March 1996 - Volume 2, Number 3

Cover Art By: *Tom McKeith*

Dynamic Delphi

Building and Using DLLs

ON THE COVER

FEATURES

REVIEW

DEPARTMENTS

Catching up with the Hype

We have heard *object this* and *object that* for years, but how much impact has object technology really had on most software developers? Object-oriented programming (OOP) techniques have certainly made inroads into conventional software development, particularly in C++ and Delphi's Object Pascal. But other concepts — such as business objects and object databases — are typically considered bleeding edge by the mainstream. (Take, for instance, the share of blank looks I received carrying an object database book around at a developer's conference last year.) The hype surrounding objects is certainly not new. Nonetheless, as we begin 1996, I believe the technology is finally catching up with all the hyperbole; and when it does, objects will have a direct impact on how we design software. I've noticed four signs of change.

First, object orientation is emerging as a requirement for visual programming tools. For years, tools like PowerBuilder, Paradox, and Visual Basic were "object-based," while OOP remained in the 3GL world with C++ and SmallTalk, or with niche products like Actor. With Delphi, Borland bridged two worlds by building a rapid application development (RAD) tool on top of an object-oriented language. Other vendors are now playing catch up. The next version of PowerBuilder is touted as truly object-oriented. Even Visual Basic, while its programming language is definitely not OO, sports distributed object capabilities.

Second, the technology of object storage is maturing. One of the dilemmas with developing objects within Delphi has been the issue of storing object data. Because objects instantiated at run time are stored only in memory, all information related to an object is lost when you close an application. Perhaps this is inconsequential when you are dealing with a UI component, but persistent storage of objects is essential when you want to work with customers or employees as objects rather than as database elements. You can use a traditional relational database, but the mapping required to store complex objects can be daunting. Object databases offer another storage solution, but ODBMSs — which hold a tiny fraction of the database market — are typically sold as class libraries for C++ or SmallTalk.

Fortunately, the ability to store persistent objects in Delphi is beginning to improve. Object database vendors will be releasing versions of their products that can integrate with Delphi and other visual programming tools. More important, however, is the trend of major relational database vendors (particularly Oracle, Sybase, and Microsoft) to add "object extensions" to their databases. When this technology appears, you will be able to store instances of objects as effortlessly as you can now store a record in a database.

Third, the allure of business objects is growing. In the RAD world, objects are often thought of as either UI widgets or other "system" components like *TQuery* or *TTimer*. (If you don't believe me, take a look on the Web and see just how many variations of Delphi edit controls there are!) However, while that form of object-orientation is good, there is potentially a far more significant aspect to OOP that can make lasting differences to companies today: business objects. A business object encapsulates logic and rules relevant to a specific business process or entity. While more expensive to develop than RAD applications, business objects offer the promise of reuse and "changability" for long-term benefits.

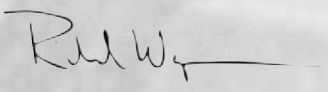
Fourth, the technology to develop multi-tier applications on the Windows platform is emerging. Until recently, *client/server* meant two-tiered architecture; the application would reside on the client PC, while the data would be stored in a SQL database on the server. The inherent flaw with this approach, however, has been determining where the business rules of the application should be stored: within the client's UI, or as stored procedures on the server? Neither choice is ideal. Mixing presentation code with business logic is a blueprint for disaster from a maintenance standpoint. But at the database level, you are forced to rely on a relatively unsophisticated language — SQL — to manage complex business rules. In such an environment, the ability to deploy business objects becomes problematic. From an OO standpoint, the solution is to partition the application, adding a business logic layer in between the presentation and database levels. Undoubtedly, some partitioning can occur simply at the client PC level (perhaps through dynamic link libraries). But in distributed environments, this type of middleware cannot to be restricted to residing solely on the client.

Until now, little technology has existed for deploying objects in the Windows environment. The release of Visual Basic 4.0 Enterprise Edition, however, features a Component Manager that allows you to access remote OLE objects from within a Visual Basic application. Borland and PowerSoft will surely be offering similar capabilities in the future with their products.

If you are still skeptical of the significance of objects, think of the hype surrounding CD-ROM media in the early 1990s. For years, industry pundits heralded "The year of the CD-ROM," when in fact the number of PCs that even had CD drives was a fairly small percentage. Nonetheless, in 1996, CD-ROM dominance is being realized: CD-ROM software revenues will for the first time exceed that of floppy software this year. The world finally caught up with the CD-ROM hype; the same will happen with object technology.

In the coming months, we'll look together at more trends that will have an impact on Delphi developers in a new column called ***File | New***.

Richard Wagner, Contributing Editor



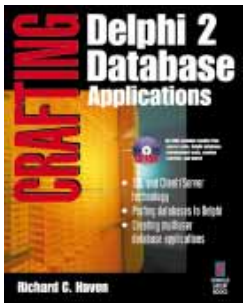
Richard Wagner is the chief technical officer of Acadia Software in Boston, MA and Contributing Editor to Delphi Informant. He welcomes your comments at rwagner@acadians.com.

Delphi TOOLS

New Products and Solutions

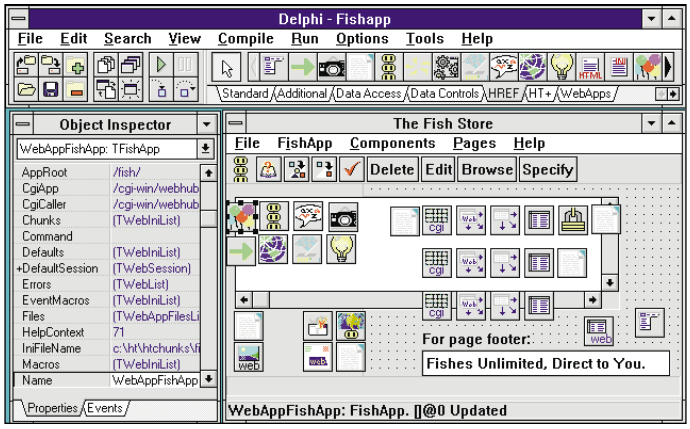
New Delphi Book

Crafting Delphi 2 Database Applications
Richard C. Haven
The Coriolis Group



ISBN: 1-883577-62-4
Price: US$44.99, Canada $62.99 (1000 pages, CD-ROM)
Phone: (800) 410-0192

A Delphi Web Framework: WebHub for Windows



HREF Tools Corp., of New York, NY, announced an early release of ***WebHub***, a Delphi-based framework for building and maintaining dynamic, database-driven World Wide Web sites.

WebHub includes over 30 Web-specific components that simplify database-driven Web site development and maintenance. Its architecture provides features such as macro expansion, reusable HTML chunks, and the ability to "refresh" a site without recompiling the .EXE, resulting in minimal downtime.

WebHub solves the tough CGI (Common Gateway Interface) challenges, according to the company. For example, WebHub augments Netscape's "cookie" solution for persistent client state with a session object that automatically tracks surfers and their data. WebHub bypasses CGI's critical performance problem — a WebHub-based .EXE can remain running with IDAPI channels open, while a tiny 4KB .EXE loads and unloads for each http request.

A free trial version is available from HREF's Web site.

Price: US$665

Contact: HREF Tools Corp., c/o Shine & Company, 9 East 40th Street, New York, NY 10016
Phone: (800) 365-9533, or (707) 542-0844
Fax: (707) 527-5373
E-Mail: Internet:webhubsales@-href.com
Web Site: http://www.href.com/

SkyLine Tools Debuts ImageLib 3.1 and ImageLib 95

SkyLine Tools Inc., of North Hollywood, CA, recently announced the release of ***ImageLib 3.1 Portfolio*** (16-bit), and ***ImageLib 95*** (32-bit), a set of 15 true Object Pascal VCLs that provide an interface to a central DLL. The Delphi components allow programmers to customize and modify features, thereby creating new programs. Using these tools, the novice programmer can develop "code free" applications.

ImageLib supports the following graphics file formats: TIFF, JPEG, PNG, GIF, PCX, BMP, ICO, WMF, CMS, and SCM. Images can also be TWAIN scanned, rotated, flipped, zoomed, and sized.

The new 16- and 32-bit versions support AVI, MOV, RMI, MID, and WAV multimedia formats, as well as scrolling text messages in BLOB fields. Text may also appear in other sections of the application, and can be placed, and/or rotated over an image. The ImageLib zoom tool enlarges any portion of an image with a simple mouse click. Image, video, and sound files can be previewed via its new Open dialog box.

ImageLib is royalty-free and allows the developer to supply applications in Windows 3.1 and Windows 95. With ImageLib you can enable all image formats with 4- and 8-bit dithering, as well as 24-bit "true color." ImageLib provides access to features like JPEG Quality and JPEG Smooth, and images can be converted to JPEG gray scale.

Price: ImageLib 3.1 Portfolio 16-bit version, US$139; ImageLib 95 32-bit version, US$169; ImageLib Combo 16- and 32-bit versions, US$199.

Contact: SkyLine Tools Inc., 11956 Riverside Drive, Suite 206, North Hollywood, CA 91607
Phone: (800) 404-3832
Fax: (818) 766-9027
E-Mail: CIS: 72130,353
CIS Forum: GO DELPHI
Web Site: http://theclassifieds.com/-skyline_tools

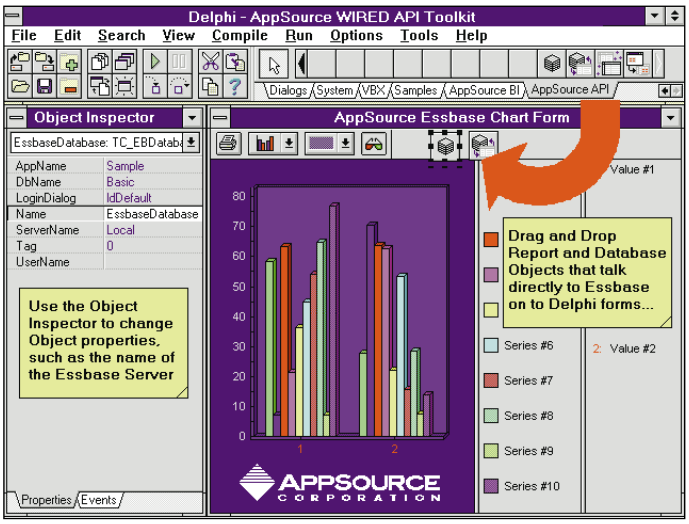
Delphi TOOLS

New Products and Solutions

Delphi Training

Financial Dynamics is a Borland Premier Connections Partner headquartered in Tysons Corner, VA, with offices in New York, NY; Richmond, VA; Raleigh, NC; and Greensboro, NC. Financial Dynamics offers a full suite of Delphi training and consulting services, including monthly three-day Introductory (US$1,125), and two-day Advanced (US$750) courses. To complement the Borland-certified courses, Financial Dynamics offers a series of client-tailored Boot Camp workshops. During the Boot Camp, a development team and their mentor analyze, design, build, and deploy a real-world application selected by the client. For more information contact Paul Fischer at (800) 486-5201, ext. 124; or browse their Web site at http://www.findyn.com/findyn/.

AppSource Announces WIRED API Toolkit: An OLAP Resource for Delphi



AppSource Corp. of Orlando, FL announced the release of ***WIRED API Toolkit,*** which includes object class libraries and visual controls released from **Borland International** and **Arbor Software** of Sunnyvale, CA. This toolkit allows Delphi developers to create OLAP (On-Line Analytical Processing) applications that use the power of Arbor's Essbase multidimensional database server.

WIRED API Toolkit accelerates the process of developing OLAP applications by providing a set of Delphi component objects and class libraries for Essbase. Data-aware database and report objects can be dragged from the Delphi Component Palette and dropped onto a form, where their properties can be set at design time. Dimension objects are created at run time, allowing the application to sense and react to changes in users' Essbase databases.

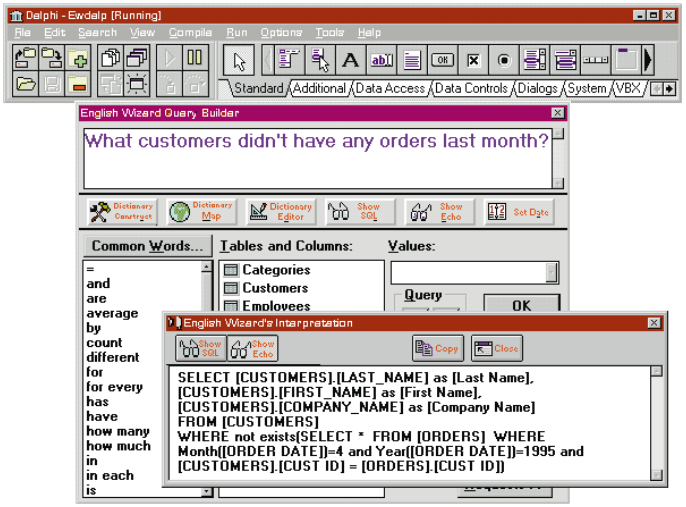
The WIRED API Toolkit enables third-party developers and consultants to build OLAP applications for budgeting, sales analysis, product planning, and financial analysis, leveraging both the flexibility, speed, and ease of development of Delphi, with the analytical power and response time of Essbase. (WIRED API Toolkit was initially created for AppSource Corp.'s WIRED for OLAP product, also built using Delphi and Essbase.)

The base version features database and report objects with dynamic dimension object creation. The enhanced version provides additional objects and pre-built forms to front-end the Essbase API.

Price: Base version with unlimited run-time license, US$995; enhanced version, US$2,995.

Contact: AppSource Corp., 4751 Rosewood Drive, Orlando, FL 32806-7811
Phone: (800) 611-5664, or (407) 888-8050
Fax: (407) 888-8070
E- Mail: Internet:appinfo@-appsource.com

English Wizard VBX Translates English into SQL



Linguistic Technology Corporation, of Acton, MA, has announced their ***English Wizard*** VBX for use with Delphi. English Wizard translates ordinary English database requests into SQL. It can enable any database reporting tool to understand English requests for information, so users don't need to understand SQL to request data.

For the most flexibility, developers can invoke English Wizard functions by directly calling its API. Calling the DLL directly allows programmatic changes to English Wizard's dictionary.

Additional features of English Wizard include its ability to "English-enable" ODBC-compliant reporting tools, including Microsoft Access, Forest & Trees, Impromptu, InfoMaker, ReportSmith, Microsoft Query, and Excel.

Price: US$149

Contact: Linguistic Technology Corporation, 179 Great Road, Suite 220, Acton, MA 01720
Phone: (800) 425-8200, or (508) 266-1818
Fax: (508) 266-1828
E-Mail: Internet:support@lingtech.com or CIS: 75407,3367
Web Site: http://world.std.com/-~engwiz

table of CONTENTS

News LINE

March 1996

Java Track Added to Borland Developers Conference

Borland International Inc. has added a Java track to its 7th Annual Borland Developers Conference scheduled for July 27-31, 1996 at the Anaheim Convention Center in Anaheim, CA. For more information contact Borland at (408) 431-1000, or http://www.borland.com.

Five-Year Ordeal Ends: Borland Prevails in Lotus Copyright Suit

Scotts Valley, CA — Borland International Inc. has won its five-year copyright infringement suit with Lotus Development Corporation. The US Supreme Court, which had agreed to hear the case last year, affirmed a decision by the First Circuit Court of Appeals which had ruled in Borland's favor.

Lotus initially filed its suit in 1991. In August 1992, the US District Court in Boston ruled that an optional feature in Borland's spreadsheet products, called the Command Hierarchy, infringed the copyright of Lotus 1-2-3. Borland voluntarily removed this feature from shipping versions of the product following the decision. The court reaffirmed its decision in July, 1993.

In August, 1993, the Federal District Court ruled that another compatibility feature in Quattro Pro and Quattro Pro for Windows infringed the copyright of Lotus 1-2-3. The court subsequently entered an injunction against Borland against further sales or distribution of then current versions of Borland's spreadsheet products. In response, Borland shipped new versions of Quattro Pro that did not include the features found to be infringing, and announced it would seek an immediate appeal.

In March of 1995, The US Court of Appeals for the First Circuit reversed the District Court ruling that Quattro and Quattro Pro infringed the copyright of Lotus 1-2-3. In written opinions, three appellate judges held in favor of Borland. The court concluded: "Because we hold that the Lotus menu command hierarchy is uncopyrightable subject matter, we further hold that Borland did not infringe the copyright by copying it."

Borland sold its Quattro Pro spreadsheet to Novell Inc. in March of 1994.

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

Delphi Web Seminar Announced

Fairfax, VA — HREF Tools Corp. has scheduled a free introductory seminar "Building World Wide Web Applications with Delphi and WebHub" for March 25, from 7PM to 9PM. Hosted by BRTRC, the seminar will be led by HREF Tools CEO Ann Lynnworth. For more information, visit the HREF Web site at http://super.sonic.net/ann/-events.html, or send a message to seminars@href.com for an automatic reply.

Delphi Named Finalist in 1996 Excellence in Software Awards

Washington, DC — The Software Publishers Association (SPA) has named Borland's Delphi a finalist in the Best Programming Tool category for the 1996 Excellence in Software Awards — the 11th annual Codie Awards. The awards will be presented during SPA's 11th Annual Codies gala in San Francisco, CA on March 4, 1996.

Over a period of seven weeks, 100-plus reviewers from the software industry and national technology media selected five finalists in each of the 34 categories.

For more information, visit SPA's CompuServe forum (GO SPAFORUM), or Web site at http://www.spa.org/.

Delphi Informant Reader's Choice Banquet

Elk Grove, CA — Informant Communications Group, Inc. (ICG) has announced the first annual *Delphi Informant* Reader's Choice Awards Banquet, scheduled for Tuesday, March 26, 1996, immediately following the first full day of Software Development '96 West in San Francisco.

The banquet will spotlight the winners in each of twelve product categories. The event will be held at Chevy's restaurant, near Moscone Center, site of Software Development '96 West. The event is co-sponsored by ICG and Borland International.

"We're very excited about holding the *Delphi Informant* Reader's Choice Awards Banquet during Software Development '96 West" said Mitchell Koulouris, president of ICG. "It's the ideal forum to celebrate the resounding success enjoyed by Borland's Delphi product over the past year, and an opportunity for third-party vendors to receive the attention they deserve."

The product receiving the most votes overall will be named Product of the Year. The results of this survey will be published in the April issue of *Delphi Informant,* which will be distributed at Software Development '96 West.

News LINE

March 1996

Oracle Sponsors Sun Microsystems Java Cup International Developers Contest

Redwood Shores, CA — Oracle Corp. is sponsoring Sun Microsystem's Java Cup International, a developer contest which began last December and ends March 31, 1996. Winners will be announced in San Francisco in April 1996. Sun hopes to have 500 applets in the public domain following the contest. Additional information about the contest, including prizes to be awarded, can be found on Sun's Java Cup International World Wide Web page at http://javacontest.sun.com.

Borland Names New Member and Chairman to Board of Directors

Scotts Valley, CA — Borland International Inc. has appointed Dr William F. Miller, 70, chairman of the company's board of directors. Dr Miller, a professor at Stanford University, succeeds Philippe Kahn who resigned as chairman effective January 1, 1996. Borland also announced that Dr Harry J. Saal, 51, chairman of Network General Corp., has been appointed to its board of directors.

Dr Miller is professor of public and private management, Graduate School of Business, at Stanford University; president emeritus of SRI International; professor of computer science, School of Engineering; and senior fellow, Institute of International Studies.

In 1990, Dr Miller retired following 11 years as president and CEO of SRI International. He also served as chairman and CEO of the SRI Development Co. and chairman and CEO of the David Sarnoff Research Center. Prior to his role at SRI, he was vice president and provost of Stanford University and vice president for research and associate provost of computing. He has served on many government commissions and as director of several corporations. He is currently a member of the board of directors of Varian Associates, Inc., First Interstate Bancorp, First Interstate Bank of California, Pacific Gas & Electric Co., Regis McKenna, Inc., SciosNova, and the BHP International Advisory Council.

Dr Miller received his graduate and undergraduate education at Purdue University, where he received B.S., M.S., Ph.D., and D.Sc. honoris causa degrees.

Dr Saal is chairman of Network General Corp., a company he founded in 1986. Before founding Network General, Dr Saal founded Nestar Systems, Inc., a pioneer in systems for personal computers. Dr Saal was the founding president and CEO of Smart Valley, Inc., a nonprofit organization chartered to create a regional electronic community by promoting an advanced information infrastructure and the collective ability to use it.

Dr Saal is a magna cum laude graduate of Columbia University where he received his Ph.D. in high energy physics in 1969. He then served as deputy director of the Stanford Linear Accelerator Center's Computation Group, and later as visiting associate professor of computer science at the State University of New York at Buffalo. From 1973 to 1978, he worked for IBM at the IBM Scientific Center in Haifa, Israel, and the IBM General Products Division in San Jose.

Dr Saal also serves on the board of several other private and public high technology firms. Ernst & Young named Dr Saal the Bay Area 1990 Software Entrepreneur of the Year.

Other members of Borland's board of directors include Gary Wetsel; Stephen J. Lewis, managing director of Generation Ventures, L.C.C.; David Heller, director and president of Pacific Technology Capital Corp.; George Hara, partner, Accel Partners; and Philippe Kahn.

New *Delphi Power Tools* Catalog Published

Elk Grove, CA — Informant Communications Group, Inc. (ICG) has published the Spring 1996 and Spring/Summer 1996 editions of its *Delphi Power Tools*, a catalog of third-party add-in products and services that complement Borland's Delphi product. Compiled for the serious Delphi developer, *Delphi Power Tools* is independently produced by ICG and inserted into each copy of Delphi 2.0.

Both new editions of the catalog feature third-party VCLs, OCXes, and DLLs that can be used with Delphi 2.0. The catalog also contains a comprehensive listing of third-party add-in products, consultants, and trainers. The Spring 1996 edition of *Delphi Power Tools* increased in size from 52 to 84 pages.

"*Delphi Power Tools* is a low-cost, highly effective method of reaching thousands of Delphi developers" said Mitchell Koulouris, president of ICG. "The Delphi add-in community has grown and solidified over the last year and *Delphi Power Tools* has been an important vehicle in growing the Delphi market."

ICG plans to produce the next edition of *Delphi Power Tools* in June, 1996. The company is also planning new editions of both the *Borland C++ Power Tools* and *Paradox Power Tools* catalogs. Advertisers can obtain a media kit and more information by contacting Advertising Director Lynn Beaudoin at (916) 686-6610, ext. 17 (Internet:lbeaudoin@informant.com); or Advertising Sales Associates Sheri Birkmaier at (916) 686-6610, ext. 21 (Internet:sbirkmaier@informant.com), or Joe Krack at (916) 686-6610, ext. 27 (Internet:jkrack@informant.com).

table of CONTENTS

On the Cover

Delphi 1.0 / Object Pascal

By Andrew J. Wozniewicz

DLLs: Part I

Introducing Dynamic Link Libraries and How to Create Them Using Delphi

Programs running in Windows can share subroutines located in executable files called *dynamic link libraries* (DLLs). Typically, DLLs are used in the context of a large system or application, where many common routines are shared among many programs.

This is the first article in a series that describes how to create and use DLLs with Delphi. In particular, the topics include:

- Creating DLLs using Delphi.
- Declaring import units to automatically make the subroutines in a DLL available to Delphi programs.
- Using subroutines implemented in DLLs.
- Loading DLLs, and resolving subroutine addresses at run time.

First we'll cover basic DLL terminology. Then we'll build some string-handling functions.

Understanding DLLs

A DLL is an executable module containing subroutines that Windows programs can call to perform common, useful tasks. DLLs are one of the more important elements of Windows, and exist primarily to provide services to applications. Windows itself uses DLLs to make Windows subroutines and resources available to Windows applications. It's probably fair to say that DLLs are the fundamental concept of Windows architecture.

Windows itself consists largely of three DLLs named KERNEL, USER, and GDI. These libraries contain the code and data for the Windows application programming interface, or API.

A Delphi application is an executable file that consists of one or more forms or windows. A program interacts with the user at run time and retrieves messages and events from a message queue maintained on its behalf by Windows.

In contrast, DLLs are separate libraries of subroutines, containing procedures and functions that can be called by other programs. However, DLLs are not themselves directly executable and come into play only when another Windows module calls one of the subroutines in the DLL. Specifically, DLLs do not have their own message queues and rely on the flow of messages and events in the application being used.

A Waste of Space

A user can simultaneously run several copies of a program, called *instances*. These instances share the same code in memory. However, if you run several programs containing the same subroutine

library, the same executable code will be cloned in memory. The same routines will reside in memory in multiple copies.

In other words, if you *statically link* the common routines (shared among many programs) as units to each individual program, you end up with identical code that is linked into many executable modules. If you consider that each of the various individual programs can be running simultaneously on the same system, you see that virtually identical code may be loaded unnecessarily into memory in multiple copies. This results in space being wasted instead of used for the needs of applications.

DLLs to the Rescue

The solution to this problem is the DLL, and the process of dynamic linking that eliminates the duplication of common code. *Dynamic linking* is the process that Windows uses to link a subroutine call in one executable module to the actual function in a dynamic library module at run time. Furthermore, if you change any of the subroutines in a common, static subroutine library, you must rebuild all the programs that use the library. However, if you extract the common subroutines and put them into a separate dynamic library, then only the library contains the routines required by all programs.

There is only one copy of a dynamic library loaded in memory at run time, no matter how many programs use it. Additionally, if you change a subroutine in the common library, you need only recompile the library module. The programs that use the library remain intact, unless you have changed the interfaces to the library. DLLs are similar in concept to Object Pascal units. They represent an additional step in the quest for separating concerns among different subsystems, or modules comprising a program. Object Pascal units are only separate until compile/link time, when they are combined to form a single executable file.

DLLs remain separate even when an application is deployed. Therefore, DLLs are more autonomous than Object Pascal units, and they use the operating system's (Windows) facilities rather than the language/compiler facilities to achieve modularization and integration of separate modules. The integration occurs at run time, late in the development cycle. This is desirable from the perspective of reusability and flexibility.

In addition to letting applications share executable code, you can use DLLs to share other resources, such as data and hardware. Windows fonts, for example, are actually shared text-drawing data, and Windows device drivers actually are special DLLs that enable applications to share hardware resources.

Although a DLL can have any valid file name extension, the standard extension is .DLL. Only libraries with the default extension of .DLL can be loaded automatically by Windows. If the library file has a different extension, it must be loaded explicitly by the program that wants to use it. An application can invoke the standard API **LoadLibrary** function to explicitly load a DLL. We'll see an example of this later.

Dynamic Linking

When a Windows program is loaded into memory, the calls to the Windows API will point to the entry of the subroutines in the appropriate DLLs, which also are loaded into memory. Hence, the executable application module relies on the presence of other supplementary modules. Typically, the program cannot run in the absence of the required external DLLs.

Contrast this with the process of static linking, when the addresses of all subroutines are known beforehand — at compile time. The compiler/linker can appropriately resolve any static references throughout the program and substitute actual addresses for symbolic names used in the source code. Static linking results in a fully self-contained executable file.

Creating Custom DLLs

So far, as you have developed Delphi applications, you have been writing Windows programs. Now it's time to develop DLLs *for use by* programs. Many of the principles you have learned for writing Delphi programs are also applicable to writing Delphi DLLs, but there are some important differences to understand.

As mentioned, DLLs enable you to more economically share code and resources among many applications. Although they are important in the overall architecture of Windows, custom DLLs are not absolutely necessary for every Windows application. After all, you have probably built a fair number of Delphi applications without worrying about DLLs. Although each one of these applications relied on the access to Windows DLLs, you did not have to deal with the details of loading and binding to DLLs. This is about to change.

There are several distinct advantages of DLLs that become increasingly important as your applications become larger, more sophisticated, and complex:

- DLLs enable you to share code and resources among many applications.
- Using DLLs, you can easily customize your application for different purposes.
- DLLs facilitate the development of large, complex applications by allowing a more strict separation of subsystems.
- DLLs enable you to streamline access to data and data-like resources (such as business model infrastructures), and to hardware devices.

Note that you can use DLLs written in other language environments, and you can write custom DLLs in Delphi. The interfaces between a DLL and a program are usually transparent and independent of the implementation language. Therefore, any DLL-capable program can call a subroutine in a Delphi-created DLL — whether that program was written in C, C++, Visual Basic, ObjectPAL, dBASE, etc., or an extension to an off-the-shelf application, including macro-language extensions written in Microsoft Word for Windows or Excel.

Your Delphi programs also can call subroutines in DLLs written in other languages. As mentioned, they do it continually whether or not you are aware of it. Every Delphi program must interact with Windows, and that involves making calls to one, and typically more, of the three main Windows DLLs.

Declaring Libraries

Conceptually, DLLs can be treated simply as more autonomous units. However, from a Pascal programmer's perspective, a DLL source code module bears a striking resemblance to a program module.

A DLL is defined in the same manner as a program, with the reserved word **library** replacing the word **program**:

```
library LibMane;
  < declarations and implementation >
begin
  < optional initialization code >
end.
```

Figure 1 shows an example of a library module. It begins with a header, consisting of the reserved word **library**, followed by the library name and a semicolon. The library implementation follows much of the same rules as the program implementation. You can insert **uses** clauses; declare types, constants, and variables; and declare and implement subroutines and classes — in other words, everything you can do within a program.

```
library StrUtils;

  function UpperCaseStr(S: string): string; export;
    begin
      for I := 1 to Length(S) do
        S[I] := UpCase(S[I]);
      Result := S;
    end;

  exports
    UpperCaseStr;

begin
end.
```

Figure 1: An example of a library module.

Again, similar to a program module, an executable block is enclosed within a pair of **begin** and **end** keywords at the end of a library module. Its role is similar to the role of a **unit** initialization block, rather than that of a program's executable block. It's meant to initialize any local data and objects needed by the library.

The **begin..end** initialization block of a library module may (and often does) remain empty. In a library, these keywords must be present, even if there are no data structures to initialize. Conversely, with units, the initialization block's **begin** keyword can be omitted if there is no initialization code.

Using the exports Clause

One of the main reasons for having a library is to reuse common subroutines from different applications. This implies that you must first implement the subroutines to use them. Fortunately, the way you implement subroutines inside a DLL is not much different from how you implement them in a regular program. In fact, you can use the same source code units that implement subroutines in both programs and DLLs. Subroutines that are made public by listing them in a unit's **interface** section can be used directly in a program by being statically linked to it via the **uses** clause. These subroutines can also be linked into a DLL and made available to a greater number of applications simultaneously by being "exported" from the DLL.

Most importantly, when building dynamically linked collections of subroutines (from the perspective of making the subroutines available outside the library), you must explicitly export those subroutines you intend to be visible outside the library by creating an **exports** clause. The syntax for the **exports** clause, in its simplest form, follows this convention:

```
exports
  Subroutine1,
  Subroutine2,
  Subroutine3,
  ...
  SubroutineN;
```

The **exports** clause begins with the reserved word **exports**, followed by a list of exported subroutines separated by commas. The clause is terminated by a semicolon. Here's an example:

```
exports
  FillStr,
  LTrimStr,
  RTrimStr;
```

Only one **exports** clause may be present in a project. You cannot place an **exports** directive inside a unit belonging to a project; **exports** can only be placed inside the main library or program module.

There may be any number of **exports** clauses in a project's main source file. All the subroutines listed inside all the **exports** clauses will be made visible to other modules "by name," and by other means (we'll discuss this more later). When the module that contains an **exports** clause is compiled, the resulting executable file (.EXE or .DLL) contains special entries that enable Windows to dynamically link the exported subroutines — at run time — to all other applications and modules that may need them.

The key to successfully exporting a subroutine from a DLL is to make the subroutine itself exportable. A subroutine cannot be exported unless you explicitly define it as exportable by placing the **export** directive at the end of the subroutine's declaration heading.

export Directive

The **export** directive makes a subroutine exportable and must be placed after the regular heading in the subroutine's declaration:

```
function Name(<Parameters>): ReturnType; export;
```

or

```
procedure Name(<Parameters>); export;
```

Here are some examples:

```
function StripStr(S: string): string; export;

procedure RefreshBuffers; export;
```

The **export** directive instructs the compiler to create special code at the beginning and end of the compiled routine that enables Windows to use it from another executable module.

In summary, to make a global subroutine inside a DLL project available for use by other Windows executable modules, you must ensure that:

- The subroutine is declared with the **export** directive, i.e. it is *exportable*.
- The subroutine's name is listed inside an **exports** clause of the library module, i.e. it is actually *exported*. A subroutine may be made exportable (with an **export** directive) but not actually exported (with an **exports** clause).

Architecting DLLs

As a matter of principle, when building reusable DLLs you should avoid implementing the subroutines being exported as part of the main library module. There's a temptation to begin creating the implementations of the exported subroutines directly inside the main library module — you could simply start writing code in the main library project file. This approach, however, quickly leads to large and unmanageable library files.

You should separate libraries into units for the same reason you divide programs into separate units. That is, managing smaller units with well-defined interface sections is far easier than keeping track of all the subroutines implemented in one place. Remember that you may be implementing both the subroutines you intend to export from the library, and the auxiliary subroutines for the private use of the library itself that are not exported. By using separate units that group closely related subroutines, you impose an additional layer of organization on the library's code.

An additional benefit to organizing your implementation code in units is that you can either use dynamic linking, or statically link the implementing unit into an application. On some occasions, you may decide that linking the code directly into the application is a better approach, or you'll be content to use the services encapsulated by your DLL. In either case, it's better to have one unit source file that you can use in both scenarios.

Hence, a better approach to organizing a library's code is to have all the routines, both exported and private, implemented in a separate unit, or collection of units. The main library module would have the units it incorporates listed in its **uses** clause, and would implement the **exports** clause.

The following example demonstrates how to organize your subroutines into separate units, yet make them exported from the library.

A Simple Library Example

The best way to illustrate how to build a DLL is to step through a practical example. You'll see that the task of DLL writing has been made reasonably simple by Delphi (much simpler than in more traditional environments). However, you must remember that writing DLLs is a traditional coding exercise and that Delphi 1.0 does not support you in any special way beyond providing the raw capabilities of creating and using DLLs.

Let's build an example DLL that implements several useful string-handling subroutines. Here are the steps:

1. Create a new, blank Delphi project.

2. Close it without saving the default form-unit file, Unit1, which Delphi creates by default. Right-click in the Code Editor and select **Close Page** from the pop-up menu. Answer **No** when you are prompted to save the unit. The Code Editor closes because there are no files to edit.

3. Create a new directory named DLLFIRST. Save the project as DLLFIRST.DPR in the new directory. At this point, your project only consists of the main program file, DLLFIRST.DPR.

4. Select **View | Project Source** from the Delphi menu. The Code Editor reopens and displays the DLLFIRST program file (see Figure 2). Now you'll convert this minimal Delphi program to an equally minimal DLL.

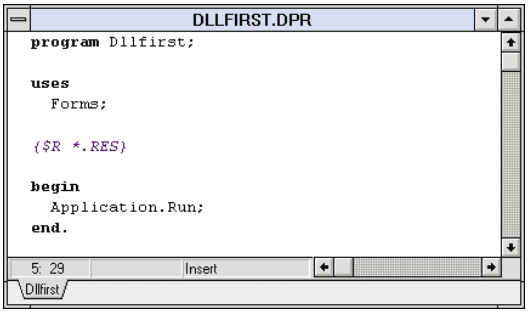


Figure 2: The DLLFIRST project before being modified into a DLL.

5. On the first line of code, replace the reserved word **program** with the word **library**. Then remove:

```
uses
  Forms;

{$R *.RES}
```

Now remove the *Application.Run* statement. Your main source code file is now reduced to:

```
library Dllfirst;

begin
end.
```

6. Congratulations! You've just created the world's simplest DLL. Don't forget to save the project.

7. Select **Compile | Compile** from the Delphi menu. It takes only a fraction of a second to compile this empty library, but you now can verify that instead of generating an executable program file (.EXE) you have succeeded in creating your first dynamic link library: DLLFIRST.DLL.

Of course, this library is not particularly useful in its current state. It does not provide any services that a program might use. Soon, you will develop a set of simple character-string manipulation routines that will be available from this DLL.

An interesting point to note, however, is that the library may be used almost "as is" to store various binary Windows resources, such as bitmaps, icons, and so on. In this context, the DLL is a repository of dynamically installable resources rather than a collection of subroutines. Note that if you had chosen **Run | Run** instead of **Compile | Compile** to compile your DLL, it still would have been compiled, but because it knows that DLLs are not directly executable, Delphi would have displayed the error message "Cannot run a unit or DLL."

If you close the project at this point and attempt to open it later, Delphi opens the file but also displays an error message, "Error in module DllFirst: uses clause missing or incorrect." Delphi is telling you that a **uses** clause is absent from your project file. Delphi needs the **uses** clause to maintain the list of source files belonging to the project, but currently your project consists of the single, main project file. Although there is no true need for the **uses** clause at this point, you have to reintroduce one to placate Delphi's code-generation facilities. So modify the project source to add a **uses** clause for the WinTypes unit:

```
library DllFirst;

uses
  WinTypes;

begin
end.
```

Listing WinTypes in the **uses** clause of the library is safe because it does not introduce any real dependencies. WinTypes is a unit that declares various data types used by the Windows API. It does not implement any code, so you are not adding overhead to your library. Also, you can later delete WinTypes from the **uses** clause entirely. You cannot do that right now, because there are no other units in the project and this would eliminate the **uses** clause altogether. Remember, you cannot leave the reserved word **uses** without unit names listed after it.

Using DLLFirst String-Handling Functions

Now let's implement some string-handling functions inside the DLL. First, let's create a separate unit, named XString, that implements the subroutines. To do this, create a new unit for the project by selecting **File | New Unit** from Delphi's menu. Then save it as XSTRING.PAS by selecting **File | Save File As**.

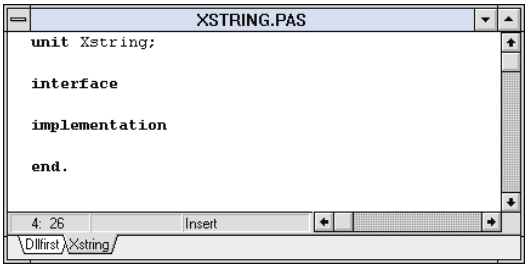


Figure 3: The shell for our sample unit.

The newly-created unit (see Figure 3) is an empty shell.

Now we're ready to provide the actual string-handling routines inside the XString unit. We'll implement the following functions:

- The *FillStr* function returns a string of a single, repeated character value.
- The *UpCaseFirstStr* function capitalizes the first letter of every word of the passed string argument.
- The *LTrimStr* function removes leading blanks from the passed string argument.
- The *RTrimStr* function removes trailing blanks from the passed string argument.
- The *StripStr* function removes all blanks, whether leading, trailing, or embedded, from the passed string argument.

(This month, we'll build and analyze *FillStr* and *UpCaseFirstStr*. We'll move on to the other functions in April.)

In the following sections, note that the declarations of the string-handling subroutines to place inside the **interface** section of the XString unit include the **export** directive, while the implementations no longer repeat and must *not* repeat the directive. As noted earlier, the **export** directive makes these subroutines exportable across the Windows-module (.DLL/.EXE) boundaries.

The *FillStr* Function

The first function to include in your DLL's XString unit is *FillStr*. This classic library function returns a string of the specified length consisting of the indicated character, repeated throughout the entire string a specified number of times.

FillStr is declared as follows:

```
function FillStr(C: Char; N: Byte): string; export;
```

The function takes two parameters: 1) the character to be repeated, `C`; and, 2) the desired length of the string, `N`, or the number of times to repeat the character `C`. Enter this declaration in the **interface** section of the XString unit we just created.

Here's a suggested implementation for the *FillStr* function:

```
function FillStr(C : Char; N : Byte): string;
{ Returns a string with N characters of value C }
begin
  FillChar(Result[1],N,C);
  Result[0] := Chr(N);
end;
```

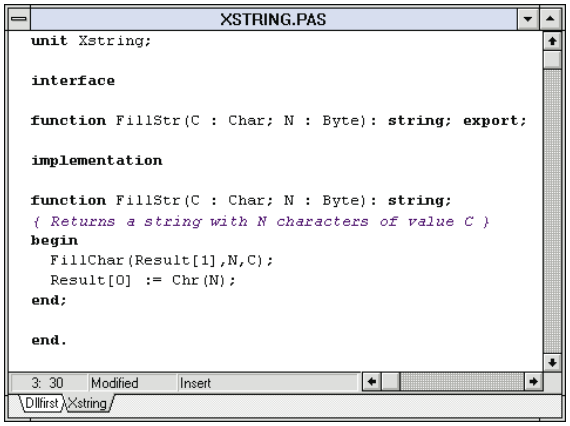


Figure 4: The XString unit with the *FillStr* function implemented.

Enter this code in the **implementation** section of the XString unit. (The entire unit is shown in Figure 4.) The function uses a few tricks to get the job done more efficiently. First, it uses the standard procedure *FillChar* to quickly fill the *Result* string with the appropriate character value. The alternative approach is to set up a **for** loop that counts through the subsequent character positions in the result string and then "stuffs" the character to be repeated in the character cells. *FillChar* accomplishes the same task in a single step.

Second, *FillStr* forces the returned string to be of the appropriate length by explicitly setting the length byte. As you may recall, the length byte of a string variable is the first cell (byte) of the string's character array. *FillStr* sets the length byte directly to ensure that the returned string contains the correct number of characters.

Here's an example of the *FillStr* function in use:

```
var
  S1, S2: string;
begin
  S1 := FillStr(' ',12);
  S2 := FillStr('#',80);
end
```

After the two calls to *FillStr*, S1 will contain a string of 12 blanks, and S2 will contain 80 number-sign (#) characters.

The *UpCaseFirstStr* Function

Another useful addition to the set of standard string-management routines already available in Delphi is the *UpCaseFirstStr* function. It complements the standard *UpperCase* and *LowerCase* string-handling functions from the SysUtils unit. *UpCaseFirstStr* capitalizes the first letter of every word. *UpCaseFirstStr* takes a single string parameter and returns the same string, so it is declared as follows:

```
function UpCaseFirstStr(const S: string): string; export;
```

Enter this declaration in the **interface** section of the XString unit. The body (i.e. the implementation) of *UpCaseFirstStr* mainly consists of a **for** loop that runs through all the characters in the string (see Figure 5). If the character is the first letter of a new word, it is capitalized, if not, it is skipped.

```
function UpCaseFirstStr(const s: string): string;
var
  Index: Byte;
  First: Boolean;
begin
  Result := S;
  First  := True;
  for Index := 1 to Length(S) do
    begin
      if First then
        Result[Index] := UpCase(Result[Index]);
      if Result[Index] = ' ' then
        First := True
      else
        First := False;
    end;
end;
```

Figure 5: The *UpCaseFirstStr* function.

The *First* variable is used to track if the *Index* variable marks the position of the beginning of a word inside string S at any given point. The *First* flag becomes *True* whenever the current *Index* corresponds to the first letter of a word inside the string S. The flag is set with this **if** statement:

```
if Result[Index] = ' ' then
  First := True
else
  First := False;
```

The new value of *First* to be used in the next iteration of the **for** loop is *True* whenever the current location in the string S is occupied by a blank (ASCII 32 decimal or ' '). This way, after a blank character is encountered, any non-blank character that follows is treated as the beginning of a new word.

The following condition determines if the current character should be capitalized:

```
if First then
```

This checks if the *First* flag is set (i.e. whether the previous character in the string was blank) and if so, attempts to capitalize the current character. Note that the *First* flag is initially set to *True* just before the loop begins execution. This implicitly assumes that the first non-blank character in a string also marks the beginning of a new word and forces it to be capitalized. (Remember to always initialize your variables.)

Place the code of the *UpCaseFirstStr* function (see Figure 6) inside the **implementation** section of the XString unit file.

Here's an example of how *UpCaseFirstStr* could be used:

```
var
  S1: string;
begin
  S1 := UpCaseFirstStr('teach yourself delphi in 21 days.');
end
```

```
unit Xstring;

interface

function FillStr(C : Char; N : Byte): string; export;
function UpCaseFirstStr(const s: string): string; export;

implementation

function FillStr(C : Char; N : Byte): string;
begin
  FillChar(Result[1],N,C);
  Result[0] := Chr(N);
end;

function UpCaseFirstStr(const s: string): string;
var
  Index: Byte;
  First: Boolean;
begin
  Result := S;
  First  := True;
  for Index := 1 to Length(S) do
    begin
      if First then
        Result[Index] := UpCase(Result[Index]);
      if Result[Index] = ' ' then
        First := True
      else
        First := False;
    end;
end;

end.
```

Figure 6: The XString unit with the *UpCaseFirstStr* function implemented.

After the assignment statement is executed, the string `S1` contains:

```
'Teach Yourself Delphi in 21 Days.'
```

Until Next Time

In the next article, we'll implement several more Delphi functions into our sample DLL. These functions include *LTrimStr*, *RTrimStr*, and *StripStr*. We'll also discuss how to export the DLL functions we've created, and how to call them from a program. See you then. Δ

This article was adapted from material for *Teach Yourself Delphi in 21 Days* [SAMS, 1995], by Andrew Wozniewicz

Andrew J. Wozniewicz is president and founder of Optimax Development Corporation (http://www.webcom.com/~optimax), a Chicago-based consultancy specializing in Delphi and Windows custom application development, object-oriented analysis, and design. He has been a consultant since 1987, developing primarily in Pascal, C, and C++. A speaker at international conferences, and an early and vocal advocate of component-based development, he has contributed articles to major computer industry publications. Andrew can be contacted on CompuServe at 75020,3617 and on the Internet at optimax@optidevl.com.

By Craig L. Jones

PQA: Part I

Practical Quality Assurance Techniques for Delphi

For 'tis the sport to have the engineer
Hoisted with his own petard. — William Shakespeare, *Hamlet*

A new application has just been installed on a user's workstation and it performs flawlessly — at least as long as the programmer is demonstrating it. Then, almost immediately after the keyboard was handed back to the user — boom! Like a crudely manufactured medieval bomb, the new program explodes before the programmer even had a chance to step away. How embarrassing!

It seems that during the development and testing of the program, it never occurred to our developer to try a particular combination of values, hit a certain key at a certain time, or leave a certain field blank. But the user naturally does not think the same way, and readily wanders into untried waters. Perhaps the programmer did think to test for those conditions once upon a time, but the code subsequently changed during later development, allowing new bugs to be introduced that were never caught. In either case, the developer must now spend unbudgeted time on further debugging.

The solution is to apply some QA savvy. There's nothing secret or mysterious about quality assurance. In fact, it's really quite simple — just think of everything that can go wrong and test for it. Of course, that's easier said than done. The problem is that such work can be tediously time consuming. The trick then, is to eliminate the tedious aspects by writing *test drivers* that automate as much of the testing as possible.

This is the first installment of a three-part series on assuring the quality of Delphi programming projects. This series is primarily directed at Delphi programmers and assumes no prior knowledge of quality assurance (QA) techniques, although experienced QA engineers may pick up some pointers on plying their trade in the Delphi environment. Even programmers who previously disdained the "necessary evil" of QA will discover some enthusiasm for applying these techniques because they are quick to implement, easy to maintain, and automatically reusable.

In this installment, we'll explore some general QA theory and how it applies to the different stages of program development. We'll then begin to build a QA tool kit for testing Delphi programs — starting with a style of test drivers that are useful for testing simple functions. We'll also discuss the use of conditional compiler directives to keep the added test driver code from impacting the final executables.

The second installment will expand our QA tool kit to cover the testing of more complicated procedures and object methods. We'll also add facilities for running multiple test drivers consecutively (unattended), and using a compare program to check the results against an established baseline.

In the third and final installment, we'll look at a class of commercially available testing software that can record all user interaction during a test run of a completed program, and then play it back at high speed. Screen snapshots are taken along the way that are later compared to baseline images.

A series wrap-up will then show how the combination of proper planning with the right tools can make for a dramatic difference in the quality of a delivered program.

A Little Forethought

The watchword of quality assurance theory is *planning*. A good software engineer knows in advance how a program will be tested before writing a single line of code. A complicated (multiple man-year) project can have as many as eight test plans, covering each phase of the development cycle (see table in Figure 1).

For small projects, one formal integration test can suffice as long as:

- the test plan is comprehensive, precise, and developed in advance of the software to be tested (or at least simultaneously),
- it is backed up with systematic unit testing (as described below), and
- other good software engineering practices are followed (coding conventions, peer reviews, detailed documentation, etc.).

The importance of developing a test plan — before any coding begins — lies in the fact that the planning process itself can have as much, if not more, of an influence on the software development as actually executing the test. For one thing, by simply having specific testing objectives in mind, a programmer is bound to be more thorough. Coding decisions can most definitely be influenced by knowing which eventualities to allow for. Even major design issues can be influenced in this way, such as designing a report function to double as an export-to-ASCII function so that you can use a file compare utility to test it.

1	Requirements Verification	Using precisely detailed, written statements to verify a clear understanding of the user's requirements (overall goals and specific objectives).
2	Design Validation	A mental exercise of running through a program design, looking to ensure that all the contingencies are covered.
3	Unit Testing	Testing individual modules as they are developed. (Usually performed by the programmer.)
4	Integration Testing	Testing program modules in combination. (Often performed by a QA specialist.)
5	User Acceptance Testing	Testing performed by a User Advocate or User Analyst during the final phase of development, particularly focusing on user-interface details and performance issues.
6	Alpha Testing	Comprehensive testing of the completed program by the developers (or QA specialists) according to a comprehensive test plan.
7	Beta Testing	(a.k.a. "field testing") Ad hoc testing performed by a select group of users, often requiring two or three rounds.
8	Gamma Testing	Final field test of the debugged program, perhaps only now being provided with completed documentation.

Figure 1: Testing phases for a large project.

Test Planning Strategies

Like a journalist or a detective fresh at the scene of a crime, the best way to build a test plan is by thinking in terms of the "5 Ws" — who, what, when, where, why (and how):

- What functions must be tested?
- How are they going to be tested?
- Why are the tests necessary? What are the possible problems?
- Who's going to conduct the tests? The system designer? The programmer? A tester? The user advocate? Someone else?
- When in the development cycle do the tests need to be executed?
- Where (in what environment) should the tests be performed?

Enumerating what functions need to be tested is as easy as cannibalizing whatever serves as the system's requirements analysis document. The hard part is determining the "whys" (i.e. listing everything that can go wrong with each of those functions). However, once you have that, determining who, when, where, and how is fairly straightforward.

One way to unearth the possible problems is to take a "black box" approach at enumerating the combinations of conditions that might be troublesome; black box meaning no matter how the software operates internally. Let's say, for example, that you are developing an application to be used by a librarian.

A typical concern might be how well the software can process two different books that happen to have the same title — or for that matter, one book that was reprinted with a different title than the first edition. After a short brain-storming session, you might come up with something on the order of 200 such concerns for a small application.

Second, knowing how the code is designed (taking a white box approach), seek out additional, technical concerns. The table in Figure 2 describes four areas of consideration here.

A good test plan will ensure complete path coverage of the source code during testing, and that all significant boundary conditions are considered. Any performance issues, especially critical ones, should be thoroughly explained. Finally, any bugs that slip through the cracks of the formal testing, but are later caught by somebody's ad hoc testing, should be formalized to prevent regression.

Path Coverage	Testing the various combinations of logic flow through a program. At minimum, making sure that each line of code is executed at least once during testing. At the extreme, testing each and every combination of conditional expression values.
Boundary Conditions	Testing the code's ability to handle variable/parameter values that are at, or near, specific thresholds (e.g. 0, 1, 32767, -32768, $9,999.99, $0.005, 00:00, 23:59, 12/31/99, 2/29/2000).
Performance	Testing program response times.
Regression	Testing to ensure that previously eliminated bugs do not reappear, either exactly as they had before, or in some mutated form.

Figure 2: General areas of concern for a test plan.

A good test plan will list not only what needs to be tested, but how the developer intends to conduct the tests. This is especially important for anyone who is new to writing test plans, because it forces an examination of the issues from multiple perspectives. With a little experience, however, a developer can go directly to building test driver code.

Unit Test Drivers

In his article "Think Objects, Not Reuse" Richard Wagner [in the November 1995 *Delphi Informant*] touts the benefit of using inheritance as a "fire-wall" to keep an existing application from breaking while a related application is under development, even though they share a common code base. Wagner states:

> First, in creating a subclass you avoid touching the bulletproof code of [the parent class], yet at the same time, you have full access to this code. Second, you can maintain a link between the ancestor and descendant class.

Thus, if the parent class itself should need to be modified for some reason (e.g. to accommodate a new operating system), the subclassed object will automatically benefit from the change.

Wagner makes a valid point, but it begs the question: How do we know that the parent class code is bulletproof in the first place? Furthermore, how do we keep it bulletproof on those occasions when the parent class code does need to be maintained?

In both cases, the answer is to formalize a *unit test* for the object, and then to automate it with a *test driver*. (Note: Here, a *unit* is merely a generic term for a portion of code, such as an object class or an individual function. It's not to be confused with a Pascal *unit*, i.e. a source code file). Such a test driver would handle path coverage as well as boundary condition testing for us by exercising the code, calling it repeatedly under different situations.

Unit Testing of Simple Functions

For a simple function — one that takes a given number of arguments and returns a single result value without affecting anything else — writing a test driver is straightforward. You just need to make a series of calls to the function, passing it different combinations of arguments each time, comparing the result to an expected value.

```
function TitleToSortable(sText: string): string;
var
  iNumeral: integer;
begin

  { Convert to upper case. }
  sText := UpperCase(sText);

  { Strip leading A, AN, THE. }
  if Copy(sText,1,2) = 'A ' then
    begin
      Delete(sText,1,2);
      sText := sText + ', A';
    end
  else
  if Copy(sText,1,3) = 'AN ' then
    begin
      Delete(sText,1,3);
      sText := sText + ', AN';
    end
  else
  if Copy(sText,1,4) = 'THE ' then
    begin
      Delete(sText,1,4);
      sText := sText + ', THE';
    end;
  { Convert numerals to words. }
  result := '';

  repeat
    iNumeral := Pos(sText[1],'0123456789')-1;
    if iNumeral >= 0 then
      begin
        Delete(sText,1,1);

        case iNumeral of
          0: result  := result + 'ZERO';
          1: result  := result + 'ONE';
          2: result  := result + 'TWO';
          3: result  := result + 'THREE';
          4: result  := result + 'FOUR';
          5: result  := result + 'FIVE';
          6: result  := result + 'SIX';
          7: result  := result + 'SEVEN';
          8: result  := result + 'EIGHT';
          9: result  := result + 'NINE';
        end;
      end;
  until iNumeral<0;

  result := result + sText;

end;
```

Figure 3: A function to convert book titles so they can be properly sorted.

For example, let's continue with the idea of developing an application for use by a librarian. This librarian will be entering book titles into a database exactly as they appear on the title page. The system will then need to produce reports sorted by title. To accomplish this, you decide that you need a function to translate a given title into a sortable version (see Figure 3). It will have to move the words "A", "An", and "The" to the end of the title, and make the whole title upper- case. Thus, "The Programmer's Bible" becomes "PROGRAMMER'S BIBLE, THE".

Furthermore, any leading numerals in the title should be translated to words, so that "3 Tier Solutions" will be sorted alongside "The Three Most Common Mistakes Made By C++ Programmers." For now, the routine only handles one digit at a time (e.g. translating "30" to "threezero" as opposed to "thirty"), which the user agreed would be good enough for the first phase implementation.

A minimal test sequence for this function will ensure that each line of code is executed at least once, satisfying the conditions of each of the three **if** statements (A, An, The). It will also test for boundary conditions such as a lengthy title, a brief title, and an empty string. Such a test driver, written from scratch, might resemble the listing in Figure 4.

Building a Unit-Test Tool Kit

A more generalized approach might be to write a function, which we'll call *QA_sAssert*, that is responsible for comparing the actual and expected results (of type string). It would report positive as well as negative findings, logging both to a memo field (or to a disk file as we'll see in the next installment), and displaying the complete values of the actual and expected results (see Figure 5).

A test driver that would take advantage of this procedure might then resemble the code shown in Figures 6 and 7. The results of running such a test driver are shown in Figure 8. The test driver code might be located in the same source code file (.PAS) as the function being tested, or it might be kept in a separate unit with other test drivers. The *QA_Start* and *QA_Log* procedures are defined in Figure 9.

To run the test driver, you simply need to create a form called `formQALog` that contains a memo field named `memoQALog` and a button that runs the test driver. If you have more than one test driver to run, you can add a button for each one, or use one button plus a radio button group (as shown in Figure 8), or some other control structure.

Conditional Compiler Directives

It's recommended that the test driver code be placed in the same source code file (unit) as the functions that are being tested. This makes for a handy means to develop and maintain the test drivers in sync with the functions.

Unfortunately, Delphi executables are large enough already. So, we really don't want to see the test driver code adding to the bulk when the final program is released. Fortunately, the Delphi compiler supports conditional compiler directives, which allow us to specify whether certain portions of Object Pascal code are to be compiled.

Compiler directives take the form of special comments. They are enclosed in braces (`{}`) like all other comments, and begin with a dollar sign (`$`). The three directives of interest here are: `{$DEFINE x}`, `{$IFDEF x}`, and `{$ENDIF}`, where `x` is any identifier. Code that appears between an `{$IFDEF x}` directive and a corresponding `{$ENDIF}` directive will only be considered part of the source code by the compiler if the `x` identifier was previously defined.

```
procedure Test1;
begin
  if sTitleToSortable('The Programmer''s Bible') <>
     'PROGRAMMER''S BIBLE, THE' then
    MessageDlg('Error: case = The',
               mtInformation,[mbOk],0);
  if ...
  if ...
end;
```

Figure 4: A test driver that might be written without the QA tool kit. It could be called from a single-button form.

```
procedure QA_sAssert(sActual, sExpected: string);
var
  sLine: string;
begin
  if (sActual = sExpected) then
    begin
      sLine := '= ' + sActual;
      formQALog.memoQALog.Lines.Add(sLine);
    end
  else
    begin
      sLine := 'X ' + sActual;
      formQALog.memoQALog.Lines.Add(sLine);
      sLine := ' ' + sExpected;
      formQALog.memoQALog.Lines.Add(sLine);
    end;
end;
```

Figure 5: The *QA_sAssert* procedure. Calling it declares an assertion that the two arguments should be equal. Comparison results are added to a memo field in the QALog form. *QA_sAssert* only compares strings. Another procedure (e.g. *QA_iAssert*), would be needed to compare integer values, and yet another for real numbers, and so on.

```
procedure QA_SortableTest0;
var
  sTitle1, sTitle2: string;
  iCount: integer;
begin
  QA_Start('Title');
  QA_Log('TitleToSortable()');
  QA_Log('-----------------');

  { Path Coverage: A, An, The, numerals, none. }
  QA_sAssert(TitleToSortable('The Programmer''s Bible'),
             'PROGRAMMER''S BIBLE, THE');
  QA_sAssert(TitleToSortable('Personal Computers'),
             'PERSONAL COMPUTERS');
  QA_sAssert(TitleToSortable(
             'A Guide to Quality Assurance'),
             'GUIDE TO QUALITY ASSURANCE, A');
  QA_sAssert(TitleToSortable('3 Tier Solutions'),
             'THREE TIER SOLUTIONS');
  QA_sAssert(TitleToSortable('50 Tips & Trick'),
             'FIFTY TIPS & TRICKS');
  QA_sAssert(TitleToSortable('An Overview of Pascal'),
             'OVERVIEW OF PASCAL, AN');
end;
```

Figure 6: A path coverage test driver that uses *QA_sAssert*.

```
{ Boundary Conditions: empty, short, and long titles. }
QA_sAssert(TitleToSortable(''),'');
QA_sAssert(TitleToSortable('A'),'A');
QA_sAssert(TitleToSortable('b'),'B');

sTitle1 := 'A Long Title';
sTitle2 := 'LONG TITLE, A';

for iCount := 1 to 10 do
  begin
    insert('Very ',sTitle1,3);
    insert('VERY ',sTitle2,1);
  end;

QA_sAssert(TitleToSortable(sTitle1),sTitle2);
```

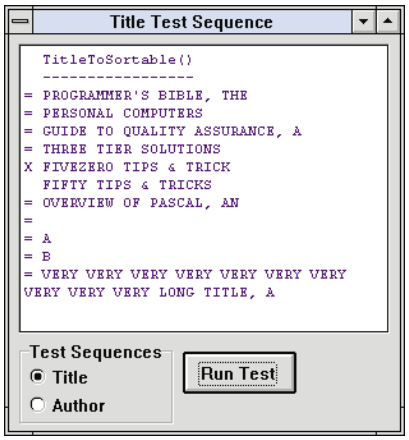


Figure 7 (Top): Additional test driver code to check for boundary conditions.
Figure 8 (Left): A test driver control panel showing the results of one of the two test sequences that have been programmed. An equal sign (=) denotes tests that have passed, where the actual and expected results agree. An "X" denotes failed tests, where the actual result on the first line disagrees with the expected result on the next line.

So, to control whether the test driver code is compiled, use something like the code in Figure 10 which uses `{$IFDEF QA_MODE}` ... `{$ENDIF}` to enclose code that should only be compiled during testing.

Then, to turn off `QA_MODE` (thus causing the compiler to ignore the test driver code as if it were commented out) simply disable the `$DEFINE` directive. To do this, you can remove the dollar sign (making it an ordinary comment), or change it into an explicit `$UNDEFINE` directive.

Finally, you will find that as the number of units in your Delphi project grows, it can become unwieldy to revisit the `$DEFINE` directives at the top of each file, converting them each into `$UNDEFINE`s and back again.

Another directive, `{$I FileName}`, tells the compiler to logically include the source code contained in a file called `FileName` as if it were physically present in the source code file that specified the `$I` directive. You could thus replace all the `$DEFINE` directives in each unit with an `$I` directive that names a new file (e.g. PROJDEF.PAS). This new file would then contain a single `$DEFINE` statement that, when changed into an `$UNDEFINE`, would automatically affect all units at once.

```
procedure QA_Start(sTestName: string);
begin
  formQALog.Caption := sTestName + ' Test Sequence';
  with formQALog.memoQALog do
    begin
      Lines.Clear;
      Font.Color := clNavy;
      Font.Name  := 'Courier New';
      Font.Size  := 8;
    end;
end;

procedure QA_Log(sMessage: string);
begin
  formQALog.memoQALog.Lines.Add(' ' + sMessage);
end;
```

```
unit UnitName
{$DEFINE QA_MODE}
...

implementation

  < program source code here >
{$IFDEF QA_MODE}
  < test driver code here >
{$ENDIF}

 end.
```

Figure 9 (Top): The *QA_Start* procedure initializes the memo field log for a new test run. *QA_Log* adds information directly into the log (as opposed to going through *QA_sAssert*, for example).
Figure 10 (Bottom): Using conditional compiler directives to control excluding test code from a final build of the program.

Conclusion

With the proper tools and proper planning, assuring software quality can be relatively painless. We've seen how to systematically develop a comprehensive test plan, some of which appears in the form of test driver code that is machine-executable, and therefore easily repeatable, and we began to build a QA tool kit for testing Delphi code.

Future installments will add to the tool kit and discuss other, commercially-available testing software that can be used with Delphi programs. Δ

The demonstration project referenced in this article is available on the Delphi Informant Works CD located in INFORM\96\MAR\DI9603QA.

Craig L. Jones is a contract software engineer in Southern California with over 14 years of programming and consulting experience. He is a charter member of the San Diego Delphi Users Group and is currently serving as a SIG leader for the Orange County Delphi Users Group. He is also a member of Team Borland, supporting Paradox and Delphi on the GEnie network. Mr Jones can be reached at craig.jones@genie.geis.com, or on CompuServe at 71333,3515.

By Karl Thompson

A Table Documentor

Documenting Database Tables with a Delphi Utility

Delphi is a wonderful tool, but it does have a few omissions. For example, Delphi cannot print table structures. The Database Desktop allows you to view a table's structure, but cannot print it for you. Of course, you could always save the structure information to a table, and then create and print a report of its contents.

The Table Documentor utility presented in this article (see Figure 1) does just that. It can display or print Paradox or dBASE table structures with just a few mouse clicks.

Furthermore, it isn't limited to showing or printing just one structure at a time. The user can navigate the hard disk, select a number of tables (within memory limits), and the Table Documentor will analyze and process all the tables. This makes the utility particularly handy for comparing table structures.

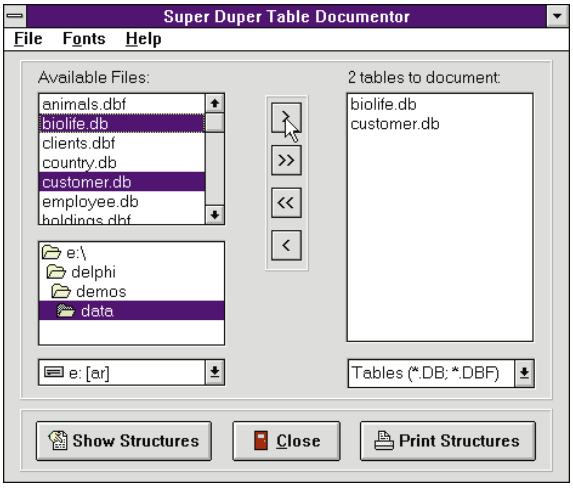


Figure 1: The Table Documentor at run time.

Using the Utility

Before viewing its code, test the program. Due to Windows user interface conventions, the program is easy to use, but you should note some of its more subtle features. For example, you can select files using one of three techniques:

- Double-click on each file name to add it to the list box on the right. (There is no limit to the number of files that can be documented at one time.)
- Click on a file and hold Ctrl or Shift while continuing to select files. When all the appropriate files are selected, click on the > button. The selected files will move to the ... **tables to document** list.
- Click the >> button and all the tables in the current directory will be documented. If the directory contains Paradox and dBASE tables, you may limit the selection to one or the other via the pick list above the **Print Structures** button. (Figure 2 shows the structures of two tables being displayed.) You can follow similar steps to deselect files.

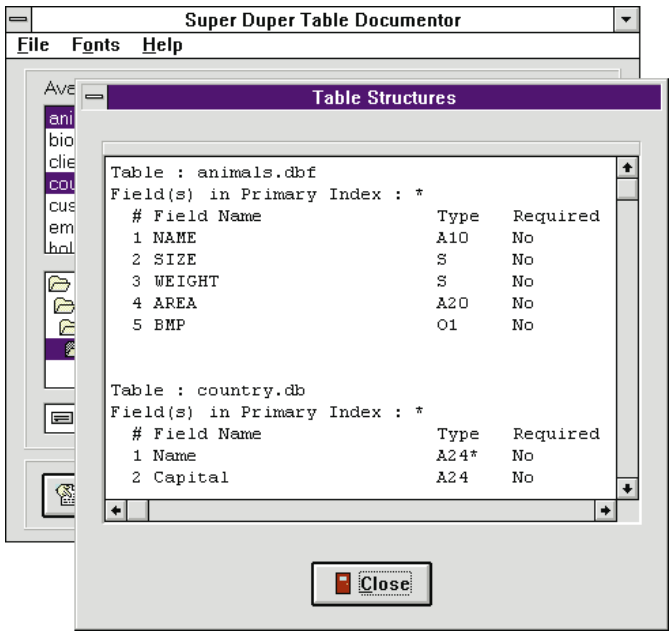


Figure 2: *Form2* (Structure Form) in action.

By default, the Table Documentor does not save the path to the file. However, to document tables in different directories, the Table Documentor must keep track of the tables' locations. You can change the default behavior under the **File** menu. When **Save Path** is **ON**, the path will be prepended to the file name. (Storing the table's path is optional since if the files are nested several directories deep, then the table name may not always be visible in the list box on the right. For aesthetic reasons, I did not want to widen the main form.)

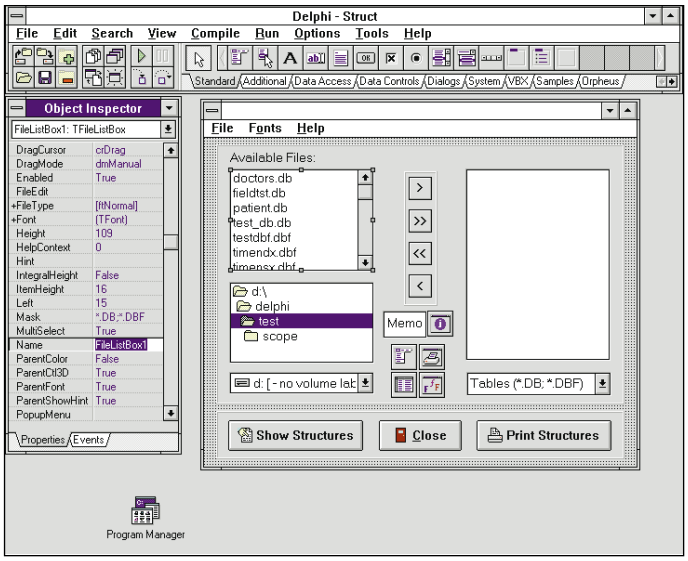
Another **File** menu option should also be noted. When **Form Feed** is **ON** (the default), each structure begins printing on a new page. When **Form Feed** is **OFF**, the Table Documentor prints continuously, regardless of page breaks.

A Look at the Code

Many of the classes, properties, and techniques used by the Table Documentor have been thoroughly discussed in previous issues of *Delphi Informant* and do not require further explanation. In this article, we'll:

- closely examine the *FieldDef* and *IndexDefs* properties of the *TTable*. They provide all the information we need about a table's structure.
- concentrate on how to handle printing using only Object Pascal, thus eliminating the complication of using ReportSmith or another report writer.
- discuss how to provide users with simple, online help that does not require the use of the Window's Help engine. This can be handy for small applications such as the Table Documentor, since supplying or installing a .HLP file is unnecessary.

The Table Documentor consists of four forms. The Main Form is the interface for the user to select tables for documenting (see Figure 3). *Form2* has a *TMemo* component containing the text that describes the structure of the tables (see Figure 4). *Form3* contains



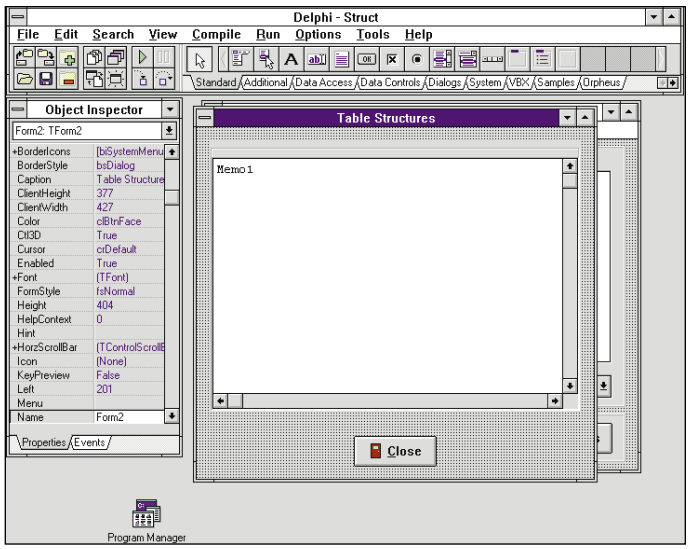


Figure 3 (Top): *Form1* in design mode.
Figure 4 (Bottom): *Form2* at design time.

another *TMemo* component for displaying the help text. A fourth form is used for an About box. (Note: This About box is not a standard Delphi component — it's freeware written by Jeff Atwood. For more information, see the downloadable message at the end of the article.)

Form1 is the most complex form in this project. It uses 14 components arranged on three *TPanels*. *Panel1* contains two *TLabels*, as well as a *TFileListBox*, *TDirectoryListBox*, *TDriveComboBox*, *TListBox*, and *TFilterComboBox*. *Panel2* is placed on *Panel1* and contains four *TSpeedButtons*. A third *TPanel* contains three *TBitBtns*. Additionally, four non-visual components are used: *TMainMenu*, *TPrintDialog*, *TTable*, and *TFontDialog*. The *TMemo* component that you see in design mode has its *Visible* property set to *False* (we'll discuss this more later).

The program's main functionality is contained within the *GetStructures* function (see Figure 5). A table's structure is analyzed in *GetStructures* and it's called whether the structure is being displayed or printed.

```
function TForm1.GetStructures(var M: TMemo) : Boolean;
const
  KeyFieldIndicator: String1 = '*';
var
  TblCnt, FldCnt, X, Y: integer;
  SizeStr: String3;
  KeyChar: String1;
  PrimIndxStr: string;
begin
  Result := True;
  Screen.Cursor := crHourglass;
  try
    with M, Table1 do begin
      TblCnt := ListBox1.Items.Count - 1;
      Text   := ''; { Remove ListBox's default string }
      PrimIndxStr := 'None';

      for X := 0 to TblCnt do begin
        { 'Add' methods will raise an exception if not
          enough memory }
        DatabaseName    := FileListBox1.Directory;
        TableName       := ListBox1.Items[ X ];
        { Update raises an expection if index can't be
          found or is unknown }
        { For example: TTable doesn't understand
          Rock-e-t's NSX indexes }
        FieldDefs.Update;
        IndexDefs.Update;
        for Y := 0 to IndexDefs.Count - 1 do
          { Find primary index }
          if ixPrimary in IndexDefs.Items[Y].Options then
            { Save the fields in the index }
            PrimIndxStr := IndexDefs.Items[Y].Fields;
        { Items uses a 0 offset }
        FldCnt:= FieldDefs.Count - 1;
        Lines.Add('Table : ' + TableName);
        Lines.Add('Field(s) in Primary Index : ' +
                  KeyFieldIndicator);
        Lines.Add(format('%3s %-25s %-6s %-s',
                  ['#','Field Name','Type','Required']));
        for Y := 0 to FldCnt do begin
          if (FieldDefs.Items[Y].Size > 0) then
            SizeStr := IntToStr(FieldDefs.Items[Y].Size)
          else
            { Size 0 makes no sense for Date type, etc. }
            SizeStr := '';
          if (Pos(FieldDefs.Items[Y].Name,
                  PrimIndxStr) = 0) then
            KeyChar := ' '
          else
            { Indicate keyed field }
            KeyChar := KeyFieldIndicator;
          Lines.Add(format('%3d %-25s %1s%-5s %-3s',
                    [FieldDefs.Items[Y].FieldNo,
                    FieldDefs.Items[Y].Name,
                    FieldTypeToAbbrevstring(
                    FieldDefs.Items[Y].DataType),
                    SizeStr+KeyChar,
                    BooleanToYNstring(
                      FieldDefs.Items[Y].Required)]));
        end; { Inner field loop }

        { Don't FormFeed after last table, or if
          menu's FormFeed is not selected }
        if Printing              and
           X < TblCnt            and
           FormFeed1.Checked     then
          begin
            Lines.Add(#12); { FormFeed character }
          end
        else
          begin
            Lines.Add('');  { Blank lines }
            Lines.Add('');
          end;
      end; { Outer table loop }
    end;
  finally
    Screen.Cursor := crDefault;
  end;
end;
```

Figure 5: The *GetStructures* function.

In general, Pascal-type strings (managed by a *TString* object) are being initialized so that one string contains detail information about each field in a table. There are also three header strings that store the table's name, a legend row, and a title row. The string object that is being initialized is a property of a *TMemo* object.

The *GetStructures* function contains an outer loop and two inner loops that are not nested. The outer loop is used to iterate through each table in the *ListBox1*, and to initialize or update four *TTable* properties. The current table being documented is assigned to the *Table1.TableName* property. Its path is assigned to *Table1.DatabaseName*.

Table1.FieldDefs and *Table1.IndexDefs* are the two key properties that provide information about a table's structure. The *FieldDefs* object is of type *TFieldDefs* and contains two properties: *Count* stores the total number of fields in the table; and *Items* is an array of pointers to *FieldDef* objects. The *FieldDef* objects are of type *TFieldDef* and define properties such as *DataType*, *FieldNo*, *Name*, and *Size*. These hold information about the individual fields. To initialize each *FieldDef* and subsequently *Table1.FieldDefs*, a call must be made to the *FieldDefs.Update* method.

At this point, you may wonder why the *TTable.Field* property was not used to gather the structure information. The reason lies in the way *Field* and *FieldDefs* are initialized. It's important to note that *FieldDefs* does not require the underlying table to be open when it's initialized.

FieldCount's value could have been easily used to initialize an index for iterating through *Table1.Field* — which would obtain field names and datatypes. However, this requires that *Table1* be opened (i.e. that *Table1.Active* be set to *True*) and obviously, opening the table is not always possible.

Because updating *FieldDefs* does not require the table to be open, the Table Documentor can always show the table's structure regardless of another application's interaction with the table.

The second *TTable* property that we are most concerned with is *IndexDefs*, an object of type *TIndexDefs*. This class has the same two properties as *TFieldDefs*, but in this case the *Items* property is an array of pointers to instances of *TIndexDef*. One instance of *TIndexDef* is created for each table index. *TIndexDef.Field* is a string containing the name of all the fields that comprise the index. The field names are separated by a semicolon.

Again, there may be a more direct approach to obtaining information about the table's primary index. However, as with *TTable.Fields*, if we had used the *IndexFields* property instead of *IndexDefs*, we would have been forced to work with an open table.

Additionally, *IndexFields* only includes the names of the fields in the currently active index. If the primary index is not the active index, more work is needed to make the primary index the active index.

Another advantage of using *IndexDefs* is that only minimal work is needed to increase the Table Documentor's functionality so it can include the names of all the indices and fields that are used in each index.

Like *FieldDefs*, *IndexDefs* requires a call to its *Update* method to initialize each *IndexDef* object, and itself. Once *FieldDefs* and *IndexDefs* are initialized, the first of the two inner loops determines which table's index is the primary index. The *IndexDefs.Items[n].Options* property is evaluated to see if it's of type *ixPrimary*. If so, the field names comprising the index are assigned to *PrimIndxStr*.

Questions and Answers

Next, we must store the text strings that will be printed or displayed. But first, we need to answer this question: What is the best way to store, display, and print the strings?

The *TMemo* component on the Standard page of the Component Palette provides the answer. Many Delphi programmers may not realize that a Memo component has two properties for storing text — *TMemo.Text* and *TMemo.Lines*.

For our purposes, the *Text* property is too restrictive because the maximum length of its string is 255 characters. Conversely, the *Lines* property (a *TStrings* object) has no such limit, and initializing the strings with the *Add* method is easy. Therefore, the Table Documentor uses *TMemo.Lines* for string storage.

Before entering the final inner loop, three lines of header information are prepared. The first contains the table's name and path. The second describes the meaning of the *KeyFieldIndicator*. The third call to *Add* initializes the column title line.

The string's display to the user is controlled by the call to *Format*. If you are unfamiliar with Delphi's *Format* function, the online help provides a good overview of its use (search on "Format Strings"). (Note that when using *Format*, any white space included in the format specifier list is interpreted literally.)

In the final loop, two tests are performed to obtain information about the field before the string is concatenated. The first is performed so that if the data field's size is returned as a zero, a blank can be displayed rather than the illogical zero value. The second performs a call to the *Pos* function. If the name of the current field is in the *PrimIndxStr* then *Pos* finds the match and returns the position of the first character. Therefore, for values greater than zero indicating that the field is included in the primary key, the *KeyChar* is initialized with the *KeyFieldIndicator*.

After these two tests, the string is ready to be built and added to *Memo1.Lines*. One more call is made to *Format* to concatenate the string that is used for the line containing the field information.

Printing Table Structures

Often, adding printing functionality to a program is a dreaded task because, at best, it tends to make for tedious programming. At worst, it adds a lot of bulk to a program.

```
{ This function is based on code from
  "Delphi Developer's Guide" by Xavier Pacheco &
  Steve Teixeira [Borland Press, 1995], pp. 297. }
function TForm1.PrintStructures(var M: TMemo) : Boolean;
var
  X: integer;
  StructText: TExtFile;
begin
  Result := True;
  if (M.Lines.Count = 0) then begin
    Result := False;
    MessageDlg('No tables selected for documenting...',
               mtInformation, [mbOK], 0);
    Exit;
  end;
  { If PrintDialog1.Execute then we could give user
    option to setup printer }
  AssignPrn(StructText);
  try
    Rewrite(StructText);
    Printer.Canvas.Font.Assign(M.Font);
    for X := 0 to M.Lines.Count - 1 do
      writeln(StructText, M.Lines[X]);
  finally
    CloseFile(StructText);
  end;
end;
```

Figure 6: The *TForm1.PrintStructures* function.

While the technique used by the Table Documentor may not be suitable for all projects, it eliminates the arduous aspects of preparing reports. In addition, the 24-plus line function that handles the printed output does not add any appreciable bulk to the program.

As we've seen, *GetStructures* initializes a *TMemo* component's *Lines* property with data that describes a table. Now that the *Lines* object is properly formatted, our final task is to output each string's value to the printer. This is accomplished by the *PrintStructures* function (see Figure 6). *PrintStructures* depends upon the Printers unit being listed in the program's **uses** clause. (Note that the code does not use the *AssignPrn* procedure declared in the WinPrn unit.)

After some initial data integrity checking, the function assigns a text file to the current window's default printer:

```
AssignPrn(StructText);
```

AssignPrn simply tells Windows that any output written to the text file should be sent to the printer using the *TPrinter.Canvas.Font* property. For this to happen, there must be a call to *Rewrite* that will create and open a new file. If the file exists, it will be overwritten. (*Canvas.Font* is initialized in *Form1's FormShow* event handler to the fixed pitch font set at design time in the *TFontDialog* component.)

The *Writeln* statement accomplishes printing. It optionally accepts the name of a text file variable for its initial parameter and then writes all its output to that file. Next, a **for** loop is used to iterate through each string in the *Memo1.Lines* object and send the value of the strings to the printer via the *StructText* text file.

As mentioned earlier, the *GetStructures* function handles all the formatting. Note that the *TMemo's WordWrap* property was set to *False* at design time.

Online Help

Another aspect of the Table Documentor deserves review. It's often nice to provide the user with some simple online help. Unfortunately, the application does not warrant the time required to create a Windows Help (.HLP) file complete with keywords and hypertext links.

One way to handle this is to write a series of string constants that provide the necessary help, and then use a *TMemo* component to display the strings. The Table Documentor uses this technique.

The code from the *Info1Click* procedure in the UStruct unit is shown in Figure 7. It initializes *Form3* which contains the *TMemo* component that displays the help text. The text string's constants are actually declared in the UHelp unit. A similar technique for initializing the *Form3.Memo1*'s *Lines* property was used in *GetStructures*.

```
procedure TForm1.Info1Click(Sender: TObject);
begin
  { Basic instructions: simple programs don't require
    help files}
  { Create form only as needed so as not to take up
    system resources }
  Form3 := TForm3.Create(Application);
  { Initialize font }
  Form3.Memo1.Font := Form2.Memo1.Font;
  Form3.ShowInstructionsModal;
  { Release resources }
  Form3.Free;
end;
```

Figure 7: The *TForm1.Info1Click* procedure.

A Note About Installation and the BDE

Borland insists that the complete BDE and its installation utility is distributed with all Delphi database applications. However, you may want to perform a partial install of the BDE when installing a program (for your use) requiring the BDE. Therefore, to add the Table Documentor to your arsenal of programming utilities, the entire BDE is not required.

Assuming the BDE is not installed, to use the Table Documentor, you must create a directory into which you install the DLLs that make up the BDE, e.g. C:\IDAPI. Then edit WIN.INI to add an `[IDAPI]` group and add one line to that group:

```
DLLPATH=C:\IDAPI
```

Then copy the following files to the directory you just created and assigned to the DLLPATH variable:

- IDAPI01.DLL
- IDR10009.DLL
- ILD01.DLL
- IDAPI.CFG (with Delphi's minimal default values)
- IDPDX01.DLL (Paradox access)
- IDDBAS01.DLL (dBASE access)

After rebooting, the Table Documentor can be used without a complete BDE installation. These six files only occupy about 1MB (uncompressed) as opposed to the entire BDE that occupies over 2MB.

Concluding with a Challenge

A how-to article should also challenge the user. Assuming that no program is ever complete, what features could be added to enhance the Table Documentor? If I were to write v2.0, it would report more details about a table's properties. It would be cool to know about validity checks, table lookups, and secondary indices, to name a few.

I would also alter the interface slightly by using a tabbed notebook metaphor. One page would contain the components on the current main form and the other page would contain the memo object showing the table structures. When the user clicks on **Show Structures**, the code would change pages. I think that this would be slicker and cause less screen clutter.

Additionally, the form could be minimized and later opened to reference the structure listings without having to locate the child form among all the open windows. If anyone makes these modifications, please e-mail a copy of your efforts to me. Thanks! Δ

Figure 6 is based upon Object Pascal code shown on page 297 of *Delphi Developer's Guide* by Xavier Pacheco and Steve Teixeira [SAMS, 1995].

The sample Table Documentor application described in this article, and the About box freeware written by Jeff Atwood, are available on the Delphi Informant Works CD located in INFORM\96\MAR\DI9603KT.

Karl Thompson is an independent Delphi and Paradox developer serving clients from New York City to Philadelphia. He has been working with Borland's Pascal development environments since 1984. He can be reached at (609) 470-1430, or on CompuServe at 72366,306 (Internet: 72366.306@compuserve.com.).

By Bill Todd

Error Handling: Part II

A Closer Look at Delphi's Exception Handling Functionality

Last month, we began this series by discussing the basics of Delphi's exception handling mechanism. Our topics included allocating resources and ensuring proper clean-up, trapping for RTL exceptions, and understanding the RTL exception hierarchy.

This month, we'll take a closer look at exception handling in Delphi by covering these subjects:

- Understanding the scope of exceptions
- Reraising exceptions
- Using the *Exception* object
- Using silent exceptions
- Creating a custom exception
- Creating a custom default exception handler

Understanding Exception Scope

You must understand the scope of exceptions to determine where you need **try..except** blocks in your application, and what will happen if you do not handle an exception. Consider the procedures shown in Figure 1.

```
procedure procB;
var
  i: Integer;
begin
  i := 2 div 0;
end;

procedure procA;
begin
  try
    procB;
  except
    on EIntError do
      MessageDlg('Integer math error.',mtError,[mbOK],0);
  end;
end;
```

Figure 1: The *procA* and *procB* procedures.

When *procA* calls *procB*, an integer math exception will be raised by the attempt to divide by zero. When an exception is not handled in the block where it occurs, execution in the block that caused the exception ends and control returns to the block that called it. In this case, when the divide by zero exception occurs in *procB*, control returns to *procA* and execution resumes with the exception still raised. Therefore, the code in the **except** clause in *procA* will execute.

Put another way, exceptions "travel" through each nested **try..except** block, moving back up the procedure and function calling chain until they are handled by one of your **try..except** blocks, or until they reach Delphi's default exception handler.

You can provide a default exception handling mechanism in your Delphi application. Doing so, however, is dangerous. In Figure 2, note that an **else** clause appears in the **except** block. The **else** clause will handle all exceptions other than the integer math exceptions. These are handled by the **on** `EIntError` **do** statement. Since you may not know how to handle every type of exception safely, you almost certainly do not want to write a **try..except** block with an **else** clause.

```
procedure procA;
begin
  try
    procB;
  except
    on EIntError do
      MessageDlg('Integer math error.',mtError,[mbOK],0);
    else
      MessageDlg('Some other exception occurred.',
                 mtError,[mbOK],0);
  end;
end;
```

```
procedure procA;
begin
  try
    procB;
  except
    on EIntError do
      MessageDlg('Integer math error.',mtError,[mbOK],0);
    else
      begin
        MessageDlg('Some other exception occurred.',
                   mtError,[mbOK],0);
        raise;
      end;
  end;
end;
```

Figure 2 (Top): This version of *procA* attempts to handle all errors — a bad idea. Excluding math integer errors (these are handled by the **on..do** statement), the **else** clause in the *procA* procedure handles all exceptions. **Figure 3 (Bottom):** This version of *procA* uses a **raise** statement to reraise an exception.

Reraising Exceptions

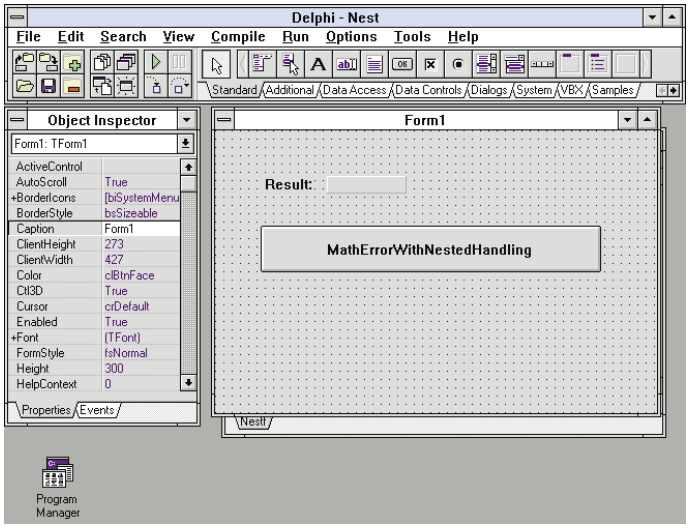
When you handle an exception with a **try..except** block, the exception ends *there* because Delphi destroys the exception object.

However, you can raise the exception again with the keyword **raise**. For example, the code in Figure 3 contains an **else** clause that displays a message for all exceptions except integer math exceptions. However, after the message dialog box has been displayed, the **raise** statement "reraises" the exception so it will be handled again further up the calling chain.

The ability to reraise an exception lets you easily write nested **try..except** blocks without having to duplicate code. You can provide specific handling in the inner block and reraise the exception to let the more generic code in the outer **try..except** block execute. This technique is demonstrated in the sample project NEST.DPR. Its main form is shown in Figure 4, and code from the *OnClick* event of the form's Button is shown in Figure 5.

This *OnClick* event involves a two-step calculation. The value assigned to the variable `k` must be 23 if an error occurs in the first step of the computation, and 0 (zero) if an error occurs in the second step. Therefore, each of the computation statements is enclosed in its own **try..except** block that sets `k` to the proper value if an error occurs and reraises the exception.

The outer **try..except** block handles displaying the correct message so that the message display code does not have to be duplicated in the **try..except** for each step in the calculation.




```
procedure TForm1.MathErrorWithNestedHandlingClick(
  Sender: TObject);
var
  i,j,k,l,m: Integer;
begin
  i := 23;
  j := 0;
  l := 2;
  m := 4;
  try
    try
      k := i + j;
    except
      on EIntError do
        begin
          k := 23;
          raise;
        end;
    end;
    try
      k := k + i div j * (l div m);
    except
      on EIntError do
        begin
          k := 0;
          raise;
        end;
    end;
  except
    on EDivByZero do
      MessageDlg('Divide by zero error',mtError,[mbOK],0);
    on EIntError do
      MessageDlg('Integer math error.',mtError,[mbOK],0);
  end;
  { Display the result. }
  Result.Caption := IntToStr(k);
end;
```

Figure 4 (Top): The main form of NEST.DPR.
Figure 5 (Bottom): Implementing nested **try..except** blocks.

Using the *Exception* Object

One of the most useful features of Delphi's exception mechanism is that you can access the *Message* field in the *Exception* object using a special form of the **on..do** syntax in a **try..except** block.

The sample project IOERR.DPR demonstrates this. Figure 6 shows the project's main form. It has two buttons, each of

which contain code that tries to open a non-existent file. Figure 7 shows the *OnClick* event handler for the **I/O Error** button. This code tries to open the non-existent file FOO.TXT. In the **except** block, the **on..do** statement:

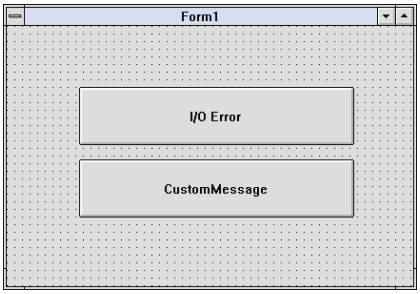


Figure 6: The main form of the sample IOERR project.

```
on Ex: EInOutError do
```

performs two functions. First, it traps the *EInOutError* exception. Second, **on..do** initializes the variable *Ex* to give you access to the *Exception* object. The type of variable *Ex* is the type of the *Exception* object you are trapping, *EInOutError* in this case.

In the *MessageDlg* call, *Ex.Message* inserts the message text so that the standard message text for this error is displayed in the custom dialog box. This field is available for all exceptions. In addition, *Ex.ErrorCode* is used to add the operating system error code to the message. *ErrorCode* is available in *EInOutError* exceptions only.

If that was all you could do with the message text it would not be useful. However, you can change the text of the message and reraise the exception. For example, look at the code from the **CustomMessage** button (see Figure 8). In this case, the code changes the text of the *Message* property and reraises the exception.

Thus the standard exception dialog box will appear, but with a custom, and more descriptive, message. This allows you to take full advantage of the default exception handling in Delphi and still easily provide custom error messages that will be more helpful to your users and yourself.

There is one final piece of information that you can use when working with an exception. When an exception is raised, the variable *ErrorAddr* in the System unit is set to the address in your code where the error occurred. You can display this information to the user just as the default exception handler does.

Using Silent Exceptions

As you have seen, the default exception handling mechanism in Delphi displays a dialog box that contains an error message. However, there is another class of exception, *EAbort*, that does not display an error dialog box. This is called a *silent* exception, and it's so useful that a special procedure called *Abort* is provided so you can easily raise a silent exception in your code.

In database applications, the most common use for *Abort* is to block an event. An example of this is shown in Figure 9. In this case the code ensures that if the State field value is two characters long, and the Country field does not contain "America", the record cannot be posted. The call to *Abort* prevents the record from being posted by raising a silent exception.

```
procedure TForm1.IOErrorBtnClick(Sender: TObject);
var
  TestFile: TextFile;
begin
  AssignFile(TestFile,'foo.txt');
  try
    Reset(TestFile);
  except
    on Ex: EInOutError do
      begin
        MessageBeep(0);
        MessageDlg(Ex.Message + '  Code: ' +
                   IntToStr(Ex.ErrorCode),
                   mtError,[mbOK],0);
    end;
  end;
end;
```

```
procedure TForm1.CustomMessageBtnClick(Sender: TObject);
var
  TestFile: TextFile;
begin
  AssignFile(TestFile,'foo.txt');
  try
    Reset(TestFile);
  except
    on Ex: EInOutError do
      begin
        Ex.Message :=
          'The file FOO.TXT could not be found';
        raise;
      end;
  end;
end;
```

```
procedure TCustForm.CustTblBeforePost(DataSet: TDataset);
var
  State: string;
begin
  State := CustTbl.FieldByName('State').AsString;
  if (Length(State) = 2) and
     (CustTbl.FieldByName('Country').AsString <>
     'America') then
    begin
      MessageDlg('Country must be America for this State.',
                 mtError, [mbOk], 0);
      Abort;
    end;
end;
```

Figure 7 (Top): The **I/O Error** button's *OnClick* event handler.
Figure 8 (Middle): Code attached to the **CustomMessage** button.
Figure 9 (Bottom): In this example, the *Abort* procedure blocks a record from being posted by raising a silent exception.

Creating Custom Exceptions

You can also create custom exception objects and raise them to handle error conditions. The code in Figure 10, from the sample CUSTOM.DPR project, shows how to do this. Figure 11 shows the application's main form.

Exceptions are objects and you can raise any object as an exception. However, there is no reason to do so. Delphi's exception handling mechanism will only handle exception objects that are descendants of *Exception*. Therefore, you should derive your exceptions from the *Exception* base class.

```
implementation

{$R *.DFM}

type
  ETestError = class(Exception);

procedure TForm1.CustomExceptionClick(Sender: TObject);
begin
  try
    raise ETestError.Create('This is a custom exception');
  except
    on ETestError do
      MessageDlg('A custom exception occurred at ' +
                 IntToHex(Seg(ErrorAddr),4) + ':'  +
                 IntToHex(Ofs(ErrorAddr),4),
                 mtError,[mbOK],0);
  end;
end;
```

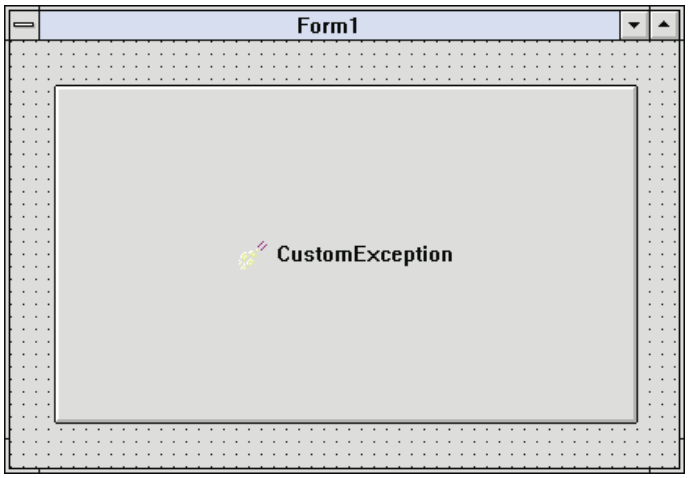


Figure 10 (Top): Creating a custom exception handler.
Figure 11 (Bottom): The main form for the sample CUSTOM project.

The code in Figure 10 derives a new exception, *ETestError*, from the *Exception* class in the **type** declaration at the beginning of the **implementation** section of the unit. (Note that your custom exception **type** declaration must be global to the unit.)

To raise your custom exception, use the **raise** command to raise the exception and call its constructor, as shown here:

```
raise ETestError.Create('This is a custom exception');
```

Raising the exception causes the **try..except** block to handle the exception and display the custom error dialog box. Note the use of the *ErrorAddr* variable to display the address where the exception was raised.

Delphi's components handle errors with their own exceptions and you can trap and handle those exceptions just as you can run-time library exceptions.

Creating a Custom Default Exception Handler

So far this discussion of error handling has been limited to handling errors where they occur in your code. If an error occurs that you do not explicitly trap for in your code, it's handled by the default exception handler. So what is the default exception handler? It's a method of the *Application* object.

The top of the hierarchy in a Delphi application is the *Application* object. Its type is *TApplication* and its name is always *Application*. If you look at the project file for a Delphi application, you'll find the following two lines in the **begin..end** block:

```
Application.CreateForm(TForm1, Form1);
Application.Run;
```

Delphi starts your application by calling the *Application* object's *CreateForm* method to create the main form, and the *Application*'s *Run* method to start the application. (For more information about the *TApplication* object, see *TApplication* in the Delphi online help.)

Another *TApplication* method is *HandleException*. This is the default exception handler for an application. *TApplication* also generates an *OnException* event whenever an exception occurs that your code does not handle. If you write an event handler for the *OnException* event then your event handler will be used in place of the default handler.

Creating an event handler for one of the *Application* object's events is a bit more difficult than creating other event handlers. This is because *Application* does not appear in the Object Inspector. To create an event handler for an *Application* event you must perform these three steps:

- Write the procedure to handle the event.
- Add the event handler as a method to the **type** declaration for the main form.
- Assign the name of the event handler to the event in the main form's *OnCreate* event handler.

The easiest way to understand this process is to look at an example provided in the sample application, APP.DPR. The only trick to writing the event handler is knowing the parameters. For *OnException,* the event handler declaration is:

```
procedure TForm1.AppOnException(Sender: TObject;
                                E: Exception);
```

The *OnException* handler takes two parameters. The first is the object that called it, and the second is the *Exception* object that contains the error message for the exception.

The second step is to add the handler, named *AppOnException* in this example, to the main form's **type** declaration. Figure 12 is the **type** declaration for APP.DPR's main form. Notice that *AppOnException* has been added to the form's list of methods.

The third step is to assign the name of the exception handler to the *OnException* event in the form's *OnCreate* handler.

```
type
  TForm1 = class(TForm)
    Table1: TTable;
    CreateAnException: TButton;
    procedure AppOnException(Sender: TObject; E:
Exception);
    procedure CreateAnExceptionClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

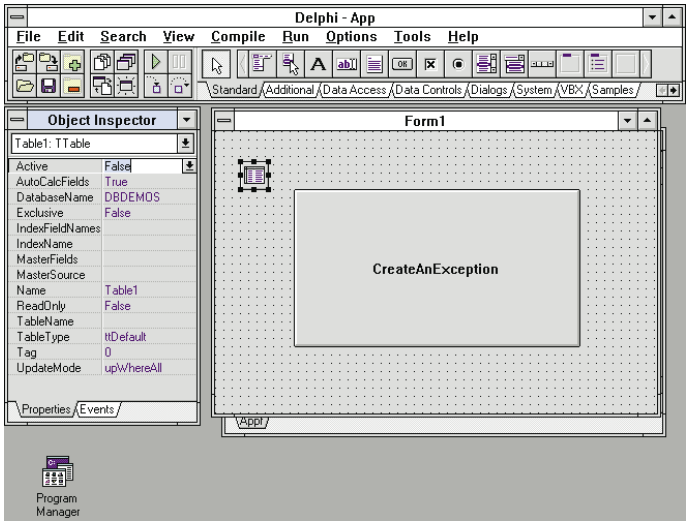


Figure 12 (Top): The **type** declaration for the APP project's main form. **Figure 13 (Bottom):** The main form for the sample APP project.

Select the form and double-click the *OnCreate* event in the Object Inspector. Then add the following code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  { Assign the custom OnException handler. }
  Application.OnException := AppOnException;
end;
```

This tells the *Application* object to call *AppOnException* whenever an unhandled exception occurs. Figure 13 is the main form for APP.DPR. It contains a *TTable* object with its *DatabaseName* property set to *DBDEMOS*. It also contains a Button with the following code in its *OnClick* handler:

```
procedure TForm1.CreateAnExceptionClick(Sender: TObject);
begin
  Table1.TableName := 'foo.db';
  Table1.Open;
end;
```

This code causes an exception by trying to open a table that does not exist. The exception handler itself, *AppOnException*, contains the code in Figure 14.

The *AppOnException* exception handler displays the error message in a dialog box and also writes an entry to an error log text file. The log file entry includes the date, time, user's network login name, the error message, and the address at which the exception occurred.

```
procedure TForm1.AppOnException(Sender: TObject;
                                E: Exception);
var
  Addr:     string[9];
  ErrorLog: System.Text;
begin
  AssignFile(ErrorLog,'errorlog.txt');
  { Open the file for appending. If it does not exist
    then create it. }
  try
    System.Append(ErrorLog);
  except
    on EInOutError do
      Rewrite(ErrorLog);
  end;
  { Write the date, time, user name, error message,
    and address to the error log file.}
  Addr := IntToHex(Seg(ErrorAddr),4) + ':' +
          IntToHex(Ofs(ErrorAddr),4);
  Writeln(ErrorLog,
          format('%s [%s] %s %s',[DateTimeToStr(Now),
          GetNetUserName, E.Message, Addr]));
  { Close the error log file. }
  System.Close(ErrorLog);
  { Display the error message dialog box. }
  MessageDlg(E.Message + '. Occurred at: ' +
             Addr, mtError, [mbOK], 0);
end;
```

Figure 14: The *AppOnException* exception handler.

The first task this code performs is to open the error log file:

```
AssignFile(ErrorLog, 'errorlog.txt');
{ Open the file for appending. If it does not
  exist then create it. }
try
  System.Append(ErrorLog);
except
  on EInOutError do
    Rewrite(ErrorLog);
end;
```

This code tries to open the file for appending. If the file does not exist, the call to *Append* raises an *EInOutError* exception that is handled in the **except** block by calling *Rewrite* to create and then open the file. The next snippet of code:

```
Addr := IntToHex(Seg(ErrorAddr),4) + ':' +
        IntToHex(Ofs(ErrorAddr),4);
```

converts the *ErrorAddr* pointer to a hex string by extracting and converting first the *Segment* then the *Offset*.

The next step is to write the date, time, user name, error message, and address to the file and close the file:

```
Writeln(ErrorLog,
        format('%s [%s] %s %s',[DateTimeToStr(Now),
               GetNetUserName,E.Message,Addr]));
{ Close the error log file. }
System.Close(ErrorLog);
```

GetNetUserName is a custom procedure that obtains the user's network login name by making a direct call to a BDE (Borland Database Engine) function (which is beyond the scope of this article). The final step in this *OnException* han-

dler is to display the dialog box with the error message to the user. You could also log exceptions to a table in your database as an alternative to using a text file.

Conclusion

This series has explored Delphi's exception mechanism for handling run-time errors. As you have seen, exceptions provide a flexible way to implement error handling in your Delphi programs.

You can handle exceptions yourself, let Delphi handle them, take some custom action and then reraise the exception to let the default exception handler process it, or just change the text of the message to add information. Δ

This article was adapted from material from *Delphi: A Developer's Guide* by Bill Todd and Vince Kellen [M&T Books, 1995 — (800) 488-5233].

The demonstration projects — Nest, IOErr, Custom, and App — referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\MAR\DI9603BT.

Bill Todd is President of The Database Group, Inc., a Phoenix area consulting and development company. He is co-author of *Delphi: A Developer's Guide* [M&T Books, 1995], *Creating Paradox for Windows Applications* [New Riders Publishing, 1994], and *Paradox for Windows Power Programming* [Que, 1993]; Technical Editor of *Paradox Informant*; a member of Team Borland; and a speaker at every Borland database conference. He can be reached at (602) 802-0178, or on CompuServe at 71333,2146.

By Cary Jensen, Ph.D.

BDE Basics

Examining Delphi's Support for the Borland Database Engine

Although Delphi is not a database application, it's well suited to building them. Specifically, Delphi is a powerful development environment that permits you to create applications that store, manipulate, and retrieve data. In most cases, support for accessing this data is provided by the Borland Database Engine (BDE), the same database engine used by Paradox for Windows, Visual dBASE, and Borland C++.

However, BDE support is not a product of Delphi's compiler. Instead, it's provided through DLL calls to the BDE. The Delphi run-time library (RTL) supplies three compiled units that support these DLL calls: DBIPROCS.DCU, DBITYPES.DCU, and DBIERRS.DCU.

This month's DBNavigator takes a closer look at the contents of these units, Delphi's support for direct BDE calls, and two approaches to calling BDE procedures and functions from within your Delphi applications.

Backstage Operations

Typically, the BDE's presence is not obvious to the Delphi database developer until an application is delivered. Then the BDE must also be delivered if it's not already on the target computer. This is because calls to the BDE are, for the most part, encapsulated by components on the Data Access and Data Controls pages of the Component Palette, or their ancestors. (A notable exception is the Report component on the Data Access page, which initiates a DDE link with ReportSmith, and does not involve the BDE directly.)

The primary advantage of this encapsulation is that you can add data access capabilities to your Delphi applications by simply adding one or more data-aware objects to your form. These objects, in turn, contain methods and properties that provide you with indirect, yet greatly simplified, access to the BDE's procedures and functions.

For example, the *Open* method of a Table component is a simple method call, requiring no arguments. This method, however, ultimately results in a call to the BDE function *DbiOpenTable*, which requires 12 parameters, in addition to several prerequisite calls (specifically, *DbiInit* and *DbiOpenDatabase*).

Wanted: Documentation

There are times, however, when you need to get information from, and control features of, the BDE not surfaced by these controls. Fortunately, most BDE methods and procedures are directly available to your applications with a minimum of effort. I say *most* because some BDE calls do not

appear to be available. For example, the BDE-related units of the RTL do not contain a method for removing an alias from the IDAPI.CFG file. Since the Borland Database Configuration utility can remove an alias, there must be a function or procedure that provides this capability.

As far as the supported BDE calls are concerned, the one problem that Delphi developers must overcome is obtaining valid information about these functions and procedures. Although Delphi ships with the BDE, it contains no written documentation for the BDE. Furthermore, examples of BDE calls from Delphi applications are difficult to find.

It should be noted that Borland sells the BDE as a separate product, permitting C/C++, Pascal, and other language developers to access the BDE's features. This stand-alone BDE product does ship with complete documentation that would satisfy the needs of most Delphi developers. As far as I know, Borland does not sell this documentation separately. And although it was not available at the time of this writing, I would dearly love to see Borland make an Adobe Acrobat version of this documentation available for download (just as they currently do with the Object Pascal Language Reference and VCL Reference).

But Delphi is not totally devoid of help. While the RTL does not ship with the actual import units (units that import the functions and procedures of a DLL) for the BDE API (the formal name for this is IDAPI — Independent Database Application Programming Interface), it does ship with the **interface** section of these units. By inspecting these **interface** sections you can see the syntax of every BDE procedure and function supported by Delphi, as well as the types and constants used and returned by BDE calls.

If you installed Delphi using the default directories, these interface files are located in the directory \DELPHI\DOC, and are named DBIPROCS.INT, DBITYPES.INT, and DBIERRS.INT (the .INT extension stands for Interface). Figure 1 shows part of the DBIPROCS.INT file. The source code (.PAS) files are not shipped with Delphi — only the compiled unit (.DCU) files are shipped.

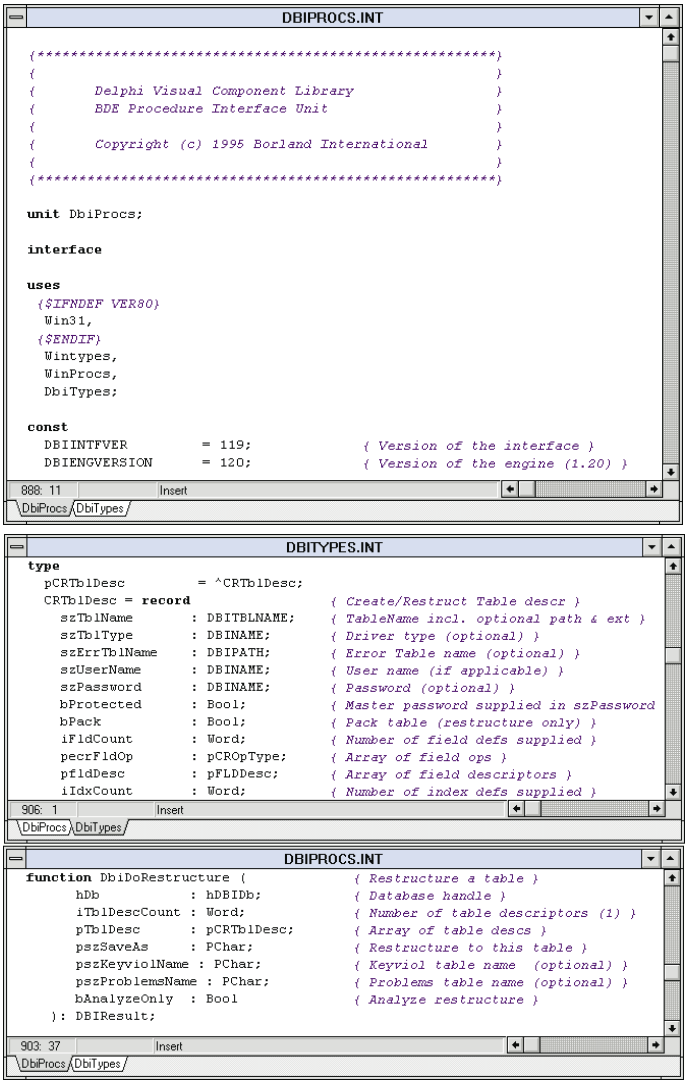


Figure 1 (Top): The file DBIPROCS.INT contains the **interface** section from the DBIPROCS.PAS import unit (the unit that declares and imports the functions and procedures from a DLL). **Figure 2 (Bottom):** The DBIPROCS.INT open to the *DbiDoRestructure* function on the bottom editor window, and the DBITYPES.INT file open to the *CRTblDesc* record declaration on the top editor window.

If you're planning any development using direct BDE calls, I think you'll find these .INT files to be invaluable. Although I do have a copy of the *BDE User's Guide* (I got mine from the product Borland C++ and Database Tools), I still use the .INT files extensively. You can see why by inspecting Figure 2, which displays two editor windows, one containing the DBIPROCS.INT file entry for *DbiDoRestructure*, and the other containing the DBITYPES.INT declaration for the *CRTblDesc* record object (which is used by *DbiDoRestructure*).

The online help file for the BDE (BDE.HLP) is available from Borland's CompuServe forums. The best place to look for it is the Borland Developers Tools forum (accessed with GO BDEVTOOLS). Search the files using the keywords "HELP" and "BDE". It's the closest thing to the BDE documentation available without purchasing the BDE. For each BDE function and procedure call, the BDE help file contains a description of the call, the syntax, a list of the parameters, a description of the usage, identifies any prerequisites for use, describes the completion state, and, if the call is a function, describes the return value. As you can imagine, this is a very important file for the BDE user. A screen from the BDE.HLP file is shown in Figure 3.

Of course, if you plan to do any important work with the BDE, you'll need to buy the BDE proper from Borland. It's the only way to get all the documentation you need — including the *BDE User's Guide* — unless you already have the Borland C++ and Database Tools product. [To purchase the BDE call Borland Customer Service at (510) 354-3828. The price is US$395.]

Making BDE Calls from Delphi Applications

There are two general approaches for working with the BDE. The first is to provide for all BDE calls directly, without the intervention of data-aware controls.

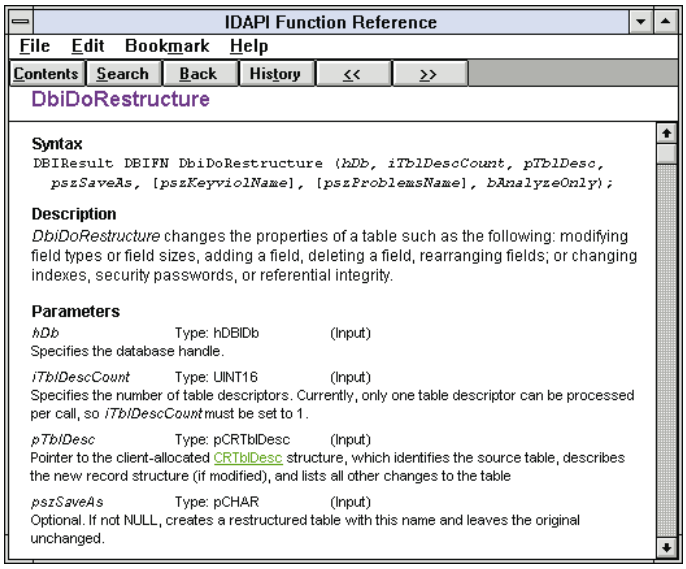


Figure 3: The BDE.HLP entry for *DbiDoRestructure*.

Accessing BDE functions and procedures this way is a lot of work. This is because you must take responsibility for initializing the BDE, as well as establishing database handles, cursor handles, and record handles. These steps are demonstrated in the following section. Please use Listing One beginning on page 33 as a reference.

Example 1: Packing Tables

Before any calls to the BDE can be made, you must give your unit access to the BDE import, type, and constant units. This is achieved by adding *DbiProcs*, *DbiTypes*, and *DbiErrs* in the **uses** clause of the unit from which the BDE calls will be made. In this example, these units are listed in the **interface** section's **uses** clause.

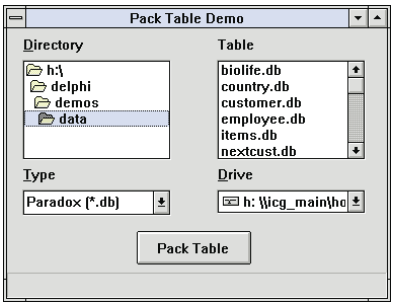


Figure 4: The main form of a project named PACKTAB.DPR. This form permits you to pack Paradox or dBASE tables. (Packing releases space from Paradox tables occupied by deleted records, and removes records marked for deletion from dBASE tables.) The code for this form's unit is shown in Listing One on page 33.

Since this form (see Figure 4) contains no data-aware components, all access to the BDE must be performed with Object Pascal. This is achieved by using three basic BDE calls: *DbiInit*, *DbiOpenDatabase*, and *DbiOpenTable*. Each of these calls is demonstrated in the *PackTable* procedure. In this case, the table that is selected from the main form is opened for exclusive use. This means that no other user can access this table while this procedure has the table open. It's also possible, and usually desirable, to open a table for shared use.

The *PackTable* procedure then demonstrates how to pack a table. For a Paradox table, you use the function *DbiDoRestructure*, while a dBASE table requires the use of *DbiPackTable*. An example of both of these procedures is demonstrated. The use of *DbiDoRestructure* in this example is as simple as this function can get. To actually change the structure of a Paradox table with this function would require a much more complex argument list.

Once your work with the BDE is done, it's necessary to clean up after the application. In this case, both the table cursor handle and the database handle need to be released (using *DbiCloseCursor* and *DbiCloseDatabase*), and then the BDE must be deactivated (using *DbiExit*).

Example 2: Displaying Table Information

As you can see from the preceding example, managing access to the BDE is a lot of work. It's much easier if you permit data-aware components to do some of this work for you.

For example, if you place a Table component on a form, and then open that table, the BDE will already be initialized, a database handle will be established, and a cursor handle will also exist. (The database handle can be obtained from the *DBHandle* property of the table, and the cursor handle can be obtained from the *Handle* property.) Furthermore, when the application no longer needs the BDE, the data-aware components take responsibility for releasing the handles and deactivating the BDE.

This combination of data-aware controls and BDE calls is demonstrated in the project named RECINFO (see Figure 5). The source code for this form is shown in Listing Two beginning on page 34.

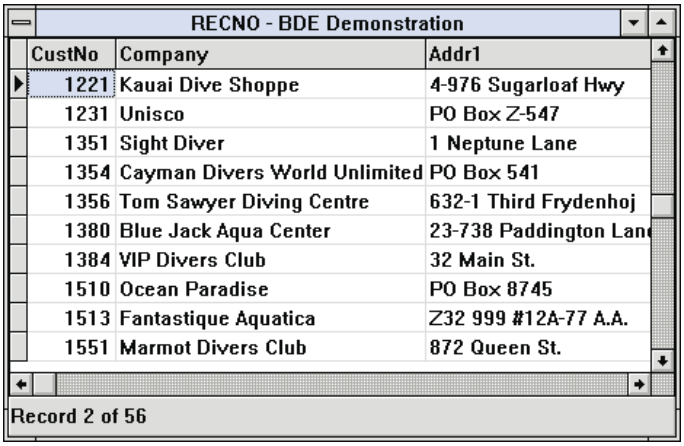


Figure 5: The main form of RECINFO. Notice the record information displayed in the panel at the bottom of the form.

You will immediately recognize two characteristics in Listing Two. First, it's short, requiring far less code than that shown in Listing One. This is because the internal code for the Table and DataSource components placed on this form take responsibility for the initialization of the BDE. The second important feature is that the *DbiProcs*, *DbiTypes*, and *DbiErrs* units must still appear in the **uses** clause, even when data-aware components are present.

The calls to the BDE in this example are found in the event handler assigned to the *OnDataChange* event property of a DataSource component. This event handler is called any time there is a change to the current record. Within this event han-

dler, the BDE procedures *DbiGetSeqNo* and *DbiGetRecordCount* are used to get the current record number and total number of records, respectively. These values are then used to update a message displayed in a Panel component. Note that both of these BDE calls require the cursor handle. As mentioned previously, these can be obtained from the *Handle* property of the table.

Conclusion

While most of the features of the BDE are encapsulated in data-aware controls, it's possible to make BDE calls directly to access features not otherwise provided by these controls. This article was intended to provide you with a brief overview of some of the issues involved with making BDE calls from within your Delphi applications. Δ

The demonstration forms referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\MAR\DI9603CJ.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, and is Contributing Editor of *Paradox Informant* and *Delphi Informant*. Cary is this year's Chairperson of the Paradox Advisory Board for the upcoming Borland Developers Conference. He has a Ph.D. in Human Factors Psychology, specializing in human-computer interaction. You can reach Jensen Data Systems at (713) 359-3311, or through CompuServe at 76307,1533.

Begin Listing One — The Packtabu Unit

```
unit Packtabu;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics,  Controls, Forms, Dialogs, StdCtrls,
  DbiProcs, DbiTypes, DbiErrs, FileCtrl, ExtCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    FilterComboBox1: TFilterComboBox;
    DriveComboBox1: TDriveComboBox;
    FileListBox1: TFileListBox;
    DirectoryListBox1: TDirectoryListBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Panel1: TPanel;
    procedure PackTable(Sender: TObject;TabName: PChar);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var
  Tab: PChar;
begin
  if FileListbox1.FileName = '' then
    begin
      MessageDlg('No table select',mtError,[mbOK],0);
      Exit;
    end;
  GetMem(Tab,144);

  try
    StrPCopy(Tab,FileListBox1.FileName);
    PackTable(Sender,Tab);
  finally
    Dispose(Tab);
  end;
end;

procedure TForm1.PackTable(Sender:  TObject;
                           TabName: PChar);
var
  hDb       :hDBIDb;     { Database handle }
  hCursor   :hDBICur;    { Cursor handle }
  dbResult  :DBIResult;  { Return value from BDE calls }
  PdxStruct :CRTblDesc;  { Paradox restructure record }
begin
  { Initialize the BDE }
  dbResult := DbiInit(nil);
  if dbResult <> DBIERR_NONE then
    begin
      case dbResult of
        DBIERR_MULTIPLEINIT :
          ShowMessage('DBIERR_MULTIPLEINIT');
      else
        ShowMessage('DbiInit failure');
      end;
      DbiExit;
      Exit;
    end;

  { Open a Database }
  dbResult :=
    DbiOpenDatabase('','STANDARD',dbiREADONLY,
                    dbiOPENSHARED,'',0,nil,nil,hDB);
  if dbResult <> DBIERR_NONE then
    begin
      case dbResult of
        DBIERR_UNKNOWNDB      :
          ShowMessage('DBIERR_UNKNOWNDB');
        DBIERR_NOCONFIGFILE   :
          ShowMessage('DBIERR_NOCONFIGFILE');
        DBIERR_INVALIDDBSPEC  :
          ShowMessage('DBIERR_INVALIDDBSPEC');
        DBIERR_DBLIMIT        :
          ShowMessage('DBIERR_DBLIMIT');
      else
        ShowMessage('DbiOpenDatabase failure');
      end;

      Exit;
    end;

  { Open a table. This returns a handle to the table's
    cursor, which is required by many of the BDE calls. }
  dbResult :=
    DbiOpenTable(hDB,TabName,'','','',0,dbiREADWRITE,
                 dbiOPENEXCL,xltNONE,False,nil,hCursor);
  if dbResult <> DBIERR_NONE then
    begin
      case dbResult of
        DBIERR_INVALIDFILENAME     :
          ShowMessage('DBIERR_INVALIDFILENAME');
        DBIERR_NOSUCHFILE          :
```

```
        ShowMessage('DBIERR_NOSUCHFILE');
      DBIERR_TABLEREADONLY       :
        ShowMessage('DBIERR_TABLEREADONLY');
      DBIERR_NOTSUFFTABLERIGHTS  :
        ShowMessage('DBIERR_NOTSUFFTABLERIGHTS');
      DBIERR_INVALIDINDEXNAME    :
        ShowMessage('DBIERR_INVALIDINDEXNAME');
      DBIERR_INVALIDHNDL         :
        ShowMessage('DBIERR_INVALIDHNDL');
      DBIERR_INVALIDPARAM        :
        ShowMessage('DBIERR_INVALIDPARAM');
      DBIERR_UNKNOWNTBLTYPE      :
        ShowMessage('DBIERR_UNKNOWNTBLTYPE');
      DBIERR_NOSUCHTABLE         :
        ShowMessage('DBIERR_NOSUCHTABLE');
      DBIERR_NOSUCHINDEX         :
        ShowMessage('DBIERR_NOSUCHINDEX');
      DBIERR_LOCKED              :
        ShowMessage('DBIERR_LOCKED');
      DBIERR_DIRBUSY             :
        ShowMessage('DBIERR_DIRBUSY');
      DBIERR_OPENTBLLIMIT        :
        ShowMessage('DBIERR_OPENTBLLIMIT');
    else
      ShowMessage('DbiOpenTable failure');
  end;
  DbiCloseDatabase(hDB);
  DbiExit;
  Exit;
end;

{ The BDE is initialized, a database is open, and
  a cursor is open for a table. We can now work with
  the table. The following segment shows how to pack
  a dBASE or Paradox table. Note that before we can
  pack the Paradox table, the table's cursor handle
  must be closed, otherwise we would get a 'Table in
  use' error. }

try
  Panel1.Caption := 'Packing ' + FileListBox1.FileName;
  Application.ProcessMessages;

  if AnsiUpperCase(ExtractFileExt(
                  FileListBox1.FileName)) = '.DB' then
    begin
      { Close the Paradox table cursor handle. }
      DbiCloseCursor(hCursor);
      { The method DoRestructure requires a pointer
        to a record object of the type CRTblDesc.
        Initialize this record. }
      FillChar(PdxStruct,SizeOf(CRTblDesc),0);
      StrPCopy(PdxStruct.szTblName,
               FileListBox1.Filename);
      PdxStruct.bPack := True;
      dbResult :=
        DbiDoRestructure(hDB,1,@PdxStruct,nil,
                         nil,nil,False);
      if dbResult = DBIERR_NONE then
        Panel1.Caption := 'Table successfully packed'
      else
        Panel1.Caption :=
          'Failure: error ' + IntTostr(dbResult);
    end
  else
    begin
      { Packing a dBASE table is much easier. }
      dbResult := DbiPackTable(hDB,hCursor,'','',True);
      if dbResult = DBIERR_NONE then
        Panel1.Caption := 'Table successfully packed'
      else
        Panel1.Caption :=
          'Failure: error ' + IntTostr(dbResult);
    end;
  finally
    DbiCloseCursor(hCursor);
    DbiCloseDatabase(hDB);
    DbiExit;
  end;
end;

end.
```

End Listing One

Begin Listing Two — The Recnou Unit

```
unit Recnou;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,

  Graphics, Controls, Forms, Dialogs, Grids, DBGrids,
  ExtCtrls, DB, DBTables, DbiProcs, DbiTypes, DbiErrs;

type
  TForm1 = class(TForm)
    DataSource1: TDataSource;
    Table1: TTable;
    Panel1: TPanel;
    DBGrid1: TDBGrid;
    procedure DataSource1DataChange(Sender: TObject;
                                    Field: TField);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.DataSource1DataChange(Sender: TObject;
                                       Field: TField);
var
  i,j: LongInt;
begin
  Table1.UpdateCursorPos;
  DbiGetSeqNo(Table1.Handle,i);
  DbiGetRecordCount(Table1.Handle,j);
  Panel1.Caption :=
    'Record '+ IntToStr(i) + ' of ' + IntToStr(j);
end;

end.
```

End Listing Two

By Gary Entsminger

Approaching OLE Automation: Delphi 2.0 & Word Basic

State of the Object Art: Part III

Another thing I dislike: A writer who says he's going to write about one thing and then writes about another the first chance he gets (see Part II in the February 1996 issue of Delphi Informant*). But I can't help myself — something more interesting has come up.*

I started this discussion on the "State of the Object Art" using 1988 as a benchmark (see Part I in the January 1996 issue of *Delphi Informant*). Then, objects were mostly small timers in programming — hardly in the computing lingo. Only a few languages spoke "object" — Smalltalk, Actor, and C++ — and serious developers traveled elsewhere.

But in 1996, "object" is spoken all over the map — by OOP programmers, by system software, by users, by the computing media. Even on the Internet, it's an object-oriented language — Java — that's suddenly got our attention.

In this installment (Part III), we'll explore a new darling of the object art — OLE Automation. It's a way to manipulate other applications *as objects* from your Delphi applications. It's a powerful concept, and I think you'll like it when you try it.

In the first part of this article, you'll learn the basics of OLE Automation. In the second part, you'll create a little application that shows how OLE Automation works. This application, called the OLE Automation Explorer, connects a Delphi application to Word Basic via OLE Automation.

(Note that Delphi 1.0 does not have OLE Automation capabilities. Delphi 2.0 does, and it's one of many good reasons to switch to Delphi 2.0. However, even with Delphi 1.0, you can easily follow along.)

OLE Automation, a Gentle Introduction

You use OLE Automation to control other applications, which are sometimes called "programmable Windows objects" in this context, from your Delphi applications. The key words are *programmable* and *objects*. Programmable, in this context, means "accessing and manipulating another application's commands." In object-oriented terms, commands are methods and properties, and an object is an application that supports OLE Automation.

Before your Delphi application can use OLE Automation to "program" another application in the Windows environment, the application must support OLE Automation. For example, applications in the Microsoft Office group (Word, Excel, etc.) support OLE Automation. You can use programming languages such as C++, Visual Basic 4.0, and

Delphi 2.0 to create applications that support OLE Automation.

An application (i.e. *object*) that supports OLE Automation provides methods or properties that other applications can access through code. Using OLE Automation, your applications can request that other applications perform tasks from within your applications. Thus you don't have to write the code for the task yourself. When your application needs to perform the task, it invokes the OLE Automation object's method that handles the specific task. You request service, task by task. Thus you use only as much of the OLE Automation object's functionality as you need, when you need it.

For example, if your application requires some spreadsheet functionality that already exists in Microsoft Excel, you can create a reference to an Excel spreadsheet — an OLE Automation object — and have Excel handle the task.

Similarly, you can add word processing capability to your Delphi application by using the Microsoft Word for Windows (WinWord) Basic command language to manipulate a WinWord file. For example, you can use WinWord's proofing tools (spelling, thesaurus, word count) and its ability to create and manipulate tables within text. The possibilities are limited by your imagination and the capabilities of specific OLE Automation Servers.

Although a relative newcomer to the computing arena, OLE Automation is one cool way to open up a system. In an easy ridin' kind of object-oriented world, no one would have to recreate existing functionality. He or she could simply tap into what's already there and avoid writing some code. It might even save some programming time.

Using OLE Automation, the Basics

In general, making a Delphi application OLE Automation-capable requires the following general steps:

- In the **uses** part of the unit, make Delphi 2.0 OLE Automation support accessible to your application by adding the OLEAuto.pas unit supplied by Delphi 2.0:

```
uses
  OLEAuto;
```

  This file contains the low down, dirty OLE Automation details. Check it out if you're so inclined.

- In the procedure that accesses the OLE Automation object, declare a *Variant* variable to represent the OLE Automation server. The Variant type is new to Delphi 2.0:

```
MSWord: Variant; { Represents OLE Automation Server. }
```

- Use the Delphi 2.0 *CreateOLEObject* function to create the OLE Automation object and return a reference to it:

```
MSWord := CreateOLEObject('Word.Basic');
```

- Access OLE Automation properties and methods as needed.

Those are the basics. Each OLE Automation server will have its own set of properties and methods you can use. Find out which ones are available by consulting the application's documentation or online help.

The Example Project: OLE Automation Explorer

Now, for a no-bells-and-whistles OLE Automation application: OLE Automation Explorer, Take One.

From the Delphi 2.0 main menu, create a new project. Use the default form as the main (and only) form for this project.

Use the Component Palette to add the following components to the form:

- six Edit components
- two Button components

Next, use the Object Inspector to change the *Text* property of each of the Edit components. By default, these components contain the strings: `Edit1`, `Edit2`, `Edit3`, `Edit4`, `Edit5`, and `Edit6`. Change each of these strings to a number. Your choice.

Next, change the *Caption* property of the two Buttons to `Use Fields` and `Random`. Change the *Caption* of the Form to `Auto Explorer`. Then, change the *FormStyle* property of the Form to *fsStayOnTop*, to allow the Delphi application to float over the OLE Automation server. Figure 1 shows this form at design time.

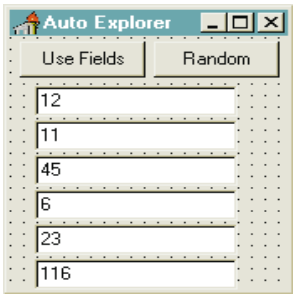


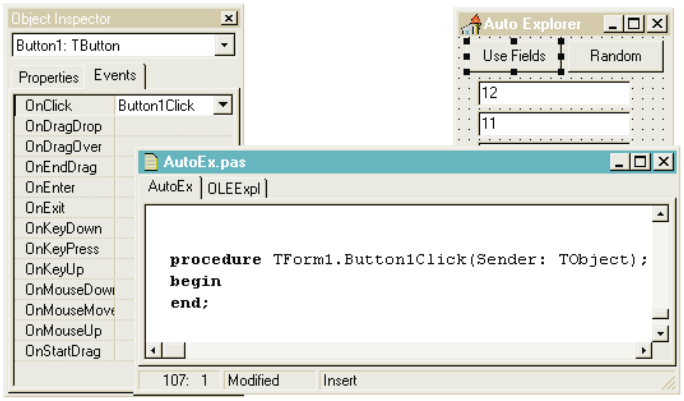
Figure 1: The main form of the example application, OLE Automation Explorer, at design time.

Use the Object Inspector's Events page to create *OnClick* events for the Buttons. To do this, double-click the *OnClick* event edit box in the Object Inspector for each Button to create templates for the events (see Figure 2). (Or you can simply double-click on each Button component.)

Figure 3 shows the class description for the form. Everything in this application occurs in response to the two *OnClick* events. The *OnClick* procedures behave similarly; each creates an OLE Word Basic object and uses Word Basic commands to create a file and manipulate data within the file.

The Use Fields Button

The **Use Fields** button *OnClick* event procedure (*TForm1.Button1Click*) builds a string of data delimited (separated) by commas. It inserts the resulting string of data in the new WinWord file. It then moves to the beginning of the data and converts the data to a WinWord table.




```
type
  TForm1 = class(TForm)
    Edit1: TEdit;   { 6 edit boxes to hold fields. }
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Edit5: TEdit;
    Edit6: TEdit;
    Button1: TButton;  { Use fields button. }
    Button2: TButton;  { Generate random table button. }
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
end;
```

Figure 2 (Top): Creating the template for an *OnClick* event handler.
Figure 3 (Bottom): The class description for the form.

The *Button1Click* procedure also declares a *Variant* variable to represent MSWord and uses the Delphi *CreateOLEObject* function to create the OLE Automation object:

```
MSWord: Variant; { Represents OLE Automation Server. }
begin
  { Create an OLE Word Basic object. }
  MSWord := CreateOLEObject('Word.Basic');
```

It then uses WinWord's **FileNew** method (or command) to create a new file within Word:

```
{ Use Word Basic commands to create a new Word file. }
MSWord.FileNew;
```

builds a string from the contents of the six Edit components:

```
S :=     Edit1.Text + #13;
S := S + Edit2.Text + #13;
S := S + Edit3.Text + #13;
S := S + Edit4.Text + #13;
S := S + Edit5.Text + #13;
S := S + Edit6.Text + #13;
```

uses Word Basic to insert the string into the current Word file:

```
MSWord.Insert(S);
```

uses Word Basic to move six lines up, to the beginning of the file:

```
MSWord.LineUp(6,1);
```

and uses Word Basic to post a message to user:

```
MSWord.MsgBox('Converting fields to table.');
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S : string;
  MSWord: Variant; { Represents OLE Automation Server. }
begin
  { Create an OLE Word Basic object. }
  MSWord := CreateOLEObject('Word.Basic');
  { Use Word Basic commands to create a new Word file. }
  MSWord.FileNew;

  { Build a string from edit boxes. }
  S :=     Edit1.Text + #13;
  S := S + Edit2.Text + #13;
  S := S + Edit3.Text + #13;
  S := S + Edit4.Text + #13;
  S := S + Edit5.Text + #13;
  S := S + Edit6.Text + #13;

  { Use Word Basic to insert the string
    into the current Word file. }
  MSWord.Insert(S);
  { Use Word Basic to move 6 lines up,
    to the beginning of the file. }
  MSWord.LineUp(6,1);
  { Use Word Basic to post a message to user. }
  MSWord.MsgBox('Converting fields to table.');
  { Use Word Basic to convert text to a table. }
  MSWord.TextToTable(ConvertFrom := 2,
                     NumColumns := 1);
end;
```

Figure 4: The **Use Fields** button's *OnClick* event procedure, *TForm1.Button1Click*.

Of course, we could have used Delphi to display this message, but this way you can see how similar the task is in the two languages.

The procedure also uses Word Basic to convert the text into a WinWord table:

```
MSWord.TextToTable(ConvertFrom := 2,
                   NumColumns := 1);
```

Again, note that you must get the method details for an OLE Automation-capable application from the application itself. Delphi doesn't supply this information for you. In this case, I used the Word Basic Program Help System from the WinWord 7.0 Help system to get information about the methods and their accompanying parameter requirements. In most cases, you'll need to obtain the equivalent information for any OLE Automation-capable application you want to manipulate from your Delphi applications.

The full listing for the *Button1Click* event-handler procedure is shown in Figure 4.

The Random Button

The **Random** button's *OnClick* event procedure works similarly (see Figure 5). The main differences are that it uses a random number generator to build the string to insert in the Word file, and that it builds a slightly different table (one with three columns).

```
procedure TForm1.Button2Click(Sender: TObject);
var
  S, S1,
  S2, S3: string;
  MSWord: Variant; { Represents OLE Automation Server. }
  I, Num: Integer;
begin

  { Create an OLE WordBasic object. }
  MSWord := CreateOLEObject('Word.Basic');
  { Use Word Basic commands to create a new Word file. }
  MSWord.FileNew;
  { Build random string of values to use for table. }
  Randomize;
  S := '';

  for I := 1 to 20 do
    begin
      Num := Random(99);
      Str(Num,S1);
      Num := Random(999);
      Str(Num,S2);
      Num := Random(9999);
      Str(Num,S3);
      S:= S+ S1 + ',' + S2 + ',' + S3 + ',' + #13;
    end;

    { Use Word Basic to insert the string
      into the current Word file. }
    MSWord.Insert(S);
    { Use Word Basic to move up
      to the beginning of the file. }
    MSWord.LineUp(I,1);
    { Use Word Basic command to post a message to user. }
    MSWord.MsgBox('Converting random numbers to table.');
    { Use Word Basic command to convert text to a table. }
    MSWord.TextToTable(ConvertFrom := 2, NumColumns := 3);
end;
```

Figure 5: The *Button2Click* event handling procedure.

Over and Out

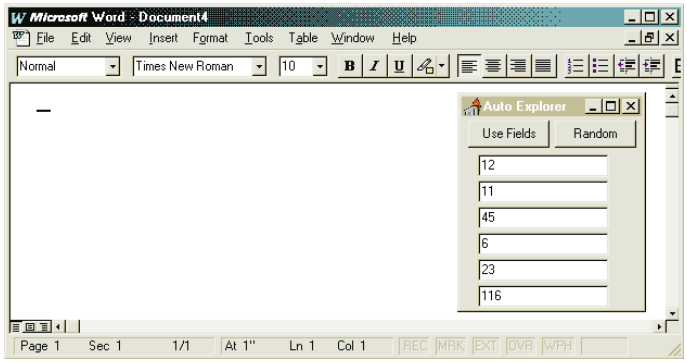
Listings Three and Four, beginning on page 39, provide the complete listing for this project.

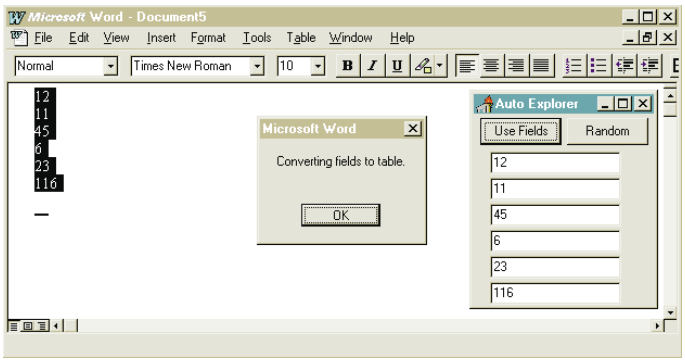
Figures 6, 7, 8, 9, and 10 show the OLE Automation Explorer at several stages of operation.

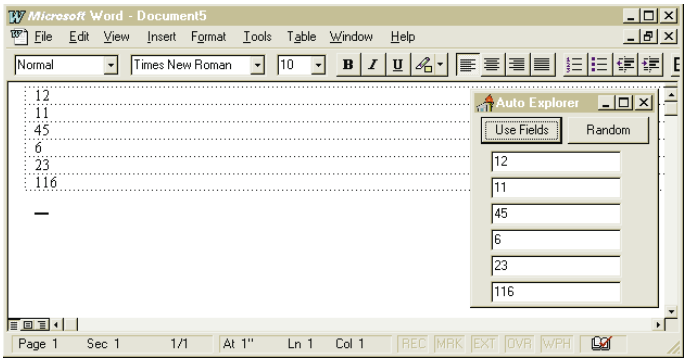
I think you're going to hear a lot about OLE Automation as applications learn to rely on each other for functionality instead of doing it all themselves. Δ

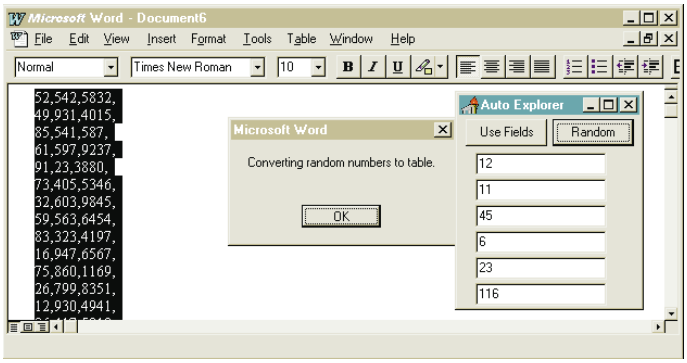
The demonstration project referenced in this article is available on the Delphi Informant Works CD located in INFORM\96\MAR\DI9603GE.

Gary Entsminger is the author of *The Way of Delphi* [Prentice-Hall, 1996], *The Tao of Objects* [M&T Books, 1995], *Secrets of the Visual Basic Masters* [Sams, 1994], and *Developing Paradox Databases* [M&T Books, 1993].









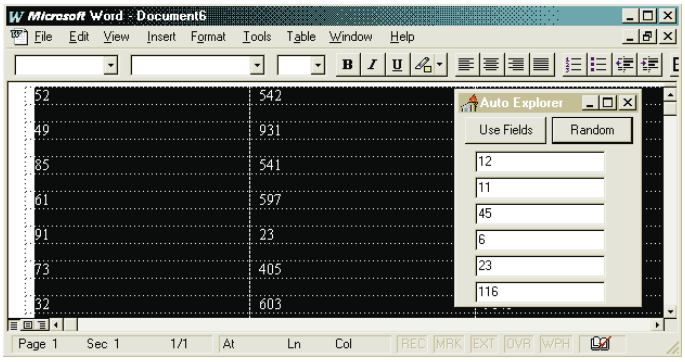


Figure 6 (Top): The example application, OLE Automation Explorer, at run time before making an OLE connection. **Figure 7:** After making the OLE connection to WinWord and creating a new file. **Figure 8:** After inserting the string of data into the Word file. **Figure 9:** After creating the first table. **Figure 10:** After creating the second table.

Begin Listing Three — The Project File: OLEExpl.DPR

```
program OLEExpl;

uses
  Forms,
  AutoEx in 'AutoEx.pas' {Form1};

{$R *.RES}

begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

End Listing Three

Begin Listing Four — The Form File: AutoEx.PAS

```
unit AutoEx;

{ Uses Delphi's automation control to
  use Word Basic to insert tables into
  Word documents.
  Microsoft Word as the automation server. }

interface

uses
  Windows, Classes, Graphics, Forms,
  Controls, DB, DBGrids, DBTables, Grids,
  StdCtrls, ExtCtrls, ComCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;    { 6 edit boxes to hold fields. }
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Edit5: TEdit;
    Edit6: TEdit;
    Button1: TButton;  { Use fields button. }
    Button2: TButton;  { Generate random table button. }
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
end;

var
  Form1: TForm1;

implementation

uses OleAuto; { Delphi 2.0 OLEAuto support file. }

{$R *.DFM}

{ Use fields button click event procedure. }
procedure TForm1.Button1Click(Sender: TObject);
var
  S : string;
  MSWord: Variant; { Represents OLE Automation Server. }
begin

  { Create an OLE WordBasic object. }
  MSWord := CreateOleObject('Word.Basic');
  { Use Word Basic commands to create a new Word file. }
  MSWord.FileNew;
  { Build a string from edit boxes. }
  S :=     Edit1.Text + #13;
  S := S + Edit2.Text + #13;
  S := S + Edit3.Text + #13;
  S := S + Edit4.Text + #13;
  S := S + Edit5.Text + #13;
  S := S + Edit6.Text + #13;
  { Use Word Basic to insert the string
    into the current Word file. }
  MSWord.Insert(S);
  { Use Word Basic to move 6 lines up,
    to the beginning of the file. }
  MSWord.LineUp(6,1);
  { Use Word Basic to post a message to user. }
  MSWord.MsgBox('Converting fields to table.');
  { Use Word Basic to convert text to a table. }
  MSWord.TextToTable(ConvertFrom := 2, NumColumns := 1);
end;

{ Random button click event procedure. }
procedure TForm1.Button2Click(Sender: TObject);
var
  S, S1,
  S2, S3: string;
  MSWord: Variant; { Represents OLE Automation Server. }
  I, Num: Integer;
begin

  { Create an OLE WordBasic object. }
  MSWord := CreateOleObject('Word.Basic');



  { Use Word Basic commands to create a new Word file. }
  MSWord.FileNew;
  { Build random string of values to use for table. }
  Randomize;
  S := '';

  for I := 1 to 20 do
    begin
      Num := Random(99);
      Str(Num,S1);
      Num := Random(999);
      Str(Num,S2);
      Num := Random(9999);
      Str(Num,S3);
      S:= S+ S1 + ',' + S2 + ',' + S3 + ',' + #13;
    end;

    { Use Word Basic to insert the string
      into the current Word file. }
    MSWord.Insert(S);
    { Use Word Basic to move up
      to the beginning of the file. }
    MSWord.LineUp(I,1);
    { Use Word Basic command to post a message to user. }
    MSWord.MsgBox('Converting random numbers to table.');
    { Use Word Basic command to convert text to a table. }
    MSWord.TextToTable(ConvertFrom := 2, NumColumns := 3);
  end;
end.
```

End Listing Four

By Robert Vivrette

From the Start

Passing Command-Line Parameters to a Delphi Application

Occasionally, I see questions from users about how to add various capabilities to their Delphi applications. Interestingly, one of the recurring questions is also one of the easiest to answer: "How can I read parameters from the command line?"

Many of you have used the properties pages for an application in Windows 3.1 or Windows 95. To display an application's properties page in Windows 95, right-click on the application's icon and select **Properties** from the pop-up menu. (In Windows 3.x, click on the application's icon and select **File | Properties** from the Program Manager menu.)

Figure 1 shows the Windows 95 properties page for a hypothetical application called ACCTPOST. (Figure 2 shows the Windows 3.1 Properties dialog box for the same application.) You'll notice that the **Target** edit box identifies the application's name. More importantly for the purposes of this tip, it also allows you to enter command-line parameters.

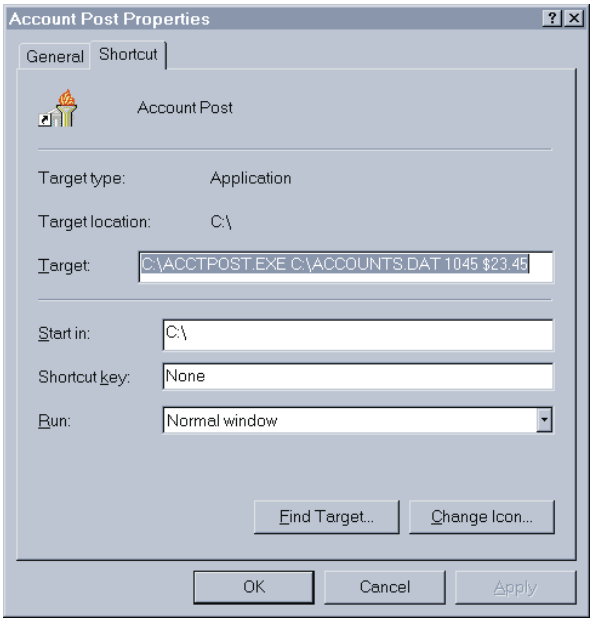


Figure 1: The Properties sheet of the sample ACCTPOST program, showing the command-line parameters (i.e. the Accounts.DAT file, and the values 1045 and $23.45) that will be passed to the application.

Often, such a parameter represents some file that you want to have the application use (such as a word processor opening a file). Such a capability is typically managed by means of the file type associations that Windows provides. This is, for example, how Windows knows to launch the Notepad application when you click on a file with a .TXT extension.

There are situations, however, when file associations aren't enough. For example, an application might require multiple file names to be passed to it, or the parameters aren't files at all, but are drive letters, values

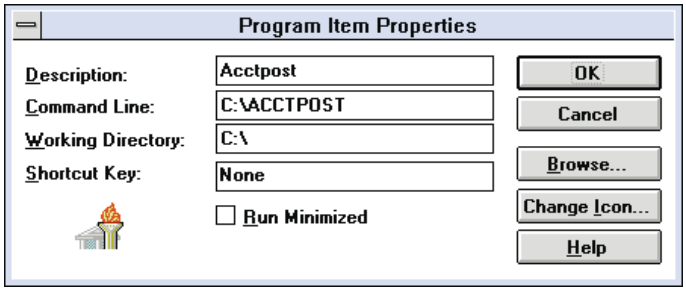


Figure 2: The Windows 3.1 Properties dialog box for the sample ACCTPOST program (no command-line parameters are shown).

(e.g. account balances), or the like. Figure 1 illustrates just this situation.

The Functions

Delphi provides two functions that allow you to get access to these parameters: *ParamCount* and *ParamStr*. *ParamCount* is a function that returns the number of parameters on the command line. *ParamStr* is a function that accepts a single parameter specifying the particular one you are interested in.

For example, *ParamStr(1)* would return a string value of the first parameter, *ParamStr(2)* would return the second, and so on. As a bonus, if you ask for *ParamStr(0)*, you get the fully qualified name (including drive letter and path) of the application itself. Sometimes this is a handy way to have your program find out which directory it's running from.

A Demonstration

To illustrate these features, place a *TEdit* and a *TMemo* on a form (as shown in Figure 3) and add the following code to the *FormCreate* method:

```
procedure TForm1.FormCreate(Sender:
                            TObject);
var
  a : Integer;
begin
  Edit1.Text := IntToStr(ParamCount);
  for a := 0 to ParamCount do
    Memo1.Lines.Add(IntToStr(a) +
    ': ' + ParamStr(a));
end;
```

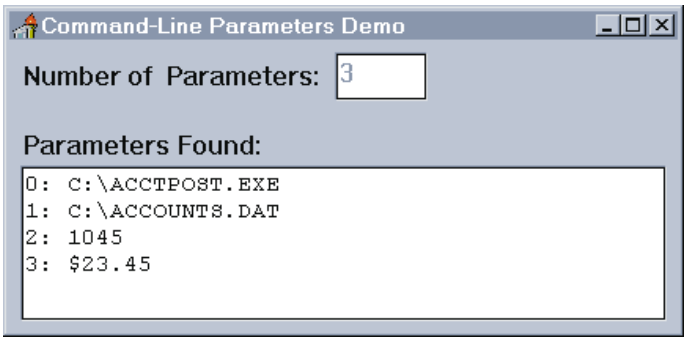


Figure 3: The demonstration project, ACCTPOST, at run time.

The code is straightforward. First, the *ParamCount* function is used to determine the number of parameters and display them in an Edit component. Then a **for** loop uses the value returned by *ParamCount* to iterate through the *ParamStr* string array and load each string into a Memo component.

When the program runs, the Edit component will indicate the number of parameters that were passed and the Memo will list each of the parameters found. Figure 3 gives you an indication of how it operates.

There you have it! A quick and easy way to read values from an application's command-line. Δ

The demonstration project referenced in this article is available on the Delphi Informant Works CD located in INFORM\96\MAR\DI9603RV.

Robert Vivrette is a contract programmer for Pacific Gas & Electric and Technical Editor for *Delphi Informant*. He has worked as a game designer and computer consultant, and has experience in a number of programming languages. He can be reached on CompuServe at 76416,1373.

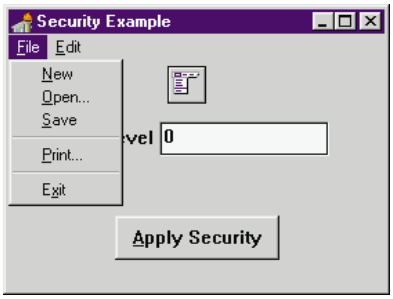
At Your Fingertips

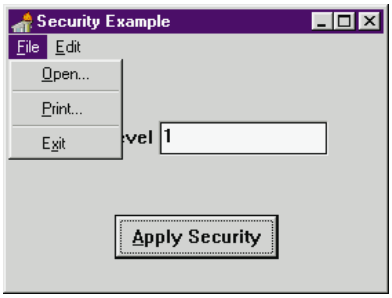
BY DAVID RIPPY
DELPHI / OBJECT PASCAL

Opportunities are usually disguised as hard work, so most people don't recognize them. — Ann Landers

How can I create a simple menu-level security system?

Virtually every application needs some sort of security. For typical Delphi applications that use the MainMenu component, you can easily add menu-level security with just a few lines of Object Pascal code. In a menu-level security system, the user's security level (represented here by the variable *iSecLevel*) is compared to the security level of each menu item (via the menu item's *Tag* property). If the user's security level is less than the menu item's *Tag* property, the *Visible* property of the menu item is set to *False*, making it unavailable to the user.





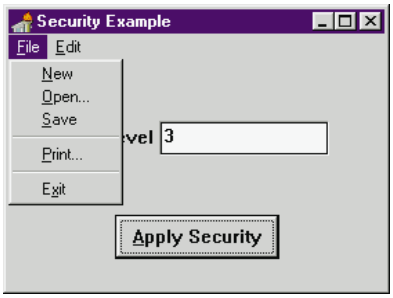


Figure 1 (Top): Create a MainMenu component similar to this one.
Figure 2 (Middle): A security level of 1 hides several menu items.
Figure 3 (Bottom): A security level of 3 shows every menu item.

First, create a main menu similar to the one shown in Figure 1. As you create each menu item (**New**, **Open**, **Save**, etc.), set its *Tag* property to the desired security level. For simplicity, make 0 (the *Tag*'s default) the lowest level of security, and 3 the highest level of security. For the **File** menu, a typical security system would set **Open**, **Print**, and **Exit** to 0, and **New** and **Save** to 3. This would allow a person with a security level of 0, 1, or 2 to view or print a document, but not create or save a new one (see Figure 2). For users with a security level of 3 or higher, all menu items would be available (see Figure 3).

Take a look at the Button's *OnClick* procedure:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i, j, iSecLevel : Integer;
begin
  { Your app would not include this line. }
  iSecLevel := StrToInt(Edit1.Text);

  for i := 0 to MainMenu1.Items.Count - 1 do begin
    if iSecLevel < MainMenu1.Items[i].Tag then begin
      MainMenu1.Items[i].Visible := False;
      Continue;
    end
    else
      MainMenu1.Items[i].Visible := True;
    for j := 0 to MainMenu1.Items[i].Count - 1 do begin
      if iSecLevel < MainMenu1.Items[i].Items[j].Tag then
        MainMenu1.Items[i].Items[j].Visible := False
      else
        MainMenu1.Items[i].Items[j].Visible := True;
      end;
  end;
end;
```

The code first assigns the value of *Edit1* to the variable *iSecLevel*. (Obviously, your application would assign the user's security level in a different manner.) It then iterates through the *MenuItems* and compares each *MenuItem*'s *Tag* property to the user's security level (*iSecLevel*). Any *MenuItems* with *Tag* values greater than *iSecLevel* are made invisible.

Note that you can make an entire pull-down menu (**File**, **Edit**, **Window**, etc.) invisible by setting its *Tag* property in the same manner. For example, if **Edit**'s *Tag* was set to 3 and the user's security level was 2, the entire **Edit** pull-down would be invisible. — *D.R.*

How can I stretch an .AVI image to completely fill a Panel using the MediaPlayer component?

Have you ever used the MediaPlayer component to display an .AVI file, but the .AVI is too small to fit the Panel on a form (see Figure 4)? It would be nice if the MediaPlayer component had a *Stretch* property similar to the Image component. Fear not! With just one line of Object Pascal, we can simulate the *Stretch* property, allowing the size of an .AVI to grow or shrink as needed.

Add this Object Pascal to the Form's *OnCreate* event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MediaPlayer1.DisplayRect :=
    Rect(0,0,Panel1.Width,Panel1.Height);
end;
```

This code sets the *DisplayRect* property of the MediaPlayer to the dimensions of the *Panel* object. This gives the MediaPlayer the intelligence to automatically shrink or expand to fill the *Panel* and eliminate the ugly empty space shown in Figure 4. The modified form is shown in Figure 5. — *D.R.* (Special thanks to Randy Haben and Jeff Hopper of Borland.)

How can I disconnect an event handler from an event?

Have you ever inadvertently called an event from within that same event's event handler? For example, if you have a field called Amount, and whenever the field is changed, you want to throw out the old value, and set the actual value to 42. You might first try the following code:

```
procedure TForm1.Table1AmountChange(Sender: TField);
begin
  Table1Amount.AsInteger := 42;
end;
```

This code will get you into trouble! Setting the value of Amount to 42 triggers the event that, in turn, sets the value to 42 a second time. The simple way to avoid this recursion (without using those messy global variables) is to temporarily disconnect the event handler from the *OnChange* event as follows:

```
procedure TForm1.Table1AmountChange(Sender: TField);
begin
  Table1Amount.OnChange   := nil;
  Table1Amount.AsInteger  := 42;
  Table1Amount.OnChange   := Table1AmountChange;
end;
```

Now, when the value is set to 42, the *Table1AmountChange* procedure doesn't get called a second time, because it has been disconnected from the *OnChange* event. The important point here is that event handlers are really properties that can be modified at run time! — *David Faulkner, Silver Software Inc.*

Figure 4 (Top): This .AVI is too small for the Panel component.
Figure 5 (Bottom): That's much better! The .AVI fits the Panel perfectly.

Quick Tip: Using the Windows 95 Taskbar

If you're a Windows 95 user, you have probably found that the Windows 95 Taskbar gets in the way of Delphi's Object Inspector and code editor. To remedy this, drag the Taskbar to the right side of the screen where it's out of the way. — *D.R.* Δ

The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\MAR\DI9603DR.

David Rippy is a Senior Consultant with Ensemble Corporation, specializing in the design and deployment of client/server database applications. He has contributed to several books published by Que, and is contributing writer to *Paradox Informant*. David can be reached on CompuServe at 74444,415.

BY LARRY CLARK

ABC for Delphi

A Low-Cost Collection of VCL Components

ABC for Delphi is a collection of more than 30 VCL components. Although "ABC" is the acronym for "Advanced Business Components," nothing seems specifically business-oriented about the collection.

The package is supplied on a single diskette and includes an 82-page saddle-bound manual. The manual contains installation directions, a description of each component, and references for the various properties, methods, events, types, and units.

The documentation is extremely brief. Within the component descriptions, the narrative description generally occupies less space than the simple enumeration of the component's properties, methods, and events. With derived components, there's no indication of which properties differ from those of the component's ancestor.

Although the manual contains no code examples, the diskette comes with several demonstration programs that effectively illustrate most of ABC's features. Unfortunately, there is a tendency to wrap several controls into a single demonstration, making it difficult to see the essence of each component's use. I prefer to see smaller, self-contained examples that highlight the critical aspects in both the manual and help file.

The installation of ABC follows the usual procedure for adding Delphi component libraries and Help files. The brief installation instructions worked well with no surprises. Figure 1 shows a Component Palette with the demonstration components installed.

Functional Descriptions

The components of ABC for Delphi are a mixed bag; I found no unifying theme. Rather, they seemed to be whatever the developers found they could do with Delphi that might prove useful to other developers. The manual's Component Overview section (it's less than one page, including a 1/3-page table listing the components) divides the components into three groups. For the purpose of this discussion, however, I have created my own classifications.

Exception Handling

I found ABC's exception-handling components among the best in the package. They provide important capabilities that cannot be achieved nearly as easily without them.

TAppException provides a centralized facility for displaying and logging error messages. You simply add a *TAppException* component to the main form of your application. When an application raises an exception that is not handled in a **try..except** block, *TAppException* takes over.

Depending on property settings, it can display a message box and/or log the error to a file, and optionally terminate the program. A custom *OnException* event that gains control before *TAppException* assumes control allows you to take corrective actions and cancel *TAppException*'s normal behavior if necessary.

TDBAppException uses a database to extend the behavior of *TAppException*. It allows you to use a lookup table within your database to define exception messages. It also logs each exception it encounters in another table.

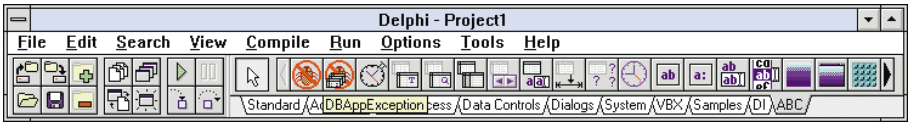


Figure 1: After installing the demonstration .PAS file, ABC components appear on their own page of the Component Palette. In this example, the non-visual DBAppException component displays its balloon help.

Database Components

ABC's database components provide marginal improvements over the capabilities in the standard Delphi VCL. For the most part, they offer convenience, but little in the way of new functionality.

TDBSuperNav is an enhanced version of Delphi's *TDBNavigator*. It allows you to set the scroll rate for the scroll buttons to approximately one, two, or four times the normal speed. A new property causes some of the buttons to be colored red or green, or can be set to show everything in black as *TDBNavigator* does.

Four new buttons are added to *TDBNavigator*'s set. Two are used to move forward or backward, within a data source, by multiple records at a time (the exact number of records being controlled by a property). A new up/down pushbutton manipulates bookmarks within the data source. Depressing it sets a bookmark and restoring it to the up state jumps to the marked position and frees the bookmark. Finally, the fourth new button is for appending. It has the same functionality as Insert, but also automatically scrolls to the end of the data set.

TDBRecordCount provides a label that automatically displays the current record number and the number of records in a data source. This control operates only with Paradox databases. The format of the presentation is controlled by a mask such as "Record #R of #N."

TDBState provides another label, showing the current state of the data source (e.g. Browse, Edit, or Insert). The text used to describe the states can be changed.

Data-Aware Grids

Three controls deal with grids. One enhances the standard *TDBGrid*, while the other two combine several components to reduce the effort in displaying data-aware grids.

TDBFixedGrid enhances *TDBGrid* by adding properties to support fixed-grid columns. The fixed columns cannot be edited, and thus are suitable for displaying record keys. The new properties allow you to specify the number of fixed columns and the color in which they are displayed.

TDBTableGrid facilitates the creation of simple grids by combining the functions of *TTable*, *TDataSource*, *TDBGrid*, and *TDBSuperNav* into a single component. To create a grid, you simply add the component and set the *DatabaseName*, *TableName*, and *Active* properties that appear at the top of the Object Inspector. Delphi performs all the usual links automatically. The *TDBSuperNav* appears by default immediately above the grid, but can be moved to a position immediately below it as well.

TDBQueryGrid performs a similar function for query-based grids, combining the functions of *TQuery*, *TDataSource*, *TDBGrid*, and *TDBSuperNav*, and only requires you to set the *DatabaseName*, *SQL*, and *Active* properties.

Labels

ABC offers five variations on Delphi's Label control. The first two combine labels with editable fields, while the remaining three provide minor embellishments to standard labels.

TEditLabel combines an Edit and a Label into a single component that can be moved and sized as a unit. Properties control the size, alignment, and offset of the label portion. This can be helpful in creating forms with numerous labeled editable fields. However, the need to set the size and alignment properties on each control seems less convenient than other approaches (e.g. the Elastic control of VS/VBX), where such properties can be set on a form-wide basis.

TDBEditLabel is the data-aware equivalent, combining a DBEdit and a Label.

TAutoLabel adds properties to Delphi's *TLabel* to enable 3D appearance and to make the border visible in a specified color. A single-character separator can be automatically appended to the caption text. Unlike *TLabel*'s behavior, double-clicking on a *TAutoLabel* toggles its *AutoSize* property, rather than jumping to the control's *Click* event.

TSuperLabel provides a similar set of border enhancements to Delphi's *TLabel*, but without the separator and with no AutoSizing capability. In exchange, *TSuperLabel* provides top, bottom, left, and right margins, and a vertical alignment property, allowing precise positioning of label text.

TClockLabel, as the name implies, provides a label that shows the time and date. A variety of default formats are supplied, or you can specify a format string. The clock is updated automatically at intervals controlled by a property.

Window Backgrounds

Four components provide background graphics for windows. *TBackground* provides a gradient fill from one color to another. The fill may be horizontal, vertical, diagonal, or radial. This feature seems to interact badly with the *TClockLabel* component (or with other components that require it to repaint its background).

If the ClockLabel is transparent, for example, the color of the underlying form flashes for an extended time between clock updates. The problem is especially severe when using radial fills, which seem to take longer to redraw. Aside from this flaw, *TBackground* seems able to produce some attractive effects.

TTiledImage is an enhanced version of Delphi's *TImage* component with additional sizing options. The original image can be stretched in both directions to fill the dimension of the *TTiledImage*, or it can be repeated à la Windows wallpaper. *TMDIBackground* and *TMDITiledImage* provide similar func-

tions and options as *TBackground* and *TTiledImage*, but for the main form of an MDI application.

Buttons

ABC offers three new types of button controls:

- *TPicBtn* replaces Delphi's *TBitBtn* component. It allows pictures to be specified in icon, metafile, or bitmap format, and can be centered or tiled, in addition to the standard positioning options. Text can be given 3D effects, either raised or lowered.
- *TPicSpeedBtn* adds the same properties to Delphi's *TSpeedButton* as *TPicBtn* adds to *TBitBtn*.
- *TRepSpeedBtn* extends *TPicSpeedBtn* by adding timer functions to cause the *MouseDown*, *MouseUp*, and *Click* events to fire repeatedly as long as the user holds down the mouse button. Two intervals can be specified, indicating how long before the first repeated event fires and how long between repeated events.

Miscellaneous

ABC's remaining components are too diverse to classify, even within this mixed bag.

TMouseRegion defines a transparent region that can be superimposed on other objects (e.g. a map) to create "hotspots." The borders of the region may be visible or invisible. Standard events are triggered for *Click*, *DblClick*, *MouseDown*, *MouseMove*, and *MouseUp*.

TPopupSelect combines the functions of Delphi's Edit control with a variation on the popup menu. When the user clicks and holds down the mouse button over a PopupSelect field, the popup menu appears. (This interferes with the ability to edit the field via the mouse.) Unlike Delphi's *TPopupMenu*, the items on *TPopupSelect*'s menu are specified via a *TStringList*.

TStopWatch provides a timing function that can be used for performance testing. The timer can be started and stopped by invoking the *TStopWatch*'s *Start* and *Stop* methods. Multiple tests can be timed and averaged. The *Show* method displays the number of tests, the time of the last test, the total time for all tests, and the average time. This data can then be logged automatically to a specified file.

TSingleInstance is an unusual component. Simply placing one of these on your application's startup form prevents multiple instances of the application from running concurrently. If another instance is detected, *TSingleInstance* optionally displays a customizable message, then terminates execution or attempts to transfer control to the other instance.

TLauncher allows your application to execute a Windows program at a specified date and time. Launch times can be specified as a combination of date, time, day of week, and day of month (e.g. run payroll on the fifth of the month). Events are provided to gain control immediately before and after launching the program. The program's return code indicates the results, including those returned by the *WinExec* function.

TRSExit can be used to force ReportSmith to close. The control can close the design-time or run-time version, or both. ReportSmith can be forced to close programmatically, or automatically when the application ends.

TWin3D encapsulates CTL3DV2.DLL to automatically subclass Windows 3.x forms in the 3D style. Operating under Windows 95, I had no need for this component and was unable to verify its operation. The documentation was far too sketchy to give me any idea about its ease of use.

INFORMANT FACT FILE

ABC for Delphi is an assortment of VCL components that provides a variety of exception-handling, user-interface, and utility functions. The user manual is rather sparse, but a collection of demonstration programs helps illustrate use of the components. A demonstration version can be obtained from various online sites. Source code is available at extra cost.

Objective Software Technology
P.O. Box E138
Queen Victoria Terrace
CT 2600 Australia
Phone: +61 6-273-2100
E-Mail: CIS:100035,2441
Price: US$179; without source code US$89

TWinAbout makes it easy to create an attractive About box containing the project name and icon, the user's name and affiliation (from the `[MS User Info]` section of WIN.INI), and the amount of Windows memory and system resources available, plus up to two lines of customized text.

Unfortunately, the box is derived from a Microsoft standard, so (at least under Windows 95) it prefaces the project name with "Microsoft" and inserts a Microsoft copyright notice.

TWinMsgBox encapsulates the *MessageBox* function, allowing the relevant parameters to be specified by properties. Setting the *Beep* property causes the system sound associated with the *MsgType* to play immediately before displaying the message.

Pricing and Evaluation

ABC for Delphi is marketed by Objective Software Technology, an Australian company. (It's available in the US from ZAC Catalogs, (800) GO-DELPHI). It sells for US$179, without source code it's available for US$89.

On a component-per-dollar basis, this makes ABC a bargain. Even better, you can download a complete working version of the program (design-mode only) from CompuServe. Use the command GO DELPHI or GO ICGFORUM, then download ABC1DEMO.EXE and ABC1COMP.ZIP (see Figure 1).

Despite the low cost, I'm ambivalent about the package's value. None of the components strike me as "must-haves." A

few seem fairly helpful, but most provide no new functionality and offer only marginal improvements in convenience. Still, each seems to do something a little better or easier than would be possible without the package.

Conclusion

With this product, the whole is greater than the sum of its parts. When confronted with a problem that ABC could help to solve, having the components pre-installed on your Component Palette can save time and effort.

How valuable this is to you will depend on the nature of your application, whether you want to perform one of the functions that ABC provides, and how comfortable you feel trying to create the same functionality without ABC's help.

Considering the price, it would be difficult to argue that the purchase is a foolish one. Δ

Larry Clark is a principal of Logic Concepts, Inc., a Sacramento, CA-based software development firm. He has been programming since 1959, and now specializes in Delphi, Visual Basic, and Microsoft Access. An active member of the Sacramento PC Users Group, he leads the group's Delphi SIG, as well as its Visual Basic/Access SIG. He can be contacted at larry.clark@sacpcug.org.