanced migration capabilities.

InstallShield Professional 6.1 also includes Microsoft Millennium compatibility. Professional 6.1 provides built-in support for the System Restore feature found in Microsoft's upcoming Millennium operating system (the next consumer Windows release). With this feature, users can restore PCs corrupted during software installation. The System Restore feature automatically monitors and records key system changes to the user's PC. This functionality lets the end user undo a change that may have harmed their system.

With version 1.1 of InstallShield for Windows Installer, InstallShield provides enhanced functionality that allows developers to take advantage of Microsoft's Windows Installer-based service.

InstallShield for Windows Installer 1.1 also provides support for the new features in Microsoft's Windows Installer 1.1, including COM+ support, side-by-side components, and nested installation support. The Windows Installer 1.1 engine will be included in an InstallShield for Windows Installer installation by default, providing support for Microsoft Windows 95, Windows 98, Windows NT 4, and Windows 2000.

InstallShield Software Corp.
Price: US$995
Phone: (800) 374-4353
Web Site: http://www.installshield.com

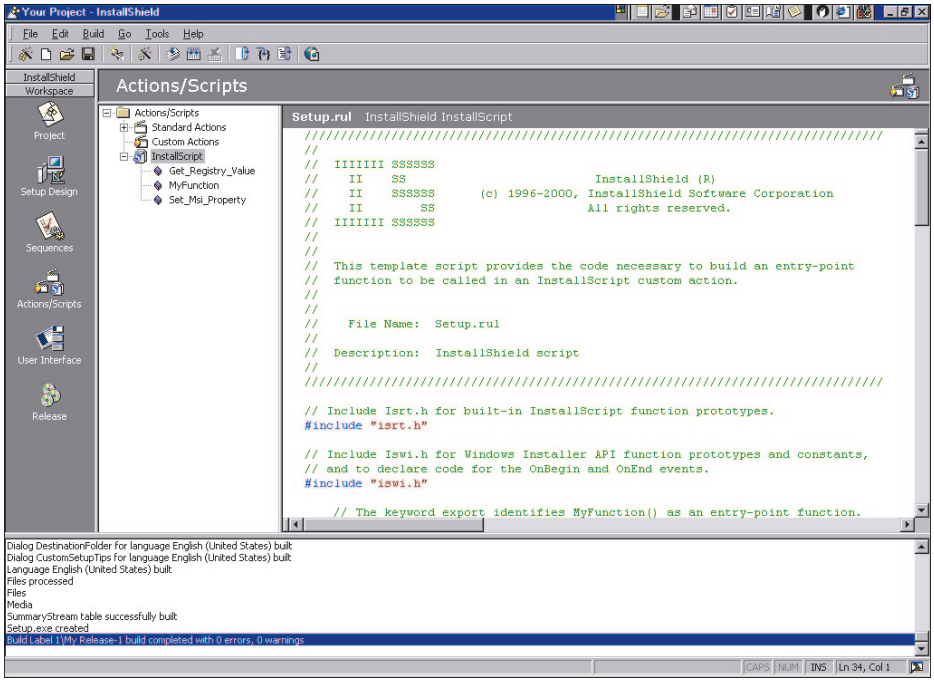


table of CONTENTS

News LINE

June 2000

C. Robert Coates Announces Resignation from Inprise Board of Directors

Dallas, TX — C. Robert Coates, CEO of Management Insights, Inc., has resigned from the Board of Directors of Inprise Corp.

Mr Coates resigned on Sunday, February 6, at 8:00 AM, prior to the vote on the merger of Inprise and Corel Corp. Mr Coates plans to communicate the reasons for his resignation in a future letter to shareholders and in a detailed letter to Inprise that Inprise must then disclose in an SEC Form 8-K filing.

Mr Coates says he will oppose the merger unless it results in substantially higher prices for Inprise shareholders and/or can be shown by Inprise's CEO to clearly benefit Inprise's customers.

He hopes that the Inprise Board will consider other offers for parts or all of Inprise, or continue to operate Inprise as an independent company so that it can follow through on the many positive press releases issued in the last five months.

Mr Coates currently owns 3,005,440 shares of Inprise Corp.

Management Insights, Inc. is a diversified management consulting, venture capital, and investment firm. Its customers include a large number of Fortune 500 companies. Among its public investments are large equity stakes in Inprise Corp., Evolving Systems, and Northfield Labs.

Inprise to Supply Ericsson for Network Management Integration

Stockholm, Sweden — Inprise Corp. announced that Ericsson has selected VisiBroker CORBA technology as a key part of its Operation Support System (OSS). Inprise's object request broker will form the basis of the future architecture within Ericsson's OSS products for managing both GSM networks and forthcoming broadband CDMA networks.

As operators look to upgrade their mobile phone network infrastructure, they need to provide an industrial-strength way of ensuring network availability and performance.

The Ericsson OSS is a tool for monitoring mobile networks and responding quickly to network service issues. The goal of OSS is to provide mobile phone users with the highest quality of service. VisiBroker is planned to enable Ericsson to specify Integration Reference Points for various systems, which are defined in Interface Definition Language. By adopting an industry-standard CORBA implementation, third-party vendors of network management software will be able to integrate their network management offerings with Ericsson's OSS in a consistent way.

The CORBA architecture allows operators to both manage individual network nodes, as well as interface with third-party management systems from other equipment vendors. Inprise is also working with Prism Technology to provide a number of additional CORBA services to Ericsson.

The financial terms of the agreement were not disclosed.

Inprise/Borland Announces Borland C++Builder 5

Scotts Valley, CA — Inprise/-Borland announced Borland C++Builder 5, the new version of its ANSI C++ development system for Windows 95/98/NT/2000. C++Builder 5 is available in three editions: Enterprise, Professional, and Standard.

Inprise/Borland also announced that the free Borland C++ Compiler, the foundation of C++Builder 5, has received over 160,000 downloads since its release. Developers who start with the compiler are able to seamlessly move into the full C++Builder environment for visual, database, Internet, and distributed development.

C++Builder 5 brings together Borland's C++ development environment with the latest Internet standards: XML and HTML 4. A detailed list of features included in C++Builder 5 can be found at http://www.borland.com/bcppbuilder/.

Customers may order C++Builder 5 on the Web at http://www.borland.com and from major software distribution channels. C++Builder 5 Enterprise has an estimated street price (ESP) of US$2,499 for new users. C++Builder 5 Professional has an ESP of US$799. C++Builder 5 Standard has an ESP of US$99.95. (These prices are in US dollars and apply only in the United States. Customers outside the US should contact their local Inprise/Borland office, distributor, or representative.)

Inprise/Borland Announces JBuilder 3.5

London, England — Inprise/-Borland announced JBuilder 3.5. JBuilder 3.5 Enterprise is a rapid application development tool for creating business, database, and distributed applications based on the Java 2 platform. It supports development on the Linux, Windows, and Solaris platforms.

JBuilder 3.5 includes support for J2EE (Java 2 Platform, Enterprise Edition) so programmers can deliver reliable and scalable enterprise Java applications. JBuilder 3.5 provides wizards and visual tools for creating reusable JavaBeans and Enterprise JavaBeans technology. Included is a development license for VisiBroker for Java and the Inprise Application Server 4.0, which supports J2EE technology standards. A detailed matrix of features included in JBuilder 3.5 is located at http://www.borland.com/jbuilder/feamatrix.

JBuilder 3.5 is available in two versions: Enterprise and Professional. Both are available on the Web at http://shop.borland.com and from major software distribution channels.

JBuilder 3.5 Enterprise has an estimated street price (ESP) of US$2,499 for new users. JBuilder 3.5 Professional has an ESP of US$799. JBuilder 3.5 Foundation, a standard version of JBuilder, is available for free download at http://www.borland.com/jbuilder/foundation.

table of CONTENTS

By Ron Loewy

Palm Conduits

Part I: An Introduction to Palm Programming

Colin Chapman would have loved Palm handheld computing devices. It's a simple machine that sacrifices a lot of un-needed features to excel in performing its task.

Chapman (for the un-initiated) is the genius that created Lotus, the famous race and sports car manufacturer. While his Formula 1 competitors at the start of the 1960s (Ferrari and Porsche, among others) concentrated on creating bigger, more powerful engines that required bigger and heavier cars, Chapman was busy inventing race cars that used an engine that started its life as a fire-fighting water pump. He overcame the engine's lack of power and sophistication by creating smaller, lighter, and wind-cheating cars. He used to joke that to make a car competitive, he would add "lightness." This out-of-the-box thinking saw a dominance of Lotus race cars during the 1960s and early 1970s.

As Delphi programmers, we're used to the PC industry's fast expanding machines. Faster processors, bigger hard disks, and more memory run bigger operating systems, sophisticated middleware, and capable database servers. Our applications take advantage of sophisticated GUI widgets and gizmos, and use life-like multimedia elements. And, as we all know, no application is ever finished without a cool splash screen and scrolling credits in the About box.

Palm devices — for the few that retreated to their undercover Y2K shelters several years ago and might not know — are the small, mostly black-and-white devices (Palm Computing just announced a color version at the time of this writing) marketed by 3Com that include a set of pre-defined applications, such as a ToDo list, Memo Pad, Address list, and, in newer models, Expense Report and E-mail. Unlike earlier organizers that were limited to a pre-defined set of applications, the Palm devices can be extended; new applications can be written or purchased and installed on the machine.

The Palm designers learned from the mistakes of the Apple Newton, and provided a convenient way to exchange data between the Palm device and a PC. This is the reason for the success of the Palm platform, where other handheld devices failed: The Palm devices aren't replacements for the PC, but mobile extensions of it. They're not designed for massive data entry or big processing capabilities, but as viewers of data with limited updating capabilities.

The design of Palm applications emphasizes simple operation and encourages data entry on the PC when possible. Consider a traveling salesman that needs his products and contact information when he's on the road, but would rather create his customer list and inventory/marketing information on a PC. When he's in the office, he uses general-purpose applications (such as Office) or custom applications; when he's on the road, he carries the important information for his trip on his lightweight Palm device, which will last for several months on a set of AAA batteries.

This two-part series will introduce you to the concepts of Palm programming and describe a method of exchanging information with Palm devices using Delphi. It's out of the scope of this series to teach you Palm programming (there are many good references, books, and tools that can be used for this). I will, however, explain the theory behind Palm applications, the databases they use, and the methods used to exchange information with the Palm on the PC. In Part II of this series, we'll continue by writing a sample conduit for the ToDo application that comes standard with the Palm OS.

Palm Programming

Before we jump into the technical details of exchanging information with Palm devices from a Delphi application, let's try to understand a bit

more about Palm applications, the way Palm devices store data, and how the exchange process occurs.

Device Programming

Palm devices are based on Motorola processors. They run an operating system named Palm OS that provides simple user interface, memory, and data-storage services. Newer versions of the Palm OS (the Palm III and later) also provide network services via TCP/IP, Infrared Beaming inter-device information exchange, and Web-clipping services (Palm VII).

The most important — and disappointing — information about Palm device programming is that Delphi cannot be used to create Palm applications. The official language supported by Palm Computing is C or C++, using the CodeWarrior compiler/IDE from Metrowerks Corp. A free GNU C/C++ compiler is also available, and several other tools (mostly using a version of Basic) are available from third-party vendors.

A Palm application uses "forms" to display its user interface. Think of a form as a Windows window. Forms (like the Windows we're familiar with) contain controls, such as edit boxes, buttons, list boxes, tables, etc. In other words, forms contain everything you've come to expect from a GUI interface. If you look at a Palm application's C source code, you'll see it's not very different from early Windows or Mac source: a big event loop for the application routes messages to form event handlers. If you remember the (good?) old WinMain days of Windows applications, you know what Palm application source code looks like.

The small amount of memory and storage available reunite other old programming friends you thought you said good-bye to, such as segmented memory, near and far jumps, and other small-footprint development gotchas.

Coming from Delphi's RAD development environment — with its two-way tools, visual object browser and wizards, helpers, and experts — you'll find that programming Palm applications in CodeWarrior is like camping: It's a lot of fun to "rough it" for a while, but I still like to come back to the luxury of hot water for my showers and a refrigerator I don't need to carry on my back. Not that CodeWarrior is like being thrown out to the sharks armed with VI as your editor, but it's closer to the old days of Borland Pascal for Windows for code writing, with its Constructor visual form designer, which resembles the old Resource Workshop we used before Delphi came along.

Databases

A big shock to anyone that came from conventional PC/Server/-Mainframe programming to Palm device programming is the fact that a Palm device does not have traditional persistent storage devices, like hard disks, tape drives, CD-ROMs, or diskette drives. Instead, a Palm device partitions its memory into two parts: a persistent part, used to store applications and databases which remain in memory even when the device is turned off (at least as long as there is power in the batteries); and dynamic memory, used like dynamic memory on any operating system.

A Palm application can't write directly to the "protected" partition of memory, thus the applications and databases are protected from memory overwrite bugs. The Palm OS provides a set of database access API functions that allow you to read, write, and search a database.

As discussed in the introduction, the Palm device was designed to work as an extension of the PC, with easy-to-synchronize data. The database structure reflects this design. A Palm database is a collection of (variable size) records in memory. Every record has a record index used to read, write, and search.

Every Palm application has a "Creator" ID associated with it. Likewise, a database has a "Creator" ID and type associated with it, and, of course, its name. This ensures that it's easy to associate a database with the application that uses it.

Every database has more "global" information associated with it (for example, a list of 0 to 15 categories associated with the database). Every record in the database can be associated with one of the categories, and allows for easy filtering of data. For example, the Address Book contains pre-defined categories for business-related entries, private entries, and untitled entries. You can define new categories, such as Family, People I Play Soccer With, or whatever strikes you, as categories you need.

Each record in the database is identified using a unique record ID and has a category ID associated with it. Thus you can associate your mother's second cousin's address with the Family filter, and the guy from whom you bought a radiator for your 1936 Morgan with your Car Buddies category. Each record can have a different size; to save space, records are packed in memory. A string field, for example, will be stored by its length, plus 1 character (the null character used in C for an end-of-string) instead of a pre-defined length with spaces in the unused characters.

Every record in the database also has an associated attribute field. This field includes attributes used for synchronization of data with the PC. Some of the flags found in this attribute field are:

- Deleted flag: When a user deletes a record on the Palm device, the record remains in the database until synchronization with this flag is turned on. When synchronization happens, the PC database is updated and the record can be physically removed from the database.
- Modified flag: When a user modifies the values of the record on the Palm device, this flag is turned on. During synchronization, the PC application knows it needs to update its own database based on this flag.
- Private property: Determines if the record is always shown. If the record is marked private, you need to use a password in the Palm security application to display the record. This ensures that if your Palm is stolen, some of the information can be hidden from prying eyes.
- Archived property: This flag is turned on for deleted records that need to be archived on the PC during synchronization.

Conduits

A conduit is a piece of code that sits on the PC and performs the data synchronization between the handheld and the PC. Palm Computing provides the CDK (Conduit Development Kit) in Windows and Mac versions.

When you install the Palm Desktop on your PC, an application named HotSync is added to the system tray of your PC. When your Palm device is connected to the cradle you installed on your machine, and you click the synchronize button, this application is activated and starts the data synchronization with the device using a propriety protocol.

The HotSync application activates conduits registered with it to perform the data synchronization. If a conduit is registered with the HotSync application, this conduit is called using the conduit

API to do its job. All databases found on the device that have no conduits associated with them are backed up to the PC using a simple copy operation.

If your Palm application doesn't need to exchange information with a PC, you don't need to develop a conduit for it; you can be sure that the data will be backed up automatically every time the user performs a HotSync operation. However, if your application is an extension of a PC application (as most Palm applications are), you need to write a conduit to exchange the requested information.

In a surprising move, the Palm CDK requires Microsoft's Visual C++ 5.0 or later. This is strange because the official development tool for Palm applications is CodeWarrior, which cannot be used (at the time this article was written) to write conduits on the PC.

When I was faced with the need to write a conduit for my application, I decided to investigate the option of using Delphi. My research lead me to EHAND Connect, a product offered by EHAND AB, a company in Sweden. EHAND Connect is a COM-based product that allows you to write conduits with every Windows development tool that can create automation objects. The price for EHAND Connect is free for public applications, or a small fee for internal enterprise applications. The manual that comes with the product provides Visual Basic samples, but it's easy to use Delphi to create your conduits, as we'll see in this article.

Getting Started with Conduit Development

Before we can discuss conduit development, you should own a Palm device, preferably one with Palm OS V3.0 or later. (The Palm IIIe that is the entry product from 3Com when this article was written is an example of one such device.) You should install the Palm software on your PC, connect the cradle, and perform HotSync to ensure that everything works. Palm Computing offers the POSE (Palm OS Emulator) for people who want to develop without access to a real device, but in reality, I found it much easier to work with a real Palm device than with the POSE.

To make it easy, we'll write a conduit for the ToDo application that comes standard with the Palm OS, so you won't need to write a new Palm application, or even install one on your device. If you use the ToDo application on your device, I would suggest performing a HotSync before you start playing with the conduit we will write; this will ensure that the data you need will remain intact.

EHAND Connect

EHAND Connect can be downloaded from http://www.ehand.com/ehand/d.asp (version 1.0 was the version available when this article was written). Unzip the distribution file and run the setup application to add the EHAND Connect SDK to your computer.

You now need to import the EHAND Connect type library into Delphi. I used Delphi 5's TlibImp.exe application (available in Delphi's \Bin sub-directory), and executed it with the -P+ parameter (for Pascal output) on EHConnect.tlb, which can be found in the EHAND Connect installation directory. Assuming you installed using the standard directory structure, the command line looks like this:

```
Tlibimp -P+
  "C:\Program Files\EHand\EHand Connect\EHConnect.tlb"
```

The result is two files, EHConnect_TLB.pas and EHConnect_TLB.dcr, that I moved to Delphi's \Imports sub-directory.

The Conduit Objects

When the conduit object is called by EHAND Connect, a conduit core object is passed to our code. This object provides access to an object model used to represent that synchronization process, and provide access to source and destination information.

We'll begin writing conduit code in Part II of this series. For now, let's inspect the different objects that our code can use.

The core conduit class. The core conduit class is the root class passed to our code. It provides access to the conduit synchronization object model and provides many services and access to information about the synchronization process. This class is referenced as the interface *Icore* (or Conduit) in the Delphi type library import unit.

The class properties provide information like the type of synchronization that needs to be performed (*SyncType*), the type of connection (*ConnType* — the device is connected via a cradle or over a modem), and more.

The class provides the methods to open and define the database we want to synchronize with. Use *OpenDatabase* to open a database on the device, and *CloseDatabase* to close it when you're done with it. You can write information to the HotSync log with *AddLogEntry*, remove a database with *DeleteDB*, or get information about the device using *GetHHOSVersion*. You can read a record (returning a data record class) using functions such as *ReadRecordByIndex*, *ReadNextModifiedRec*, and more. You can create new records with *GetEmptyRecord*, and write a record to the database using *WriteRec*. The *DefineField* method is used to define the structure of records (schema) in the database (the Palm device does not include this meta information in the database).

There are more methods and properties supported in the class, and we'll discuss some of them when we inspect our sample application in Part II of this series.

The user information class. The user information class (referenced as *IComUserIDInfo* or *UserInfo* in the Delphi type library import unit) is used to provide information about the user of the device being synchronized. The class provides information, such as the user id, user name, password, time of last synchronization, etc.

The database information class. The database information class is used to represent the global database information. The Delphi type library import unit represents this class using the *IComDBGenInfo* interface, but you can also use the name *DBInfo* to access the same interface.

This class provides access to global information associated with the database, for example, the categories defined for the database or global database fields. The methods of the object allow you to read information or create information in this part of the database.

The data record class. The data record class is used to represent a single record in a database. The Delphi type library import unit represents this class using the *IComRecordInfo* interface.

The data record class provides information about the data record, including the unique record identifier (*RecID*), the record position in the database (*RecIndex*), the record's category (*CatID*), the record's size (*RecSize*), the data portion size (*TotalBytes*), and the attributes associated with the record (*Deleted*, *Modified*, *Private*, and *Archived*). Use the *GetField* method to get the value of a field in the record, or *SetField* to set the value of the field in the record.

The collection class. The collection class is used to store an array of elements in one variant. It's represented as the *ICollection* interface in the Delphi type library import unit. This class provides the same kind of functionality that a Delphi *TList* provides to access and set items in a collection. The EHAND Connect Visual Basic samples use this class to hold memory images of the records they manipulate. I chose to use the familiar *TStringList* class instead.

Conclusion

The next part of this series demonstrates the use of EHAND Connect to create a conduit in Delphi by writing a simple ToDo application that will store information from the Palm device ToDo application to a Paradox database. We will also write a conduit that synchronizes the Paradox database and the Palm device. Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JUNE\DI200006RL.

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

By Nikolai Sklobovsky

Waking from Threadmare

An Agent-based Approach to Multi-threaded Programming

Multi-threaded programming has never been an entry-level subject. Even with the proper wrapper classes, which modern development tools provide so generously, creating an elegant, smoothly working multi-threaded application isn't always an easy task.

Two major problems a novice programmer is apt to encounter when dealing with multiple threads are synchronization and visualization. And debugging multiple threads can hardly be called pleasurable. Even if your favorite IDE provides thread-debugging support, the extremely volatile nature of threads-based processes often prevents you from using an IDE and makes you go back to the good old "debug print" technique. And sometimes even this doesn't help.

Several years ago, I reached a point where multi-threaded programming ceased to be just a buzz-word and became the *sine qua non* condition for my projects. After having had enough painful "fun" with my first threads, especially on multi-processor machines, I developed a technique for dealing with them. Since that time, neither my colleagues (who gladly employed this technique in a variety of projects) nor I have had a single problem with multiple threads. Multi-threaded programming has now become routine, bringing with it many perks, and — if properly used — no pain. During these years this technique has been refined, revised, and thoroughly tested. We now have many mission-critical applications that use it all the time.

The purpose of this article is to share this robust, reliable, and relatively simple method with a broad community of fellow programmers. If you're already familiar with the subject and have enough multi-threaded experience under your belt, you can jump right to the implementation section.

You'll notice that only a few code excerpts are included in the article's text (the rest of the code is available for download; see the end of the article for details). Although the technique is non-language specific and can be implemented in different languages in various ways, all the examples here are written in Delphi. You can use any 32-bit version of the compiler, although the latest version is preferable. You should also be familiar with basic Delphi and threads-related terminology.

Why on Earth Use Multiple Threads?

One of the most important questions for a developer to ask about any new technology is "Why should I use it?" or "How would I (or my application) benefit from it?" This question arises naturally because most, if not all, of these technologies promise a lot in some foggy future while the suffering starts almost immediately. Unfortunately — or quite fortunately for IT professionals — in programming, as in geometry, there is no king's way to the bright shiny tops of today's "cutting edge." The learning curve is usually very steep, which means you must acquire tons of new information before your application even starts working at all, let alone produces any benefits.

The standard rule is: If your application already works the way you want, don't touch it. This simple policy works pretty well until that sad moment when you realize that you've already used 100 percent of your existing arsenal to make an application work — and it still doesn't. This is the moment of truth. Now you'd better be familiar with the new technology that you're about to use. Otherwise you might make an erroneous choice with disastrous results.

Microsoft Windows began providing support for multiple threads with its first 32-bit version. The benefits were very attractive. After 16-bit, event-driven-only cooperative multi-tasking, when any extra millisecond in a tight loop could easily result in a system-wide "freeze," programmers were finally given a nice opportunity to separate hard working — and mostly sequential — data-crunching business processes from relatively slow, random, unpredictable event-based GUIs. It also opened a door for smooth background processing. The only thing

it seemed you needed to do was to isolate those processes, put them into separate threads, and *voilà*! You can forget about taking complex and sophisticated measures, allowing the other parts of your application, and the system itself, to breathe while you're busy digging your data. Everything became simple and straightforward, almost like in the old DOS world. Each part was simply doing its job while the operating system provided each with its fair share of the CPU cycles. Simpler implementations meant fewer bugs and a shorter path to market. Long live multiple threads!

Delphi has offered thread support starting with its first 32-bit version, namely Delphi 2, which hit the market soon after Windows 95. Although Delphi provided a wrapper class conveniently named *TThread*, you could also choose to use bare-metal, Win32 API thread routines. Examples and online Help articles provided some decent information on the subject, so hundreds of programmers rushed to use — and to fight — this fascinating innovative technology. And many were slain.

New World Order

Almost all thread-specific problems arise from a single paradigm change: You no longer live in a synchronous world. You can no longer assume the next line of code will be reached right after the current one has executed. To make things worse, you can't even assume any given single line of your high-level Delphi code will be executed as a whole. Instead, a thread or process switch might happen at any moment, sometimes even during a simple assignment operation.

In the old sequential world, you could think of yourself as a mighty lone warrior. Your errands were tough, but at least you knew your exact position at any particular moment. Now, suddenly, you're leading a commando squad. Although your troops still accept your commands instantaneously, it takes an unpredictable amount of time for them to be implemented. You immediately experience a certain lack of control. Each of your troopers being properly trained for a specific job, they can carry it out very effectively, much more effectively than you used to do. But this also means that they no longer report their every step, or ask your permission to move. Once they've accepted the task, they disappear into the jungles of the CPU. The only way to get them back is to wait until they're done, or to kill them.

Under the new conditions, you need to radically revise your strategy. Rather than continuing to think about the whole campaign every moment, you have to design the campaign in advance, discuss each part with the relevant participant, provide a reliable communications link between them when joint efforts are required, and then sit back and relax, watching their progress and making small corrections, as needed. Sound easy? It is, provided you've managed to complete your part, i.e. to design the campaign, establish the communications, and monitor the action.

Design

Many people have recognized that multi-threaded applications typically require much more planning than conventional single-threaded ones. Basically, threads usage is very similar to units usage in any strategic computer game. The overall success depends not only on the strength and training of each unit, but also on how well the plan was designed and thought out. The best troops can lose to the weakest enemy if not properly deployed; the best plan can fail if the tools aren't good enough.

The benefits and desirability of good design were generally recognized in conventional programming. In the world of multi-threading, good design is pure necessity. Proceeding here without good design makes failure inevitable.

Synchronization (Terminate Traps)

Communication between the deployed units and HQ (headquarters) is required for victory. You need to know, for instance, that the bridge ahead is secured before you advance your main forces to the river. For multiple threads, this essential communication is called synchronization. Unfortunately, this part of Delphi's implementation of thread support is unsafe and unclear.

Let's consider the *OnTerminate* event. At first sight, this appears to be a perfectly convenient way for a thread to say: "Hey, I'm done; you can proceed safely." Unfortunately, in real life it's not so simple. The problem is that this event is fired (via the *Synchronize* method) by default while the thread is still alive. As a result, all you can do is set up a flag, or send yourself an asynchronous message. You definitely *cannot proceed*. If you try, your application will be caught in a deadlock, or you will encounter some nasty, random audio/visual effects when you try to close it. Of course, you might think you could override the virtual *DoTerminate* method. This seems to be a great idea, until you start implementing it.

Another easy trap to fall into concerns the similarly named couple *Terminated*/*Terminate*. You might think that once you've called the *Terminate* method for a thread, that thread will terminate. You may even be so naive as to expect it to stop working immediately. It isn't so. *Terminate* is a static procedure that simply sets the protected read-only *Terminated* property to True, and that is all. Basically, Delphi provides a programmer with a free, Boolean, thread-located variable, and a decent way to set its value to True. All further responsibility for checking this variable value falls to the application programmer, making him or her insert endless conditional operators in loops, thus obfuscating a formerly clear and simple algorithm.

As you can see, Delphi's support for "polite" thread termination is quite limited. Besides, if you use a native thread API rather than the *TThread* object, you can't use this support at all.

Visualization

Another important aspect of multi-threaded applications is feedback — visual or otherwise — from the working threads. With the VCL being officially non-thread-safe, visual feedback draws especially hot attention from the programmers. Delphi developers have suggested a simple solution to this problem, namely the *Synchronize* method. This method acquires a VCL-global critical section, effectively making the VCL thread-safe for this particular thread for the duration of each call. You can think of this as two highways that merge at some point, run together for a while, then finally part and go their separate ways until the next merge.

This simple solution works surprisingly well until you decide to check for your end-user reaction during this synchronize-based call. Suddenly, your calling (working) thread stops and waits with you. Quite often this is exactly the opposite of your intentions. You had hoped the thread would work steadily regardless of innocent end-user actions. And even if you didn't launch any message boxes, simply moving the mouse (and what else is a poor user supposed to do while looking at your progress bar but play with the mouse?) can slow your "synchronized" application manifold, thus immediately eliminating one of threading's most important benefits: performance.

Another problem may also arise if, for some reason, you decide not to use *TThread* object, and switch to native Win32 API calls instead. You've already lost the *Terminate* group, and now there's no *Synchronize* method for you either. This is unfair! You didn't say you weren't going to use the VCL at all; you just didn't want to use its poorly developed thread support.

A Way Out

A cure for these problems can be found when you realize that your multi-threaded designs have a lot in common. Let's consider the simplest case: a two-level multi-threaded application. The main thread, which always exists, will be responsible for the GUI and user interaction. One or more secondary threads will perform some useful data-processing work, such as sorting thousands of rows, or copying multi-megabyte files.

At this point, the design seems to be solid, leaving us with only two things remaining to be worked out: communication (synchronization) and feedback (visualization). We would like for each thread's progress to be smoothly displayed with its own labeled progress bar, and for each thread to have a humble **Cancel** button in case something goes wrong. By "smooth display" of the thread's progress, we mean that a working thread should be free to inform its host about its status whenever it's most suitable for the thread, not necessarily for the host. At the same time, we also expect the host to report the current status in a uniform way (e.g. two times per second) regardless of the thread's ability to speak. Implementing a **Cancel** button means that any secondary thread — *TThread* based or not — can be easily and almost instantly terminated at any moment. "Almost" is good enough (a thread may need some time for closing files, releasing resources, etc.) provided our user can have immediate feedback that the thread has accepted the command and is carrying it out as soon as possible.

We also want all our secondary threads to terminate peacefully if the user decides to close the application, or shut down the whole system. And, of course, we would like to have a chance to get rid of all the thread-associated GUI components once the corresponding thread has finished its job. It goes without saying that, during all this data processing, your main window should remain as sensitive to user action as if it were doing nothing else.

In other words, we want our working secondary thread to be free to do its primary job, and to diligently report its progress without any severe loss of performance. At the same time, we want our primary GUI thread to be able to display this information, and have ultimate control over the working thread's life and death. This means a certain amount of work needs to be done by somebody to accomplish our wishes. We need an agent.

Implementation

Actually, we need two agents. One agent should be available to the working processes. It needs to be small and simple and, theoretically speaking, could even be absent altogether. Our troopers cannot carry a home theater on a mission. Likewise, they certainly should not slow down if their radio goes dead. On the contrary, they should simply throw the whole set away and move faster.

The other agent must be more bulky. It will be a command center or central headquarters (HQ). It should take care of all incoming signals from multiple field devices (which may reside in different threads), serialize them, and provide a convenient way for the GUI to retrieve this valuable information.

Field Agent

The primary roles of the field agent are to provide a working thread with an easy way to inform HQ about its progress, and to allow HQ to send it a self-destruction signal, if necessary. This agent should be very lightweight, and its absence should not prevent a working thread from doing its job.

That last part may sound tricky to a novice Delphi developer, but there's a simple solution for this kind of problem. Delphi itself uses it in the ubiquitous *Free* method. The only thing you must do is ensure that all the methods you are about to publish are static (i.e. not virtual nor dynamic) and that the first statement of all these methods looks like this:

```
if Self = nil then
  Exit;
```

Simple enough! Now we can develop our business algorithms without bothering to check whether our agent is present. Its code will handle a **nil** situation automatically. The **public** portion of our field agent — let's name it Client — is shown in Figure 1. See also Listing One (on page 14).

Here, the *Start* and *Finish* methods serve to denote the entry and exit points of some logical process, while the *Report* methods obviously do

```
function Start(const csCaption: string;
  bProgressable: Boolean = False;
  pUserData: Pointer = nil): Integer;
// Always accepted if pid is okay.
procedure Finish(pid: Integer = pidCurrent);

procedure Report(const csText: string = '';
  bImportant: Boolean = False;
  pid: Integer = pidCurrent); overload;
procedure Report(Index, Count: Integer;
  const csText: string = '';
  pid: Integer = pidCurrent); overload;
// Always accepted if pid is okay.
procedure ReportAlways(const csText: string;
  ErrorClass: ExceptClass = nil;
  pid: Integer = pidCurrent);
```

Figure 1: The public portion of our field agent, named Client.

```
procedure DoTheJob;
var
  i, iCount: Integer;
begin
  iCount := 10 0 0 0 ;
  for i := 0 to Pred(iCount) do begin
    DoSomething(i);
  end;
  DoSomethingElse;
end;
```

Figure 2: If the working method of the original business process looks like this ...

```
procedure DoTheJob(Client: TOurCustomFieldAgent);
var
  i, iCount: Integer;
begin
  Client.Start('Doing something', True);
  try
    iCount := 10 0 0 0 ;
    for i := 0 to Pred(iCount) do begin
      Client.Report(i, iCount);
      DoSomething(i);
    end;
    Client.Report('Doing something else', True);
    DoSomethingElse;
  finally
    Client.Finish;
  end;
end;
```

Figure 3: ... then our agent-aware version would look like this.

a humble servant's job in providing HQ with valuable progress data. Thus, if the working method of the original business process looks like the code in Figure 2, then our agent-aware version would look like that in Figure 3.

Fair enough. If this thing really works this way, you could possibly buy it. At this point you might have a couple of questions, such as why do we need to insert a **try..finally** statement? And what are those suspicious comments about "acceptance" all about? Finally, how are we supposed to terminate this process in case we need to? We'll answer these and other questions in the next section.

How It Works

Now it's time to raise the curtain, or, should I say, the "exception"? Our agents use this powerful technique to maximize both the robustness and the safety of the application. The idea is simple. The business process doesn't monitor our agent; it simply calls its methods. However, it must be prepared for an agent to raise an exception in response to one of these calls. This, in turn, would result in immediate quitting from all the loops and call stacks, through all the **except** and **finally** parts. Thus, we can solve both problems by having the thread use the same line to both report its status and receive an "abort" feedback via exception.

Exceptions are very powerful. This means they shouldn't be abused. Our agents are smart enough not to raise an exception in response to a *Finish* method, or to a special method named *ReportAlways*. Sometimes you need to send some information to HQ even when you know the mission has been aborted, such as when you know who the mole is.

Let's briefly scan over other features of our Client before we switch to its more complicated HQ partner.

Start and *Finish*

Each time you're about to start some logically related group of actions, it's wise to give this group a human-readable name, and call the *Start* method for the available Client. If you know this action can theoretically be associated with some progress-bar-like UI component, you can set its second argument to True. This will give you an opportunity to conveniently report on its progress.

The *Start* method always returns an Integer: a Process ID or PID. Note that this is not a Windows process ID, but rather our own logical ID, which we can use to identify each of our actions. In the vast majority of cases, you can simply ignore it; all other methods by default accept the so-called current ID, which is supported by HQ on your behalf. This current ID is defined by a constant named *pidCurrent*. Think of PID as a radio frequency. You generally don't need to know its exact value to be able to say a few words. If you want ultimate control, however, it exists for that purpose.

There is another special ID available. If you don't feel like using the *Start*/*Finish* pair, but still need to report something, you can use any of the *Report* methods with a constant, *pidMain*. This process always exists and everybody is aware of its presence. You can treat it as a common open frequency. *Finish* will never raise an exception (except for the weird case when you try to finish a process with an unknown ID).

Reports

Client allows us to send three kinds of reports back to HQ. First, and most commonly used, is a text report. This text information can be either important or unimportant. Importance guarantees delivery. Imagine a situation where you're scanning several drives. You would probably want to notify the end user about each drive, which could be conveniently handled within the time required to scan each one. However, it wouldn't be important to display all the file names. Important messages are queued and eventually delivered. All unimportant ones use a single storage place, each new message thus overwriting the previous one.

A special kind of report is a progress report. If you declared your process to be "progressable," you can report its progress in a very convenient way. Naturally, this kind of report is treated as unimportant.

Last, but not least, is the *ReportAlways* method. Like *Finish*, it doesn't raise an exception, even if the process or the whole mission has been cancelled. You can use it to send some "famous last words." Its usual place is in the **except** clause of the thread's *Execute* method.

At Headquarters

This part of our work is essentially more complex. HQ is responsible for all its field agents. It should provide radio channels (IDs), separate important information from the unimportant, inform the GUI, and solve many other problems "just to keep things moving." Here's how it works.

We have a class (component) that usually resides in a main application thread. Let's name it *Log*. Internally it creates a critical section. When initialized, it creates an upper-level "activity process" with *pidMain*. All secondary "activity processes" are created during *Start* calls. For each process, there is a queue for important messages, as well as some other data slots, e.g. time of start, current progress, error status, etc. *Log* also starts an internal timer. This timer defines how often the GUI will be notified about new information.

Whenever a working thread calls one of the client's methods, the client acquires *Log*'s critical section and performs all the actions needed. This guarantees data consistency. These notifications are very fast. Hence they don't affect overall performance, and the threads usually don't have to wait for each other.

When *Log*'s timer ticks, it first checks whether there was any new data since the last tick. If there was, *Log* generates a series of events. Here is the list, which is defined in Listing Two (beginning on page 14):

- property *OnDataUpdateStart*: *TActivityProcessLogEvent*
- property *OnDataUpdateFinish*: *TActivityProcessLogEvent*
- property *OnProcessStart*: *TActivityProcessEvent*
- property *OnProcessUpdate*: *TActivityProcessEvent*
- property *OnProcessFinish*: *TActivityProcessEvent*

First, *Log* generates an *OnDataUpdateStart* event to signal "got some data, be ready." Then it sends process-specific events for each activity process available. Lastly, it sends *OnDataUpdateFinish*, thus informing us that there is no more new information at the moment. Event handlers can retrieve the information via several of *Log*'s methods, all of which use the same critical section and are quite fast. This way, the GUI doesn't wait for its secondary threads, and the threads don't have to wait for a slow GUI, or for each other.

Here are some other events you may find useful:

- property *OnInitialize*: *TActivityProcessLogEvent*
- property *OnFinalize*: *TActivityProcessLogEvent*
- property *OnIdle*: *TActivityProcessLogEvent*

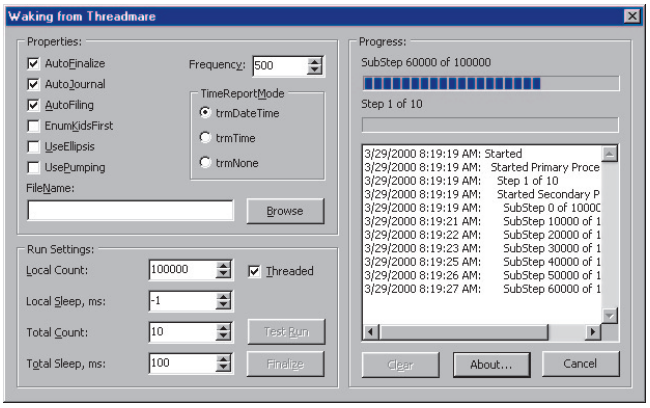


Figure 4: A program demonstrating all of the techniques discussed in this article is available for download.

The first two are fired when *Log* is initialized/finalized. *OnIdle* occurs when no new information has been received between two consecutive timer ticks.

No More *Synchronize/OnTerminate*

As you can well guess, there is no need to use *Synchronize* anymore. Anything you do inside *Log*'s event handler is perfectly thread-safe. *Log* takes care of it for you. There is also no need to use an *OnTerminate* event. Whenever *Log* detects that a thread's master activity process has finished, it waits until the thread is actually terminated (via the *WaitForSingleObject* API call) before firing an *OnProcessFinish* event.

A fully-functional program demonstrating these techniques accompanies this article (see Figure 4). It's available for download; see end of article for details.

Conclusion

This suggested approach provides an easy-to-use way for multi-threaded programming. All you need to do is transfer the data-processing algorithms into secondary threads, supply them with the *Log*'s client, and then drop the *Log* component into your progress dialog form and hook up a few event handlers. Happy multi-threading! Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JUNE\DI200006NS.

Nikolai Sklobovsky is a senior system analyst for Retail Technologies International, house of RetailPro (http://www.retailpro.com), one of the world's best POS systems, where he developed a sophisticated, yet easy-to-use, DSS (Decision Support System) for OLAP analysis of merchant data. He has over 10 years of experience in applied mathematics and teaching at the university level, as well as over 10 years of experience in IT. You can contact Nik at delphi@sklobovsky.com or at his Web site at http://www.sklobovsky.com.

Begin Listing One — Client field agent

```
const
  pidMain =  0 ; // Internal main process, always exists.
  pidCurrent = -1;  // Current process.
  // All other negative process ids are invalid.
  pidInvalid       = -2;

  cbNoProgress     = False;
  cbProgress       = True;
  cbUnimportant    = False;
  cbImportant      = True;
  cbNotJournalOnly = False;
  cbJournalOnly    = True;

type
  EActivityProcessLogError = class(Exception) end;
  EActivityProcessLogAbort =
    class(EActivityProcessLogError) end;
  EActivityProcessLogUserCancel =
    class(EActivityProcessLogAbort) end;

  TActivityProcessStat = (
    apsNewData,   // Something has been changed.
    apsBrandNew,  // Process has just been created.
    apsFinished,  // Process terminated.
    apsError,     // Some (external) error occurred.
    apsAborted,   // Process was stopped (programmatically)
                  // for some reason.
    apsCancelled  // Process was stopped because of user.
  ); // TActivityProcessStat.

  TActivityProcessStatus = set of TActivityProcessStat;
  TActivityProcessError = apsError..apsCancelled;
  TActivityProcessErrors = set of TActivityProcessError;

  TCustomActivityLogClient = class(TObject)
  public
    // Working process side.
    function Start(const csCaption: string;
      bProgressable: Boolean = False;
      pUserData: Pointer = nil): Integer; register;
    // Always accepted if pid is ok.
    procedure Finish(pid: Integer = pidCurrent); register;
    procedure Report(const csText: string = '';
      bImportant: Boolean = False;
      pid: Integer = pidCurrent); overload; register;
    procedure Report(Index, Count: Integer;
      const csText: string = '';
      pid: Integer = pidCurrent); overload; register;
    procedure Journal(strsText: TStrings;
      pid: Integer = pidCurrent); overload; register;
    procedure Journal(const csText: string = '';
      pid: Integer = pidCurrent); overload; register;
    // Important message.
    procedure Notify(const csText: string;
      pid: Integer = pidCurrent); register;
    // Always accepted if pid is ok.
    procedure ReportAlways(const csText: string;
      ErrorClass: ExceptClass = nil;
      pid: Integer = pidCurrent); register;
  end; // TCustomActivityLogClient.
```

End Listing One

Begin Listing Two — *TActivityProcess*

```
const
  // Standard timer frequency (1/2 sec).
  ciDefaulLogFrequency = 50 0 ;

  cbNextInLine         = False;
  cbLastOnly           = True;
  cbImportantOnly      = False;
  cbIncludeUnimportant = True;
  cbAppendToOldLog     = False;
  cbOverwriteOldLog    = True;

type
  TActivityProcess = class;
  TActivityProcessLog = class;

  TTimeReportMode = (
    trmDateTime,  // Timestamp of absolute date and time.
```

```
  trmTime,      // Timestamp of relative time.
  trmNone       // No timestamp.
); // TTimeReportMode.

TActivityMessage = record
  ID: Integer;
  Indent: Integer;
  TimeStamp: TDateTime;
  JournalOnly: Boolean;
  Error: ExceptClass;
end; // TActivityMessage.
TPActivityMessage = ^TActivityMessage;

TActivityProcessEvent =
  procedure (Sender: TActivityProcess) of object;
TActivityProcessLogEvent =
  procedure (Sender: TActivityProcessLog) of object;

TActivityProcess = class
public
  destructor Destroy; override;
  // Reading info (client area) - sequential access.
  // False means no new data available.
  function GetMessage(var Text: string;
    bIncludeUnimportant: Boolean = cbIncludeUnimportant;
    bLastOneOnly: Boolean = cbLastOnly;
    pMsg: TPActivityMessage = nil): Boolean;
  // Unimportant messages buffer.
  function GetText: string;
  // Random access methods - are not used - deleted.
  // aborting methods.
  procedure Terminate(
    Reason: TActivityProcessError = apsAborted;
    const csErrorText: string = '';
    ErrorClass: ExceptClass = nil);
  procedure Abort(const csText: string = '');
  procedure Cancel(const csText: string = '');
  // Properties.
  property Caption: string read FCaption;
  property Progressable: Boolean read FProgressable;
  property pUserData: Pointer read FUserData;
  property Log: TActivityProcessLog read FLog;
  property Parent: TActivityProcess read FParent;
  // Original thread ID which created this process.
  property ThreadID: THandle read FThreadID;
  property ID: Integer read FID;
  // Internally assigned ID - unique within one Log's
  // Init-Done session.
  property TimeStart: TDateTime read FTimeStart;
  // Initially TimeStart.
  property LastUpdate: TDateTime read GetLastUpdate;
  // Initially zero.
  property TimeFinish: TDateTime read GetTimeFinish;
  // 0  is the top-most.
  property Level: Integer read FLevel;
  // If this is a main process in this thread.
  property Master: Boolean read FMaster;
  property Status: TActivityProcessStatus read GetStatus;
  property Position: Integer read GetPosition;
  property Max: Integer read GetMax;
  property KidsCount: Integer read GetKidsCount;
  property Kids[Index: Integer]: TActivityProcess
    read GetKids;
  // Read write.
  property Tag: Integer read GetTag write SetTag;
end; // TActivityProcess

TActivityProcessLog = class(TComponent)
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  // Actual methods.
  procedure Initialize;
  // Asynchronous call - use OnFinalize event.
  procedure Finalize;
  function WaitForAllThreads: Cardinal;
  // Ref-counted.
  procedure StartFiling(bOverwrite: Boolean = True);
  procedure StopFiling;  // Ref-counted.
  procedure GetJournal(strs: TStrings;
    ReportErrors: TActivityProcessErrors =
      [apsCancelled]);   // Termination.
  procedure Kill(pid: Integer = pidCurrent;
    Reason: TActivityProcessError = apsAborted;
    const csErrorText: string = '';
    ErrorClass: ExceptClass = nil); register;
  procedure Abort(const csText: string = '');
  procedure Cancel(const csText: string = '');
  // Properties.
  property Client: TCustomActivityLogClient read FClient;
  property Journal: TStrings
    read FJournal write SetJournal;
  property TimeStart: TDateTime read FTimeStart;
  property LastUpdate: TDateTime read GetLastUpdate;
  property TimeFinish: TDateTime read GetTimeFinish;
  // Timer is ticking.
  property Active: Boolean read GetActive;
  property ErrorStatus: TActivityProcessErrors
    read GetErrorStatus;
  property ThreadsCount: Integer read GetThreadsCount;
  property ActiveCount: Integer read GetActiveCount;
  property ProcessCount: Integer read GetProcessCount;
  property Process[Index: Integer]: TActivityProcess
    read GetProcess; default;
published
  property AutoFinalize: Boolean read FAutoFinalize
    write SetAutoFinalize default True;
  property AutoJournal: Boolean read FAutoJournal
    write SetAutoJournal default True;
  property AutoJournalErrors: TActivityProcessErrors
    read FAutoJournalErrors write SetAutoJournalErrors
    default [apsCancelled];
  property AutoFiling: Boolean read FAutoFiling
    write SetAutoFiling default True;
  property Frequency: Cardinal read GetFrequency
    write SetFrequency default ciDefaulLogFrequency;
  property TimeReportMode: TTimeReportMode
    read FTimeReportMode write SetTimeReportMode
    default trmDateTime;
  property EnumKidsFirst: Boolean read FEnumKidsFirst
    write SetEnumKidsFirst default False;
  property UseEllipsis: Boolean read FUseEllipsis
    write SetUseEllipsis default False;
  // Call Application.ProcessMessages or not.
  property UsePumping: Boolean read FUsePumping
    write SetUsePumping default False;
  property FileName: string read FFileName
    write SetFileName;
  // Event handlers.
  property OnInitialize: TActivityProcessLogEvent
    read FOnInitialize write SetOnInitialize;
  property OnFinalize: TActivityProcessLogEvent
    read FOnFinalize write SetOnFinalize;
  property OnIdle: TActivityProcessLogEvent
    read FOnIdle write SetOnIdle;
  property OnDataUpdateStart: TActivityProcessLogEvent
    read FOnDataUpdateStart write SetOnDataUpdateStart;
  property OnDataUpdateFinish: TActivityProcessLogEvent
    read FOnDataUpdateFinish write SetOnDataUpdateFinish;
  property OnProcessStart: TActivityProcessEvent
    read FOnProcessStart write SetOnProcessStart;
  property OnProcessUpdate: TActivityProcessEvent
    read FOnProcessUpdate write SetOnProcessUpdate;
  property OnProcessFinish: TActivityProcessEvent
    read FOnProcessFinish write SetOnProcessFinish;
end; // TActivityProcessLog.
```

End Listing Two

By Jeremy Merrill

Manipulating Events

Manipulating Event Properties Using *TMethod*

To understand how to manipulate event properties using *TMethod*, you must understand some pointer basics. As its name implies, a pointer is a variable that points to something else. It does this by holding the memory address of what it is pointing to. A pointer is, therefore, simply a memory address, and can reference a variable, object, procedure, or anything else that occupies memory.

We use pointers in Delphi all the time; we just don't think of them as pointers. When we define an object variable like *Form1*, for example, what we're really doing is defining a pointer to an area of memory that holds a *TForm1* object. The *Form1* variable by itself is useless, until we create the actual form that it points to. Figure 1 is an example of pointers in action.

While this isn't exactly useful code, it does illustrate the use of pointers. The declaration:

```
pInt: ^Integer;
```

defines *pInt* as a pointer to an integer. We assign *pInt* the memory address of *x* using the @ operator, and we can dereference what *pInt* is pointing to using the *pInt*^ syntax (where the ^ operator follows the pointer variable).

```
var
  pInt: ^Integer;  // Declare Integer pointer.
  ptr: Pointer;    // Declare generic pointer.
  x: Integer;
  obj: TObject;    // Declare object pointer.
begin
  x := 10 ;
  // The @ operator returns the address of a variable,
  // i.e. a pointer to the variable.
  pInt := @x;
  // The ^ operator returns the value stored at an address
  // i.e. it dereferences the pointer.
  pInt^ := pInt^ + 2;  // x = 12

  ptr := @x;
  Integer(ptr^) := Integer(ptr^) + 2;  // x = 14

  obj := TObject.Create;
  obj.Free;
end;
```

Figure 1: Pointers in action.

The *ptr* variable works the same way. The difference is that *ptr* is defined as a generic pointer, so we must typecast it in order to tell Delphi what it's referencing. The *obj* variable works the same way as well. The *TObject.Create* constructor returns the memory address of the newly created object. We reference the *Free* method indirectly, through the *obj* pointer. Fortunately, Delphi has removed the need to use ^ and @ operators when using object pointers.

Events and *TMethod*

Now let's look at the variables that point to methods. A method is a procedure or function that is part of an object. On the Events tab of the Object Inspector, when we select a control's *OnMouseDown* event and press F1 for help, we get the following type information:

```
type TMouseEvent = procedure (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer) of object;
property OnMouseDown: TMouseEvent;
```

This tells us three things, the most obvious being the arguments the *OnMouseDown* event handler expects. It also tells us that the event is a property. The only reason it shows up on the Events tab instead of the Properties tab is because the Object Inspector is smart enough to figure out which properties are events.

The third thing this tells us is that the value this property holds references a method. We know that it references a method, and not just a standalone procedure, because of the last two words of the *TMouseEvent* type declaration: **of object**. This is an important distinction. A single pointer can reference a procedure or function, but it

takes two pointers to reference a method. All method variables follow the same structure as the basic *TMethod* type, defined in the SysUtils unit as follows:

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

Therefore, all method variables, including all event properties, can be typecast as *TMethod*. When we reference the *Code* pointer of a method property, we're pointing to the object's procedure or function. *Data* points to the instance of the object, and is used to populate the *Self* pointer that's built into all methods. Therefore, when we point the *OnMouseDown* event property to an appropriate method, we not only specify the routine to call, but also the object (commonly a form) that owns the routine.

Why do we care? Because when we typecast a method variable as a *TMethod*, we can do some rather interesting things, as we'll see.

Calling Methods by Name

The first use for *TMethod* we'll look at is its ability to execute a method using its string name. This is accomplished by using a function of *TObject* named *MethodAddress*. Delphi uses *MethodAddress* to convert the method name of an event handler (defined in the .dfm file of a form) into the method pointers assigned to an event property.

To use it, we need to remember that the methods found by *MethodAddress* must be **published**, not just **public** (only **published** methods have their names stored in the object definition). Figure 2 shows an example of how to call methods by name.

In Figure 2, we're storing the values of the method names, such as *RateCalculation03011998* or *RateCalculation09151999*, as *ComboBox1* items. The method name could just as easily have come from a database, parameter, or any other string source (perhaps even a method-name generation routine). Whatever the source, the method's address can be found by using *MethodAddress*. If no such published method exists, a warning message is generated. The actual method call to whatever routine was chosen occurs when the *Rate* variable is assigned the result of the *CalcRate* function.

Retrieving the Object from the Event Property

In my article "Modifying VCL Behavior" (see the February, 2000 issue of *Delphi Informant Magazine*), I wrote about dynamically modifying the VCL. That article included the source code for a *TLinkedLabel* component, which redirected the *WindowProc* event property of another control, named *Associate*. While not mentioned in that article, *TLinkedLabel* has a few problems. If you link two or more LinkedLabels to the same *Associate*, then remove one or more of the LinkedLabels, it's possible to end up with invalid pointers. Here's one possible sequence of events that illustrates this problem:

1) *LinkedLabel1* links to the *Associate*. This sets *LinkedLabel1.FOldWinProc* to *Associate.WindowProc*, and *Associate.WindowProc* to *LinkedLabel1.NewWinProc*.
2) *LinkedLabel2* links to the *Associate*. This sets *LinkedLabel2.FOldWinProc* to *Associate.WindowProc* and *Associate.WindowProc* to *LinkedLabel2.NewWinProc*. However, because of step 1, *Associate.WindowProc* was already redirected, so *LinkedLabel2.FOldWinProc* now points to *LinkedLabel1.NewWinProc*. Note that linking multiple LinkedLabels to the same *Associate* does not cause a problem in and of itself, because the *WindowProc* routines simply chain together.
3) *LinkedLabel1* gets deleted. This causes *LinkedLabel1* to set *Associate.WindowProc* to *LinkedLabel1.FOldWinProc* prior to deletion. While this sets *Associate.WindowProc* back to its original *WindowProc* method, the link between *LinkedLabel2* and *Associate* is now broken.
4) *LinkedLabel2* gets deleted. This causes *LinkedLabel2* to set *Associate.WindowProc* to *LinkedLabel2.FOldWinProc*. However, as we saw in step 2, *LinkedLabel2.FOldWinProc* points to *LinkedLabel1.NewWindowProc*. Therefore, *Associate.WindowProc* now points to *LinkedLabel1.NewWindowProc*. Because *LinkedLabel1* has been deleted, *Associate.WindowProc* now points to a non-existent object, resulting in probable access violations.

How does *TMethod* help us resolve this problem? By allowing us to interrogate the *Associate* to determine if it's already linked to another object. The following is one possible solution that simply

```
type
  TRateFunc = function: Double of object;

  TCalcRoutines = class(TObject)
  private
    FCalcRate: TRateFunc;
  public
    property CalcRate: TRateFunc
      read FCalcRate write FCalcRate;
  published
    function RateCalculation03011998: Double;
    function RateCalculation06151998: Double;
    function RateCalculation01011999: Double;
    function RateCalculation04011999: Double;
    function RateCalculation09151999: Double;
  end;

var
  CalcRoutines: TCalcRoutines;

implementation

procedure TForm1.ComboBox1Change(Sender: TObject);
var
  rc: TMethod;
  Rate: Double;
begin
  rc.Code := CalcRoutines.MethodAddress(ComboBox1.Text);
  if (not Assigned(rc.Code)) then
    ShowMessage('Invalid Rate Calculation Requested.')
  else
    begin
      rc.Data := CalcRoutines;
      CalcRoutines.CalcRate := TRateFunc(rc);
      ...
      // Selected method is executed.
      Rate := CalcRoutines.CalcRate;
      ...
    end;
end;

...

initialization
  CalcRoutines := TCalcRoutines.Create;

finalization
  CalcRoutines.Free;

end.
```

Figure 2: Calling methods by name.

prevents a LinkedLabel from linking to an associate control, if that associate is already linked to a different object:

```
procedure TLinkedLabel.SetAssociate(Value: TControl);
begin
  if (Value <> FAssociate) then
    begin
      if ((Assigned(Value)) and
         (TMethod(Value.WindowProc).Data <> Value)) then
        Exit;
      if (Assigned(FAssociate)) then
        FAssociate.WindowProc := FOldWinProc;
      FAssociate := Value;
      ...
```

The *if ((Assigned(Value))* statement, and adding SysUtils to the **uses** statement, are the only things we need to add to *TLinkedLabel* to get this to work. Another possible approach could be to make the LinkedLabels aware of each other and cleanly handle chaining. Although this is a more complicated solution (the details of which I will defer to the reader), this could be done by referencing the existing LinkedLabel through the *Data* pointer, as we did in the previous example.

Using Data/Self as a Hidden Parameter

Another possible use of *TMethod* typecasting is to use the *Data* pointer, and *Self*, as hidden parameters. An example is shown in Figure 3.

First, let's look at *FormCreate*. Notice that when we set *Code* to the address of *TestProc*, we must reference *TestProc* by its class. This isolates the procedure from any object, and allows the @ operator to return a single memory address (using *@TestProc* instead of *@TForm1.TestProc* will result in a compile-time error). The next line of code may look a little strange. This takes advantage of the fact that pointers and integers both use four bytes of memory. By typecasting an integer as a pointer, we can store the integer value in a pointer variable. This same approach can be used to reference objects in *Tag* properties, store integer values in *TStrings.Objects*, and so on.

When *Button1* is pressed, *TestProc* will be executed, but the *Self* pointer within *TestProc* will hold the integer value of 123 instead of referencing *Form1*. To retrieve the value stored in *Self*, we simply typecast it as an integer. We set *Button1*'s caption to 123 in the second line to demonstrate that the integer value was actually passed and retrieved. Also notice that *Self* can be reassigned, just as any other variable. Here we reset *Self* to *Button1*'s parent form, *Form1*, by using Delphi's *GetParentForm* function, which takes a *TControl* parameter. This step is important, because the last line of *TestProc* relies on *Self* being set correctly.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  tmp: TNotifyEvent;
begin
  TMethod(tmp).Code := @TForm1.TestProc;
  // Store the integer value 123 in Data.
  TMethod(tmp).Data := Pointer(123);
  Button1.OnClick := tmp;
end;

procedure TForm1.TestProc(Sender: TObject);
var
  i: Integer;
begin
  i := Integer(Self);  // i = 123
  // Set button caption to 123.
  TButton(Sender).Caption := IntToStr(i);
  // Set Self to Form1.
  Self := (GetParentForm(Sender as TControl) as TForm1);
  // Set Form1.Caption to 'Parent Form'.
  Caption := 'Parent Form';
end;
```

Figure 3: Using the *Data* pointer, and *Self*, as hidden parameters.

The important thing to understand when using this technique is that it can be potentially dangerous if done incorrectly. Remember that *Self* is assumed at compile time to be a *TForm1* object. The compiler doesn't know when *Self* points to something else. If we had not reset *Self* to *Form1* in the previous example, the code would have tried to set *TForm1(123).Caption* to *Parent Form*, which would result in an access violation. To safely use this technique, therefore, you need to reset *Self* before any references to the parent object's fields, properties, or methods.

Using a Stand-alone Procedure as an Event Handler

TMethod allows us to set the *Code* portion of an event property to the address of a stand-alone procedure. If you've ever wanted to use a procedure as an event handler, Figure 4 shows you how.

While this works essentially the same way as the code in the previous section, assigning the address of the procedure to the *Code* pointer, there is one very important distinction. Notice in Figure 4 that the *Evnt* local variable of *FormCreate* is defined as a *TNotifyEvent*. This makes it type compatible with an *OnClick* event property. *TNotifyEvent* is defined as:

```
TNotifyEvent = procedure (Sender: TObject) of object;
```

TNotifyEvent has only one parameter, *Sender*, but the *BtnClick* procedure has two parameters, *Data* and *Sender*. *Evnt* and *BtnClick* are actually type compatible. How? Because *Evnt* is a method, and *BtnClick* is a stand-alone procedure. When we call a method, Delphi passes whatever is defined in the *Data* pointer as a hidden parameter. The assembled code for a method automatically reads this hidden parameter and assigns *Self* to it. The assembled code for a stand-alone procedure, however, doesn't know anything about a hidden parameter, so we have to insert it at the beginning of the parameter list.

In the example shown in Figure 3, pressing *Button1* will bring up the message "Data Empty," while pressing *Button2* will display "Data Found." Both buttons will change their captions to "Clicked" after the message has been displayed.

```
procedure BtnClick(Data: Pointer; Sender: TObject);
begin
  if (Assigned(Data)) then
    ShowMessage('Data Found')
  else
    ShowMessage('Data Empty');
  if (Sender is TButton) then
    (Sender as TButton).Caption := 'Clicked';
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  Evnt: TNotifyEvent;
begin
  TMethod(Evnt).Code := @BtnClick;
  TMethod(Evnt).Data := nil;
  Button1.OnClick := Evnt;
  TMethod(Evnt).Data := Button2;
  Button2.OnClick := Evnt;
end;
```

Figure 4: Using a procedure as an event handler.

TNotifyList

Now we're going to create a new class, named *TNotifyList*, that acts like a *TList* for procedures and methods, as shown in Listing One (beginning on page 19). While this class demonstrates many of the techniques we've already discussed, I'm including it here to provide another practical example of how manipulating method pointers can add additional power and flexibility to your applications. The main purpose of this class is to create lists of events that can be modified on the fly, and to call all the events in any given list with a single method call. Because the purpose of this class is one of notification, we'll restrict the procedures and methods to those matching the *TNotifyEvent* structure.

The first thing you'll notice when looking at the *TNotifyList* class is that we use two *TList* objects to hold the list of methods — one to hold the *Code* pointers, and the other to hold the *Data* pointers. By keeping these two lists "in sync," we can combine the *Code* and *Data* pointers from the two lists to form a single method pointer.

Another important feature of *TNotifyList* is that it can contain methods and procedures. Notice the declaration of *TNotifyProc* near the top of the unit. Although similar to *TNotifyEvent* it's missing the **of object** syntax that distinguishes methods from procedures. By using method overloading (introduced in Delphi 4), we can add, remove, and reference entries using the same commands, whether they are procedures or methods. Internally, we simply distinguish between methods and procedures by checking to see if the *Data* pointer is **nil**.

One point that merits attention is the apparent duplication of code in the class definition. While most of the overloaded methods are similar, the *Remove* methods appear to be identical. They are, however, different in two respects: The passed parameter is a *TNotifyEvent* in one method and a *TNotifyProc* in the other method. This causes the second difference in the two routines in that the calls to *IndexOf* are referencing different overloaded methods. Because Delphi resolves overloaded calls at compile time by looking at parameter types, two separate *Remove* methods are required by the compiler. We should also mention possible uses of a *TNotifyList* object. If you have a large application, you may have many places in the code that need to know when particular application-wide events occur, perhaps a change in status or mode, or a configuration setting. Having a single event handler that tries to notify everything that cares about the change can be a daunting task, especially if some of those notifications are dependent on whether objects or forms have been created or not. By using a *TNotifyList*, you can shift the burden from the event handler to the individual forms, objects, and other code that care about the event. An object's constructor can add its own notification method to the list, and remove it when the object is destroyed. All the event handler needs to do is call the NotifyList's *Notify* event to call all the associated methods and procedures.

Conclusion

Manipulating method properties by typecasting them as a *TMethod* can provide interesting capabilities, and increased power and flexibility to an application. It can also help resolve difficult programming problems. The *TNotifyList* class, in particular, can help resolve many event synchronization issues. I'm sure there are many more uses of these techniques than I have presented. Hopefully, you will find these techniques useful. Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JUNE\DI200006JM.

Jeremy Merrill is an EDS contractor in a partnership contract with the Veteran's Health Administration. He is a member of the VA's Computerized Patient Record System development team, located in the Salt Lake City Chief Information Officer's Field Office.

Begin Listing One — *TNotifyList*

```
unit NotifyList;

interface

uses
  SysUtils, Classes;

type
  TNotifyProc = procedure(Sender: TObject);

  TNotifyList = class(TObject)
  private
    FCode: TList;
    FData: TList;
  protected
    function GetIsProc(index: Integer): Boolean;
    function GetMethods(index: Integer): TNotifyEvent;
    function GetProcs(index: Integer): TNotifyProc;
    procedure SetMethods(index: Integer;
      const Value: TNotifyEvent);
    procedure SetProcs(index: Integer;
      const Value: TNotifyProc);
  public
    constructor Create;
    destructor Destroy; override;
    procedure Add(const NotifyProc: TNotifyEvent);
      overload;
    procedure Add(const NotifyProc: TNotifyProc); overload;
    procedure Clear;
    function Count: Integer;
    procedure Delete(index: Integer);
    function IndexOf(const NotifyProc: TNotifyEvent):
      Integer; overload;
    function IndexOf(const NotifyProc: TNotifyProc):
      Integer; overload;
    procedure Notify(Sender: TObject);
    procedure Remove(const NotifyProc: TNotifyEvent);
      overload;
    procedure Remove(const NotifyProc: TNotifyProc);
      overload;
    property IsProc[index: Integer]: Boolean
      read GetIsProc;
    property Methods[index: Integer]: TNotifyEvent
      read GetMethods write SetMethods;
    property Procs[index: Integer]: TNotifyProc
      read GetProcs write SetProcs;
  end;

implementation

{ TNotifyList }

constructor TNotifyList.Create;
begin
  inherited;
  FCode := TList.Create;
  FData := TList.Create;
end;

destructor TNotifyList.Destroy;
begin
  FData.Free;
  FCode.Free;
  inherited
end;

procedure TNotifyList.Add(const NotifyProc: TNotifyEvent);
var
```

```
  m: TMethod;
begin
  if (Assigned(NotifyProc) and
     (IndexOf(NotifyProc) < 0 )) then
  begin
    m := TMethod(NotifyProc);
    FCode.Add(m.Code);
    FData.Add(m.Data);
  end;
end;

procedure TNotifyList.Add(const NotifyProc: TNotifyProc);
begin
  if (Assigned(NotifyProc) and
     (IndexOf(NotifyProc) < 0 )) then
    begin
      FCode.Add(@NotifyProc);
      FData.Add(nil);
    end;
end;

procedure TNotifyList.Clear;
begin
  FCode.Clear;
  FData.Clear;
end;

function TNotifyList.Count: Integer;
begin
  Result := FCode.Count;
end;

procedure TNotifyList.Delete(index: Integer);
begin
  FCode.Delete(index);
  FData.Delete(index);
end;

function TNotifyList.GetIsProc(index: Integer): Boolean;
begin
  Result := (not Assigned(FData[index]));
end;

function TNotifyList.GetMethods(index: Integer):
  TNotifyEvent;
begin
  TMethod(Result).Code := FCode[index];
  TMethod(Result).Data := FData[index];
end;

function TNotifyList.GetProcs(index: Integer): TNotifyProc;
begin
  Result := FCode[index];
end;

function TNotifyList.IndexOf(
  const NotifyProc: TNotifyEvent): Integer;
var
  m: TMethod;
begin
  if (Assigned(NotifyProc) and (FCode.Count > 0 )) then
    begin
      m := TMethod(NotifyProc);
      Result := 0 ;
      while ((Result < FCode.Count) and
            ((FCode[Result] <> m.Code) or
            (FData[Result] <> m.Data))) do
        Inc(Result);
      if Result >= FCode.Count then
        Result := -1;
    end
  else
    Result := -1;
end;

function TNotifyList.IndexOf(
  const NotifyProc: TNotifyProc): Integer;
var
  prt: ^TNotifyProc;
begin
  prt := @NotifyProc;
  if (Assigned(NotifyProc) and (FCode.Count > 0 )) then
    begin
      Result := 0 ;
      while ((Result < FCode.Count) and
            ((FCode[Result] <> prt) or
            (FData[Result] <> nil))) do
        Inc(Result);
      if Result >= FCode.Count then
        Result := -1;
    end
  else
    Result := -1;
end;

procedure TNotifyList.Notify(Sender: TObject);
var
  i: Integer;
  evnt: TNotifyEvent;
  proc: TNotifyProc;
begin
  for i := 0 to FCode.Count-1 do
    if (FData[i] = nil) then
      begin
        proc := FCode[i];
        if (Assigned(proc)) then
          proc(Sender);
      end
    else
      begin
        TMethod(evnt).Code := FCode[i];
        TMethod(evnt).Data := FData[i];
        if (Assigned(evnt)) then
          evnt(Sender);
      end;
end;

procedure TNotifyList.Remove(
  const NotifyProc: TNotifyProc);
var
  idx: Integer;
begin
  idx := IndexOf(NotifyProc);
  if (idx >= 0 ) then
    begin
      FCode.Delete(idx);
      FData.Delete(idx);
    end;
end;

procedure TNotifyList.Remove(
  const NotifyProc: TNotifyEvent);
var
  idx: Integer;
begin
  idx := IndexOf(NotifyProc);
  if (idx >= 0 ) then
  begin
    FCode.Delete(idx);
    FData.Delete(idx);
  end;
end;

procedure TNotifyList.SetMethods(index: Integer;
  const Value: TNotifyEvent);
begin
  FCode[index] := TMethod(Value).Code;
  FData[index] := TMethod(Value).Data;
end;

procedure TNotifyList.SetProcs(index: Integer;
  const Value: TNotifyProc);
begin
  FCode[index] := @Value;
  FData[index] := nil;
end;

end.
```

End Listing One

By Jason Perry

Exploiting SQL Server 7 DMO

Part I: Building a SQL Server Scripting Tool

As enterprises grow more dependent on distributed database technology, the need for powerful management applications becomes critical in keeping a multi-server environment healthy. Because it's impossible for a tool vendor to know all the strategies a company will use, the tools supplied with the products are often severely limited in capability. Microsoft recognized this problem and created a set of COM objects, called Microsoft SQL Server Distributed Management Objects (SQL-DMO), to aid in the management of their SQL Server DBMS.

SQL-DMO is a part of Microsoft's SQL Distributed Management Framework (SQL-DMF). SQL-DMF is a framework of objects and services that are used to manage Microsoft SQL Server. It enables you to perform unattended tasks, such as database backup, by providing objects that work with SQL Server. The SQL Enterprise Manager is an example of a Microsoft product that uses the SQL-DMF and SQL-DMO objects. For our purposes, we'll stick to exploiting SQL-DMO objects.

This two-part series demonstrates how to use Microsoft's SQL Server 7 DMO objects in the development of database management tools and COM-based business objects for enterprise Microsoft SQL Server applications. While learning how to use SQL-DMO objects, I will complete a script-writing tool for you SQL developers. The next installment of the series will present a database reconciliation tool to aid in cross-database object comparisons, and a simple security object to demonstrate how the object can be used in your application development. Readers are expected to be familiar with OLE Automation, and Microsoft SQL Server 7.

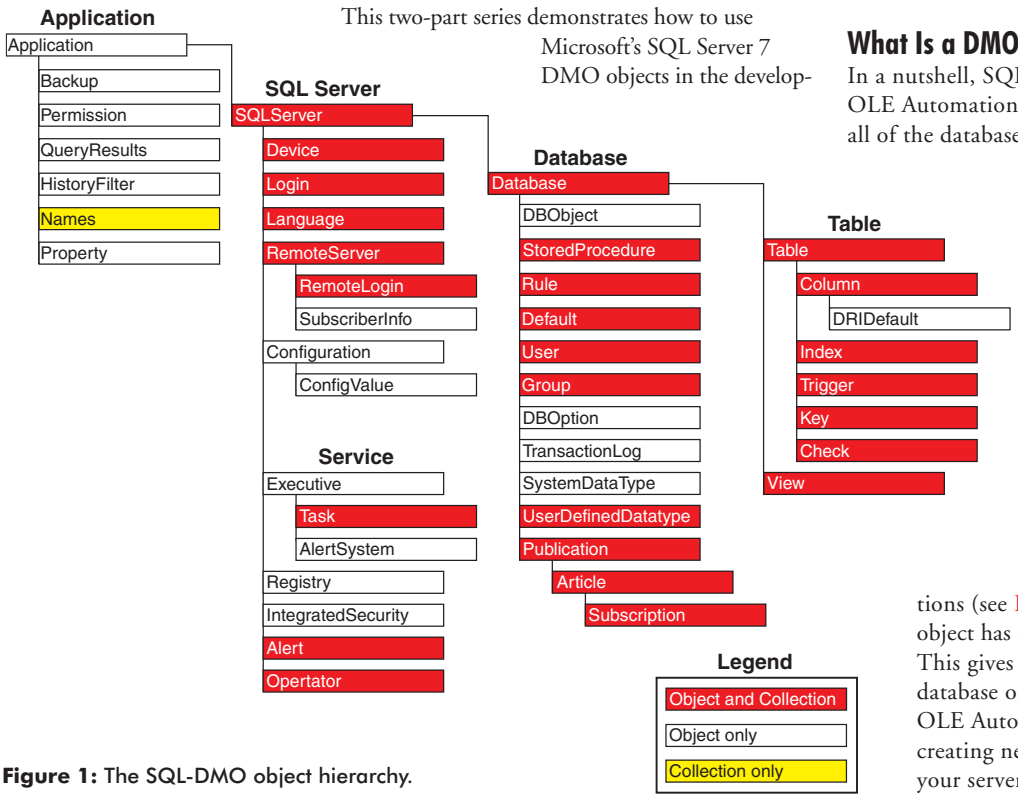


Figure 1: The SQL-DMO object hierarchy.

What Is a DMO Object?

In a nutshell, SQL-DMO objects are in-process OLE Automation servers. These servers expose all of the database objects in Microsoft SQL Server that you would normally manipulate by writing Transact-SQL by hand (in Enterprise Manager for instance), or by using a third-party tool, such as Erwin Modelmart. Each of these objects have properties and methods that you would access in code, very much like a Delphi VCL component. The SQL-DMO object hierarchy is logically arranged and makes excellent use of collections (see Figure 1). For instance, a *Server* object has a collection of *Database* objects. This gives you the ability to evaluate each database on a server through a single OLE Automation server, as opposed to creating new objects for each database on your server.

File name	Description
redist.txt	Redistribution file list and licensing policy.
sqldmo.hlp	The DMO help file. This contains just about everything you'd like to know about SQL-DMO objects, their methods, and properties.
sqldmo.dll	The SQL-DMO in-process server DLL and COM object.
sqldmo.rll	The SQL-DMO resource file.
sqlresld.dll	SQL Enterprise Manager Resource DLL Loader.
sqlsvc.dll	Database Service Layer.
sqlsvc.rll	Database Service Layer Resource DLL.
sqlwoa.dll	SQL Server Unicode/ANSI Translation Layer.
sqlwid.dll	SQL Server Unicode/ANSI Translation Layer.
w95scm.dll	SQL Service Control Manager Abstraction Layer.
pre60to7.SQL	Creates 6.0 MSDB tables in order to "stage" the data being imported by Convert.
pre65to7.SQL	Modifies the tables created by PRE60TO7.SQL to make them look like the 6.5 versions of those tables. Must be run after PRE60TO7.SQL.

Figure 2: SQL-DMO required files.

Note how the object collections are arranged logically. The server object has a collection of databases, the database objects have a collection of tables, the table objects have a collection of triggers, etc.

Create and Access SQL-DMO Objects from Delphi

The first thing to do is get the development environment working. Complete source code for the three projects described in this series are available for download in Delphi 4 and Delphi 5 versions (see end of article for details). The code works in Delphi 3, but the IDE/menu options differ slightly. Note that all three projects must be run from Windows NT. There are some other requirements:

- You must have Microsoft SQL Server 7 Client Tools installed.
- You must have DBO SQL Server 7 security rights.
- The Borland Database Engine (BDE) must be installed.

The easy answer to getting the required files is to install the Microsoft SQL Server Client Tools on your development machine. These files (see Figure 2) are required to create and access SQL-DMO objects, and can be browsed on your Microsoft SQL Server 7 CD.

The files in Figure 2 are redistributable, according to Microsoft's redistribution policy in redist.txt (be sure to read this). Note the .SQL script files. These give you the ability to update older servers to use the SQL-DMO objects in Microsoft SQL Server 7. Be sure to open each script and read the headers to learn more.

Creating the SQL-DMO Objects

Once you've set up your development environment, you can create and manipulate the SQL-DMO objects just as you would any other OLE Automation object. The first thing to do is to import the type library for the SQL-DMO COM object. From Delphi, start a new application and save it in a meaningful directory.

Then select **Project | Import Type Library** to display the Import Type Library dialog box (see Figure 3). Scroll down the window of COM objects, select **Microsoft SQLDMO Object Library (Version 7.0)**, then click **Install**. A file named SQLDMO_TLB.pas will be created in your application directory. This file contains all the interfaces for the COM object, and all the enumerated types that the objects use. Even if you are not an expert at COM, the imported type library is easy to read and understand.

The first object we'll create is the *SQLServer* object. There are several ways it can be created. First, put SQLDMO_TLB.pas in your **uses** clause. The first way to create the *SQLServer* object is to call the *coCreate* method, which is located in the **implementation** section of the type library:

```
uses
  SQLDMO_TLB.pas
var
  SQL_DMO : _SQLServer;

SQL_DMO := CoSQLServer.Create;
```

There are some benefits to doing it this way. One is that the Delphi IDE uses the unit to do type checking and provide code completion for you. The other is that it uses a vTable lookup to call the COM objects' methods. This is much faster than doing it the other way, which uses the IDispatch interface to call the methods:

```
uses
  SQLDMO_TLB.pas
var
  SQL_DMO : Variant;

SQL_DMO := CreateOLEObject('SQLDMO.SQLServer');
```

This should look familiar to VB converts. For this article, I am going to do it the second way. Why? Mostly so I can use the object's methods more loosely. Working with variants will make implementing the SQL Scripting Tool easier. More on that later.

Once you've created the *SQLServer* object (by whatever method), you will have access to all of its methods and properties, just as you would any other Delphi VCL component. The next step is to

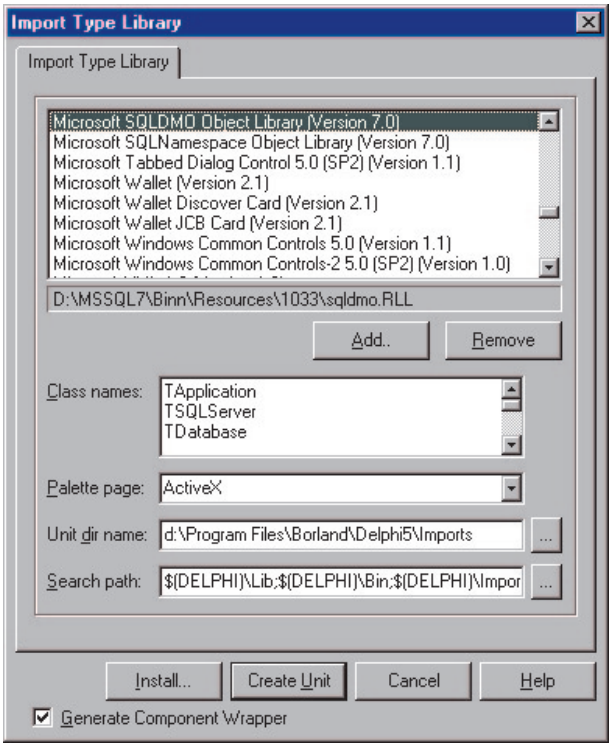


Figure 3: The Delphi 5 Import Type Library dialog box (Delphi 4's differs slightly).

open a connection to the server. You may want to set a few of the properties before opening the connection:

```
// Use NT Authentication (as opposed to
// SQL Server Authentication).
SQL_DMO.LoginSecure := True;
// Set a reasonable timeout.
SQL_DMO.LoginTimeout := 30 ;
// Autoreconnect if connection is lost.
SQL_DMO.AutoReconnect := True;
// Assign application name so server knows who I am.
SQL_DMO.ApplicationName := 'SQL Script Builder';
```

The *LoginSecure* property tells the object to use NT Authentication, or both NT Authentication and SQL Server Authentication. Consult your data administration team to set the property correctly. The *ApplicationName* property is used to name the connection to the SQL Server. When viewing active server connections, this value will be seen on the screen. The other properties are self-explanatory. Next, open the connection and verify that it opened (see Figure 4).

The *Connect* method takes one argument — the server name we want to connect to. Hint: If you're using the desktop SQL Server installation, hard-code a period (.) as the parameter. This tells it to connect to the local server. If you create your object using the *coCreate* method, you'll have to pass it the login name and password. This is a direct example of why I wanted to use the objects more "loosely" and therefore chose the *CreateOLEObject* method. In this case, I rely on the trusted connection to allow access to the server.

The *VerifyConnection* method requires an argument of type SQLDMO_VERIFYCONN_TYPE (see Figure 5). I chose SQLDMOConn_ReconnectIfDead from the type library that we imported. This option will make an attempt to reconnect the server object if it did not connect properly.

You can see how easy it is to get started. You now have a connected *SQLServer* object for which you can access various object collections, methods, and properties.

```
// Connect.
SQL_DMO.Connect('My Server Name Here');
// Test the connection.
if not SQL_DMO.VerifyConnection(
         SQLDMOConn_ReconnectIfDead) then begin
  raise Exception.Create(
    'An error has occurred while attempting to' + #10 #13 +
    'connect to the SQL OLE Server.' + #10 #13#10 #13 +
    'Be sure to load the SQL Server 7.x tools' + #10 #13 +
    'before attempting to use SQL Script Builder.' +
    #10 #13 + '(msg:dmSQL.pas/SetSQLObjectsDatabase)');
end;
```

Figure 4: Open and verify the connection to server.

Constant	Description
SQLDMOConn_ReconnectIfDead	Attempt to reconnect if not connected.
SQLDMOConn_LastState	If not connected, return to the last known state.
SQLDMOConn_CurrentState	If not connected, stay in the current state.
SQLDMOConn_Valid	Validate the connection.

Figure 5: The SQLDMO_VERIFYCONN_TYPE enumerated constants.

SQLServer Object Collections

Collections are exactly what they sound like: a homogeneous grouping of objects referenced by a variable. They're used throughout the entire SQL-DMO hierarchy. They have an *Item* property that returns specific objects. Note that the collections are 1-based, not 0-based, as you're used to. Some collections to look at include *Databases*, *Tables*, *Columns*, *StoredProcedures*, *Triggers*, *Indexes*, *Users*, *Logins*, *DatabaseRoles*, etc. When you use the *Item* property of each collection, you return the *Database*, *Table*, *Column*, *StoredProcedure*, *Trigger*, *Index*, *User*, *Login*, and *Role* objects, respectively. Pay particular attention to the high usage of collections as we develop the three applications.

Building a SQL Scripting Tool

One of the things developers do on any size project is write tons of stored procedures, views, and triggers. Unless you have a tool like Erwin to create your scripts, you'll have to enter them by hand in the Enterprise Manager ISQL window. Even though it works, it doesn't promote consistency, nor does it have configuration management support for the scripts you write. What I've done is demonstrate the use of SQL-DMO objects in a fully functional, template-based, script-building tool I call SQL Script Builder (SSB). It allows developers to create their own templates from which to build scripts, and also provides a template for standard *Get*, *Insert*, *Update*, and *Delete* procedures. Data administration teams love this because they can get the developers to write consistent script. Developers love it because they can create their own templates to aid their scripting, and don't have to be hassled by typing standards and comments for every script they create.

I want to start by showing you the tool and explaining its functionality. As we go along, I'll show you how I did it. Be sure to create a System DSN, named "PUBS," pointing to "PUBS" in your ODBC administrator. I will use this database for testing purposes throughout the article. When you start SSB, you will see a hierarchical view of the servers available to you (see Figure 6). This uses the *TSession* object (which exists in every database application by default) to get a list of SQL Servers. The source is shown in Listing One (on page 25).

Note the *dmoObject* variable. It is a pointer to a record of type *TdmoObject*:

```
type
  PdmoObject = ^TdmoObject;
  TdmoObject = record
    SQL_DMO    : _SQLServer;
    SQL_DB     : _Database;
    SQL_OBJ    : Variant;
    lConnected : Boolean;
  end;
```

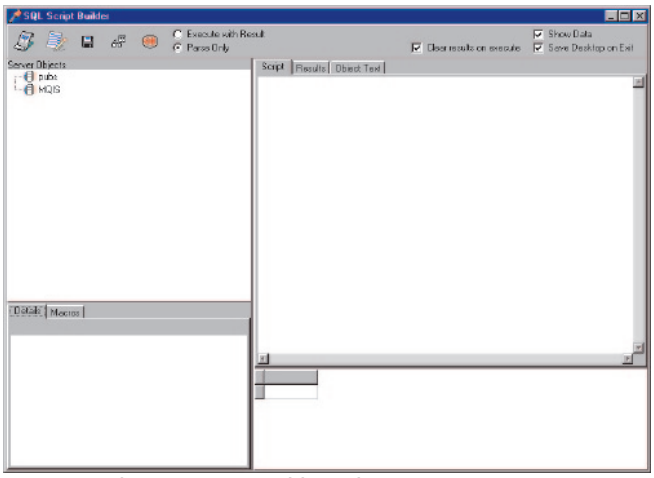


Figure 6: The SQL Script Builder utility at run time.

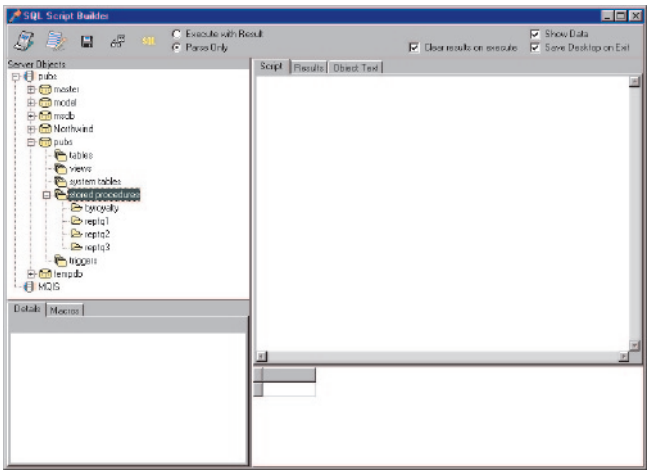


Figure 7: The PUBS database viewed through SSB.

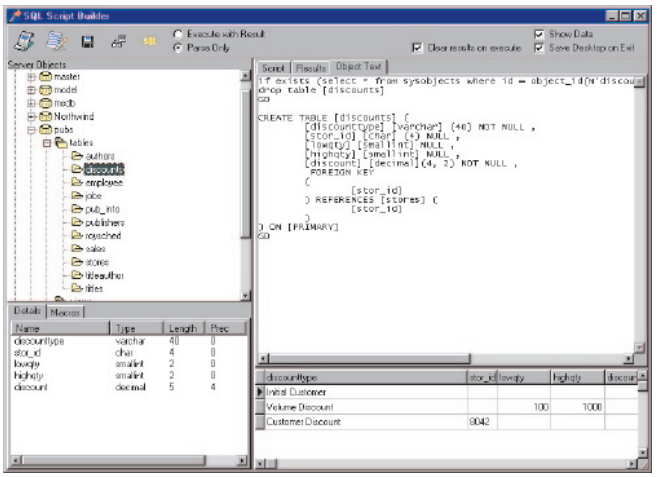


Figure 8: The *tables* collection and Object Text window.

```
procedure Tbo_SQL.LoadTables(oTreeNode: TTreeNode);
var
  lcv : Integer;
  db : _Database;
  dmoObject : PdmoObject;
begin
  // Delete all of the child nodes first.
  oTreeNode.DeleteChildren;
  // Get the node's SQL_DB property.
  db := PdmoObject(oTreeNode.Data)^.SQL_DB;
  db.Tables.Refresh(True);
  for lcv := 1 to db.Tables.Count do begin
    // Table list (non-system).
    if not db.Tables.Item(lcv, Null).SystemObject then
      begin
        new(dmoObject);
        dmoObject.SQL_DMO :=
          PdmoObject(oTreeNode.Data)^.SQL_DMO;
        dmoObject.SQL_DB := db;
        dmoObject.SQL_OBJ := db.Tables.Item(lcv, Null);
        dmoObject.lConnected :=
          PdmoObject(oTreeNode.Data)^.lConnected;
        oTreeNode.Owner.AddChildObject(oTreeNode,
          db.Tables.Item(lcv, Null).name,
          dmoObject).StateIndex := 4;
      end;
  end;
  oTreeNode.expanded := True;
end;
```

Figure 9: The *LoadTables* procedure.

A pointer to each object gets created (new method) and added to the tree node (*TTreeView.AddObject*). Don't forget to clean up your pointers in the *TTreeView.OnDeletion* event. All of the methods in the dmSQL.pas unit take a *TTreeNode* as an argument. Now any object is easily accessed and manipulated.

Click on the PUBS database. Navigate your way down to the stored procedures for this object (see Figure 7). Note that as each node is clicked, a method is called with the node.

The tree nodes under the PUBS node represent SQL-DMO collections for the *SQLServer* object. As you click on each node, SSB spins through the collections and presents the data to you. At the same time, it stores a pointer to each object in each collection, just as I did in the *LoadServers* procedure. This is really handy because as the tree nodes are selected and expanded, the pointer to each object is readily available.

Now click on the tables node. A list of available tables appears (see Figure 8). Next, select the "discounts" table and click on the Object Text page. The script that created the table is presented to you. This works for all the SQL Server objects. At the same time, the attributes of the table are presented in the page control under the tree view (see Figure 9).

System tables have their own node by virtue of the *SystemObject* property. Notice that as I spin through the *Tables* collection, I test each table object's *SystemObject* property. This is how I separate the system tables. Then I create the pointer to the record object and assign its values. Every node has access to the *SQLServer* object, the selected database object, and then itself. I also threw in a "connected" variable that refers back to the *SQLServer* object. That's it! Some simple UI manipulation and you end up with a cool *TTreeView* to navigate your SQL servers and their objects.

So, how did I get the cool script information to appear in the right-hand page control? Each SQL Server object has a "script" method. All I did was pass the *TTreeNode* into the method and invoke the method:

```
procedure Tbo_SQL.GetSQLScriptText(oMemo: TMemo;
  oTreeNode: TTreeNode);
begin
  // Prevent unselected nodes from entering.
  if (oTreeNode = nil) or
     (not assigned(oTreeNode.Parent)) or
     VarIsEmpty(PdmoObject(oTreeNode.data)^.SQL_OBJ) then
    Exit;
  // Get the script.
  oMemo.text := PdmoObject(oTreeNode.data)^.SQL_OBJ.Script(
    SQLDMOScript_Default + SQLDMOScript_Drops);
end;
```

I also pass in a *TMemo* control so the method can fill it. In my example, I call the script method with two arguments. This returns the script that created the object and creates the DROP script as well. You can pass any of 38 possible arguments of type SQLDMO_SCRIPT_TYPE=TOleEnum, which can show the permissions, constraints, DRI, indexes, and even create the DROP statements for you. This is an extremely powerful feature in itself. It would be a good UI enhancement to add the capability to modify the arguments. (If any of you do it, please send it to me.)

Now let's get into the business of creating new scripts. The tool I wrote is very drag-and-drop oriented. When you select a table

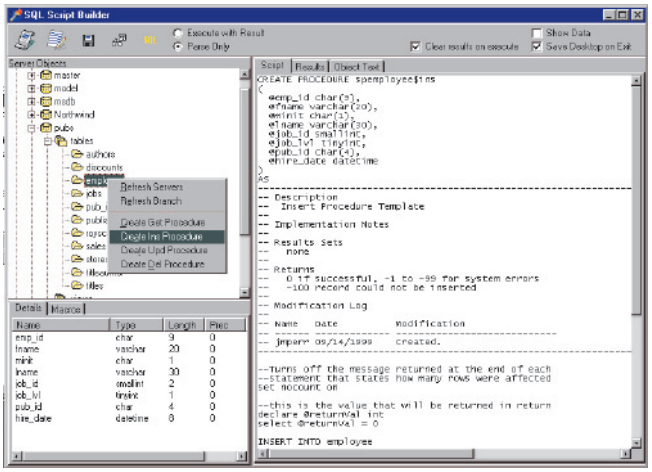


Figure 10: SSB scripting options are presented in a pop-up menu.

and right-click on it, you are presented with several options (see Figure 10).

Select the **Create Ins Procedure** menu option. You will see a brand new INSERT procedure created. The template name is templates.ini. Defined at the top are some macros with which you can create your own templates. To build this template, a method is again called with *TTreeNode*, and a simple macro substitution is done to fill in the fields.

Select the **Parse Only** radio button on the tool bar, then click the **SQL** speed button. The script will execute on the server. If any errors occur, you are notified in a message dialog box. If it's successful, it will tell you "Successful Parse." When the speed button is clicked, it calls the *Execute* method, then calls the *Database* object's *ExecuteImmediate* method with the argument *SQLDMOExec_NoExec*. This tells the server to parse the script and not execute it. It's perfect for testing the syntax of your script before executing it. The source for the *Execute* method is shown in Listing Two (on page 26).

Because that's working well, select the **Execute with Result** radio button and execute the script again. Now the script is committed to the server. Refresh your stored procedure list and you will see it. If you view the object text, you will see the procedure you just created.

The *Execute* method is also called if you type in your own SQL script statement. Type in:

```
SELECT * FROM Employee
```

and execute it. SSB will change pages to the Results page with the table's contents. In the *Execute* method described previously, when the *Database* object's *ExecuteWithResult* method is invoked, any results are returned in another collection object. These results are traversed and the properties used to fill the *TMemo* with the rows that are returned. Piece of cake, huh?

Conclusion

In this first installment of a two-part series, we began with the basics of SQL-DMO objects. Then, we discussed a script-writing tool for SQL developers, named SSB.

Next month, in Part II, we'll continue with a look at a database reconciliation tool for cross-database comparisons, and a simple security application that demonstrates how a custom security object can be used in application development. Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JUNE\DI200006JP.

Jason 'Wedge' Perry is a System Architect for OOP.COM in Chesapeake, VA. Before accepting this position, Wedge was a self-employed consultant in development positions ranging from grunt programmer to system architect. In his spare time, Wedge races a Kawasaki KX250 moto-cross motorcycle for the Elizabeth City MX Club.

Begin Listing One — The *LoadServers* procedure

```
procedure Tbo_SQL.LoadServers(oTreeView: TTreeView);
var
  lcv : Integer;
  oAliasList : TStrings;
  dmoObject : PdmoObject;
begin
  oAliasList := TStringList.Create;
  try
    // Since servers are being determined,
    // clear and reload the treeview.
    oTreeView.Items.Clear;
    Session.GetAliasNames(oAliasList);
    // Spin thru and fill aliases.
    for lcv := 0 to oAliasList.Count - 1 do begin
      if Session.GetAliasDriverName(
           oAliasList[lcv]) = 'SQL Server' then
        Continue
      else
        begin
          new(dmoObject);
          dmoObject.SQL_DMO := CoSQLServer.Create;
          // Name it.
          dmoObject.SQL_DMO.Name := '.';// oAliasList[lcv];
          // Use NT Authentication.
          // (as opposed to SQL Server Authentication).
          dmoObject.SQL_DMO.LoginSecure := True;
          // Set a reasonable timeout.
          dmoObject.SQL_DMO.LoginTimeout := 3;
          // Autoreconnect if connection is lost.
          dmoObject.SQL_DMO.AutoReconnect := True;
          // Assign application name so server
          // knows who I am.
          dmoObject.SQL_DMO.ApplicationName :=
            'SQL Script Builder';
          // Not connected yet.
          dmoObject.lConnected := False;
          // Login. (Uncomment if not loginSecure).
          // dmoObject.SQL_DMO.Login := 'sa';
          // Password.
          // dmoObject.SQL_DMO.Password := 'sa';
          // Add the object (not connected yet).
          oTreeView.Items.AddObject(oTreeView.Selected,
            oAliasList[lcv], dmoObject).StateIndex := 1;
        end;
    end;
  finally
    oAliasList.free;
  end;
end;
```

End Listing One

Begin Listing Two — The *Execute* method

```
function Tbo_SQL.Execute(oSourceMemo, oResultMemo: TMemo;
  oTreeView: TTreeView; lClearResultsFirst: Boolean;
  sDelimiter: string; lExecute: Boolean): Boolean;
var
  QueryResults : Variant;
  lcv, lcv2, lcv3 : Integer;
  s : string;
  db : Variant;
begin
  Result := False;
  // Empty script won't execute.
  if osourceMemo.text = '' then
    Exit;
  db := PdmoObject(oTreeView.Selected.data)^.SQL_DB;
  // Execute the SQL and return any results.
  if lExecute then
    begin
      // Execute the script and return any results.
      if oSourceMemo.SelLength > 0 then
        QueryResults := db.ExecuteWithResults(
          oSourceMemo.SelText, Length(oSourceMemo.SelText))
      else
        QueryResults := db.ExecuteWithResults(
          oSourceMemo.Text, Length(oSourceMemo.Text));
      // Spin through result sets, displaying data to user.
      if QueryResults.ResultSets > 0 then
        begin
          // Clear the results if the user wants.
          if lClearResultsFirst then
            oResultMemo.Lines.Clear;
          for lcv := 1 to QueryResults.ResultSets do begin
            // Select the result set.
            QueryResults.CurrentResultSet := lcv;
            for lcv2 := 1 to QueryResults.Rows do begin
              s := '';
              for lcv3 := 1 to QueryResults.Columns do
                s := s + TrimRight(
                  QueryResults.GetColumnString(
                  lcv2, lcv3)) + sDelimiter;
              oResultMemo.lines.Add(s);
            end;
          end;
          Result := True;  // Has results.
        end
      else
        begin
          ShowMessage('Successful execution.');
          Result := False; // Don't have results.
        end;
    end
  else
    begin
      // Only parse the SQL.
      if oSourceMemo.SelLength > 0 then
        db.ExecuteImmediate(oSourceMemo.SelText,
          SQLDMOExec_NoExec, Length(oSourceMemo.SelText))
      else
        db.ExecuteImmediate(oSourceMemo.Text,
          SQLDMOExec_NoExec, Length(oSourceMemo.Text));
      ShowMessage('Successful Parse.');
    end;
end;
```

End Listing Two

By Ken Revak

Augmenting a Control

Four Approaches to Extending Native VCL Objects

Developers often need to add properties or behaviors to a number of classes. Ideally this feature would be coded at the appropriate place in the VCL hierarchy, but this isn't possible due to the design of Delphi. This article investigates several techniques that can be used to add a feature to a number of controls.

One of my petty annoyances with Delphi is that there are several different properties used to access the caption, text, or value of a control, so the various techniques will be demonstrated by implementing a *CurrentText* feature for selected controls. Accessing *CurrentText* permits you to get and set the current textual value of a control. These techniques support information hiding by preventing the client code from dealing directly with the various *Text*, *Caption*, or *Cell* properties of individual controls. *CurrentText* will be added to Button, Edit, Memo, StringGrid, and ListBox controls.

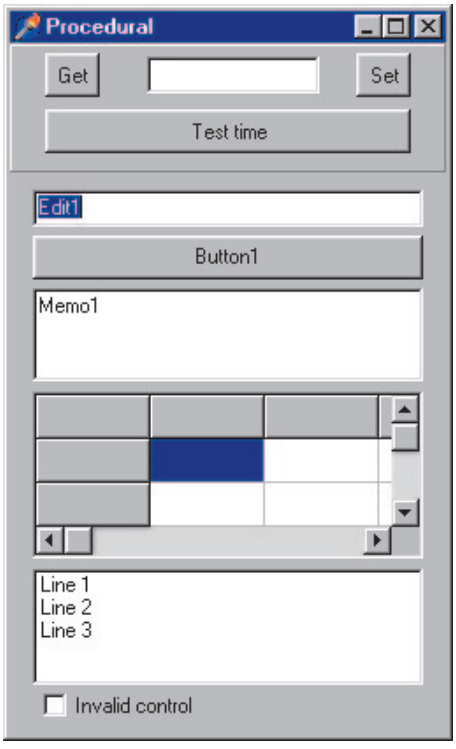


Figure 1: This demonstration form will be implemented four ways.

Demonstration Form

Figure 1 shows the form that will be re-implemented to demonstrate the four approaches to solving this problem. The **Get** button will retrieve the *CurrentText* value of the active control and place it in the edit box. The **Set** button will assign the text of the edit box to the *CurrentText* property of the active control. Note the use of SpeedButton components to avoid the focus shifting to the button when you click on it.

The *CurrentText* feature will be implemented on the Edit, Memo, and StringGrid components in the lower panel. The **Invalid control** check box doesn't implement the *CurrentText* feature; it's used to test the handling of that error condition. The **Test time** button will report on performance by timing how long it takes to retrieve the *CurrentText* value 10,000 times.

Each example will use *GetCurrentText* and *SetCurrentText* routines to perform housekeeping appropriate for that example. These routines accept the control as a *TComponent*, but end up calling routines defined at the lowest levels of the hierarchy.

The Procedural Approach

Listing One (on page 29) demonstrates the procedural approach to solving this problem. Within the *GetCurrentText* and *SetCurrentText* routines, a series of **if** statements accesses the property appropriate for that type of control. In this case, we're wrapping the control in access routines, rather than modifying the control. Note that, even in this non-object-oriented approach, we're using the Run-time Type Information (RTTI) capabilities of Delphi.

The advantages of this technique are that it can be grafted onto existing applications without

changing the controls already present in the UI, testing for a class also works for its descendants (e.g. testing for *TEdit* also handles *TDBEdit*), and it's easy to implement and debug. Its disadvantages are that it can cause the linker to drag in "dead" code if you don't use all the controls you're checking for, it permits accessing existing facilities, it doesn't permit you to add data storage, and it's not object-oriented.

The Message-based Approach

The VCL neatly encapsulates Windows message processing. This is done by a dispatch mechanism that accesses routines based on a numeric index. Normally this number represents a Windows message, but we can use our own numbers and corresponding routines. This dispatch mechanism is present in *TObject* and can be used by any class in the system. Listing Two (beginning on page 29) demonstrates using this message-processing mechanism.

First we define our message numbers based on WM_USER and a record structure whose first field is a two-byte integer (Cardinal) that contains the message number to be called. I prefer to emulate the *TMessage* structure, but it isn't necessary to do so; you may add as many fields as you need. In this case, *Value* will contain the string value to be assigned or retrieved, and the predefined *Result* variable is used to indicate if the operation has been performed.

The *GetCurrentText* and *SetCurrentText* routines initialize the message structure, call the dispatch routine for the component, and then interpret and act on the result returned. Each control implements routines to respond to the messages, and fills in the appropriate fields in the message record.

These routines don't know or care what type of control they're dealing with. The key point is that we're effectively using the dispatch mechanism at the *TObject* level to call a procedure defined at the individual control level. The *GetCurrentText* and *SetCurrentText* procedures don't know anything about the implementation of the individual components.

With this technique, you can pass and return complex information, but you must implement the message routines in each control and revise your UI to use these components. This technique can be used for any class. However, you must be careful to prevent message number collisions, i.e. ensure that message numbers are unique.

The Introspection Approach

Listing Three (beginning on page 30) demonstrates the introspection approach. Here we leverage Delphi's property-publishing capabilities to access the *CurrentText* feature.

Each component implements a standard get/set property method, and defines the property in the **published** section of the component. *GetCurrentText* and *SetCurrentText* use routines and structures from the TypInfo unit to find the property, and then access the appropriate routine.

This technique is generally limited to simple types, because it can quickly become complex when trying to pass/return complex information. It's also based on the TypInfo unit, which may change from release to release and is limited to *TPersistent* descendants.

The Interface Approach

Listing Four (beginning on page 31) accomplishes the *CurrentText* feature by using interfaces. Interfaces were introduced by Delphi 4 to support COM, and provide an alternative to multiple inheritance. Essentially, an interface is a list of routines that a class agrees to implement. Delphi provides the facilities to define and access the routines defined in an interface.

Approach	Total access time
Procedural	230
Message	240
Introspection	251
Interface	241

Figure 2: Performance comparisons of the four approaches (in milliseconds).

This implementation begins by defining an interface containing the routines to be implemented. The long hex number:

```
['{6298A451-1D1E-11D3-937B-0 0 80 C8E717EA}']
```

is inserted using Ctrl Shift G and is called a globally unique identifier (GUID). This identifies this interface to the universe and is guaranteed to be unique. Because we're not using this GUID outside the application, we don't need to concern ourselves with entering it into the registry.

Each control implements the interface by including it in its definition and implementing the specified routines. *GetCurrentText* and *SetCurrentText* separately query the system to obtain the interface for the passed component, and then use it to call the appropriate routine.

This technique is quite flexible regarding parameters and types, and will please object purists. It also provides a convenient mechanism to group similar functionality, and ensures you have implemented the interfaces you have specified at compile time.

Performance

Figure 2 contains the performance comparisons as performed on my machine. To my surprise, the results for the various approaches are similar, and — of themselves — don't provide a compelling reason to select one over the other.

Conclusion

The richness of the Delphi environment provides many ways to solve programming problems. Here, we've investigated and compared various ways of adding a feature to several controls.

The interface approach provides the best overall approach due to its support for rich data types, compile-time checking, and smooth implementation. The procedural approach can be useful when you have an extensive existing code base and requirements for a limited feature set. Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JUNE\DI200006KR.

Ken Revak is a principal of Catalyst Systems Ltd located in southern Alberta, Canada, where he does custom programming and develops applications using Delphi. He enjoys leveraging the features of Delphi while sneering at Visual Basic. Ken can be reached at catalyst@telusplanet.net.

Begin Listing One — Procedural approach

```
unit Listing1;
{ Procedural approach to CurrentText feature. }

interface

uses
  Classes, StdCtrls, Grids, SysUtils, Controls;

function GetCurrentText(Component: TComponent): string;
procedure SetCurrentText(Component: TComponent;
  const Value: string);

type
  ECurrentText = class(Exception);

implementation

function GetCurrentText(Component: TComponent): string;
var
  lb : TListBox;
  sg : TStringGrid;
begin
  if Component is TEdit then
    // Use the Text property for TEdits.
    Result := TEdit(Component).Text
  else if Component is TMemo then
    // Use the selected Text for a memo.
    Result := TMemo(Component).SelText
  else if Component is TButton then
    // Use the Caption for a button.
    Result := TButton(Component).Caption
  else if Component is TListBox then
    // Use the currently selected item.
    begin
      lb := TListBox(Component);
      if lb.ItemIndex = -1 then
        Result := ''
      else
        Result := lb.Items[lb.ItemIndex];
    end
  else if Component is TStringGrid then
    // Use the currently selected cell.
    begin
      sg := TStringGrid(Component);
      Result := sg.Cells[sg.Col, sg.Row];
    end
  else
    // Raise an error if component is unknown.
    raise ECurrentText.Create(Component.Classname +
      ' is not a type known to GetCurrentText');
end;

procedure SetCurrentText(Component: TComponent;
  const Value: string);
var
  lb : TListBox;
  sg : TStringGrid;
begin
  if Component is TEdit then
    TEdit(Component).Text := Value
  else if Component is TMemo then
    TMemo(Component).SelText := Value
  else if Component is TButton then
    TButton(Component).Caption := Value
  else if Component is TListBox then
    begin
      lb := TListBox(Component);
      if lb.ItemIndex = -1 then
        lb.Items.Add(Value)
      else
        lb.Items[lb.ItemIndex] := Value;
    end
  else if Component is TStringGrid then
    begin
      sg := TStringGrid(Component);
      sg.Cells[sg.Col, sg.Row] := Value;
    end
  else
    raise ECurrentText.Create(Component.Classname +
      ' is not a type known to SetCurrentText');
end;

end.
```

End Listing One

Begin Listing Two — Message-based approach

```
unit Listing2;
{ Message-based approach to CurrentText feature. }

interface

uses
  Classes, SysUtils, Messages, StdCtrls, Grids;

const
  MSG_GET_CURRENT_TEXT = WM_USER + 10 0 0 ;
  MSG_SET_CURRENT_TEXT = WM_USER + 10 0 1;

type
  ECurrentText = class(Exception);
  TCurrentTextRecord = record
    Msg    : Cardinal;
    Value  : string;
    LParam : LongInt;
    Result : LongInt;
  end;

  TmsgEdit = class(TEdit)
  protected
    procedure MsgGetCurrentText(
      var CurrentTextRecord: TCurrentTextRecord);
      message MSG_GET_CURRENT_TEXT;
    procedure MsgSetCurrentText(
      var CurrentTextRecord: TCurrentTextRecord);
      message MSG_SET_CURRENT_TEXT;
  end;

  TmsgButton = class(TButton)
  protected
    procedure MsgGetCurrentText(
      var CurrentTextRecord: TCurrentTextRecord);
      message MSG_GET_CURRENT_TEXT;
    procedure MsgSetCurrentText(
      var CurrentTextRecord: TCurrentTextRecord);
      message MSG_SET_CURRENT_TEXT;
  end;

  TmsgMemo = class(TMemo)
  protected
    procedure MsgGetCurrentText(
      var CurrentTextRecord: TCurrentTextRecord);
      message MSG_GET_CURRENT_TEXT;
    procedure MsgSetCurrentText(
      var CurrentTextRecord: TCurrentTextRecord);
      message MSG_SET_CURRENT_TEXT;
  end;

  TmsgStringGrid = class(TStringGrid)
  protected
    procedure MsgGetCurrentText(
      var CurrentTextRecord: TCurrentTextRecord);
      message MSG_GET_CURRENT_TEXT;
    procedure MsgSetCurrentText(
      var CurrentTextRecord: TCurrentTextRecord);
      message MSG_SET_CURRENT_TEXT;
  end;

  TmsgListBox = class(TListBox)
  protected
```

```
    procedure MsgGetCurrentText(
      var CurrentTextRecord: TCurrentTextRecord);
      message MSG_GET_CURRENT_TEXT;
    procedure MsgSetCurrentText(
      var CurrentTextRecord: TCurrentTextRecord);
      message MSG_SET_CURRENT_TEXT;
  end;

  function GetCurrentText(Component: TComponent): string;
  procedure SetCurrentText(Component: TComponent;
    const Value: string);
  procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Message', [TmsgEdit, TmsgButton,
    TmsgMemo, TmsgStringGrid, TmsgListBox]);
end;

function GetCurrentText(Component: TComponent): string;
var
  msg : TCurrentTextRecord;
begin
  FillChar(msg, SizeOf(msg), 0 );
  msg.Msg := MSG_GET_CURRENT_TEXT;
  Component.Dispatch(msg);
  if msg.Result <> 0 then
    Result := msg.Value
  else
    raise ECurrentText.Create(
      'CurrentText feature not found in ' +
      Component.Classname);
end;

procedure SetCurrentText(Component: TComponent;
  const Value: string);
var
  msg : TCurrentTextRecord;
begin
  FillChar(msg, SizeOf(msg), 0 );
  msg.Msg := MSG_SET_CURRENT_TEXT;
  msg.Value := Value;
  Component.Dispatch(msg);
  if msg.Result = 0 then
    raise ECurrentText.Create(
      'CurrentText feature not found in ' +
      Component.Classname);
end;

procedure TmsgEdit.MsgGetCurrentText(
  var CurrentTextRecord: TCurrentTextRecord);
begin
  CurrentTextRecord.Value := Text;
  CurrentTextRecord.Result := 1;
end;

procedure TmsgEdit.MsgSetCurrentText(
  var CurrentTextRecord: TCurrentTextRecord);
begin
  Text := CurrentTextRecord.Value;
  CurrentTextRecord.Result := 1;
end;

procedure TmsgButton.MsgGetCurrentText(
  var CurrentTextRecord: TCurrentTextRecord);
begin
  CurrentTextRecord.Value := Caption;
  CurrentTextRecord.Result := 1;
end;

procedure TmsgButton.MsgSetCurrentText(
  var CurrentTextRecord: TCurrentTextRecord);
begin
  Caption := CurrentTextRecord.Value;
  CurrentTextRecord.Result := 1;
end;

procedure TmsgMemo.MsgGetCurrentText(
  var CurrentTextRecord: TCurrentTextRecord);
begin
  CurrentTextRecord.Value := SelText;
  CurrentTextRecord.Result := 1;
end;

procedure TmsgMemo.MsgSetCurrentText(
  var CurrentTextRecord: TCurrentTextRecord);
begin
  SelText := CurrentTextRecord.Value;
  CurrentTextRecord.Result := 1;
end;

procedure TmsgStringGrid.MsgGetCurrentText(
  var CurrentTextRecord: TCurrentTextRecord);
begin
  CurrentTextRecord.Value := Cells[Col,Row];
  CurrentTextRecord.Result := 1;
end;

procedure TmsgStringGrid.MsgSetCurrentText(
  var CurrentTextRecord: TCurrentTextRecord);
begin
  Cells[Col,Row] := CurrentTextRecord.Value;
  CurrentTextRecord.Result := 1;
end;

procedure TmsgListBox.MsgGetCurrentText(
  var CurrentTextRecord: TCurrentTextRecord);
begin
  if ItemIndex = -1 then
    CurrentTextRecord.Value := ''
  else
    CurrentTextRecord.Value := Items[ItemIndex];
  CurrentTextRecord.Result := 1;
end;
procedure TmsgListBox.MsgSetCurrentText(
  var CurrentTextRecord: TCurrentTextRecord);
begin
  if ItemIndex = -1 then
    Items.Add(CurrentTextRecord.Value)
  else
    Items[ItemIndex] := CurrentTextRecord.Value;
  CurrentTextRecord.Result := 1;
end;

end.
```

End Listing Two

Begin Listing Three — Introspection approach

```
unit Listing3;
{ Introspection approach to CurrentText feature. }

interface

uses
  Classes, StdCtrls, Grids, SysUtils;

type
  TintEdit = class(TEdit)
  private
    procedure SetCurrentText(const Value: string);
    function  GetCurrentText: string;
  published
    property CurrentText: string read GetCurrentText
      write SetCurrentText stored False;
  end;

  TintMemo = class(TMemo)
  private
    procedure SetCurrentText(const Value: string);
```

```
    function  GetCurrentText: string;
  published
    property CurrentText: string read GetCurrentText
      write SetCurrentText stored False;
  end;

  TintButton = class(TButton)
  private
    procedure SetCurrentText(const Value: string);
    function  GetCurrentText: string;
  published
    property CurrentText: string read GetCurrentText
      write SetCurrentText stored False;
  end;

  TintListBox = class(TListBox)
  private
    procedure SetCurrentText(const Value: string);
    function  GetCurrentText: string;
  published
    property CurrentText: string read GetCurrentText
      write SetCurrentText stored False;
  end;

  TintStringGrid = class(TStringGrid)
  private
    procedure SetCurrentText(const Value: string);
    function  GetCurrentText: string;
  published
    property CurrentText: string read GetCurrentText
      write SetCurrentText stored False;
  end;
  ECurrentText = class(Exception);

  function  GetCurrentText(Component: TComponent): string;
  procedure SetCurrentText(Component: TComponent;
    const Value: string);
  procedure Register;

implementation

uses
  TypInfo;

procedure Register;
begin
  RegisterComponents('Introspection', [TintEdit, TintMemo,
    TintButton, TintListBox, TintStringGrid]);
end;

function GetCurrentText(Component: TComponent): string;
var
  ptrPropInfo : PPropInfo;
begin
  Result := '';
  ptrPropInfo :=
    GetPropInfo(Component.ClassInfo, 'CurrentText');
  if ptrPropInfo = nil then
    raise ECurrentText.Create(
      'CurrentText property not found in Object ' +
      Component.Classname)
  else if ptrPropInfo^.PropType^.Kind in [tkString,
                                          tkLString] then
    Result := GetStrProp(Component, ptrPropInfo)
  else
    ECurrentText.Create(
      'CurrentText property is not a string');
end;

procedure SetCurrentText(Component: TComponent;
  const Value: string);
var
  ptrPropInfo : PPropInfo;
begin
  ptrPropInfo :=
    GetPropInfo(Component.ClassInfo, 'CurrentText');
  if ptrPropInfo = nil then
    raise ECurrentText.Create(
      'CurrentText property not found in Object ' +
      Component.Classname)
  else if ptrPropInfo^.PropType^.Kind in [tkString,
                                          tkLString] then
    SetStrProp(Component, ptrPropInfo, Value)
  else
    ECurrentText.Create(
      'CurrentText property is not a string');
end;

procedure TintEdit.SetCurrentText(const Value: string);
begin
  Text := Value;
end;

function  TintEdit.GetCurrentText: string;
begin
  Result := Text;
end;

procedure TintMemo.SetCurrentText(const Value: string);
begin
  SelText := Value;
end;

function  TintMemo.GetCurrentText: string;
begin
  Result := SelText;
end;

procedure TintButton.SetCurrentText(const Value: string);
begin
  Text := Value;
end;

function  TintButton.GetCurrentText: string;
begin
  Result := Text;
end;

procedure TintListBox.SetCurrentText(const Value: string);
begin
  if ItemIndex = -1 then
    Items.Add(Value)
  else
    Items[ItemIndex] := Value;
end;

function  TintListBox.GetCurrentText: string;
begin
  if ItemIndex = -1 then
    Result := ''
  else
    Result := Items[ItemIndex];
end;

procedure TintStringGrid.SetCurrentText(
  const Value: string);
begin
  Cells[Col,Row] := Value;
end;

function  TintStringGrid.GetCurrentText: string;
begin
  Result := Cells[Col,Row] ;
end;

end.
```

End Listing Three

Begin Listing Four — Interface approach

```
unit Listing4;
{ Interface approach to CurrentText feature }
```

```
interface

uses
  Classes, StdCtrls, Grids, SysUtils;

type
  ICurrentText = interface
    ['{ 6298A451-1D1E-11D3-937B-0 0 80 C8E717EA }']
    procedure SetCurrentText(const Value: string);
    function  GetCurrentText: string;
  end;
  type ECurrentText = class(Exception);

  TinfEdit = class(TEdit, ICurrentText)
  protected
    procedure SetCurrentText(const Value: string);
    function  GetCurrentText: string;
  end;

  TinfMemo = class(TMemo, ICurrentText)
  protected
    procedure SetCurrentText(const Value: string);
    function  GetCurrentText: string;
  end;

  TinfButton = class(TButton, ICurrentText)
  protected
    procedure SetCurrentText(const Value: string);
    function  GetCurrentText: string;
  end;

  TinfListBox = class(TListBox, ICurrentText)
  protected
    procedure SetCurrentText(const Value: string);
    function  GetCurrentText: string;
  end;

  TinfStringGrid = class(TStringGrid, ICurrentText)
  protected
    procedure SetCurrentText(const Value: string);
    function  GetCurrentText: string;
  end;

  function  GetCurrentText(Component: TComponent): string;
  procedure SetCurrentText(Component: TComponent;
    const Value: string);
  procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Interface', [TinfEdit, TinfMemo,
    TinfButton, TinfListBox, TinfStringGrid]);
end;

function GetCurrentText(Component: TComponent): string;
var
  cti : ICurrentText;
begin
  Component.GetInterface(ICurrentText, cti);
  if Assigned(cti) then
    Result := cti.GetCurrentText
  else
    raise ECurrentText.Create(
      'ICurrentText not supported by this object');
end;

procedure SetCurrentText(Component: TComponent;
  const Value: string);
var
  cti : ICurrentText;
begin
  Component.GetInterface(ICurrentText, cti);
  if Assigned(cti) then
    cti.SetCurrentText(Value)
  else
    raise ECurrentText.Create(
      'ICurrentText not supported by this object');
end;

procedure TinfEdit.SetCurrentText(const Value: string);
begin
  Text := Value;
end;

function  TinfEdit.GetCurrentText: string;
begin
  Result := Text;
end;

procedure TinfMemo.SetCurrentText(const Value: string);
begin
  SelText := Value;
end;

function  TinfMemo.GetCurrentText: string;
begin
  Result := SelText;
end;

procedure TinfButton.SetCurrentText(const Value: string);
begin
  Text := Value;
end;

function  TinfButton.GetCurrentText: string;
begin
  Result := Text;
end;

procedure TinfListBox.SetCurrentText(const Value: string);
begin
  if ItemIndex = -1 then
    Items.Add(Value)
  else
    Items[ItemIndex] := Value;
end;

function  TinfListBox.GetCurrentText: string;
begin
  if ItemIndex = -1 then
    Result := ''
  else
    Result := Items[ItemIndex];
end;

procedure TinfStringGrid.SetCurrentText(
  const Value: string);
begin
  Cells[Col,Row] := Value;
end;

function  TinfStringGrid.GetCurrentText: string;
begin
  Result := Cells[Col,Row] ;
end;

end.
```

End Listing Four

Delphi COM Programming

I have a tendency to read two books simultaneously. One is work-related (usually containing words such as COM, Distributed, Professional, etc. in the title); the other is just for fun: a mystery-, science-fiction-, travel-, or hobby-related book. I even dabble in reading literature (the shock, the horror). The fun book I was multitasking with Eric Harmon's *Delphi COM Programming* was Mark Twain's travel book, *The Innocents Abroad* (or *The New Pilgrims' Progress*).

If you are wondering what *The Innocents* has to do with a Delphi COM programming book and wonder if I am out of my mind trying to compare Mr Clemens' finest with Mr Harmon's creation, my little ploy has worked. Instead of providing a dry analysis of a technical book that would probably bore you, I wanted to throw a little something else in here to keep you interested in reading this review.

So, you ask: "What is the connection your brain managed to find between the books?" Both books describe a voyage of the "new world" people (1867 US of Twain's time and Delphi programmers today) into the "old" world and its culture and heritage (Europe and the Holy Land and COM programming in a Microsoft environment).

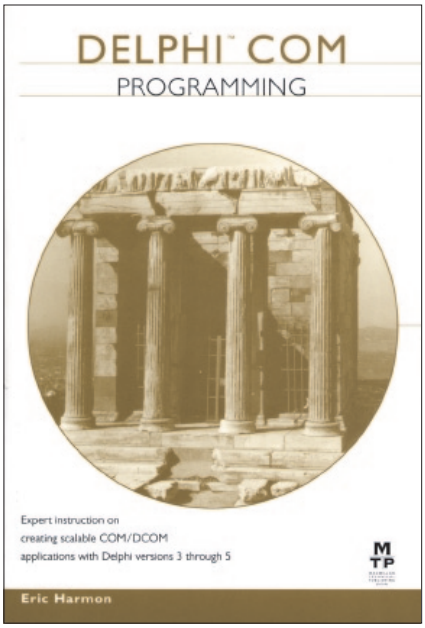
As Delphi programmers, we enjoy the ease of using the VCL, Delphi's two-way tools, and the Object Pascal language, which protect us from many hassles of programming. COM, on the other hand, is one of these overly complex Microsoft technologies that are either hidden behind the scenes in Visual Basic, or exposed with all its gory details in C++ sample code on MSDN.

However, Delphi does for COM what it did for WinAPI programming: It provides the functionality in a developer-friendly manner. *Delphi COM Programming* is an excellent book that provides the detailed COM information you'll find in the MSDN documentation presented in such a way that any Delphi programmer will be comfortable.

Delphi COM Programming starts with a chapter describing interfaces as a language option. The beginner COM programmer will be introduced to interfaces, the difference between interfaces and abstract classes, and the way interfaces are implemented and used in Delphi classes. This chapter also introduces the mother of all interfaces — *IUnknown* — and how to implement it. The chapter also describes advanced interface issues, including interface delegation.

I was surprised that, even as an experienced COM developer, I learned something new in Chapter 1: the existence of the Delphi *TAggregatedObject* class that helps with interface delegation.

Chapter 2 discusses simple COM objects and provides information about in-process vs. out-of-process COM objects, COM threading models, server registration, and an issue that often appears on the Delphi



Internet newsgroups — creating and accessing variant arrays.

Type libraries — the cross-language binary "meta-data" about classes, interfaces, enumeration, and other COM entities — are described in detail in Chapter 3. In addition to a tutorial on how to use the Delphi Type Library editor, the author provides an excellent example that uses the *ITypeInfo* interface to interpret the contents of a type library. This subject has always interested me, but I never bothered to learn more, because I had no immediate need for its use. It's nice to know that I now have a well-documented sample at my disposal, so if the need arises, I won't need to sift through MSDN documentation to find it.

Chapter 4 is aptly described by the author as, "probably the most important chapter in the book." Especially if you're a novice COM programmer, I'll have to agree with the author. But if you're an experienced COM developer, I think Harmon sells some of the other chapters short. Automation is presented in this chapter. The topics covered include automation objects, dispinterfaces, dual interfaces, and the performance differences between them.

Chapter 4 also includes the best discussion of COM events and callbacks programming in Delphi I've seen, and introduces an example of Microsoft's ActiveX Data Objects (ADO).

ActiveX controls are described in the next chapter. The Microsoft Agent ActiveX control set is used as an example of hosting an ActiveX control in a Delphi application. The VCL to the ActiveX Control wizard is described, as are simple ActiveX controls created using Active Forms.

Unfortunately, this chapter doesn't offer the same in-depth information found in the previous sections. Important issues, such as the way ActiveX controls work behind the scenes, a discussion of the important ActiveX control hosting interfaces, and some of the problems of the VCL-to-ActiveX-Control wizard (e.g. no support for *IPropertyBag* that makes these controls unusable in MSIE) are omitted.

Of all the chapters in the book, Chapter 5 is the one that is only of interest to a complete novice, and it may leave an experienced developer looking elsewhere for additional information.

Using COM servers and clients in a distributed environment is presented in Chapter 6. The text and code examples provide a walk through of installing a DCOM Server application and configuring it, which is the most difficult part in DCOM application deployment.

The accompanying sample demonstrates how a server application can be used to provide access to database information where the client machines don't have access to the BDE. The sample uses nothing more than simple COM code to perform the task without taking advantage of a middleware product like MIDAS. The sample uses variant arrays — which are described earlier in the book — to marshal record information across the network.

The next couple of chapters in *Delphi COM Programming* are the most interesting, as they discuss a topic with which I wasn't at all familiar: structured storage and some of its uses in OLE.

Structured storage is a file system in a file that allows you to store diverse object data in multiple streams (think of them as files), using (optionally) a directory like the hierarchy of storage bins (folders) in one physical file on the hard disk. COM is an object-enabling technology, and applications that need to persist object data are limited to streaming all the information to/from a file or using multiple files. With structured storage, your application can still take advantage of random access via the hierarchical structure of your needs, with the benefit for the end user being one physical file to manage.

In these chapters, readers are introduced to the structured storage utility methods, the *IStorage* and *IStream* interfaces, and some of the applications of these interfaces, such as property sets. The *TOleContainer* component is the last thing discussed in the chapter, and issues, such as menu merging, Clipboard support, and object insertion/removal, are discussed.

The concluding chapter in *Delphi COM Programming* discusses the Windows shell extensions. The topics covered include shell context menu handlers, copy hooks, links, tray icons, and property sheets.

The old-style API that Windows is built on is slowly fading as Microsoft is moving to object-based APIs implemented as COM interfaces and objects. Though Delphi does a wonderful job of wrapping COM functionality and shielding us from the low-level details of COM programming, sooner or later you'll find yourself in need of interfacing with a new technology that will most likely be presented as an object model, COM interfaces, etc.

If you haven't started learning about COM, Harmon's *Delphi COM Programming* will be a great addition to your library. It provides clear definitions of the basics and will get you started in a hurry.

If you're familiar with COM, but have never spent any time learning the details, *Delphi COM Programming* will be a good way to become familiar with many of the foundational building blocks, as well as many of the more advanced topics. You'll get a much better understanding of COM and the way Delphi helps you work with it.

If you're an experienced COM developer, you could still find topics you're probably not familiar with that will be worth the price of *Delphi COM Programming*. This was definitely the case with me.

All that said, *Delphi COM Programming* still has room for improvement. The ActiveX chapter isn't as in-depth as I would have liked it to be, and I feel that a chapter discussing MTS and COM+ would have been a great addition, because they show the way COM is evolving.

Despite these "shortcomings," I wouldn't hesitate to recommend this book to any Delphi developer interested in an introduction to COM development.

Going back to *The Innocents Abroad*, Twain doesn't discuss COM+, nor MTS for that matter. But if you want a description of every church in Italy during the late 19th century, it's definitely the book for you.

— *Ron Loewy*

Delphi COM Programming by Eric Harmon, Macmillan Technical Publishing, 201 West 103rd St., Indianapolis, IN 46290, http://www.newriders.com.

ISBN: 1-57870-221-6
Price: US$45 (510 pages)

Coding Styles of the Cool and Famous

I never watch the TV show "Lifestyles of the Rich and Famous." Who has time for that claptrap? I prefer to examine other people's code to learn tricks (after all, imitation *is* the sincerest form of flattery). I don't spend a lot of time poring through just anybody's code, however; I'm very selective. I don't want to be led down the "garden path" by some cowboy, hacker, or wannabe. The code I'm studying must flow from the brain of one of the best. An expert!

During forays into code created by my clever colleagues, I have often thrust my fist into the air, yelling "Eureka!" or "That's the ticket!" There's nothing like finding a block of code you can file away for later reuse. There are giants who walk among us, the stars of the Delphi community — those who have a natural affinity for coding and are bright, creative, and able to solve the thorniest of problems with ease (or so it appears).

Besides elegant algorithms, there's something else I enjoy noting as I probe the work of these ingenious coders: the style. Coding is an art form. Artists, once well versed in their field, develop an identifiable style. You can listen to a composition by Bruce Springsteen or J.S. Bach and, if you're familiar with their work, recognize it immediately. The same goes with paintings by Van Gogh, Dali, Picasso, etc. Is it not to be expected that style would be identifiable in our profession? Lest you think I jest, here's some code to illustrate my point:

```
procedure TForm1.Button1Click(Sender: TObject);
VAR B1, B2, B3 : ARRAY[0 ..20 ]OF Char;
begin
  FormPtr := CreateTheForm;
  StrPCopy(B1, Edit1.Text);
  StrPCopy(B2, Edit2.Text);
  StrPCopy(B3, Edit3.Text);
  LoadTheForm(FormPtr, B1, B2, B3);
  IF ShowTheForm(FormPtr) THEN
    BEGIN
      ReadTheForm(FormPtr, B1, B2, B3);
      Edit1.Text := StrPas(B1);
      Edit2.Text := StrPas(B2);
      Edit3.Text := StrPas(B3);
    END;
  DestroyTheForm(FormPtr);
end;
```

Can you guess who wrote this? It's the handiwork of Neil Rubenking, author of *Delphi Programming Problem Solver* [IDG Books Worldwide, 1996] and *Delphi 3 for Dummies* [IDG Books Worldwide, 1997], and columnist for *PC Magazine*. His style is recognizable because of the uppercased keywords. He does this so he can easily see which code he wrote and which was generated by Delphi. Here's another example:

```
Application.OnException := FOldExceptionHandler;

if ( E is EDBEngineError ) and
   ( EDBEngineError( E ).
       Errors[ 0  ].ErrorCode = DBIERR_KEYVIOL ) then
  begin
    { Handle event processing on key violation. }
    if Assigned( FOnKeyViolation ) then
      FOnKeyViolation( TableName )
    else
      MessageDlg( 'Key Violation Error on Table ' +
        TableName, mtError, [ mbOK ], 0  );
  end
  else
    Application.ShowException(E);
```

If you've read the book *Developing Custom Delphi 3 Components* [Coriolis Group Books, 1997] and/or the column "Delphi by Design" in the now-defunct *Visual Developer* magazine, you should recognize this code as belonging to Ray Konopka. He also happens to be the creator of Raize Components, the Delphi debugging tool CodeSight, and other goodies for Delphi developers. The spaces used in the parentheses and brackets give him away.

One final example:

```
...
    for i := Pred( Count ) downto 0 do begin
      if Items[i].Action = aaDelete then
        FItemList.Delete( i )
      else if Items[i].Action <> aaFailed then
        Items[i].Action := aaNone;
    end;

    DoArchiveProgress( 10 0 , Abort );
  finally {NewStream}
    NewStream.Free;
  end;
end;
{ ------------------------------------------------------ }
procedure TAZipArchive.SetZipFileComment( Value : string );
...
```

Whose code do you think this is? It's none other's than TurboPower! OK, that was a trick question, as TurboPower is a company, not a person. Sometimes an unmistakable style permeates an entire company. They also pad their method arguments with spaces fore and aft, but not so in the array elements, setting themselves apart from Konopka. I've never seen anybody else set off procedures from one another in exactly this fashion, nor is it common in Pascal/Delphi to place the **begin** keyword at the end of a line rather than on a line by itself. And who else uses the *Pred* function, for that matter!

Actually, I do, because I "stole" all three of these peculiarities from them. Developing your own style consists partly in "stealing" not only algorithms, but also various stylistic elements used in the actual presentation of code. Be selective and learn from the best. Δ

— Clay Shannon

Clay Shannon is a Delphi developer for eMake Corp., located in Post Falls, ID. Having visited 49 states (all but Hawaii) and lived in seven, he and his family have settled in northern Idaho, near beautiful Coeur d' Alene Lake. He has been working (almost) exclusively with Delphi since its release, and is the author of the book *Developer's Guide to Delphi Troubleshooting* [Wordware, 1999]. You can reach Clay at clayshannon@usa.net.

Delphi and Linux: Internet Resources

For the past two months, we've been discussing the implications surrounding Borland's decision to develop a Linux version of Delphi. Part of the Kylix Project that involves other Borland tools besides Delphi, this initiative has caused a good deal of excitement and apprehension among Borland's developer base. Since I submitted last month's column, there have been some interesting developments, which have caused further excitement and apprehension. However, our main focus here is to discuss some useful Linux Internet sites. Along the way, we'll touch on some of these new developments and list sites where you can find additional information.

General Linux sites. If you're looking for a version of Linux that installs under Windows 95/98, check out the Armed Linux site at http://www.armed.net. You can download a free copy from this site. You can also investigate the links to other Linux sites, including Linux resources, message boards/forums, and more. If you're new to Linux (as I suspect many readers of this magazine are), you should check out The Linux Knowledge Base Project, a relatively new site located at http://www.linuxkb.org. This well-organized site covers the following essential topics: Applications, Console & Shells, Editors, E-Mail, Emulators, General Linux Information, Hardware, Programming, System Administration, and X Windows. I was disappointed, however, in the low number of entries that resulted from some of my searches. Hopefully this site will continue to grow.

Of course, Borland has its own special page devoted to Linux: the Linux Borland Community at http://community.borland.com/linux. This site includes a wealth of news, articles, white papers, and other applicable information. ComputerWeekly has an entire section devoted to Linux at http://www.computerweekly.co.uk/cw_news/cw_linux_news.asp. It also has pages devoted to Microsoft Windows 2000, jobs, salaries, and other topics of interest to developers.

A programming visionary. I've always been a fan of Jeff Duntemann and still keep one of his assembly books on my active bookshelf. It was he who first got me interested in Linux with an article in his now-defunct magazine, *Visual Developer*. So when *Delphi Informant Magazine* Editor-in-Chief Jerry Coffey told me about Jeff's online diary at http://www.visual-developer.com/diary.cfm, I went there immediately and found a wealth of information about Linux, Delphi, and many other topics. Even if you have no interest in Linux, I recommend you visit the site and treat yourself to his humor and insights.

One site Duntemann mentions is the Free Pascal site at http://gd.tuwien.ac.at/languages/pascal/fpc/www. He discusses some of his experiences working with the DOS version of FreePascal 32. His assessment is very positive, calling it "an incredible piece of work ... with a WinSock library, clones of virtually all BP7 standard units (including Turbo Vision), and the beginnings of a Delphi-style component architecture." I am tempted to follow his example and run some of my ancient BP7 programs with it.

Free Pascal describes itself with these words on its site: "The language syntax is semantically compatible with TP 7.0 (Borland or Turbo Pascal version 7.0); some extensions used by Delphi (classes, RTTI, exceptions, ANSI strings) are also supported. Furthermore, Free Pascal supports function overloading and other such features." This site is definitely worth checking out, even if you have little interest in Linux at this point.

The Lazarus Project is an offshoot of Free Pascal. They describe themselves on their Web Site (http://www.lazarus.freepascal.org) as follows: "Lazarus is the class libraries for Free Pascal that emulate Delphi. Free Pascal is a GPLed compiler that runs on Linux, Win32, OS/2, 68K, and more. Free Pascal is designed to be able to understand and compile Delphi syntax, which is, of course, OOP. Lazarus is the part of the missing puzzle that will allow you to develop Delphi-like programs in all of the above platforms. Unlike Java, which strives to be a write once run anywhere, Lazarus and Free Pascal strives for write once compile anywhere. Since the exact same compiler is available on all of the above platforms it means you don't need to do any recoding to produce identical products for different platforms." Sounds pretty cool, don't you think? But let's not forget Borland. There have been some interesting developments there as well.

The merger. In the middle of writing this series of articles, a major bombshell hit our corner of the computing world: the proposed merger of Inprise and Corel. I trust most of you have read the announcement on Borland's site. Have you looked into Corel's slant on the merger? In an interview at http://www.upside.com/texis/mvm/open_season?id=389f63bb0, Derek Burney, executive vice president of engineering and CTO of Corel Corp., spoke of his company's well-known commitment to putting sophisticated Linux applications on the desktop and the desire to go further with Linux development tools. This is where Inprise comes into the picture. Among other things, he stated "We needed strength on the server side, and Borland offers that with middleware, developer tools, Java. It was perfect. That's why this came up so fast. Imagine Word Perfect Office 2000 using Delphi as a scripting language. It could truly be a back office. Now we throw in JBuilder and JavaBeans and you've got industrial-strength Web development on top of that. Just skimming the surface, we've come up with dozens and dozens of ways where the technology can be used on both sides."

There are a number of Delphi sites that have been following these developments. One of them, Richy's Delphi-Box, includes the Corel

link and several others on the following page devoted to the merger: http://inner-smile.com/delphin.htm#corinpr.

In a season of mergers, the Inprise/Corel merger is not the only new arrangement involving the makers of Delphi. A partnership between Inprise/Borland and TurboLinux has also been announced (see http://biz.yahoo.com/prnews/000216/ca_inprise_1.html). The name itself should suggest some affinity for those who remember Turbo Pascal, the great ancestor of Delphi. But consider also the TurboLinux slogan: "Integrating Linux into the Enterprise." There's a familiar ring to it. For more information on this new partner, check out their home page at http://www.turbolinux.com.

For more information on Corel's Linux strategy, check out the Corel site at http://linux.corel.com/webcast/merger/corel_linux.htm. Like Borland, this company is also pursuing its own new arrangements. It recently announced that its LINUX OS would be bundled with hardware from http://www.TheLinuxStore.com, including their Personal Internet Appliance desktop and workstation computers. You can find additional information on this arrangement at The Andover News Network (http://www.andovernews.com/cgi-bin/news_story.pl?143286,topstories).

Late-breaking news. I just learned of an excellent "Kylix Study Guide" by John Kaster on the Borland site at http://community.borland.com/article/0,1410,21122,00.html. I strongly recommend visiting it, as it's geared specifically at those of us who are getting ready to enter the world of Linux from that of Delphi.

I would be remiss if I didn't mention that these developments — Borland's Linux initiative and its merger with Corel — haven't met with universal enthusiasm and support from the developer base. Some developers worry that emphasis on Linux will undermine support for Windows development tools like Delphi. Others wonder if Corel is the best of all possible merger partners, suggesting that Sun Microsystems might have been a better choice. I remain cautiously optimistic. First, I expect the development of Linux tools to be slow and incremental, having little initial effect on Borland's support for Delphi and C++Builder. Second, I see a good deal of historical and mutually beneficial logic behind the merger. For example, Corel is still using the BDE in some of its applications. However, only time will tell. Until next time. Δ

— Alan C. Moore, Ph.D.

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

table of CONTENTS