To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming

(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Introduction

Way, *way* back in the mid-to-late 80s, the Pascal programming language was the target of a systematic slander attack by C and (later) C++ partisans, who got the ear of the media and said “Pascal is a kiddie language” so often that the media chowderheads took their word for it.

Most of these people knew nothing of Pascal, or perhaps took a college-level course in it from other chowderheads who considered drop-in code portability the sole virtue in all computer science. So the Pascal taught in schools is typically castrated Pascal that can’t do much more than iterate arrays and talk to the command line. C is no more portable than Pascal, but…c’mon, already: The whole issue is ridiculous because portability is and always was a myth. Quick, you C gurus: Write me a single, library-free C program that will locate the text cursor to 0,0 on *any* C implementation on *any* platform. See what I mean? *No es posible*. Arguing about portability is about as useful as discussing where UFOs come from.

A better measure of a language is how much it can accomplish—and how productive it makes the programmer. There was a time when C++ had a slight edge in power. But then Borland got hold of Pascal and added everything of value that C++ had. The “kiddie language” now had typecasts, pointers, objects, inline assembly, special hooks for Windows, the “woiks.” Those of us who used Pascal immediately leapt on the additional features, and before you knew it there were hordes of highly sophisticated applications everywhere you looked, all written in Borland Pascal.

Sometimes you can’t win. The C++ guys snorted and looked the other way, and the media chowderheads still call Pascal a kiddie language. It got so bad that a lot of commercial software vendors were afraid to admit that their applications were written in Pascal.

So Borland did the right thing. They dumped the P-word. Delphi, when it happened, stood on its own merits. It wasn't a language. It was a lean, mean, program-building machine. The sheer *depth* of the Delphi product is astonishing—you can wander for months in the help system and not see the same entry twice.

The potential in all that power was slow to be understood. We're only now starting to appreciate what you can do in Delphi. This book is meant to be a compendium of truly advanced Delphi techniques—stuff you can't do in a kiddie language, and stuff that isn't a cakewalk even in C++. It's proof, now and for all time, that Delphi goes all the way down to the metal and back in creating professional applications for Windows as good as anything you can create in any language you can name.

Having lost the P-word to kick around, the media chowderheads have begun repeating a new mantra, that anything you can do in Delphi takes five or six times as long in C++. It's gotten so bad I've heard tell of MIS shops where managers are forbidding the use of C++ and replacing it with Delphi and Visual Basic.

Hey, pass the chowder. There may yet be justice.

Jeff Duntemann KG7JF

Scottsdale, Arizona

July, 1996

Table of Contents

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

What's on the CD-ROM

The CD-ROM in this book exists primarily to carry the code listings for the projects developed in the book. However, there's so much room on a CD-ROM that we've added a lot of additional material that we felt might be useful to you as a Delphi programmer.

All of the projects have been tested on a 32-bit Windows operating system, either Windows 95 or Windows NT. Some will also work under Windows 3.x, but some may not. When a choice had to be made, we chose the 32-bit side of things.

If you have trouble getting one of the project executables to run, recompile it under Delphi 2. (Not all projects will compile under Delphi 1.) Make sure you have database aliases set up for projects that require them.

Listing File Updates

Every so often a source code file may have to be updated. It isn't always a matter of bugs. Sometimes, an author will send us an update that improves the code or adds new features, even if no bugs are involved. So check now and then! Typically, we'll provide an updated ZIP for the project in question rather than an update of the entire CD-ROM.

Book disk update files are available most easily through ftp, from:

ftp://ftp.coriolis.com/pub/bookdisk

There will be subdirectories for many books there. Look for a subdirectory name corresponding to KickAss Delphi. That's where the update files will be.

The CD-ROM Directory Structure

Here are the main directories on the CD and their contents:

BOBEXPRT

Collection of Delphi "Experts" by well-known Delphi guru Dr. Bob Swart of the Netherlands.

CONTROLS

Shareware and freeware VCL software components to use in creating your own Delphi applications.

DEMOS

Demo software from Delphi component manufacturers. These are *not* connected with projects in the book!

SOURCE

All the source code and other project files from the book.

TOOLS

WinG and Win32 libraries.

VDM

Sample articles and free subscription offer from *Visual Developer Magazine*, which has strong Delphi content in every issue, including Ray Konopka's column, "Delphi By Design."

The majority of the components and utilities on this CD are *shareware*, which requires that you pay the authors "on the honor system" if you intend to use the material on a regular basis. It is up to you to make sure that the authors of the program are compensated. Check the documentation files that are related to the application you are using to find out what the restrictions and regulations are. All shareware includes information on sending payment to the authors.

Coriolis Group Contact Information:

Web:	http://www.coriolis.com
Email:	techsupprt@coriolis.com
Phone:	602-483-0192
Mail:	7339 E. Acoma #7 Scottsdale, AZ 85260

If you need technical support or have a question about something in the book, please send us email describing the problem. Include a description of the involved hardware, software, and any other information that may help us clarify the difficulty.

Table of Contents

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

ABOUT THE AUTHORS

- Don Taylor is well known to the Borland Pascal community as the founder of the Turbo User Group and the host of GeTUGether, an annual party/seminar for users of Borland languages. Don has written extensively for *PC TECHNIQUES* and other magazines in recent years. He is co-author of *Delphi Programming Explorer* and *The New Delphi 2 Programming Explorer*. In addition to programming, Don enjoys writing and marketing. His company, Turbo Communications, is building a site on the Internet to serve the interests of those in the “Baby Boomer” generation (the site can be found at http://www.best.com/<turbocom/). Don and his wife Carol live in the hills above Poulsbo, Washington.
- Jim Mischel began writing on programming in 1989, and published his first book, *Macro Magic with Turbo Assembler*, in 1992. His second book, *The Developer’s Guide to WINHELP.EXE* has been the core around which Jim has developed a seminar series on developing Help content for Windows. He is a co-author of the recent Coriolis bestsellers *Delphi Programming Explorer* and *The New Delphi 2 Programming Explorer*. In his current position he develops cutting-edge games for Maxis. Jim lives in Austin, Texas, with his wife Debra, and operates amateur radio station KB7UQD when he isn’t out chasing armadillos through the bluebonnets.
- John Penman is a freelance programmer. He has been writing software since 1979, and has worked in real time and simulation software development as well as diverse activities such as crop husbandry and potato genetics in his native Scotland. John lives with his wife and two children in South Africa, and he can be reached on the Internet at jcp@iafrica.com.

- Terence Goggin is Director of Product Management at Nutshell Software Company, in Westlake Village, CA. Nutshell Software offers made-to-order programs as well as a number of add ons for Delphi, including the IM/EX ASCII™ components, which are designed to quickly and easily import and export ASCII data. He can be reached at terenceg@nutshellsotware.com. Nutshell Software can be reached at http://www.nutshellsoftware.com.
- Ray Konopka is the author of *Developing Custom Delphi Components*, published by Coriolis Group Books. Ray is the founder of Raize Software Solutions, Inc., supplier of Delphi component products and cnsulting services. Ray can be reached on the Internet at rkonopka@raize.com.
- Jon Shemitz has been a Pascal/Delphi consultant and independent developer for almost thirteen years. He lives in Santa Cruz, California with his partner and their two children, and is contemplating buying a laptop so he can work in his greenhouse. Jon can be reached on the Internet at jon@armory.com or http:/www.armory.com/<jon.
- Richard Haven is on the Delphi development team at Borland. He has been working at Borland for over five years, coming to Delphi out of the Paradox and database areas. He has been a client/server database consultant, and has presented technical sessions at many industry conferences around the world. He has over thirty technical articles in print. Contact Richard at rhaven@santacruz.com.
- Ed Jordan (whose last name is pronounced JER-din) develops custom Delphi Components and writes about programming, when he isn't teaching creative writing. He lives in Tampa, Florida.

Acknowledgments

Solomon is generally considered the wisest man in all of history. In Proverbs 3:26 he wrote, "Do not withhold good from those who deserve it, when it is in your power to act." That excellent advice also applies when "good" means thanks for a job well done.

Working with Jeff Duntemann is a pleasure that must be experienced to be fully appreciated. Jeff has a knack of leading the way and gently prodding you at the same time, all the while making sure you're tightly involved in the process of birthing a book. It's a marvelous skill that enables him to coax the very best from his authors. There may never be a way to turn the frustration of unexplained operating system quirks and beta software into something fun - but if it ever happens, Jeff will probably be the one to do it. Thanks, Jeff.

The production staff at the Coriolis Group did a fantastic job of getting everything together right and on time. Thanks Ron, Michelle, Toni and everyone else at Coriolis.

From its inception, Borland International has been a pioneer when it comes to programming tools. Delphi is the pinnacle of their efforts to date. It was hard to imagine how they could improve significantly on the original release, but they have managed to do it with Delphi 2. Thanks to Anders, Zack, Gary,

David and all the other wizards at Borland.

Collaborating on a book isn't supposed to be easy, but with the great group of guys who worked on this book you'd never know it. To Jim Mischel, John Penman and Terence Goggin: Thanks, guys, for sharing your expert knowledge, and for putting up with my *technohumor*.

Finally, thanks to everyone who purchases Coriolis Group books. In a world flooded with "formula" products from mainstream publishers, you've chosen to buy books from a company created by programmers for the express purpose of publishing meaningful stuff for other programmers. A company that dares to challenge the status quo, and is not afraid to break the rules now and then. To all of you: May you be blessed for taking the Road Less Traveled.

Don Taylor

Dedication

For Carol, who puts the wind in my sails and a song in my heart. Together we steer a course to the stars.

Don Taylor

In memory of Sandy. April 1, 1981 - June 4, 1996.

Jim Mischel

To Jocie, for being a wonderful and loving wife. To Mum and Dad for their unstinting support over the years.

John Penman

To dad.

Terence Goggin

Table of Contents

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

CHAPTER 1
32-Bit Console Applications

JIM MISCHEL

- Console I/O
- Console Filter Programs
- Console Command-Line Parsers
- Console File I/O
- Using Delphi‘s Object Repository

Now that DOS has been stuffed and hung on the wall of Win32 as a minor API, how’s a hacker to do command-line text filters? Well, the POSIX fairy waved her magic wand, and Poof! DOS became The Console, and it’s deja vu all over again.

Windows, OS/2, the Macintosh, and other graphical user interfaces have been the darlings of the computer press (both technical and non-technical) for many years now. With all that attention focused on writing applications for these GUI environments, it’s sometimes tough to remember that there is another world out there—a world of command-line tools that perform batch processing with a minimum of user input. These tools aren’t especially sexy, but they’re certainly useful. Banks, for example, still process your checks, deposits, and loan payments in batches every night. Insurance and credit card companies also perform nightly updates, as do countless other businesses. (And do they use fancy GUI environments? Ask your bank teller. Or guess.)

Command-line tools aren’t limited to companies running financial programs on big iron. Windows 95 itself comes with quite a few: ATTRIB, DISKCOPY,

FORMAT, FDISK, SORT, and XCOPY among them. Even Delphi comes with some command-line tools. A quick peek in Delphi's BIN directory reveals (among others) the Resource Compilers (BRC32.EXE and BRCC32.EXE), the Pascal compiler (DCC32.EXE), and the linker (TLINK32.EXE).

Console Applications

Windows 95 and Windows NT support *console applications*—programs that have no GUI presence but instead run in what is commonly referred to as a "DOS box." Although these applications don't have a window, they *do* have access to the Windows API and the full 32-bit Windows address space (including virtual memory). This is in contrast to Windows 3.1, where GUI programs had access to Windows' address space, and DOS programs had access to the lower 640K.

In the past, DOS applications got around the 640K limitation through the use of DOS extenders that supported standards like DPMI (DOS Protected Mode Interface) and VCPI (Virtual Control Program Interface). If you had a 16-bit extender, you had access to 16 megabytes of memory. The less common 32-bit extenders gave you access to the full 32-bit address space, and some even supported virtual memory. The problem with DOS extenders is that they are—no matter how well presented—hacks. And many users simply couldn't get their older machines to reliably run the DOS extenders, and some DOS extenders couldn't run in a Windows DOS box.

A console application running under Windows 95, on the other hand, is simply a Windows program without a window. There's no special extender software required, and any computer that can run Windows 95 or Windows NT will support console applications.

So we've got the RAM and we're free of the GUI. What can we *do* with it?

Filters

Probably the most common use of command line tools in the PC world is the broad category of programs called "filters." A filter can be anything from a very simple line counter to a complex compiler (like Delphi's Pascal compiler), a sorting utility, or a batch update program.

All filters operate basically the same way: They're invoked from the command line and passed arguments that specify options, and input and output files. The filter reads the input, applies some processing (such processing modified by the options specified on the command line), and writes an output file.

Filters typically don't access the mouse, and in fact, rarely accept user input. If they *do* accept user input, it's through a very simple text-oriented interface. Output to the user is normally limited to status reports ("Working, please wait..."), error notifications, and a final "done" message.

In this chapter, we're going to build a relatively simple filter program with Delphi, and construct a filter program "shell" that you can use to quickly build other filter programs. Along the way, we'll learn a thing or two about Delphi's

Object Repository, reusable code, and (shudder) *process-oriented* programming.

NOTE:
I find it ironic that three years ago I was explaining Windows programming to DOS programmers, telling them how to move away from their process-oriented mindset and move into the wide world of event-driven Windows programming. With the advent of visual development tools like Visual Basic and Delphi, many new programmers *started* with event-oriented programming and haven't ever written a process-oriented command line tool. Now I'm explaining process-oriented programming to event-oriented programmers. *Plus ça change.* One good thing, though: Programmers who understand event-oriented programming have little trouble understanding process-oriented code. The reverse, sadly, is not true.

Console Applications and Delphi

Although it's possible to write console applications with Delphi, the documentation is suspiciously silent on exactly how such a thing is done. Considering the excellent demo programs that explore so many other facets of Delphi, I found the lack of an example console application surprising. Fortunately, creating a console application with Delphi is not very difficult, although it would have been nice to be informed about a couple of the details. (Trial-and-error is *such* an inefficient way to learn!)

The simplest console app is, of course, the "Hello, world" program. It's not an exciting program, but it's usually the first program I ever write with a new tool because it lets me learn about the tool without having to worry too much about the program itself. And once we've created a simple console app with Delphi, we can save the code in the Object Repository so it can be used as the starting point for other console apps. Let's get started.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Hello, Delphi

Start with a new application (File|New Application). First we need to change some of the project options to tell Delphi that we're creating a console app. Select Project|Options, and on the Linker page of the Project Options dialog box, check the "Generate console application" check box, and then click on OK to save this change.

Since console applications don't have a main form (or any other form for that matter), we need to remove the Form1 that is automagically created when you start a new application. Select File|Remove from Project, and when the Remove From Project dialog box appears, highlight the line that contains Unit1 and Form1, and click on the OK button. If a message box appears and asks you if you want to save changes to Unit1, say No. You'll be left with a Delphi screen that has only the Object Inspector. No Form or Unit window will be shown. So where do you write the code?

The one thing that's left is the project source file. Select View|Project Source, and Delphi will display a text editor window that contains the source of PROJECT1.DPR. It's this file that we're going to modify in order to create our first console application. Before you do anything else, select File|Save, and save the project as HELLO.DPR.

Modify the project source in the editor so that it resembles Listing 1.1, save your work, and then press F9 to compile and run the program.

Listing 1.1 The "Hello, Delphi" program

```
{$APPTYPE CONSOLE}
program Hello;

uses Windows;
```

```
begin
  WriteLn ('Hello, Delphi!');
  Write ('Press Enter...');
  ReadLn;
end.
```

The first line in Listing 1.1 is a compiler directive that tells Delphi to create a console application, and *must* be included at the top of any console application. This line should *only* be included in programs—not units or libraries (DLLs). The **uses** statement isn't necessary for the program (after all, the program isn't making any Windows API calls), but for some reason, Delphi doesn't like to save a project that doesn't have a **uses** statement (see again my comment about trial-and-error learning). Windows seemed like a fairly innocuous unit to list here, and listing a unit doesn't mean that it's linked in—only that Delphi will search that unit if it can't find an identifier.

The rest of the program is real simple. The string "Hello, Delphi!" is output to the console (that is, the screen), and then you're prompted to press Enter. I included a prompt for the Enter key because, without it, Delphi just displays a console window ("DOS box") briefly, runs the program and displays the hello message, and then closes the console window. The prompt lets you see that the program actually works.

Saving a Program Template

The steps required to build a console app aren't especially difficult, but there are a few details to remember. Rather than building from scratch every time (and forgetting a detail or two), let's save our little Hello program in the Object Repository so that we'll have a starting point for other console apps.

Using the Windows Exploder (what we called File Mangler under Windows NT 3.51), create a subdirectory called ConsoleApp in Delphi's Objrepos subdirectory. If you installed Delphi using the standard options, the full path name will be:

```
C:\Program Files\Borland\Delphi 2.0\Objrepos\ConsoleApp
```

Then, select Project|Save Project as from Delphi's main menu, and save the project as ConsoleApp.dpr (don't you just *love* long file names?) in that new directory.

Once you've saved the project, add it to the repository by selecting Project|Add to Repository, and fill in the Add to Repository dialog box as shown in Figure 1.1.

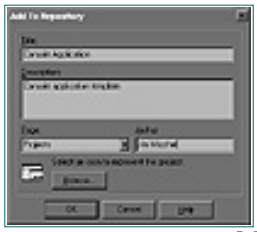

FIGURE 1.1 Adding the console app template to the repository.

Once you've added the project to the Repository, select File|New from Delphi's main menu, select the Projects page of the New Items dialog box, and

double-click on the “Console Application” icon. Delphi will prompt you for a directory and create a new project that has the options set for the console application.

NOTE:
I haven’t quite decided if it’s a good idea to store your own objects in Delphi’s Object Repository directory. Whereas it’s a handy place to put things, it’s also asking for trouble if you ever upgrade your version of Delphi. If you upgrade, it’s quite likely that Delphi’s Objrepos directory will be deleted—along with all of your cool objects. You’ll have to back up your objects before you upgrade.

You can, if you’d rather, create your own Repository directory that’s not a child of Delphi’s main directory. Either way, you’ll have to again add the projects to Delphi’s repository after you upgrade, but with a separate directory, you won’t run the risk of inadvertantly deleting your projects’ source code.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Previous | Table of Contents | Next

Console Input and Output

When a console application is started, the **Input** and **Output** standard text files are automatically associated with the console window. As a result, the **ReadLn** and **WriteLn** procedures work as expected, as do **Eof**, **Eoln**, **Read**, **Write**, and the other Text file I/O functions.

There are a number of console-specific I/O functions that can come in handy from time to time. Unfortunately, these console-specific functions are defined in the Windows console interface and there's no convenient Delphi component wrapper to insulate us from the gory details. (Now there's a good shareware project for an enterprising programmer: a Delphi class that encapsulates the Windows console interface.) The Windows console interface is a chapter in itself, so I'm going to conveniently ignore it here. If you're interested in learning more about **PeekConsoleInput**, **WriteConsole**, and the rest of the console API functions, check out the "Console Functions" entry in Delphi's online help.

Because we don't have space here to discuss the Console API, we're going to limit our console input and output to the standard Text file I/O routines. Don't take this the wrong way. The Console API functions are useful for some applications—just not for the kinds of applications that you'll typically write as console apps. Yeah, I know, it's confusing. It turns out that the Console API is much more useful for GUI programs that want to control a console window than it is for straight console applications controlling themselves.

Console apps aren't limited to a boring old text mode interface. Since you've got access to the full Windows API, you can display message and dialog boxes, control other windows, and even create another console from within your application.

Filter Programs in Delphi

Now that we know how to create a console application, let's put that knowledge to use. The rest of this chapter is concerned with writing filter programs as console apps. After an overview of how filter programs work, we're going to discuss processing the command line and efficient file operations. We'll be cutting a wide swath through Delphi's standard run-time library, and won't have time to discuss every function in detail. Remember that online help is your friend—use it early and often.

Your Basic Filter Program

As I mentioned at the beginning of this chapter, filter programs typically accept a command line that specifies options and input/output filenames, crunch the input as specified by the options, and produce the output file.

Given that general description, there's *lots* of room for improvisation. A line counter program, for example, could accept multiple input file names (including wildcards), and could have options that tell it to report not only the number of text lines in a file, but also the number of words and characters, and possibly a frequency distribution of words and characters. In a more involved program, the output can be a simple transformation of a single input file, a single file that combines multiple input files, or many different files created from a single input file.

Despite the differences in complexity, filters share a large amount of common functionality. They all process the command line, read input files, and write output files. Only the intermediate processing step changes significantly from one program to another. Because of this commonality, it's possible to build a group of functions that provide the common functionality and allows you to quickly build a custom filter program by simply defining how the command line is to be parsed and writing the code for the "processing" step. The input, output, and command line parsing portions are all there. Kind of a dehydrated filter program—just add processing.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Processing the Command Line

Command line processing sounds so simple. Given a text string that represents the command line, we want to parse out the file names and options, and set the program's variables accordingly. I'm continually amazed at how *difficult* such a simple-sounding thing can be. Fortunately, Object Pascal has two standard functions, **ParamCount** and **ParamStr**, that make things a bit easier.

ParamCount simply returns a count of the parameters on the command line. So if your command line is "MyFilter file1.txt file2.txt", **ParamCount** will return 2. The program name itself isn't counted as a parameter by this function.

ParamStr accepts an integer and returns a string that contains the command line argument that corresponds to that integer. For example, given the above command line, this statement

```
WriteLn (ParamStr (1));
```

will output 'file1.txt' (without the quotes).

If you pass 0 to **ParamStr**, the returned string will contain the full path and file name of the program that's currently being executed.

The Params program shown in Listing 1.2 illustrates the use of **ParamCount** and **ParamStr**. To create this program, select File|New from Delphi's main menu, select the Console Application item from the Projects page of the New Items dialog box, and then tell Delphi where to put your new application. Be sure to save the project as Params.dpr before you modify it.

Listing 1.2 The Params program

```
{$APPTYPE CONSOLE}
{
```

```
  Params -- a simple exploration of the ParamCount and
  ParamStr functions.
}
program params;

uses Windows;
Var
  i : Integer;

begin
  WriteLn ('Program:  ', ParamStr (0));
  WriteLn ('ParamCount = ', ParamCount);
  WriteLn ('Parameters');
  WriteLn ('----------');

  for i := 1 to ParamCount do
  begin
    WriteLn (ParamStr (i));
  end;

  Write ('Press Enter...');
  ReadLn;
end.
```

If you want to test the program from within Delphi, you need to select Run|Parameters from Delphi's main menu and enter the command line that you want to have passed to the program. For the above example, you'd enter the string "file1.txt file2.txt" (without the quotes) in the Run parameters dialog box.

Simple, no? Unfortunately, not so simple. Back in the days of DOS and Windows 3.1, things really *were* simple. But then along came long file names with embedded spaces. Now we have a problem. You see, **ParamCount** and **ParamStr** assume that command line arguments are separated by spaces. This works fine as long as your files don't have embedded spaces, but try this command line:

```
params c:\program files\borland\delphi 2.0\readme.txt
```

ParamCount returns 3, and the individual parameters it reports are

```
c:\program
files\borland\delphi
2.0\readme.txt
```

which is clearly *not* what we intended! (Okay, so maybe long file names aren't all peaches and cream—beer and skittles if you're British. *Warm* beer.)

I won't go into all the possible solutions to this problem. If you want a full discussion of this problem and the possible solutions (none of which are satisfactory, by the way—thank you Microsoft), get a copy of Lou Grinzo's *Zen of Windows 95 Programming*, also published by Coriolis Group Books. The book is ostensibly about C and C++ programming for Windows 95, but there's a wealth

of good information in there for all programmers, especially in the way of writing bug-free programs. This book is among the top three programming books I've ever read, along with *Writing Solid Code* and *Debugging the Development Process*, both written by Steve Maguire and published by Microsoft Press.

The only *workable* (although not satisfactory) solution to the embedded spaces problem is to require that file names containing embedded spaces be surrounded by quotation marks. So our sample command line becomes:

```
params "c:\program files\borland\delphi 2.0\readme.txt"
```

I guess you could require that your users always pass the short version of the file name, but I suspect they'd be more upset having to enter this

```
params c:\progra<1\borland\delphi<1.0\readme.txt
```

than they would be about the quotation marks.

Command Line Options

Most (definitely not all) command line programs get their parameters via the command line. Sometimes you'll see programs that receive their parameters from environment variables or configuration files, and some are hybrids that can accept parameters from the command line or from a configuration file whose name is specified on the command line. Since we don't want to get too bogged down in parameter processing, we're going to ignore the configuration files and environment variables, and concentrate solely on command line parameters.

You've probably used a command line tool (like DIR) that accepts options prefaced by a slash (/). For example, if you want a listing of files in the current directory and all of its subdirectories, you'd enter the command: DIR /S. Many programs also accept command line parameters prefaced by the dash (or minus sign, -). Both are common, and many programs will accept either.

File names, on the other hand, are specified in many different ways, depending on the tool. COPY, for example, lets you specify the name of the input and output files without prefacing them with option characters. So COPY FILE1 FILE2 copies FILE1 to FILE2. Borland's MAKE program, on the other hand, requires that you preface the input file name with the -f parameter. So, to process BUILD.MAK, you'd enter this command: MAKE -fbuild.mak.

The way MAKE processes command lines is easier, because *everything* is an option. Every option is separated from the others by at lease one space, and file names are handled just like other options—there aren't any special cases. This is the model we're going to use for our filter programs.

In general, there are four kinds of command line options: switches, numbers, strings, and file names. Switches simply turn an option on or off. A text filter, for example, might have a switch option to convert all lower-case characters to upper-case. Numbers can be integers or floating point, and can be specified in any number of ways: decimal and hexadecimal being the most common. Strings and file names are similar, although file names are often validated to ensure that they're properly formed.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

A Reusable Command Line Parser

If there's anything I dislike about programming, it's slogging through dozens (or hundreds) of lines of code to do something for the tenth (or hundredth) time. Command line parsing is like that: Every filter program has to process the command line, and command line parsing is *boring* code after you've written it once or twice. So I keep trying to come up with a generalized command line parser that will, with a minimum of effort on my part, parse a command line and fill in my program's options structure. That way, I'll have more time to spend on the filter (the real problem, after all) rather than on the command line parser.

A generalized command line parser is not an easy piece of code to write, and even a minimal one can be a bit involved. The one we'll develop here is minimal, but functional for many applications.

The basic idea is to define the valid option characters, the type of each option, and the default value for each option. The structure that contains this information is passed to the command line parser, which chews up the command line and fills in the values for the individual options that it finds. If it finds an error (an invalid option or a number where it expected a switch), it spits out an error message, stops processing, and returns an error status to the calling function. Simple, right? Ahhh...but not so easy.

An individual options record takes the format of the **OptionsRec** structure shown in Listing 1.3. This listing contains the full source of the CmdLine unit. You should create a new file in your editor, and enter and save this code as CMDLINE.PAS.

Listing 1.3 The CmdLine unit

```
{
  CMDLINE.PAS -- Command line parameters parsing
}
unit cmdline;

interface

type
  OptionType = (otBool, otInt, otString, otFilename);

  pOptionRec = ^OptionRec;
  OptionRec = record
    OptionChar : char;
    case Option : OptionType of
      otBool : (OnOff : Boolean);
      otInt : (Value : Integer);
      otString : (Param : ShortString);
```

```
      otFilename : (Filename : ShortString);
  end;

  pOptionsArray = ^OptionsArray;
  OptionsArray = Array [1..1] of OptionRec;

{
  GetOptionRec -- return a pointer to the options record in the
  passed Options array that corresponds to the specified option
  character.  Returns Nil if the option character is not in
  the passed Options array.
}
function GetOptionRec
      (
        Options : pOptionsArray;
        nOptions : Integer;
        OptionChar : char
      ) : pOptionRec;

{
  ProcessCommandLine -- process the command line according to the
  parameters list passed in the Options array.  Returns True if
  successful, or False if an error occurred in processing.
}
function ProcessCommandLine
      (
        Options : pOptionsArray;
        nOptions : Integer
      ) : Boolean;

implementation

uses SysUtils;

{
  GetOptionRec -- return a pointer to the options record in the
  passed Options array that corresponds to the specified option
  character.  Returns Nil if the option character is not in
  the passed Options array.
}
function GetOptionRec
      (
        Options : pOptionsArray;
        nOptions : Integer;
        OptionChar : char
      ) : pOptionRec;
var
  i : Integer;
begin
  Result := Nil;
  for i := 1 to nOptions do begin
    if (Options^[i].OptionChar = OptionChar) then begin
      Result := @Options^[i].OptionChar;
      Break;
    end;
  end;
end;

{
```

```
  ProcessBool

  Extract the on/off state for a parameter.  If the passed Param
  is a blank string, it is assumed to be On (+).  Otherwise the
  routine expects the string to start with + or -, and sets the
  OnOff variable accordingly.
}
function ProcessBool
      (
        Param : String;
        var OnOff : Boolean
      ) : Boolean;
begin
  Result := True;

  if (Length (Param) = 0) then begin
    OnOff := True;
    Exit;
  end;

  case Param[1] of
    '+' : OnOff := True;
    '-' : OnOff := False;

    else begin
      WriteLn ('Error:  + or - expected');
      Result := False;
    end;
  end;

end;

{
  ProcessInt

  Extract an integer from the passed command line parameter.
}
function ProcessInt
      (
        Param : String;
        var Value : Integer
      ) : Boolean;
begin
  if (Length (Param) = 0) then begin
    Result := False;
    WriteLn ('Error:  integer expected');
    Exit;
  end;

  Result := True;
  try
    Value := StrToInt (Param);
  except
    WriteLn ('Error:  integer expected');
    Result := False;
  end;
end;

{
```

```
  ProcessString

  Copy the passed string to the Option variable.  No error checking
  is performed, and a blank string is considered a valid parameter.
}
function ProcessString
      (
        Param : String;
        var Option : ShortString
      ) : Boolean;
begin
  Option := Param;
  Result := True;
end;

{
  ProcessFilename

  Extract a file name from the passed command line parameter.
  Currently, this function just calls ProcessString to copy the
  string parameter to the Filename.  It could, in the future,
  check to see if the string represents a valid file name, or it
  could be used to expand a short filename to a full path/file.
}
function ProcessFilename
      (
        Param : String;
        var Filename : ShortString
      ) : Boolean;
begin
  Result := ProcessString (Param, Filename);
end;

{
  CheckParam

  Check the passed Param, representing one command-line argument, against
  the list of options.  If the option character is valid, then process
  the option based on its type (Boolean, Integer, String, or Filename).

  Returns True if option processed and stored correctly, False otherwise.
}
function CheckParam
      (
        Param : String;
        Options : pOptionsArray;
        nOptions : Integer
      ) : Boolean;
var
  Rec : pOptionRec;
  Option : String;
begin
  Result := False;
  if (Param[1] in ['-', '/']) then begin
    if (Length (Param) < 2) then begin
      WriteLn ('Invalid option');
    end
    else begin
      Rec := GetOptionRec (Options, nOptions, Param[2]);
```

```
      if (Rec <> Nil) then begin
        Option := Copy (Param, 3, Length (Param) - 2);
        case Rec^.Option of
          otBool :
            Result := ProcessBool (Option, Rec.OnOff);
          otInt :
            Result := ProcessInt (Option, Rec^.Value);
          otString :
            Result := ProcessString (Option, Rec^.Param);
          otFilename :
            Result := ProcessFilename (Option, Rec^.Filename);
          else
            WriteLn ('Invalid option specification: ', Param[2]);
        end;
      end
      else begin
        WriteLn ('Invalid option character: ', Param[2]);
      end;
    end;
  end
  else begin
    WriteLn ('Error: options must start with - or /');
  end;
end;

{
  ProcessCommandLine

  Given a list of option characters and parameter types, check each
  command line argument against the list and set the values in the
  options structure accordingly.

  Returns True if all parameters processed and stored successfully.
}
function ProcessCommandLine
      (
        Options : pOptionsArray;
        nOptions : Integer
      ) : Boolean;

var
  ParamNo : Integer;

begin
  Result := True;

  for ParamNo := 1 to ParamCount do begin
    if (Not CheckParam (ParamStr (ParamNo),
                        Options, nOptions)) then begin
      Result := False;
      Exit;
    end;
  end;
end;

end.
```

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

OptionType is an enumerated type that describes the kinds of options that **ProcessCommandLine** knows about. The **OptionRec** record has three fields: the option character, the option type, and a variant portion that has a field that will hold the value for the particular option type. (If you're not familiar with variant records, take a peek at the help topic that discusses record types, or pick up a Pascal primer at your local bookstore.)

The **OptionRec** record, as coded, is a pretty inefficient way to solve this problem because each record, regardless of the type of option, occupies the maximum possible size. A **ShortString** type takes 256 bytes, which means that most of the records are much larger than they need to be. There are several ways around this, probably the most straightforward being to use *pointers* to strings, rather than strings themselves, for string and filename types. I choose not to implement that here because of the extra coding involved.

The other problem with this implementation also has to do with the **ShortString** type. The longest possible string that can be stored in a **ShortString** is 255 characters, which is shorter than the maximum path length Windows will accept (260 bytes). I had hoped to use Delphi's **AnsiString** (i.e. "long string") type for this reason, but long string types can't be stored in the variant portion of a record. Again, the most obvious solution would be to use string pointers.

Even with those problems, CmdLine is quite useful. The extra memory required shouldn't be a problem because most programs have only a handful of options, and there's no silly 64K limit on the size of static data anymore (it's a wide 32-bit world out there!). The filename length limitation is a bit of a bother, but I don't know too many people (like, none) who're going to be typing 256-character path names into a command line tool.

The CmdLine unit makes two functions available to calling programs: **GetOptionRec** and **ProcessCommandLine**. **GetOptionRec** will return a pointer to the record that corresponds to the specified option character. If no record exists for that option, then **GetOptionRec** returns **Nil**. **ProcessCommandLine** is the real workhorse. You pass it

an array of **OptionRec** structures, and it parses the command line, filling in the value fields for the individual options. If **ProcessCommandLine** processes all of the command line arguments without encountering an error, it returns **True**. If it encounters an error at any point, it immediately stops processing the command line, displays an error message, and returns **False** to the calling program.

Testing the CmdLine Unit

In order to test the command line parsing functions, we need a test program. Start a new application using the Console Application template. Save the new project as FILTER.DPR, and copy CMDLINE.PAS (Listing 1.3) to the directory that contains the new project. Then, select File|Add to Project to add the CmdLine unit to your new project.

The Filter project will be the testbed for the CmdLine unit and the file I/O unit that we'll be working on next. When we're done with those units, we're going to save the entire thing in the repository so that we have a template for other filter programs.

To test CmdLine, we need an array of options structures and some code that will call **ProcessCommandLine**. The test program, which you should enter into FILTER.DPR, is shown in Listing 1.4.

Listing 1.4 Testing the CmdLine unit with FILTER.DPR

```
{$APPTYPE CONSOLE}
program filter;

uses Windows, CmdLine;

const
  nOptions = 4;

  Options : Array [1..nOptions] of OptionRec = (
    (OptionChar : 'i'; Option : otFilename; Filename : ''),
    (OptionChar : 'o'; Option : otFilename; Filename : ''),
    (OptionChar : 'n'; Option : otInt; Value : 36),
    (OptionChar : 'd'; Option : otBool; OnOff : False)
  );

var
  cRslt : Boolean;
  Rec : pOptionRec;

begin
  cRslt := CmdLine.ProcessCommandLine (@Options, nOptions);
  WriteLn ('ProcessCommandLine returned ', cRslt);

  Rec := CmdLine.GetOptionRec (@Options, nOptions, 'i');
  WriteLn ('i = ', Rec^.Filename);
  Rec := CmdLine.GetOptionRec (@Options, nOptions, 'o');
  WriteLn ('o = ', Rec^.Filename);
```

```
  Rec := CmdLine.GetOptionRec (@Options, nOptions, 'n');
  WriteLn ('n = ', Rec^.Value);
  Rec := CmdLine.GetOptionRec (@Options, nOptions, 'd');
  WriteLn ('d = ', Rec^.OnOff);

  Write ('Press Enter...');
  ReadLn;
end.
```

The options table is initialized in the **const** section of the program, and then **ProcessCommandLine** is called to read the command line arguments and store the options' values in the options table. The program then displays the return result of **ProcessCommandLine**, and also displays the value of each option.

Try this program with a bunch of different command lines. Be sure to try some invalid command lines along with the valid ones, just to make sure that the error handling is working properly. Here are some suggested test cases:

```
-iInFile.txt -oOutFile.txt -n995 -d {valid}
-n8.94 {Error: integer expected}
-x     {Invalid option character: x}
```

The generalized command line parser provided in CmdLine makes picking out program options very easy. Just fill in a table, pass it to **ProcessCommandLine**, and it's done for you. All you have to do is ensure that any required options are specified, and set your program's internal variables according to the options that the user specified. Believe me, it's *much* easier than writing a custom command line parser for every different program.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

A Note on Program Structure

Before we start writing more involved programs, let's move the processing code from the project (.DPR) file to a separate unit. I've found that it's best to keep my own handwritten code out of the project file and in separate units, for several reasons.

For me, the most important point is that Delphi modifies the project file from time to time. I *think* it's only when you change the project name or add a new unit to the project, but I'm not sure. I don't know what Delphi is capable of changing, I haven't seen it fully documented anywhere, and I'd sure be upset if it changed something that I thought was constant. On the flip side, I might inadvertantly change something that Delphi put there for a reason. This in itself is reason enough for me. Delphi rarely touches non-form units (to my knowledge, only when you select File|Save as to change the unit name), so I feel safer with my code in separate units.

Another reason is that the project file is difficult to debug. For some reason, I had trouble setting breakpoints and single-stepping through code that was in the .DPR file.

Finally, the project file is just that—a project file. From the structure of the example programs and the way that Delphi creates projects, I get the impression that the .DPR file was never meant to contain large amounts of executable code. The project file gathers the project's units together for the project manager, and at run-time automatically, creates specific forms, and then runs the application. Seems to me that we should use the product in the way that it was intended.

So let's split out the processing code and make FILTER.DPR essentially a one-liner. Listing 1.5 is the new FILTER.DPR, and Listing 1.6 contains FILTMAIN.PAS—the module that now contains all of the processing code.

Listing 1.5 The new Filter project file

```
{$APPTYPE CONSOLE}
```

```
program filter;

uses
  cmdline in 'cmdline.pas',
  filtmain in 'filtmain.pas';

begin
  DoFilter;
end.
```

Listing 1.6 FILTMAIN: the Filter program's processing module

```
{
  FILTMAIN.PAS -- main processing module for Filter program.
}
unit filtmain;

interface

{ DoFilter does all the processing }
procedure DoFilter;

implementation

uses CmdLine;

procedure DoFilter;
const
  nOptions = 4;

  Options : Array [1..nOptions] of OptionRec = (
    (OptionChar : 'i'; Option : otFilename; Filename : ''),
    (OptionChar : 'o'; Option : otFilename; Filename : ''),
    (OptionChar : 'n'; Option : otInt; Value : 36),
    (OptionChar : 'd'; Option : otBool; OnOff : False)
  );

var
  cRslt : Boolean;
  Rec : pOptionRec;

begin
  cRslt := CmdLine.ProcessCommandLine (@Options, nOptions);
  WriteLn ('ProcessCommandLine returned ', cRslt);

  Rec := CmdLine.GetOptionRec (@Options, nOptions, 'i');
  WriteLn ('i = ', Rec^.Filename);
  Rec := CmdLine.GetOptionRec (@Options, nOptions, 'o');
  WriteLn ('o = ', Rec^.Filename);
  Rec := CmdLine.GetOptionRec (@Options, nOptions, 'n');
```

```
  WriteLn ('n = ', Rec^.Value);
  Rec := CmdLine.GetOptionRec (@Options, nOptions, 'd');
  WriteLn ('d = ', Rec^.OnOff);

  Write ('Press Enter...');
  ReadLn;
end;

end.
```

Now the project file contains just what it's supposed to contain—project build information and a "go" command. All of the programmer-written code is in FILTMAIN.PAS.

Reading and Writing Files

Once you've got command line parsing out of the way, the next big hurdle in a filter program is file I/O. Of course, if you're doing a simple character-by-character (or line-by-line) translation of a text file, you can use **Read** and **Write** (or **ReadLn** and **WriteLn**) in conjunction with **Eof** and **Eoln** to process your file. For example, the **DoFilter** procedure shown in Listing 1.7 copies characters from input to output, translating the lower-case characters to upper-case along the way.

Listing 1.7 Translating characters from upper-case to lower-case

```
procedure DoFilter;

const
  nOptions = 2;

  Options : Array [1..nOptions] of OptionRec = (
   (OptionChar : 'i'; Option : otFilename; Filename : ''),
   (OptionChar : 'o'; Option : otFilename; Filename : '')
  );

var
  cRslt : Boolean;
  iRec : pOptionRec;
  oRec : pOptionRec;
  InputFile : Text;
  OutputFile : Text;
  c : char;
begin
  cRslt := CmdLine.ProcessCommandLine (@Options, nOptions);
  if (not cRslt) then
    Halt;

  { make sure input and output files were specified }
  iRec := CmdLine.GetOptionRec (@Options, nOptions, 'i');
  if (iRec^.Filename = '') then begin
    WriteLn ('Error: input file expected');
```

```
    Halt;
  end;
  oRec := CmdLine.GetOptionRec (@Options, nOptions, 'o');
  if (oRec^.Filename = '') then begin
    WriteLn ('Error: output file expected');
    Halt;
  end;

  { open input file -- no error checking }
  Assign (InputFile, iRec^.Filename);
  Reset (InputFile);

  { create output file -- no error checking }
  Assign (OutputFile, oRec^.Filename);
  Rewrite (OutputFile);

  { Read and translate each character }
  while (not Eof (InputFile)) do begin
    Read (InputFile, c);
    c := UpCase (c);
    Write (OutputFile, c);
  end;

  Close (InputFile);
  Close (OutputFile);
end;
```

There are two problems with this version of FILTER. First, it's slow—kinda like a snake crawling out of a refrigerator. If you've got a megabyte-sized text file somewhere and a few minutes to spare, try it. The other problem is that the program only works on text files. That's fine for a one-shot filter application, but we're writing a filter template that's going to be used by many different types of programs, some of which will have to work with non-text files, and they'll all benefit from a speed improvement. What we need is a more general and much faster way to read characters (or bytes) from the file. We have to do our own buffering, which adds some complexity, but the results are well worth the effort.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

The **TFilterFile** class shown in Listing 1.8 is designed to give filter programs quick byte-by-byte access to files. It encapsulates all of the buffering and, as much as possible, relieves the programmer from having to remember the mundane details of file handling (although you do still have to **Open** and **Close** your files).

Listing 1.8 The TFilterFile class implemented in FILEIO.PAS

```
{
  FILEIO.PAS -- File input/output for filter programs
}
{$I+}  { use exceptions for error handling }
unit fileio;

interface

type
  FileIOMode = (fioNotOpen, fioRead, fioWrite);

  BuffArray = array[0..1] of byte;
  pBuffArray = ^BuffArray;

  TFilterFile = class (TObject)
  private
    FFilename : String;
    F : File;
    FBufferSize : Integer;
    FBuffer : pBuffArray;

    FBytesInBuff : Integer;
    FBuffIndx : Integer;
    FFileMode : FileIOMode;

    function ReadBuffer : boolean;
    function WriteBuffer : boolean;
```

```
  public
    constructor Create (AName : String; ABufSize : Integer);
    destructor Destroy; override;

    function Open (AMode : FileIOMode) : Boolean;
    procedure Close;

    function Eof : Boolean;

    function GetByte : byte;
    function PutByte (b : byte) : boolean;
  end;

implementation

{ TFilterFile }
{ Create -- sets up but doesn't actually open the file }
constructor TFilterFile.Create
      (
        AName : String;
        ABufSize : Integer
      );

begin
  inherited Create;
  FFilename := AName;
  FBufferSize := ABufSize;
  FBytesInBuff := 0;
  FBuffIndx := 0;
  FFileMode := fioNotOpen;

  { Assign but don't open }
  Assign (F, FFilename);
  { allocate memory for buffer }
  GetMem (FBuffer, FBufferSize);
end;

{ Destroy -- closes the file (if open) and destroys the object }
destructor TFilterFile.Destroy;
begin
  { if the file's open, close it }
  if (FFileMode <> fioNotOpen) then
    Self.Close;

  { if the buffer's been allocated, free it }
  if (FBuffer <> Nil) then begin
    FreeMem (FBuffer, FBufferSize);
    FBuffer := Nil;
  end;

  inherited Destroy;
end;

{ Open -- open the file in the proper mode }
```

```
function TFilterFile.Open
      (
        AMode : FileIOMode
      ) : Boolean;
var
  SaveFileMode : Byte;
begin
  Result := True;
  SaveFileMode := FileMode;  { FileMode defined in system unit }

  { try to open the file }
  try
    case AMode of
      fioRead : begin
        FileMode := 0;
        Reset (F, 1);
      end;

      fioWrite : begin
        FileMode := 1;
        Rewrite (F, 1);
      end;
    end;
    FFileMode := AMode;
  except
    Result := False;
  end;

  FBytesInBuff := 0;
  FBuffIndx := 0;

  FileMode := SaveFileMode;
end;

{ Close -- close the file, flushing the buffer if needed }
procedure TFilterFile.Close;
begin
  { if the write buffer has stuff in it, write it out }
  if ((FFileMode = fioWrite) and
      (FBytesInBuff > 0)) then begin
    WriteBuffer;
  end;

  try
    { close the file }
    System.Close (F);
  finally
    FFileMode := fioNotOpen;
  end;
end;
{ ReadBuffer -- read a block from the file into the buffer }
function TFilterFile.ReadBuffer : Boolean;
begin
  Result := True;
```

```
  if (Self.Eof) then
    Result := False
  else begin
    try
      BlockRead (F, FBuffer^, FBufferSize, FBytesInBuff);
    except
      Result := False;
    end;
  end;

end;

{ GetByte -- return next byte in file.  Read buffer if necessary }
function TFilterFile.GetByte : byte;
begin
  if (FBuffIndx >= FBytesInBuff) then begin
    if (not ReadBuffer) then begin
      Result := 0;
      Exit;
    end
    else
      FBuffIndx := 0;
  end;

  Result := FBuffer^[FBuffIndx];
  Inc (FBuffIndx);
end;

{ WriteBuffer -- write block from buffer to file }
function TFilterFile.WriteBuffer : Boolean;
begin
  Result := True;
  try
    BlockWrite (F, FBuffer^, FBytesInBuff);
  except
    Result := False;
  end;
  if (Result = True) then
    FBytesInBuff := 0;
end;
{ PutByte -- put byte into buffer.  Write to file if necessary }
function TFilterFile.PutByte (b : byte) : Boolean;
begin
  if (FBytesInBuff = FBufferSize) then begin
    if (not WriteBuffer) then begin
      Result := False;
      Exit;
    end
    else begin
      FBytesInBuff := 0;
    end;
  end;
```

```
  FBuffer^[FBytesInBuff] := b;
  Inc (FBytesInBuff);
  Result := True;
end;

{ Eof -- return True if at end of input file }
function TFilterFile.Eof : Boolean;
begin
  Result := (FBuffIndx >= FBytesInBuff);
  if Result then begin
    try
      Result := System.Eof (F);
    except
      Result := True;
    end;
  end;
end;

end.
```

Because **TFilterFile** handles most of the details, using it in place of standard text file I/O is very simple. The performance, though, isn't similar at all. The new **DoFilter** procedure shown in Listing 1.9 uses **TFilterFile** for input and output. The resulting program is *much* faster than the original. The beauty is that the program isn't any harder to read or understand than the slower version.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Listing 1.9 Using TFilterFile in place of standard I/O

```
procedure DoFilter;

implementation

uses CmdLine, FileIO;

procedure DoFilter;

const
  nOptions = 2;

  Options : Array [1..nOptions] of OptionRec = (
    (OptionChar : 'i'; Option : otFilename; Filename : ''),
    (OptionChar : 'o'; Option : otFilename; Filename : '')
  );

  BigBufferSize = 65536;


var
  cRslt : Boolean;
  iRec : pOptionRec;
  oRec : pOptionRec;
  InputFile : TFilterFile;
  OutputFile : TFilterFile;
  c : char;

begin
  cRslt := CmdLine.ProcessCommandLine (@Options, nOptions);
  if (not cRslt) then
    Halt;

  { make sure input and output files were specified }
```

```
  iRec := CmdLine.GetOptionRec (@Options, nOptions, 'i');
  if (iRec^.Filename = '') then begin
    WriteLn ('Error: input file expected');
    Halt;
  end;

  oRec := CmdLine.GetOptionRec (@Options, nOptions, 'o');
  if (oRec^.Filename = '') then begin
    WriteLn ('Error: output file expected');
    Halt;
  end;

  { Create and open the input file }
  InputFile := TFilterFile.Create (iRec.Filename, BigBufferSize);
  if (not InputFile.Open (fioRead)) then begin
    WriteLn ('Error opening input file');
    Halt;
  end;

  { Create and open the output file }
  OutputFile := TFilterFile.Create (oRec.Filename, BigBufferSize);
  if (not OutputFile.Open (fioWrite)) then begin
    WriteLn ('Error opening output file');
    Halt;
  end;

  { process and each character }
  while (not InputFile.Eof) do begin
    c := char (InputFile.GetByte);
    c := UpCase (c);
    if (not OutputFile.PutByte (byte (c))) then begin
      WriteLn ('Write error');
      Halt;
    end;
  end;

  InputFile.Close;
  InputFile.Free;

  OutputFile.Close;
  OutputFile.Free;
end;
```

Using the Filter Template

If you want to add the filter to the Repository, create a new directory under the ObjRepos directory, and save FILTER.DPR, FILTMAIN.PAS, CMDLINE.PAS, and FILEIO.PAS in that directory. Then just select Project|Add to Repository, and fill in the prompts. Next time you have to write a filter, all the tedious stuff is done. Just grab the template, fix up the options and change the processing loop.

A Critique

Have you ever had the urge, after you've finished a project, to turn around and do it right? Not that

there's anything *wrong* with this filter template, but now that it's finished, I look back and see where I could have done things differently.

In all, I'm pretty happy with the way it turned out, and I've used it to create some very useful programs, ranging in complexity from the one-shot quickie to a very useful stream editor.

The command line parsing is restrictive in its format because all parameters *must* be option-based, and it doesn't allow response (or configuration) files. Requiring options for all parameters isn't too much of a hassle, and it makes the code a lot easier. Adding support for response files would be very useful, and with the current design, shouldn't be terribly difficult. The only other thing I'd change (and mostly from a cosmetic point of view) is the use of **ShortString** in string and file name parameter types. **PString** or maybe **PChar** would be a more efficient choice.

TFilterFile is another story. This class implements the bare minimum required for filter I/O. You probably noticed that it doesn't have a block read or write mechanism, and it doesn't allow random file I/O. Many filter programs require one or both of those features. Block operations are fairly simple to add using an untyped **var** parameter and a byte count, in much the same way that the standard **BlockRead** and **BlockWrite** procedures work. These procedures would have to block copy bytes between the user's data structure and the object's buffer, and be sure to handle reading from and writing to the file as required.

I used methods for the **GetByte** and **PutByte** operations, rather than using properties. With some modest changes to **TFilterFile**, I could have defined these two properties

```
property InByte : byte read GetByte;
property OutByte : byte write PutByte;
```

and made **Eof** a property too, rather than a method. I guess I'm still stuck in the past. I can see the advantage of properties in many situations, but this isn't one of them. I see *some* advantages, but not enough to make me switch.

I'm unhappy with having to cast the return value of **GetByte** to a char in my program. I could have easily defined **GetChar** and **PutChar** methods in **TFilterFile**, but dang it, a character *is* a byte and I should be able to treat it that way. Here's one case where C gets it right and Object Pascal is too restrictive. It's rare, but it happens. The typecast is okay, I guess, but I normally avoid them because, in general, typecasts are considered "bad programming practice." You're telling the compiler: "Yes, I know I'm breaking the rules. Shut up and do it anyway." Second guessing the compiler is not a habit I'd like to get into.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Previous | Table of Contents | Next

CHAPTER 2
Drag and Drop the Windows Way

JIM MISCHEL

- How File Manager Drag and Drop (FMDD) Works
- Windows Messaging Under Delphi
- Encapsulating Drag And Drop Information In An Object
- Adding Drag And Drop Functionality To VCL Components
- Windows Subclassing

You'd think dragging and dropping your way around the Windows UI would be easier—but they had to leave something around for KickAss programmers to chew on, right?

There are at least three different drag and drop interfaces that can be supported by Delphi programs. The **TControl** class, of which all Delphi controls are descendents, defines a drag and drop interface between controls. By writing event handlers for the OnDragDrop, OnDragOver, and other similar events, a Delphi program supports internal drag and drop operations. With some fancy footwork and shared memory, it's possible that this method could be extended to work between two Delphi programs, but it can't be used to support drag and drop between a Delphi application and a non-Delphi program. This interface is well covered in Delphi's documentation and example programs.

Windows' Object Linking and Embedding (OLE) API also defines a drag and drop interface, which Delphi programs can support with the use of the built-in OLE controls. These controls will allow you to build OLE client or OLE

server applications that support full drag and drop with OLE objects. Again, Delphi's documentation and example programs provide a good explanation of this topic.

The third kind of drag and drop supported by Delphi is the dragging and dropping of files from (as I like to put it) File Mangler (Windows NT 3.5) or Windows Exploder (Windows 95 and NT 4.0). This interface is minimal—the only things you can drag and drop are files—but it's suprisingly useful. *This* interface, which I'll call "File Manager Drag and Drop" (FMDD), is not covered at all in Delphi's documentation, and is the subject of this chapter.

Drag and Drop the Windows Way

FMDD is implemented in Windows through the Shell interface in SHELL32.DLL. The implementation consists of four Windows API functions: **DragAcceptFiles**, **DragQueryFile**, **DragQueryPoint**, and **DragFinish**; and one Windows message: **WM_DROPFILES**. In Delphi, the **WM_DROPFILES** message is defined in the Messages unit, and the API functions are declared in the ShellAPI unit. The documented interface supports FMDD *clients*, but not FMDD *servers*. Your programs can accept dropped files from File Manager, but they can't send files to another program.

A typical implementation of FMDD in a Windows program requires that your code perform the following steps:

1. At program startup, call **DragAcceptFiles**, passing it a window handle and a **True** flag to enable acceptance of dragged files by that window.

2. When the window receives a **WM_DROPFILES** Windows message, perform the following (the **Msg.wParam** field in the Object Pascal message structure is a handle to memory used by the **WM_DROPFILES** message):

a. Call **DragQueryPoint** to determine if the drop occurred in the client area of the window.

b. Call **DragQueryFile** with an index value of $FFFFFFFF to retrieve the number of files being dropped.

c. For each file, call **DragQueryFile** to copy the file name to an internal buffer.

d. Perform the desired action on each file name.

e. Free all internal memory allocated during processing of the dropped files.

f. Call **DragFinish** to free the memory allocated by the FMDD server (i.e. File Mangler).

3. At program shutdown, call **DragAcceptFiles**, passing it a window handle and a **False** flag to discontinue acceptance of dragged files by the window.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Drag1, the first draft of a program that implements FMDD, is shown in Listings 2.1 and 2.2. A screen shot of the completed program is shown in Figure 2.1.

Listing 2.1 DRAG1.DPR

```
program drag1;

uses
  Forms,
  dragfrm1 in 'dragfrm1.pas' {Form1};

{$R *.RES}

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

Listing 2.2 DRAGFRM1.PAS

```
unit dragfrm1;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls,
  {
     ShellAPI defines the drag and drop functions.
     The functions are implemented in SHELL32.DLL.
  }
  ShellAPI;
type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    Button1: TButton;
    Button2: TButton;
    Label1: TLabel;
```

```
    Label2: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure AppMessage(var Msg: TMsg; var Handled: Boolean);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
    procedure WMDropFiles (hDrop : THandle; hWindow : HWnd);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMessage := AppMessage;
  {
    Call DragAcceptFiles to tell the drag and drop manager
    that you're accepting files.
  }
  DragAcceptFiles (Handle, True);
end;

procedure TForm1.WMDropFiles (hDrop : THandle; hWindow : HWnd);
Var
  TotalNumberOfFiles,
  nFileLength : Integer;
  pszFileName : PChar;
  pPoint : TPoint;
  i : Integer;
  InClientArea : Boolean;
Begin
  {
    hDrop is a Handle to the internal Windows data
    structure which has information about the dropped files.
  }

  {
    Determine if the files were dropped in the client area
  }
  InClientArea := DragQueryPoint (hDrop, pPoint);
  if InClientArea then
    Label2.Caption := 'In client area'
  else
    Label2.Caption := 'Not in client area';

  {
    Find out total number of files dropped by passing -1 for
    the index parameter to DragQueryFile
  }
```

```
  TotalNumberOfFiles := DragQueryFile (hDrop , $FFFFFFFF, Nil, 0);

  for i := 0 to TotalNumberOfFiles - 1 do begin
    {
      Get the length of a filename by telling DragQueryFile
      which file your querying about ( i ), and passing Nil
      for the buffer parameter.  The return value is the length
      of the file name.
    }
    nFileLength := DragQueryFile (hDrop, i , Nil, 0) + 1;
    GetMem (pszFileName, nFileLength);

    {
      Copy a file name.   Tell DragQueryFile the file
      you're interested in (i) and the length of your buffer.
      NOTE: Make sure that the length is 1 more than the filename
      to make room for the nul character!
    }
    DragQueryFile (hDrop , i, pszFileName, nFileLength);

    Listbox1.Items.Add (StrPas (pszFileName));

    { free the allocated memory... }
    FreeMem (pszFileName, nFileLength);
  end;
  {
   Call DragFinish to release the memory that Shell allocated
   for this handle.
   NOTE: This is a real easy step to forget and could
   explain memory leaks and incorrect program performance.
  }
  DragFinish (hDrop);
end;

{
  AppMessage captures application messages.  Assign this method
  to the Application.OnMessage property in FormCreate.
}
procedure TForm1.AppMessage(var Msg: TMsg; var Handled: Boolean);
begin
  case Msg.Message of
    WM_DROPFILES :  WMDropFiles (Msg.wParam, Msg.hWnd);
  end;
end;

procedure TForm1.FormClose (Sender: TObject; var Action: TCloseAction);
begin
  { Don't accept files anymore }
  DragAcceptFiles (Handle, False);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Listbox1.Clear;
end;

procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
  Close;
end;

end.
```

FIGURE 2.1 The completed Drag1 program.

The only real trick in DRAGFRM1.PAS is the line in **TForm1.FormCreate** that reads:

```
Application.OnMessage := AppMessage;
```

This line of code tells Delphi to pass Windows messages on to the **TForm1.AppMessage** procedure. This is the Delphi way of performing traditional handling of Windows messages. We have to do it this way because neither Delphi's **TControl** class, nor any of its descendents (like **TForm**), know anything about **WM_DROPFILES**, so that message isn't wrapped up in a nice Delphi event like OnDropFiles. More's the pity.

There's nothing especially *wrong* with the code in Listing 2.2. It works, which is the most important thing, but it's large, it has lots of places where you can make mistakes, and (perhaps worst of all) it's *ugly*. All that icky Windows code in the middle of a pure Delphi program *just ain't right!* (Texans have such colorful ways of mangling the language.)

There's another problem, too, but it's caused by Delphi's message handling machinery. Suppose you have two forms that want to respond to **WM_DROPFILES** messages. If each form assigns its own message handler to **Application**'s OnMessage event, only the second form will get messages. The first form's message handler is overwritten by the second. There are a number of ways around this problem, and we'll discuss some of them after we've taken care of that ugly Windows code.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

What to Do with Windows Code?

The answer to the question is “encapsulate it.” That’s what Delphi does, and does very well. The whole *point* of Delphi is to insulate you from the sordid little details of Windows programming so that you can concentrate on the important parts of your application. We’ll do the same with FMDD—wrap it up into a pretty little Delphi unit called FMDD.

Rather than have the form handle all of the details of **WM_DROPFILES** processing, we’ll define a function in the FMDD unit that the form’s OnMessage handler can call to retrieve an object that contains all available information about the drag and drop event. This object will include everything that we obtained from the Windows FMDD interface, but it will be all packaged together in a single, more manageable structure that looks like the following:

```
TDragDropInfo = class (TObject)
  private
    FNumFiles : UINT;
    FInClientArea : Boolean;
    FDropPoint : TPoint;
    FFileList : TStringList;
  public
    constructor Create (ANumFiles : UINT);
    destructor Destroy; override;

    property NumFiles : UINT read FNumFiles;
    property InClientArea : Boolean read FInClientArea;
    property DropPoint : TPoint read FDropPoint;
    property Files : TStringList read FFileList;
end;
```

In addition to the **TDragDrop** structure, the FMDD unit defines three functions: **AcceptDroppedFiles**, **UnacceptDroppedFiles**, and **GetDroppedFiles**. The first two simply encapsulate the **DragAcceptFiles** function. The third, **GetDroppedFiles**, is called in response to the **WM_DROPFILES** message and returns a **TDropInfo** object. The code for the FMDD unit is shown in Listing 2.3.

Listing 2.3 The FMDD unit that encapsulates the Drag and Drop interface

```
{
  FMDD.PAS -- File Mangler Drag and Drop
}
```

```
unit fmdd;

interface

uses Windows, Classes;

type
  TDragDropInfo = class (TObject)
  private
    FNumFiles : UINT;
    FInClientArea : Boolean;
    FDropPoint : TPoint;
    FFileList : TStringList;
  public
    constructor Create (ANumFiles : UINT);
    destructor Destroy; override;

    property NumFiles : UINT read FNumFiles;
    property InClientArea : Boolean read FInClientArea;
    property DropPoint : TPoint read FDropPoint;
    property Files : TStringList read FFileList;
  end;

function GetDroppedFiles (hDrop : THandle) : TDragDropInfo;
procedure AcceptDroppedFiles (Handle : HWND);
procedure UnacceptDroppedFiles (Handle : HWND);

implementation

uses ShellAPI;

constructor TDragDropInfo.Create (ANumFiles : UINT);
begin
  inherited Create;
  FNumFiles := ANumFiles;
  FFileList := TStringList.Create;
end;

destructor TDragDropInfo.Destroy;
begin
  FFileList.Free;
  inherited Destroy;
end;

procedure AcceptDroppedFiles (Handle : HWND);
begin
  DragAcceptFiles (Handle, True);
end;

procedure UnacceptDroppedFiles (Handle : HWND);
begin
  DragAcceptFiles (Handle, False);
end;

function GetDroppedFiles (hDrop : THandle) : TDragDropInfo;
var
  DragDropInfo : TDragDropInfo;
```

```
  TotalNumberOfFiles,
  nFileLength : Integer;
  pszFileName : PChar;
  i : Integer;

begin
  {
    hDrop is a Handle to the internal Windows data
    structure which has information about the dropped files.
  }
  {
    Find out total number of files dropped by passing -1 for
    the index parameter to DragQueryFile
  }
  TotalNumberOfFiles := DragQueryFile (hDrop , $FFFFFFFF, Nil, 0);

  DragDropInfo := TDragDropInfo.Create (TotalNumberOfFiles);

  {
    Determine if the files were dropped in the client area
  }
  DragDropInfo.FInClientArea :=
    DragQueryPoint (hDrop, DragDropInfo.FDropPoint);

  for i := 0 to TotalNumberOfFiles - 1 do begin
    {
      Get the length of a filename by telling DragQueryFile
      which file your querying about ( i ), and passing Nil
      for the buffer parameter.  The return value is the length
      of the file name.
    }
    nFileLength := DragQueryFile (hDrop, i , Nil, 0) + 1;
    GetMem (pszFileName, nFileLength);

    {
      Copy a file name.   Tell DragQueryFile the file
      you're interested in (i) and the length of your buffer.
      NOTE: Make sure that the length is 1 more than the filename
      to make room for the nul character!
    }
    DragQueryFile (hDrop , i, pszFileName, nFileLength);

    { Add the file to the string list }
    DragDropInfo.FFileList.Add (pszFileName);

    { free the allocated memory... }
    FreeMem (pszFileName, nFileLength);
  end;

  {
   Call DragFinish to release the memory that Shell allocated
   for this handle.
   NOTE: This is a real easy step to forget and could
   explain memory leaks and incorrect program performance.
  }
  DragFinish (hDrop);
  Result := DragDropInfo;
```

```
end;

end.
```

In order to make the existing test program use the new interface, we just have to change a few lines of code. First, change the reference to unit ShellApi in the unit's **uses** statement to FMDD. Then, change the form's event handlers as shown in Listing 2.4.

Listing 2.4 Using the new File Manager Drag and Drop interface

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMessage := AppMessage;
  FMDD.AcceptDroppedFiles (Handle);
end;
procedure TForm1.WMDropFiles (hDrop : THandle; hWindow : HWnd);
var
  DragDropInfo : TDragDropInfo;
  i : Integer;

begin
  DragDropInfo := FMDD.GetDroppedFiles (hDrop);

  { Determine if in client area }
  if DragDropInfo.InClientArea then
    Label2.Caption := 'In client area'
  else
    Label2.Caption := 'Not in client area';

  { add each file to the list box }
  for i := 0 to DragDropInfo.NumFiles - 1 do begin
    Listbox1.Items.Add (DragDropInfo.Files[i]);
  end;

  { Destroy the DragDropInfo object }
  DragDropInfo.Free;
end;

procedure TForm1.FormClose (Sender: TObject; var Action: TCloseAction);
begin
  { Don't accept files anymore }
  FMDD.UnacceptDroppedFiles (Handle);
end;
```

I don't know about you, but I find it much easier to use this new interface. In the spirit of Delphi, we've removed the Windows handling code from our application's code, and we've put it into a unit where we don't have to look at it or worry about it. The FMDD unit handles all the mucking around in Windows internals and returns an object—something that we know how to handle. The result is less (and much cleaner) code that's easier to write and maintain.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Responding to Windows Messages

In most cases, Delphi's interface for handling Windows messages—**Application**'s OnMessage event handler—is sufficient. Programs can define their own **OnMessage** handler, and Delphi will dutifully pass Windows messages on to that procedure. But Delphi doesn't allow you to define multiple OnMessage handlers in a single program, making custom processing of Windows messages for multiple windows somewhat of a problem. In our case, it's a real problem—only one control in an application can be accepting dropped files.

The first solution that comes to mind is to have an event handler that knows about all of the controls that need to process Windows messages. This event handler can then compare the **Msg.hwnd** value with the **Handle** properties of each control, passing the message on to the proper control's event handler. This is certainly possible, but it requires that your program know—at compile time—all of the controls that might want to process Windows messages.

A related solution would be to define an OnMessage event chaining facility, whereby controls that want to process Windows messages link themselves to a chain of OnMessage event handlers, and the chaining facility itself hooks Application, OnMessage, and then searches the list of linked controls whenever a message comes in. Controls could then link and unlink themselves from the OnMessage chain at will, although your code will have to take care to prevent breaking the chain and leaving some controls hanging without messages.

Both of these solutions have one major drawback—they require that all controls that will be handling Windows messages know about how the main application is handling the message chain. The lower-level portions of your program have to know about the higher-level implementation. This is not a good thing.

Suppose, for example, your latest masterpiece is finished except for a really cool spreadsheet control. Flipping through the latest copy of *Hacker Monthly*, you find a review of *Spreadsheet MAX*, the coolest spreadsheet control ever invented. It's *perfect* for your application and you order a copy. When it arrives, you find that it works flawlessly except that it stomps on the Application.OnMessage event handler to do its thing—completely wiping out the OnMessage chaining that you had set up. How was the spreadsheet supposed to know that you were providing message chaining?

If there's a reliable way to make multiple controls properly handle the Application.OnMessage event handler, I certainly haven't found it. So my solution is to forget it—don't use Application.OnMessage at all if you have multiple windows that need to process Windows messages. There are alternative methods of de-furring felines.

Custom Controls

If you have a special type of control that you want to respond to particular messages, simply write a custom

version of the control. For example, if you wanted a **TForm** descendent that responds to **WM_DROPFILES**, you could create the custom **TFMDDForm** control shown in Listing 2.5.

Listing 2.5 The TFMDDForm custom component

```
{
  FMDDFORM.PAS -- implements a form that responds to the
  WM_DROPFILES Windows message.
}
unit fmddform;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, FMDD;

type
  TFMDDEvent = procedure (Sender: TObject; DragDropInfo : TDragDropInfo)
    of object;
  TFMDDForm = class(TForm)
  private
    { Private declarations }
    FOnFMDD : TFMDDEvent;
    procedure WMDropFiles (var Message: TMessage);
      message WM_DROPFILES;
  protected
    { Protected declarations }
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    { Published declarations }
    property OnFMDD: TFMDDEvent read FOnFMDD write FOnFMDD;
  end;

procedure Register;

implementation

constructor TFMDDForm.Create(AOwner: TComponent);
begin
  inherited Create (AOwner);
  FMDD.AcceptDroppedFiles (Handle);
end;

destructor TFMDDForm.Destroy;
begin
  FMDD.UnacceptDroppedFiles (Handle);
  inherited Destroy;
end;

procedure TFMDDForm.WMDropFiles (var Message: TMessage);
var
  DragDropInfo : TDragDropInfo;
begin
  if assigned (FOnFMDD) then begin
```

```
    DragDropInfo := FMDD.GetDroppedFiles (Message.wParam);
    FOnFMDD (Self, DragDropInfo);
    DragDropInfo.Free;
  end;
end;

procedure Register;
begin
  RegisterComponents('Samples', [TFMDDForm]);
end;

end.
```

If you have the stomach for it, you could dig into the source for **TWinControl** and create an OnFMDD event so that all windowed controls knew about the **WM_DROPFILES** windows message. That'd take care of drag and drop for good, but it's a rather drastic measure, and it doesn't do you any good if all of a sudden you want a control to respond to multiple user-defined messages whose values aren't even defined until run-time. There's a more general (and more involved) solution that's completely flexible.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Subclassing

The problem of custom responses to Windows messages isn't new to Delphi—it's been around for as long as Windows has been around. They've even got a name for the technique: *subclassing*. Technically, there's subclassing, and then there's *superclassing*, the difference being that subclassing restricts a windows' response to messages, and superclassing adds to the processing of a windows' message. In my mind, they're the same thing simply because you use the same technique to implement either. The technique? Well, it's kind of ugly when compared to the elegance of Delphi (okay, it's *really* ugly), but it works wonders, and we can encapsulate the ugliness so you never see it in your programs.

The idea behind subclassing is pretty simple. Every window has an associated data structure that Windows knows about. Among the many wonderous things in that structure is a pointer to the windows' *window procedure*—the procedure that processes Windows messages. When Windows receives a message that's destined for a window, it looks up the address of that windows' window procedure and calls it, passing the information pertinent to the particular message. When you subclass a window, you replace the windows' window procedure with your custom routine, and save a pointer to the old procedure so you can pass messages on to it. All of this stuff is documented in the Windows SDK manuals, and there are some fairly good examples (all in C—you can't have *everything*). True, it's a bit-twiddling and hackerish kind of thing to do, but sometimes you just have to get your hands dirty. (Have you taken a look at the VCL source code lately? If you want an eye-opening experience, glance at CONTROLS.PAS sometime.)

At any rate, Delphi gives us all the tools we need to subclass a window, and we can use those tools to create a drag and drop interface that your programs can interact with in the normal Delphi fashion. As always, we'll start with the requirements.

Defining the Interface

We want drag and drop to operate as much as possible like a normal Delphi event. Since we're not defining a new custom control, we can't define an OnFMDD event that can be assigned at design-time. So we have to simulate that behavior at run-time. To do that, we need to:

1. Define a **TFMDDEvent** type for the event handler.
2. Declare an OnFMDragDrop event handler in the form's **private** section.
3. When the form is created, pass the address of the event handler to the FMDD interface so the interface knows that we want the form to accept dropped files.
4. When a drag and drop event occurs (that is, when the form receives a **WM_DROPFILES** message), the FMDD interface will call the OnFMDragDrop event handler, passing it a **TDragDropInfo** object.
5. When the form is closed, call the FMDD interface to stop accepting dropped files.

That requirements list leads us to the **interface** section shown in Listing 2.6.

Listing 2.6 The interface section of the new FMDD unit

```
interface

uses Windows, Messages, Classes, Controls;

type
  TDragDropInfo = class (TObject)
  private
    FNumFiles : UINT;
    FInClientArea : Boolean;
    FDropPoint : TPoint;
    FFileList : TStringList;
  public
    constructor Create (ANumFiles : UINT);
    destructor Destroy; override;

    property NumFiles : UINT read FNumFiles;
    property InClientArea : Boolean read FInClientArea;
    property DropPoint : TPoint read FDropPoint;
    property Files : TStringList read FFileList;
  end;

  TFMDDEvent = procedure (DDI : TDragDropInfo) of object;
procedure AcceptDroppedFiles (Control : TWinControl;
                              AOnDrop : TFMDDEvent);
procedure UnacceptDroppedFiles (Control : TWinControl);
```

Note that the **TDragDropInfo** object remains the same. We've removed the **GetDroppedFiles** function and redefined the **AcceptDroppedFiles** and **UnacceptDroppedFiles** procedures. The result is a much cleaner interface than the previous one—one that doesn't expect you to know about ugly little things like window handles and Windows messages. Of course, *somebody* has to know about

those things. The details are hidden in FMDD's **implementation** section.

Implementing the New Interface

The devil, as always, is in the details. FMDD has to do a lot of processing behind the scenes. There are three separate, but interrelated, parts to FMDD's processing:

1. **AcceptDroppedFiles** has to save the window handle of the passed control and the OnDrop event handler for future use. This procedure also must call **DragAcceptFiles** to enable **WM_DROPFILES** processing for this window, and then subclass the window so that it can respond to the message.
2. We need a Windows message handler that will respond to **WM_DROPFILES** messages by constructing a **TDragDropInfo** object and passing it to the appropriate control.
3. **UnacceptDroppedFiles** should un-subclass the window and call **DragAcceptFiles** to prevent future **WM_DROPFILES** messages from being sent to the window.

Because FMDD should allow multiple windows to accept dropped files, we'll have to keep a list of window handles and their associated event handling procedures. When **AcceptDroppedFiles** is called, it'll save the control's information in the list. The procedure that handles the **WM_DROPFILES** message will look up the window's handle in the list so that it knows which object to send an OnFMDragDrop event to, and then the **UnacceptDropped Files** procedure has to remove the control's information from the list. Fortunately, Delphi's **TList** component is tailor-made for list processing, and it's a snap to add items to, delete items from, and look up items in a **TList**.

The tricky part of the implementation is the subclassing—mostly because it deals with Windows arcana. I touched briefly on what subclassing is previously, but purposely left out any discussion of how to do it until we got to the implementation. It's time now to get our hands dirty.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Subclassing Revisited

To subclass a window, you need to obtain and save a pointer to the window's existing window procedure, and then set the new window procedure pointer in the window's data structure. To do this, you call Windows' **GetWindowLong** and **SetWindowLong** API functions, which retrieve and set values in the window's internal data structure.

Once you've successfully subclassed a window, Windows will send all messages destined for that window to the new window procedure. It's this procedure's job to respond to the messages that it's interested in (in our case, **WM_DROPFILES**), and then pass all of the other messages on to the previous window procedure—a pointer to which you saved when you subclassed the window. You can't just call that old window procedure, though. Instead, you have to call the **CallWindowProc** API function and pass it the address of the old window procedure and the parameters that Windows passed to you.

The last part of subclassing is *un*subclassing—putting things back the way you found them. Unsubclassing is a simple matter of calling **SetWindowLong** again, this time replacing the new window procedure with the old one.

It all sounds a little more complicated than it really is, and once you've seen and puzzled over an example, it becomes as clear as mud (which is about as clear as *any* Windows programming gets).

The new FMDD unit that supports subclassing is shown in Listing 2.7.

Listing 2.7 The new FMDD unit that supports multiple controls

```
{
  FMDD.PAS – File Mangler Drag and Drop
}

unit fmdd;

interface

uses Windows, Messages, Classes, Controls;

type
  TDragDropInfo = class (TObject)
```

```
  private
    FNumFiles : UINT;
    FInClientArea : Boolean;
    FDropPoint : TPoint;
    FFileList : TStringList;
  public
    constructor Create (ANumFiles : UINT);
    destructor Destroy; override;

    property NumFiles : UINT read FNumFiles;
    property InClientArea : Boolean read FInClientArea;
    property DropPoint : TPoint read FDropPoint;
    property Files : TStringList read FFileList;
  end;

  TFMDDEvent = procedure (DDI : TDragDropInfo) of object;

procedure AcceptDroppedFiles (Control : TWinControl;
            AOnDrop : TFMDDEvent);
procedure UnacceptDroppedFiles (Control : TWinControl);

implementation

uses ShellAPI;

type
  {
    TSubclassItem stores information about a subclassed window
  }
  TSubclassItem = class (TObject)
  private
    Handle : HWND;           { the window handle }
    WindowProc : TFNWndProc; { its old window procedure }
    FOnDrop : TFMDDEvent;    { control's OnFMDragDrop event handler }
  public
    constructor Create (AHandle : HWND;
       AWndProc : TFNWndProc; AOnDrop : TFMDDEvent);
  end;

var
  SubclassList : TList;

constructor TSubclassItem.Create (AHandle : HWND;
              AWndProc : TFNWndProc; AOnDrop : TFMDDEvent);
begin
  inherited Create;
  Handle := AHandle;
  WindowProc := AWndProc;
  FOnDrop := AOnDrop;
end;

{
  WMDragDrop creates the TDragDropInfo object and calls the
  FOnDrop event handler.
}
procedure WMDragDrop (hDrop : THandle; FOnDrop : TFMDDEvent);
```

```
var
  DragDropInfo : TDragDropInfo;
  TotalNumberOfFiles,
  nFileLength : Integer;
  pszFileName : PChar;
  i : Integer;

begin
  if not assigned (FOnDrop) then
    exit;
  {
    hDrop is a Handle to the internal Windows data
    structure which has information about the dropped files.
  }
  {
    Find out total number of files dropped by passing -1 for
    the index parameter to DragQueryFile
  }
  TotalNumberOfFiles := DragQueryFile (hDrop , $FFFFFFFF, Nil, 0);

  DragDropInfo := TDragDropInfo.Create (TotalNumberOfFiles);

  {
    Determine if the files were dropped in the client area
  }
  DragDropInfo.FInClientArea :=
    DragQueryPoint (hDrop, DragDropInfo.FDropPoint);

  for i := 0 to TotalNumberOfFiles - 1 do begin
    {
      Get the length of a filename by telling DragQueryFile
      which file your querying about ( i ), and passing Nil
      for the buffer parameter.  The return value is the length
      of the file name.
    }
    nFileLength := DragQueryFile (hDrop, i , Nil, 0) + 1;
    GetMem (pszFileName, nFileLength);

    {
      Copy a file name.   Tell DragQueryFile the file
      you're interested in (i) and the length of your buffer.
      NOTE: Make sure that the length is 1 more than the filename
      to make room for the nul character!
    }
    DragQueryFile (hDrop , i, pszFileName, nFileLength);

    { Add the file to the string list }
    DragDropInfo.FFileList.Add (pszFileName);

    { free the allocated memory... }
    FreeMem (pszFileName, nFileLength);
  end;

  {
   Call DragFinish to release the memory that Shell allocated
   for this handle.
```

```
   NOTE: This is a real easy step to forget and could
   explain memory leaks and incorrect program performance.
  }
  DragFinish (hDrop);

  { call the event handler }
  FOnDrop (DragDropInfo);

  { and destroy the TDragDropInfo object }
  DragDropInfo.Free;
end;

{
  find and return the list item that corresponds to
  the passed window handle.
}
function FindItemInList (Handle : HWND) : TSubclassItem;
var
  i : Integer;
  Item : TSubclassItem;
begin
  for i := 0 to SubclassList.Count - 1 do begin
    Item := SubclassList.Items[i];
    if Item.Handle = Handle then begin
      Result := Item;
      exit;
    end;
  end;
  Result := Nil;
end;

{
  FMDDWndProc handles WM_DROPFILES messages by calling WMDragDrop.
  All other messages are passed on to the window's old window proc.
}
function FMDDWndProc (
    Handle : HWND; Msg : UINT;
    wparam: WPARAM; lparam: LPARAM) : LRESULT; stdcall;
var
  Item : TSubclassItem;
begin
  Item := FindItemInList (Handle);
  if Item <> Nil then begin
    if Msg = WM_DROPFILES then begin
      WMDragDrop (wparam, Item.FOnDrop);
      Result := 0;
    end
    else
      Result := CallWindowProc (Item.WindowProc,
          Handle, Msg, wparam, lparam)
  end
  else
    Result := 0;
end;

{
```

```
  AcceptDroppedFiles subclasses the control's window and saves
  information about the control for future use.
}
procedure AcceptDroppedFiles (Control : TWinControl;
           AOnDrop : TFMDDEvent);
var
  WndProc : TFNWndProc;
begin
  DragAcceptFiles (Control.Handle, True);
  { get the old window procedure }
  WndProc :=
    TFNWndProc(GetWindowLong (Control.Handle, GWL_WNDPROC));
  { attach the new window procedure }
  SetWindowLong (Control.Handle,
    GWL_WNDPROC, Longint (@FMDDWndProc));
  { and add it to the list }
  SubclassList.Add (
    TSubclassItem.Create (Control.Handle, WndProc, AOnDrop));
end;

{
  UnacceptDroppedFiles unsubclasses the window and removes its
  entry from the subclass list.
}
procedure UnacceptDroppedFiles (Control : TWinControl);
var
  Item : TSubclassItem;
begin
  { stop accepting dropped files }
  DragAcceptFiles (Control.Handle, False);

  Item := FindItemInList (Control.Handle);
  if Item <> Nil then begin
    { restore the old window procedure }
    SetWindowLong (Control.Handle, GWL_WNDPROC,
      Longint (Item.WindowProc));
    { remove the item from the list }
    SubclassList.Remove (Item);
    { and destroy it }
    Item.Free;
  end;
end;

{ TDragDropInfo }
constructor TDragDropInfo.Create (ANumFiles : UINT);
begin
  inherited Create;
  FNumFiles := ANumFiles;
  FFileList := TStringList.Create;
end;

destructor TDragDropInfo.Destroy;
begin
  FFileList.Free;
  inherited Destroy;
end;
```

```
initialization
  SubclassList := TList.Create;

finalization
  SubclassList.Free;

end.
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

If you've done subclassing before, you may wonder why I didn't store the old window procedure (or even a pointer to the **TSubclassItem** object) in the **GWL_USERDATA** member of the window's data structure. I thought about it, but I came up against the same arguments I used against chaining **Application.OnMessage**: There's no telling what some other program's gonna do. If FMDD were to use **GWL_USERDATA**, then any control that wanted to use FMDD could use that field. It seems like an arbitrary restriction, so I went with a **TList** of structures. This provides a more flexible implementation, at only a small cost in runtime efficiency (that is, in the time required to look up the item in the list). Handling Windows messages isn't normally an especially time-critical operation, so the small amount of time required for the lookup won't be noticed. Leave **GWL_USERDATA** for the user data, and find another way to store the saved window procedure pointer.

With the completed FMDD unit, we're now free to create applications that have multiple forms accepting dropped files, or even forms that accept dropped files in two or more different controls. The Drag3 program, shown in Figure 2.2, is one such form. The form itself doesn't accept dropped files—the individual list boxes do. Fire it up and check it out. The source form's source code, DRAGFRM3.PAS, is shown in Listing 2.8.

FIGURE 2.2 A form that accepts dropped files in multiple list boxes.

Listing 2.8 DRAGFRM3.PAS.

```
unit dragfrm3;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls,
  {
     FMDD defines the drag and drop interface.
  }
  FMDD;
```

```
type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    Button1: TButton;
    Button2: TButton;
    Label1: TLabel;
    Label2: TLabel;
    ListBox2: TListBox;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
    procedure OnListbox1FMDragDrop (DragDropInfo : TDragDropInfo);
    procedure OnListbox2FMDragDrop (DragDropInfo : TDragDropInfo);
    procedure ProcessDroppedFiles (lb : TListBox;
                DragDropInfo : TDragDropInfo);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  FMDD.AcceptDroppedFiles (Listbox1, OnListbox1FMDragDrop);
  FMDD.AcceptDroppedFiles (Listbox2, OnListbox2FMDragDrop);
end;

procedure TForm1.ProcessDroppedFiles (lb : TListBox;
            DragDropInfo : TDragDropInfo);
var
  i : Integer;
begin
  { Determine if in client area }
  if DragDropInfo.InClientArea then
    Label2.Caption := 'In client area'
  else
    Label2.Caption := 'Not in client area';

  { add each file to the list box }
  for i := 0 to DragDropInfo.NumFiles - 1 do begin
    lb.Items.Add (DragDropInfo.Files[i]);
  end;
end;

procedure TForm1.OnListbox1FMDragDrop (DragDropInfo : TDragDropInfo);
begin
  ProcessDroppedFiles (Listbox1, DragDropInfo);
end;
```

```
procedure TForm1.OnListbox2FMDragDrop (DragDropInfo : TDragDropInfo);
begin
  ProcessDroppedFiles (Listbox2, DragDropInfo);
end;

procedure TForm1.FormClose (Sender: TObject; var Action: TCloseAction);
begin
  { Don't accept files anymore }
  FMDD.UnacceptDroppedFiles (Listbox1);
  FMDD.UnacceptDroppedFiles (Listbox2);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Listbox1.Clear;
  Listbox2.Clear;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;

end.
```

Now *that* is a Delphi program. There's no mucking around with window handles and window procedures. Everything's done with components and event handlers the way Delphi programs were meant to be written. All of the gory details are hidden away in FMDD—out of sight of the application programmer, who's interested in accepting dropped files, but not particularly interested in learning how to navigate through the Windows message loop.

Don't get me wrong, I'm a firm believer in knowledge. As you learn more about how Windows and Delphi work "under the hood," you develop alternate ways of looking at and solving problems. But after you've solved the problem once, why do it again? It takes a little time to design and implement a good wrapper around a Windows feature like drag and drop, but once you've done it, you can use that wrapper for any application that needs the feature, and you don't have to step down into the bowels of Windows to do it.

The Elusive Drag and Drop Server

Unfortunately, Microsoft hasn't yet documented the functions and structures that are used to make a program into a drag and drop server. Given that information, we could easily add that functionality to FMDD, thereby giving full drag and drop support to Delphi programs. Although officially undocumented, the information for Windows 3.1 drag and drop has been published (see Jeffrey M. Richter's article "Drop Everything: How to Make Your Application Accept and Source Drag-and-Drop Files" in the May-June 1992 issue of *Microsoft Systems Journal*). However, I was unable to make Mr. Richter's example work under Delphi in time to meet the deadline for this chapter. It's possible that I was doing something wrong, but it's equally possible that Microsoft changed the structure for Windows 95 programs—it is, after all, undocumented. There is hope, though—ShellAPI defines a **TDragInfo** record that's similar to the structure discussed in Richter's article. Perhaps the Delphi team knows a thing or two about this that I haven't yet discovered.

That isn't to say that I've given up on adding drag and drop server support to my Delphi programs. I will continue to work on the problem and, if successful, publish the result. If you have interest in this area (like maybe you've done it already!), please contact me at: 75300.2525@ compuserve.com.

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Previous | Table of Contents | Next

CHAPTER 3
32-Bit Delphi DLLs—When, Why, How

JIM MISCHEL

- The Nature Of DLLs
- DLL Syntax
- Calling DLLs
- Run-time Dynamic Linking
- Storing Common Forms In DLLs
- Sharing Memory Between Applications

VCL components are the new and snazzy way to reuse your code, but even ancient mechanisms like those behind Windows DLLs can work miracles when evoked in the proper fashion.

It's been an interesting Spring. February was *cold*—it actually snowed here in Austin. The roads iced up and people were smashing their cars right and left. Great amusement unless you happened to be one of those who got smashed. Shortly after the ice storm, the old Bronco's water pump went, along with a head gasket, and we decided it was time to get a new vehicle. Have you priced inexpensive cars these days? Yikes!

Next on the agenda was the pool filter. April is swimming pool month around here, so I uncovered the thing (what an interesting shade of green) and turned on the pump. No dice. Mr. Pool Man came out and did his thing, leaving me that much poorer. And then the septic system backed up because some idiot contractor saved $20 by installing inferior pipe between the tank and the house. Mr. Plumber took more than Mr. Pool Man. All in all, it's been an

expensive couple of months here at *chez Mischel*.

I'm not looking for sympathy. The point is that you can't always anticipate what's going to happen, and you'd better be flexible, or a run of bad luck similar to this can throw your life into complete disarray. The same is true of your programs—if you don't build in flexibility, somebody else will, and your program's sales will suffer for it.

In life, the key to flexibility is usually money. When you're talking Windows programming, the key to flexibility is DLLs.

What's a DLL and Why Do I Want One?

A DLL (Dynamic Link Library) is a Windows-specific executable file that contains code or data designed to be accessed by other programs. DLLs are similar in concept to Delphi's units. Both are pre-packaged pieces of code that your programs can call to accomplish all manner of things. The concept is the same—but the implementation, that's another story.

Delphi's units are *statically* linked to your programs. This means that at compile time, a copy of the code in all of the units that your program uses is placed in the program's .EXE file. And every program that uses a particular unit has a separate copy of that unit in its .EXE file. Normally, this is a good thing—you want your programs to be as self-contained as possible. However, there are at least two good reasons why you might not want code to be statically linked.

If you have a *large* unit, and a lot of different programs that use that unit, you're going to end up with an amazing amount of duplicate code in your programs. With hard drive space going for something like 30¢ per megabyte, that's less of a problem today than it once was (we'll ignore minimum sector size issues in this discussion), but what happens if you want to have four or five of those programs running at the same time? You end up with that many copies of the unit's code in memory. RAM, as inexpensive as it is, still ain't cheap—certainly not cheap enough to waste.

The second reason you may not want static linking is flexibility. Say you've just written the newest whiz-bang word processor and you'd like to be able to import other vendor's document file formats (something you *must* do if you want to compete in the word processor market). You could code up special modules for all of the currently popular formats and send your product out the door. But six months later, when the new version of Word Grinder Max (I hope that's not the name of a real product) comes out with a new format, *your* program is obsolete! The only way to get your program to read the new Word Grinder format is to release an upgrade, which will cost you plenty and gain you little. The size issue comes into play here, too. If you're statically linking code to convert hundreds of different formats, your program has an awful lot of dead weight—code that's going to be used infrequently by a very small number of customers.

The solution to both of these problems is *dynamic* linking. Instead of linking in a copy of a unit's code to the applications main .EXE file, a DLL allows you to

place the reusable code in a separate library file that's loaded at runtime on an as-needed basis. If five different programs need to use functions in the DLL, there's still only one copy of the code on disk and, way better still, only one copy in memory. Rather than linking in a piece of code to the main .EXE, you just link in some instructions that tell the program where to find the code it needs. And rather than linking a gazillion different format-conversion routines into your word processor, you just build in the ability to specify a new format conversion DLL. Your support for new formats becomes a simple matter of writing a DLL and making it available to users on an as-needed basis.

Now *that's* flexibility.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

How Do I Do It?

There are two parts to DLLs: building them and using them. You *use* DLLs every day in your normal Windows programming, probably without even knowing it. Darn near all of Windows itself is implemented in DLLs. When you call the Windows **MessageBox** function, for example, you're calling a function that's located in a DLL called USER.EXE (or USER32.DLL—Windows 95 does some funny fooling around with automatic 32-bit to 16-bit thunking, so I'm not always certain exactly what's going on). At any rate, whether you know it or not, you're already using DLLs in your everyday programming.

There are two ways to tell your program to call a function in a DLL. You can build a DLL interface unit that names the DLL and the functions that it contains, and link that interface unit with your Delphi program. This is called (remember, I'm just the messenger) statically linking the DLL. This is how you access Windows API functions. WINDOWS.DCU, which is linked with your program when you specify the Windows unit in your **uses** statement, is the DLL interface unit that defines these functions.

The other way to call DLL functions is, you guessed it, dynamically. When dynamically loading a DLL, you don't have to link with any DLL interface units. Instead, your program uses the **LoadLibrary** and **GetProcAddress** functions at runtime to locate and link to functions in a DLL. Of the two methods, static DLL linkage is easiest to code—but dynamic DLL linkage is much more robust and flexible.

Building a DLL

Logically, a DLL is more like a unit than a program, but the code looks more like a program than a unit. This isn't too surprising when you consider that a DLL is just a special kind of program whose purpose is to provide code or data that other programs can access. Listing 3.1 shows a very simple DLL that supplies just one

function: **BeepMe**. This function does nothing more than emit a “beep” when it’s called.

Listing 3.1 A simple DLL

```
{ BEEPER.DPR -- a simple DLL example }
library beeper;

uses Windows;

procedure BeepMe; stdcall;
begin
  MessageBeep (0);
end;

Exports
  BeepMe index 1 name 'BeepMe';

begin

end.
```

DLLs start with the reserved word **library** rather than **program** or **unit**. They have a **uses** statement, but you’ll notice that, like programs, they don’t have separate **interface** and **implementation** sections. You write your procedures and functions in the DLL just as you would in a unit or a program, and then you specifically *export* those functions that you want to be available to other programs.

The **stdcall** reserved word isn’t strictly required in DLLs, but it’s not a bad idea. Exported DLL functions that are defined with the **stdcall** modifier are compatible with other languages (like C++) that can call DLLs. There’s no disadvantage in using **stdcall** for your exported functions. I recommend using **stdcall** if there’s any chance that you might want C/C++ programs to have access to your exported functions.

The **Exports** statement is what tells the compiler to make specific functions available to other programs. In this example, I’ve exported the **BeepMe** procedure by name and by ordinal number—both of which are optional. Multiple exported functions should be separated by commas. So if you had another function in the DLL called **PageMe**, your **Exports** statement would look something like this:

```
Exports
  BeepMe index 1 name 'BeepMe',
  PageMe index 2 name 'PageMe';
```

To create the DLL, select File|New and select DLL from the New Items dialog box. Enter the code as shown in Listing 3.1, save it as BEEPER.DPR, and then compile. You can’t run the DLL—you need another program to call it.

Calling DLL Functions

Once you’ve compiled the DLL, save the project and then select File|New Application. We’re going to build a quick test program to put it all together.

Drop a button onto the main form and create an event handler that looks like this:

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

Next add **BeepDLL** to the list of units in the form unit's **uses** statement. Don't try to compile—we need to create BEEPDLL.PAS first.

Create a new unit called BEEPDLL.PAS, and enter the code shown in Listing 3.2.

Listing 3.2 The BEEPER DLL interface unit

```
{ BEEPDLL.PAS -- interface unit for BEEPER.DLL }
unit BeepDLL;

interface

procedure BeepMe; external 'beeper.dll';
procedure BeepMeTwo; external 'beeper.dll' name 'BeepMe';
procedure BeepMeThree; external 'beeper.dll' index 1;

implementation

end.
```

If you've done everything right, when you compile and run the program, pressing the button should emit a beep (or whatever sound your computer makes in place of a beep).

You probably noticed that I listed three different procedures, all of which are resolved at runtime, to call the **BeepMe** procedure that's in BEEPER.DLL. If you were to change your button's event handler to call **BeepMeThree** rather than **BeepMe**, the results would be the same. This is a contrived example, but there actually are times when you might need to use the **name** or **index** clauses to link to DLL functions. For example, you might run across a DLL (I have) that has a function named something like **XY$FORMAT** that you want to link to. Since **XY$FORMAT** isn't a valid Pascal identifier, you'd be unable to link to that function if you couldn't rename the function. Same goes for the **index** clause: Some DLL functions are exported by ordinal number only—no name!

This is an example of static DLL linkage. All the interface unit, BEEPDLL.PAS, does is let the compiler know that the **BeepMe** procedure is to be dynamically linked from the BEEPER.DLL file. No code from BEEPER.DLL is actually linked with your program. If you don't believe it, delete BEEPER.DLL and run the program. If you're running from within the Delphi IDE, Delphi will report an error. If you're running from outside the IDE, Windows will report that the required BEEPER.DLL could not be found.

And *that* error message brings us to the other way of calling DLL functions—runtime dynamic linking.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Linking DLLs at Runtime

Sometimes you really don't *need* a particular DLL in order for your program to work. Take the word processor format conversion DLLs for example. It's not very often that a user will actually want to convert a file. In fact, most of your users will likely never even once convert a file. It would be criminal for the program to require the format conversion DLLs in order to perform normal editing tasks. But that's what happens with statically linked DLLs. If Windows can't find the DLL when the program's loaded, you get a pretty little error message and Windows shuts down the program.

The other problem with statically linking DLLs is that it doesn't allow flexibility. If your program has to know at compile time what DLLs exist, then you're back in the same boat you were before—the only way to supply a new format conversion is to patch the executable program. Not good.

That's where runtime dynamic linking comes in. Rather than having Windows automatically load and link to your DLL at load time, why not have the program itself explicitly load and link the DLL if it's required? That way, the program will still run if the DLL is missing, it just won't be able to perform whatever function is implemented in the DLL. One neat thing about this approach is that you can tell the user what the problem is and, if he has a copy of the DLL somewhere, he can copy it to the proper place and retry—all without having to shut down the program.

Listing 3.3 is an updated version of the BEEPDLL interface unit that can be conditionally compiled for compile-time or runtime dynamic linking.

Listing 3.3 Dynamically linking a DLL at runtime

```
{ BEEPDLL.PAS -- interface unit for BEEPER.DLL }
unit BeepDLL;
{$DEFINE DYNAMIC}  {comment this line for compile-time DLL linkage }

interface

{$IFDEF DYNAMIC}
{ procedure declarations for runtime dynamic linking }
procedure BeepMe;
procedure BeepMeTwo;
procedure BeepMeThree;
{$ELSE}
{ procedure declarations for compile-time dynamic linking }
```

```
procedure BeepMe; external 'beeper.dll';
procedure BeepMeTwo; external 'beeper.dll' name 'BeepMe';
procedure BeepMeThree; external 'beeper.dll' index 1;
{$ENDIF}

implementation

{$IFDEF DYNAMIC}
uses Windows;

type
  BeepMeProc = procedure;

var
  LibInstance : HMODULE;       { DLL module handle (if loaded) }
  BeepMePtr : BeepMeProc;

procedure BeepMe;
begin
  if (LibInstance = 0) then begin
    { DLL not loaded, try to load it }
    LibInstance := LoadLibrary ('beeper.dll');

    { if LoadLibrary returns 0, there's an error }
    if (LibInstance = 0) then begin
      MessageBox (0, 'Can''t load BEEPER.DLL', 'Error',
        MB_ICONEXCLAMATION or MB_OK);
      Exit;
    end;

    { DLL is loaded, try to link to the function }
    BeepMePtr := BeepMeProc (GetProcAddress (LibInstance, 'BeepMe'));
    { if GetProcAddress returns Nil, we've got a problem }
    if (Not Assigned (BeepMePtr)) then begin
      { unload the DLL first so the user can replace it if possible }
      FreeLibrary (LibInstance);
      LibInstance := 0;
      MessageBox (0, 'Can''t find BeepMe function in DLL.', 'Error',
        MB_ICONEXCLAMATION or MB_OK);
      Exit;
    end;
  end;

  BeepMePtr;
end;

procedure BeepMeTwo;
begin
  BeepMe;
end;

procedure BeepMeThree;
begin
  BeepMe;
end;
```

```
initialization
  LibInstance := 0;
  BeepMePtr := Nil;

finalization
  { if the DLL has been loaded, be sure to unload it }
  if (LibInstance <> 0) then begin
    FreeLibrary (LibInstance);
    LibInstance := 0;
  end;

end.
{$ELSE}

end.

{$ENDIF}
```

Hey, I *said* it was more involved!

Yes, runtime dynamic linking is a little more involved. You end up writing code to do what Windows will do automatically at startup if you select compile-time dynamic linking. But there is a serious benefit to all this code—you have much better error recovery. Let's take a moment and examine how this code works.

First of all, the procedure names in the unit aren't exported in the **interface** section, but instead correspond to real procedures that are defined in the **implementation** part of the module. It's the **export** statements in the statically-bound interface unit that cause the automatic DLL linkage at program startup, so if you remove them, Windows won't try to load and link to your DLL.

Then, we define a procedure type and two variables:

```
type
  BeepMeProc = procedure;

var
  LibInstance : HMODULE;       { DLL module handle (if loaded) }
  BeepMePtr : BeepMeProc;
```

The procedure type **BeepMeProc** is similar to Delphi's event handler types. A variable of this type (in this case, **BeepMePtr**) contains a pointer to a procedure that takes no parameters. Once we've loaded BEEPER.DLL and located the **BeepMe** procedure, we'll assign its address to **BeepMePtr**.

LibInstance is the module instance handle of BEEPER.DLL, which is returned by **LoadLibrary** if it successfully loads the DLL.

The **BeepMeTwo** and **BeepMeThree** procedures are just aliases for **BeepMe**, so in the dynamic link version of the unit, they just call the unit's **BeepMe** procedure.

BeepMe is where all of the magic is performed. It first checks to see if the DLL has been loaded. If not, it calls Windows' **LoadLibrary** API function, which finds the DLL and attempts to load it—executing the DLL's startup code (more on that later)—and then returns a module handle that uniquely identifies the DLL. If the DLL couldn't be found or Windows encountered an error while loading it, then **LoadLibrary** will return 0 and **BeepMe** will issue an error message.

Assuming that **LoadLibrary** was able to load the DLL, the function then calls **GetProcAddress**, which attempts to locate a function called **BeepMe** in the newly-loaded DLL. If the function is found, then its address is placed in **BeepMePtr**. If **GetProcAddress** can't find the function, then it returns **Nil**, causing **BeepMe** to issue an error message and unload the DLL.

If everything worked right—the DLL was loaded and the **BeepMe** procedure was found in the DLL—then it's called through the **BeepMePtr** procedure pointer.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

One last note—your program should explicitly unload (by calling **FreeLibrary**) any DLLs that it loads with **LoadLibrary**. That's the purpose of the **initialization** and **finalization** parts of the module. During unit startup, the **initialization** section sets **LibInstance** and **BeepMePtr** to known values that indicate that the DLL isn't loaded. When the program exits, the **finalization** section unloads the DLL if it was loaded.

Where Windows Looks for DLLs

If you're deploying a DLL with your application, you'll usually have your installation program place the DLL in the same directory as your program's executable file. If that's where you put it, then Windows won't have any trouble locating it when it comes time to load your program (or when you call **LoadLibrary** if you're using runtime dynamic linking). If your application installs several executable files in different directories, then you have the option of placing a copy of the DLL in each directory (which kind of defeats the purpose of the DLL), or you can put the DLL into a common directory that Windows will search by default when it tries to load the DLL.

Windows searches for DLLs in these places, in the order shown:

1. The directory from which the application loaded.
2. The current directory.
3. The Windows system directory.
4. Windows NT only: The 16-bit Windows system directory.
5. The Windows directory.
6. The directories that are listed in the PATH environment variable.

If you're using runtime dynamic linking by explicitly calling **LoadLibrary**, then you can specify a full path name to the DLL, and Windows will *only* look there. This isn't an option if you're counting on Windows to automatically load your DLLs at runtime.

DLLs: Disadvantages and Cautions

Most programmers, once they've grasped a new concept, start acting like a tool addict with a new torque wrench—they just can't *wait* to apply it to something. And they'll apply some amazingly convoluted logic in order to rationalize applying it to a particular situation. I know it's hard, but restrain yourself. DLLs are undoubtedly cool, but they can easily mutate into foot-seeking missles.

Don't even *think* of implementing required program features in a DLL. Your word processor's text formatter, for example, belongs in the program, not in an external DLL. DLLs should be reserved for optional features (including third-party add-ons) and common libraries. That's *it*. If you use a DLL for anything else, you're just begging for trouble.

The biggest disadvantage to using DLLs is type checking—or the lack thereof. When you access a DLL function, using either method of linking to it, you're telling the compiler to call functions that it knows nothing about. For example, in the BEEPDLL.PAS unit, we have this declaration:

```
procedure BeepMe; external 'beeper.dll';
```

This declaration is telling the compiler that there's a procedure called **BeepMe** that's located in the named DLL. So far, so good. Here's the kicker. *The compiler takes your word for it*. There's absolutely no way that the compiler can go find BEEPER.DLL, disassemble it, and verify that the procedure called **BeepMe** actually exists and does indeed expect to be called with no parameters. If the DLL's **BeepMe** procedure is expecting one or more parameters (or in the case of a procedure with parameters—different types of parameters), then all hell will break loose when your program calls **BeepMe**, because it got more, fewer, or different types of parameters than were expected. I guarantee that this will happen to you at some point, and I can tell you from experience that it's a very difficult bug to track down. In fact—and this is very embarrassing to admit—I ran into this problem myself shortly after I wrote that last sentence, while I was working on the code for the next section.

If you want a more detailed (and hair-raising) discussion of possible problems with DLLs, I suggest that you take a look at Lou Grinzo's *Zen of Windows 95 Programming* (Coriolis Group Books, 1995). This is an excellent book that contains a wealth of information about Windows 95 programming, and some very good advice about programming in general. Programming requires a healthy dose of paranoia and a firm belief in Murphy's Law. If you don't believe that now, you will after reading Lou's book.

I'll climb off my soap box now, but don't say I didn't warn you. Now that you know how to build DLLs, let's take a look at some of the things you can do with them.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Creating Forms in DLLs

Probably the most common use of DLLs in Delphi programming is to store common forms. If you're building a suite of programs, you've probably got dozens of forms that are common across all of the programs. Rather than building all of those forms into each program, you can put them all into a single DLL, which will save you disk space and memory, and (perhaps most importantly) maintenance problems. A DLL that contains Delphi forms does incur the overhead of the runtime library code (about 100 K), but if you put many forms into a single DLL, the overhead isn't a problem.

Accessing a form from a DLL is slightly different from accessing the form in a program. Since you're not linking the unit that contains the form, you can't just show it as you would from within a normal program (that is, by calling **Form1.ShowModal**). Instead, you have to create a wrapper function in the DLL, and then call the wrapper function from the program. The wrapper function creates the form, displays it, gathers any data, destroys the form when it's closed, and returns any required information to the calling program.

Listings 3.4 and 3.5 contain PICKCLR.DPR and COLORFRM.PAS, which implement a color selection form in a DLL.

Listing 3. PICKCLR.DPR

```
library pickclr;

uses
  SysUtils,
  Classes,
  ColorFrm in 'colorfrm.pas' {ColorSelectionForm};

Exports
  ColorFrm.PickColors index 1 name 'PickColors';

begin
end.
```

Listing 3.5 COLORFRM.PAS

```
unit colorfrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ColorGrd;

type
  TColorSelectionForm = class(TForm)
    ColorGrid1: TColorGrid;
    BtnOk: TButton;
    BtnCancel: TButton;
  private
    { Private declarations }
  public
    { Public declarations }
    function Execute : boolean;
  end;

function PickColors (var Foreground, Background : TColor) : boolean;
    stdcall; export;

implementation

{$R *.DFM}
function TColorSelectionForm.Execute : boolean;
begin
  Result := (ShowModal = mrOk);
end;

function PickColors (var Foreground, Background : TColor) : boolean;
    stdcall;
var
  ColorForm : TColorSelectionForm;
begin
  ColorForm := TColorSelectionForm.Create (Application);
  Result := ColorForm.Execute;
  if (Result = True) then begin
    Foreground := ColorForm.ColorGrid1.ForegroundColor;
    Background := ColorForm.ColorGrid1.BackgroundColor;
  end;
  ColorForm.Free;
end;

end.
```

You should note that COLORFRM.PAS can be linked with a program or with a DLL without changes. This makes moving forms from programs into DLLs fairly easy. For ease of debugging, you develop the form and test it with a program. Once you've got it working, you add it to a DLL shell that you've set up.

As you can see from Listing 3.4, the project file for a DLL is very simple. The most important part is getting the **Exports** statement right. If you want more forms in the DLL, simply add their unit names to the **uses** statement, and add definitions of their wrapper functions to the **Exports** statement.

A DLL that contains forms should have an interface unit similar to the BEEPDLL.PAS unit shown in Listing 3.3. Like BEEPDLL, the unit can provide compile-time or runtime DLL linkage. For brevity, I haven't included an interface unit for the PICKCLR DLL.

To use a form that's stored in a DLL, you simply link with the DLL's interface unit and call the form's wrapper function, which will display the form and return any required values.

Coding for Flexibility

Many products provide "hooks" to which third parties can hang on additional modules. Windows Help, for example, defines an interface through which developers can add custom macros and embedded windows that provide some very interesting features to Windows Help files. Borland's new C++ 5.0 IDE also has an add-on interface that other companies are using to add features. Version control and a Java development add-on are shipped with BC++ 5.0. Both are implemented using the DLL add-on interface.

I've offered the example of word processor format conversion in this chapter as an example of one possible use of DLLs. Let's develop that idea a little further by writing a mini text editor that offers an add-on format conversion interface. The text editor itself is very simple-minded—just a Memo component and menu options to open and save files. That's okay, though. What we're really interested in is the format conversion interface.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Creating the Text Editor

Since we're all programmers here, I'm going to move fairly rapidly through the mechanics of creating the text editor's shell. I'll slow down when we get to the add-on interface.

Starting with a new project, add a Memo component to the form and set its **Align** property to **alClient** so that it takes up the entire form. Then add MainMenu, OpenDialog, and SaveDialog components to the form. In the Menu Designer, add three items to the menu: Open, Save, and Exit. Save the unit as EDITFORM.PAS and the project file as TEXTEDIT.DPR. The completed form is shown in Figure 3.1, and the complete program listing is shown in Listing 3.6.

FIGURE 3.1 The completed text editor form.

Listing 3.6 The text editor form, EDITFORM.PAS

```
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
  Menus, StdCtrls;

type
  TForm1 = class(TForm)
    Memo1: TMemo;
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Open1: TMenuItem;
    Save1: TMenuItem;
```

```
    N1: TMenuItem;
    Exit1: TMenuItem;
    procedure Exit1Click(Sender: TObject);
    procedure Open1Click(Sender: TObject);
    procedure Save1Click(Sender: TObject);
  private
    { Private declarations }
    FileName : String;
    procedure OpenFile(Filename: String);
    procedure SaveFile(Filename: String);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Exit1Click(Sender: TObject);
begin
  Close;
end;

procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    OpenFile (OpenDialog1.FileName);
end;

procedure TForm1.Save1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
    SaveFile (SaveDialog1.FileName);
end;

procedure TForm1.OpenFile (Filename: String);
begin
  Memo1.Lines.LoadFromFile (Filename);
end;

procedure TForm1.SaveFile (Filename: String);
begin
  Memo1.Lines.SaveToFile (Filename);
end;

end.
```

Test the program and make sure that it'll load and save an ASCII file (any file with a .TXT extension will work, as will .PAS and .DPR).

Now what we want to do is have the program read other file formats, convert to straight text, and display the text. Since we don't know exactly what formats might need to be converted, we need

the ability to add new formats as the need arises. Probably the easiest way to do this is with an initialization (.INI) file.

The idea is to save a description of the file format, a default extension, and the name of the DLL that contains the format conversion function. An example .INI file is shown in Listing 3.7.

Listing 3.7 TEXTEDIT.INI

```
; TEXTEDIT.INI
; Example of file conversion add-on interface
[Text]
Extension=.TXT
ConvertDLL=textconv.dll

[Word for Windows]
Extension=.DOC
ConvertDLL=wfwconv.dll

[WordCruncher]
Extension=.WCX
ConvertDLL=wcxconv.dll
```

What we do is modify the **OpenFile** procedure so that it examines the extension of the file that you choose to open, and then calls the conversion function in the proper DLL. The DLL reads the file, converts the text, and returns the result in a string list. All of the conversion functions have a function called **Convert**, which the text editor program calls. Listing 3.8 contains the modified **OpenFile** function (be sure to add IniFiles to the form's **uses** list), and Listings 3.9 and 3.10 contain the code for the text conversion DLL (TEXTCONV.DLL).

Listing 3.8 The new OpenFile function

```
procedure TForm1.OpenFile (Filename: String);
type
  ConvertFunc = function (Filename: String;
      Strings: TStrings): boolean; stdcall;
var
  ConvertIni : TIniFile;
  ConvertList : TStringList;
  FileExt : String;
  Extension : String;
  DLLName : String;
  x : Integer;
  Found : Boolean;
  LibInstance : HMODULE;
  Converter : ConvertFunc;
  IniFileName : String;

begin
  FileExt := UpperCase (ExtractFileExt (Filename));
  IniFileName := ExtractFileDir (ParamStr (0)) + '\TEXTEDIT.INI';
  ConvertIni := TIniFile.Create (IniFileName);
  ConvertList := TStringList.Create;
  { Read the list of available conversions }
  ConvertList.Add ('Hello, world');
  ConvertIni.ReadSections (ConvertList);
```

```
{
  For each conversion, read the Extension entry and compare it
  against the extension of the selected file.
}
x := 0;
Found := False;
while ((x < ConvertList.Count) and (Not Found)) do begin
  Extension := ConvertIni.ReadString (
    ConvertList.Strings[x], 'Extension', '');
  if (UpperCase (Extension) = FileExt) then
    Found := True
  else
    x := x + 1;
end;

if Found then begin
  DLLName := ConvertIni.ReadString (
    ConvertList.Strings[x], 'ConvertDLL', '');
  {
    Load the DLL, get the address of the Convert function,
    and call it.
  }
  LibInstance := LoadLibrary (PChar(DLLName));
  if LibInstance = 0 then begin
    Application.MessageBox (
      PChar ('Can''t load DLL '+DLLName),
      'TextEdit',
      MB_ICONEXCLAMATION or MB_OK);
  end
  else begin
    Converter := GetProcAddress (LibInstance, 'Convert');
    if Not Assigned (Converter) then begin
      Application.MessageBox (
        PChar ('Can''t find Convert function in '+DLLName),
        'TextEdit',
        MB_ICONEXCLAMATION or MB_OK);
    end
    else begin
      if not Converter (Filename, Memo1.Lines) then begin
        Application.MessageBox (
          'Error loading file',
          'TextEdit',
          MB_ICONEXCLAMATION or MB_OK);
      end;
    end;
    FreeLibrary (LibInstance);
  end;
end
else begin
  Application.MessageBox (
    PChar('No conversion supplied for file type '+FileExt),
    'TextEdit',
    MB_ICONEXCLAMATION or MB_OK);
```

```
  end;

  ConvertList.Free;
  ConvertIni.Free;
end;
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Listing 3.9 TEXTCONV.DPR

```
library textconv;

{ Important note about DLL memory management: ShareMem must be the
  first unit in your library's USES clause AND your project's (select
  View-Project Source) USES clause if your DLL exports any procedures or
  functions that pass strings as parameters or function results. This
  applies to all strings passed to and from your DLL--even those that
  are nested in records and classes. ShareMem is the interface unit to
  the DELPHIMM.DLL shared memory manager, which must be deployed along
  with your DLL. To avoid using DELPHIMM.DLL, pass string information
  using PChar or ShortString parameters. }

uses
  ShareMem,
  SysUtils,
  Classes,
  textc in 'textc.pas';

Exports
  textc.Convert index 1 name 'Convert';

begin
end.
```

Listing 3.10 TEXTC.PAS

```
unit textc;

interface

uses Classes;

function Convert (Filename: String; Strings: TStrings) : boolean;
  stdcall; export;

implementation
```

```
function Convert (Filename: String; Strings: TStrings) : boolean;
  stdcall;
begin
  Strings.LoadFromFile (Filename);
  Result := True;
end;

end.
```

Pay particular attention to the note at the top of Listing 3.9 (TEXTCONV.DPR). The really cool thing about this note is that it's automatically placed in your project file when you select File|New|DLL. Truthfully, I'm not sure if I should be referencing the ShareMem unit or not in this case. I've tried the program without ShareMem, and it appears to work okay. And I can make the argument that I'm not passing a class to the **Convert** function—only a *pointer* to a **TStrings** object. I rather suspect, though, that the note applies to pointers to classes as well, so I've included ShareMem in the **uses** list for the program and the DLL. If you do have to use ShareMem, remember to ship the DELPHIMM.DLL file with your application.

Do note that the **OpenFile** function in Listing 3.8 is by no means good enough for a commercial program. This is an *example* that illustrates the concept. A commercial implementation would require that your program actually go read the file to determine what type it is (if possible), and prompt the user for permission to perform the conversion before actually doing anything. This example shows you one way that you could implement an add-on interface to provide support for third-party additions to your products.

Sharing Memory Between Applications

Fortunately for us Delphi programmers, Delphi's DLLs by default allow multiple instances, so there's one less worry. However, just because multiple instances are allowed doesn't mean that it's easy to share information between processes that are using the same DLL. Under Windows 95 and Windows NT, each instance of a DLL has its own data segment. You can't use a simple global variable in a DLL to share information between two running applications. For this, you need to set up a shared memory block in Windows. And to do *that*, you need to understand a little more about how Windows and Delphi load and map DLLs.

The DLLProc Variable

When Delphi loads a DLL, the DLL's startup code (the code between the **begin** and **end** at the bottom of your DLL) is executed. If your DLL needs to load resources, allocate memory, or do any other processing when it's first loaded and before any other functions are called, then that code should be placed here. This code is executed for every application that loads the DLL.

Windows will also notify your DLL when a process or thread attaches to it or detaches from it. But you have to request that notification from Delphi. The way you do that is by setting up a DLL handler function and setting the **DllProc** variable (defined in the System unit) to point to that function. Your DLL handler function should be defined like this:

```
procedure DLLHandler (Reason: Integer);
```

The **Reason** parameter will be one of four constants: **DLL_PROCESS_ATTACH**, **DLL_PROCESS_DETACH**, **DLL_THREAD_ATTACH**, or **DLL_THREAD_DETACH**.

To set up a shared memory block, you need to respond to **DLL_PROCESS_ATTACH** messages and call **CreateFileMapping** to create (or obtain a pointer to an already-created) shared memory block. Your DLL must also respond to **DLL_PROCESS_DETACH** messages and release the memory block so that Windows can release it when no more processes need it.

SHAREME.DPR (Listing 3.11), implements a shared memory block. In this example, the shared memory is just an integer that gets incremented every time a process attaches, and decremented when a process detaches.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Listing 3.11 Implementing shared memory in a DLL

```
library shareme;

uses
  Windows,
  SysUtils,
  Classes;

const
  pCounter: ^Longint = nil;

function GetProcessCount : Longint; stdcall; export;
begin
  Result := pCounter^;
end;

procedure MyDLLHandler (Reason: Integer);
const
  hMapObject : THandle = 0;
var
  fInit : Boolean;
begin
  case Reason of
    DLL_PROCESS_ATTACH : begin
      { create a named file mapping object }
      hMapObject := CreateFileMapping (
        $FFFFFFFF,        { use paging file }
        nil,              { no security attributes }
        PAGE_READWRITE,   { read/write access }
        0,                { high 32 bits of size }
        sizeof (longint), { low 32 bits of size }
```

```
        'SharedMem'       { name of object }
      );

      { the first process to attach initializes the memory }
      fInit := (GetLastError <> ERROR_ALREADY_EXISTS);

      { get a pointer to the file-mapped shared memory }
      pCounter := MapViewOfFile (
        hMapObject,         { object to map view of }
        FILE_MAP_WRITE,     { read/write access }
        0,                  { high 32-bits of offset }
        0,                  { low 32-bits of offset }
        0                   { default:  map entire file }
      );

      { initialize or increment the count }
      if (fInit) then
        pCounter^ := 1
      else
        pCounter^ := pCounter^ + 1;
    end;

    DLL_PROCESS_DETACH : begin
      { decrement the count }
      pCounter^ := pCounter^ - 1;

      { unmap shared memory from the process's address space }
      UnmapViewOfFile (pCounter);

      { close the handle to the file mapping object }
      CloseHandle (hMapObject);
    end;

    (*
      Thread attach and thread detach aren't handled

    DLL_THREAD_ATTACH :
    DLL_THREAD_DETACH :

    *)

  end;
end;

Exports
  GetProcessCount index 1 name 'GetProcessCount';

begin
  DLLProc := @MyDLLHandler;
  MyDLLHandler (DLL_PROCESS_ATTACH);
end.
```

You should take special note of the two lines of code in the DLL's initialization section. The first line of code initializes the System unit's **DLLProc** variable to point to the DLL's handler function. I thought that this was all that was required, but it appears that Delphi won't call the handler with a **DLL_PROCESS_ATTACH** value. So, the library's initialization code calls its own handler function. In my opinion, this is a bug in Delphi's handling of DLL initialization.

To test the shared memory, create a form that calls the DLL's **GetProcessCount** function when the form is created, and have it display the count in a label field on the form. If you run multiple copies of the application, you should see the counter increment once for each process that attaches to the DLL. If you close one or more of the applications and open new ones, the process counter should reflect the net effect (that is, if you opened three, closed one, and then opened another, the last one you open should have a process count of 3).

Global memory handles like those allocated by SHAREME consume valuable Windows resources, so be careful how you allocate them. If you'll be sharing a lot of different fields from a single DLL, you should put them all together into a single memory block (i.e. a record), and allocate just one memory block for the entire structure. This will minimize the Windows resources that the program uses. And be sure that your DLL correctly frees the memory blocks. If your DLL crashes or otherwise exits without freeing the memory block, that memory and the Windows resource will remain allocated until you reboot Windows. There is no way to free it once you've trashed the handle to the memory block.

Movin On!

If you're interested in digging, there's lots of stuff to learn about DLLs. In this chapter, I've given you enough information for you to go exploring. If you have the Microsoft Developer's Network CD-ROMs, you'll want to look up DLLs in the index and read everything you can find. You also should learn more about **CreateFileMapping** and related file mapping functions, paying special attention to the differences between Windows 95 and Windows NT. You can do a lot of cool things with DLLs, but you've got to be careful. Good luck!

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

CHAPTER 4
The Delphi Winsock Component

JOHN PENMAN

- Encapsulating Winsock In A Component
- Resolving Internet Address
- Asynchronous Host Name Resolution
- Cancelling An Asynchronous Operation
- Resolving Ports And Services
- Resolving Protocols

Objects are good…but components are better. Let's make Internet access for Delphi a plug-and-play proposition by putting all our Winsock baggage into a VCL component.

The Internet—and networked environments in general—becomes more popular every day. It's natural, therefore, for programmers to want to integrate network services into their applications. Under Microsoft Windows, the Winsock API is the *lingua franca* for Internet access. Using the Winsock component described in this chapter as a starting point, you'll soon be implementing many familiar TCP/IP-based programs such as FINGER, FTP, SMTP, POP3, and ECHO applications.

What Is Winsock?

Winsock is an abbreviation for Windows Sockets, an interface layer between a Windows application and an underlying TCP/IP network. The sockets interface originated with Berkeley Unix as the API to its TCP/IP network

stack. Winsock is based on the Berkeley Sockets API and includes most of the standard BSD API functions, as well as some Windows-specific extensions. Adding TCP/IP networking capability to your Windows programs is as simple as using the Winsock API and dynamically linking with WINSOCK.DLL, the library that implements Winsock.

The easiest way for a Delphi programmer to interface with the Winsock API is as a component. In this chapter, we'll do just that by creating the WSock component, encapsulating the Winsock API. This produces several immediate benefits:

- The API becomes part of the Delphi VCL;
- Encapsulation promotes easier code reuse;
- The client application sees a clean interface using properties and methods.

The WSock component is certainly Delphi-friendly, but it's not comprehensive. WSock provides the framework you'll use to create daughter components designed to handle any particular Internet protocol. A Winsock component already equipped to handle every conceivable Internet protocol would be a fat and complicated component. Instead, we'll use the WSock component as the basis for new components that deal specifically with a particular protocol.

For example, creating a component for the Hypertext Transfer Protocol (HTTP) involves three steps:

1. Derive a new component from WSock.
2. Set the **Service** property to HTTP in the new component's constructor.
3. Add the appropriate methods and properties for handling HTTP.

We'll go through these steps in the next chapter to create a specific component for an FTP client application.

Dissecting WSock

The WSock component is based on the non-visual **TWSocket** class that, it turn, descends from **TComponent**. Because it's non-visual, **TWSocket** is like a foundation of a house, normally hidden from view. The **TComponent** class provides the necessary methods and properties that WSock requires—but no more. Had I used **TGraphicControl** as an ancestor, the resulting **TWSocket** class would be more powerful, but with a corresponding increase in complexity and overhead. WSock provides the basic framework to set up and maintain a TCP/IP connection and supports both stream (TCP) and datagram (UDP) sockets.

To simplify the task of building TCP/IP networking components for Internet applications, our ideal Winsock component must perform the following basic housekeeping functions:

- Start and stop Winsock;
- Resolve host names and perform other conversion operations;

- Create, maintain and destroy a connection (both TCP and UDP);
- Send and receive data over a connection.

Like all networking life-forms, our Winsock component must initialize, clean up after itself, and report errors. Listing 4.1 shows the **TWSocket** class that performs these functions and much more. The majority of methods are in the protected section in the **TWSocket** class so that scion components can use them. These methods remain invisible to the client applications.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Listing 4.1 The TWSocket Definition

```
unit WSock;
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs;

 {$I wsapintf} (* Winsock implementation declarations *)

 const
  WSockVersionNo : String = '1.10';
  WSockBuildDate : String = '190596';

  SOCK_EVENT     =  WM_USER + 1;
  ASYNC_EVENT    =  SOCK_EVENT + 1;

type

 TAsyncOperationDone = procedure (Sender : TObject;
                                  sSocket : TSocket) of object;

 TConditions = (Success, Failure, None);

 THostAddr   = (HostAddr, IPAddr);

 TOperations = (SendOp, RecvOp, NoOp);

 TTypeOfCalls = (Blocking, NonBlocking);

 TSockTypes   = (SockStrm, SockDgram, SockRaw);

 TServices    = (NoService, Echo, Discard,
```

```
                    Systat, Daytime, Netstat,
                    Qotd, Chargen, ftp, telnet,
                    smtp, time, rlp, nameserver,
                    whois, domain, mtp, tftp, rje,
                    finger, http, link, supdup,
                    hostnames, ns, pop2,pop3,
                    sunrpc, auth, sftp,
                    uucp_path, nntp);

 TProtoTypes = (IP, ICMP, GGP, TCP, PUP, UDP);

 TAsyncTypes = (AsyncName, AsyncAddr, AsyncServ,
                AsyncPort, AsyncProtoName, AsyncProtoNumber);
const

 NULL : Char   = #0;
{ This is required for end of data sent to host }
 CRLF : array[0..2] of char = #13#10#0;

 MaxBufferSize  = 255;

 {strings for services property}
 ServiceStrings : array[TServices] of String[10]
                = ('No Service',
                   'echo      ',
                   'discard   ',
                   'systat    ',
                   'daytime   ',
                   'netstat   ',
                   'qotd      ',
                   'chargen   ',
                   'ftp       ',
                   'telnet    ',
                   'smtp      ',
                   'time      ',
                   'rlp       ',
                   'nameserver',
                   'whois     ',
                   'domain    ',
                   'mtp       ',
                   'tftp      ',
                   'rje       ',
                   'finger    ',
                   'http      ',
                   'link      ',
                   'supdup    ',
                   'hostnames ',
                   'ns        ',
                   'pop2      ',
                   'pop3      ',
                   'sunrpc    ',
                   'auth      ',
                   'sftp      ',
```

```
                    'uucp-path ',
                    'nntp      ');
{ protocol strings }
 ProtoStrings : array[TProtoTypes] of String[4] =
                   ('ip  ',
                    'icmp',
                    'gcmp',
                    'tcp ',
                    'pup ',
                    'udp ');

type

 CharArray   = array[0..MaxBufferSize] of char;

 TAddrTypes  = (AFUnspec,    { unspecified}
                AFUnix,      { local to host (pipes, portals) }
                AFInet,      { internetwork: UDP, TCP, etc. }
                AFImpLink,   { arpanet imp addresses }
                AFPup,       { pup protocols: e.g. BSP }
                AFChaos,     { mit CHAOS protocols }
                AFNs,        { XEROX NS protocols  }
                AFIso,       { ISO protocols }
                AFOsi,       { OSI is ISO }
                AFEcma,      { european computer manufacturers }
                AFDatakit,   { datakit protocols }
                AFCcitt,     { CCITT protocols, X.25 etc }
                AFSna,       { IBM SNA }
                AFDecNet,    { DECnet }
                AFDli,       { Direct data link interface }
                AFLat,       { LAT }
                AFHyLink,    { NSC Hyperchannel }
                AFAppleTalk, { AppleTalk }
                AFNetBios,   { NetBios-style addresses }
                AFMax);

const
 ServDefault     =  NoService;
 ProtoDefault    =  TCP;
 SockDefault     =  SockStrm;
 AddrDefault     =  AFINET;
 PortNoDefault   =  0;

type
   TWSocket = class(TComponent)
   private
    { Private declarations }
    FValidSocket : u_int;
    FParent      : TComponent;
    FSockType    : TSockTypes;
    FService     : TServices;
    FProtocol    : TProtoTypes;
    FAddrType    : TAddrTypes;
```

```
    FAsyncOpDone : TAsyncOperationDone;
    FAsyncType   : TAsyncTypes;
    FData,
    FRemoteName,
    FAsyncRemoteName,
    FAsyncService,
    FAsyncPort,
    FAsyncProtocol,
    FAsyncProtoNo,
    FLocalName,
    FInfo        : String;
    FBusy,
    FCancelAsyncOp,
    FOKToDisplayErrors : Boolean;
    FStatus            : TConditions;
    FTypeOfCall        : TTypeOfCalls;
    FConnected         : Boolean;
    FHand,
    FTaskHandle        : THandle;
    FHomeHostName      : String;
    FOperation         : TOperations;
    FUpDateTime,
    FOnChange,
    FOnAsyncDone       : TNotifyEvent;
    FWSALastError,
    FTimeOut,
    FRC                : Integer;
    FVendor,
    FWSVersion,
    FMaxNoSocks,
    FMaxUDPPSize,
    FWSStatus,
    FServiceName,
    FPortName,
    FProtocolName,
    FProtocolNo        : String;
    FAsyncBuff         : array[1..MAXGETHOSTSTRUCT] of char;
    FNoOfBlockingTasks : Integer;
    { access routines for properties}
    function  GetLocalName : String;
    procedure SetRemoteHostName(NameReqd : String);
    function  GetDataBuff : String;
    procedure SetDataBuff(DataReqd : String);
    function  GetDatagram : String;
    procedure SetDatagram(DataReqd : String);
    procedure SetUpPort;
    procedure SetTimeOut(TimeOutReqd : Integer);
    procedure SetPortName(ReqdPortName : String);
    procedure SetServiceName(ReqdServiceName : String);
    {winsock calls}
    procedure GetProt(Protocol : PChar);
    procedure ConnectToHost;
    function  GetOOBData : String;
```

```
    procedure SetOOBData(ReqdOOBData : String);
    function  StartUp : Boolean;
    procedure CleanUp;
   protected
    { Protected declarations }
    FPortNo      : Integer;
    FHost        : pHostent;
    FServ        : pServent;
    FProto       : pProtoEnt;
    FHostEntryBuff,
    FProtoName,
    FServName    : CharArray;
    Fh_addr      : pChar;
    FpHostBuffer,
    FpHostName   : array[0..MAXGETHOSTSTRUCT-1] of char;
    FAddress     : THostAddr;
    FMsgBuff     : CharArray;
    FSocketNo    : TSocket;
    FSockAddress : TSockAddr_In;
    FHandle      : THandle;
    FStarted     : Boolean;
    FHwnd,
    FAsyncHWND   : HWND;
    procedure SetUpAddr; virtual;
    procedure SetUpAddress; virtual;
    procedure GetHost;
    procedure GetServ;
    function  CreateSocket : TSocket;
    function  WSAErrorMsg : String;
    procedure SetProtocolName(ReqdProtoName : String);
    procedure SetProtoNo(ReqdProtoNo : String);
    procedure Change; virtual;
    procedure AsyncChange; virtual;
    procedure WMTimer(var Message : TMessage); message wm_Timer;
    procedure PostInfo(const s: String);
    procedure StartAsyncSelect; virtual;
    procedure AsyncOperation(var Mess : TMessage);
    function  GetAsyncHostName : String;
    procedure SetAsyncHostName(ReqdHostName : String);
    function  GetAsyncService : String;
    procedure SetAsyncService(ReqdService : String);
    function  GetAsyncPort : String;
    procedure SetAsyncPort(ReqdPort : String);
    function  GetAsyncProtoName : String;
    procedure SetAsyncProtoName(ReqdProtoName : String);
    function  GetAsyncProtoNo : String;
    procedure SetAsyncProtoNo(ReqdProtoNo : String);
    procedure CancelAsyncOperation(CancelOp : Boolean);
  public
  { Public methods }
    procedure GetServer;
    procedure QuitSession;
    procedure Cancel;
```

```
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
  { Public properties }
    property WSVendor      : String  read FVendor;
    property WSVersion     : String  read FWSVersion;
    property WSMaxNoSocks  : String  read FMaxNoSocks;
    property WSMaxUDPPSize : String  read FMaxUDPPSize;
    property WSStatus      : String  read FWSStatus;
    property TimeOut       : Integer read FTimeOut
                                     write SetTimeOut
                                     default 0;
    property Info          : String  read FInfo write FInfo;
    property WSErrNo       : Integer read FWSALastError
                                     default 0;
    property Connected     : Boolean read FConnected
                                     write FConnected
                                     default FALSE;
    property LocalName     : String  read GetLocalName
                                     write FLocalName;
    property Status        : TConditions read FStatus
                                         write FStatus
                                         default None;
    property HostName      : String read FRemoteName
                                    write SetRemoteHostName;
    property WSService     : String read FServiceName
                                    write SetServiceName;
    property WSPort        : String read FPortName
                                    write SetPortName;
    property WSProtoName   : String read FProtocolName
                                    write SetProtocolName;
    property WSProtoNo     : String read FProtocolNo
                                    write SetProtoNo;
    property Data          : String read GetDataBuff
                                    write SetDataBuff;
    property Datagram      : String read GetDatagram
                                    write SetDatagram;
    property OOBData       : String read GetOOBData
                                    write SetOOBData;
    {Asynchronous properties}
    property CancelAsyncOP : Boolean read FCancelAsyncOp
                                     write CancelAsyncOperation;
    property AsyncHostName : String  read GetAsyncHostName
                                     write SetAsyncHostName;
    property AsyncService  : String  read GetAsyncService
                                     write SetAsyncService;
    property AsyncPortNo   : String  read GetAsyncPort
                                     write SetAsyncPort;
    property AsyncProtocol : String  read GetAsyncProtoName
                                     write SetAsyncProtoName;
    property AsyncProtoNo  : String  read GetAsyncProtoNo
                                     write SetAsyncProtoNo;
  published
  { Published declarations }
```

```
    property OkToDisplayErrors : Boolean read FOKToDisplayErrors
                                          write FOKToDisplayErrors
                                          default TRUE;
    property HomeServer        : String read FHomeHostName
                                         write FHomeHostName;
    property SockType          : TSockTypes read FSockType
                                            write FSockType
                                            default SOCKSTRM;
    property Service           : TServices read FService
                                           write FService
                                           default NoService;
    property Protocol          : TProtoTypes read FProtocol
                                             write FProtocol
                                             default TCP;
    property AddrType          : TAddrTypes  read FAddrType
                                             write FAddrType
                                             default AFInet;
    property CallType          : TTypeOfCalls read FTypeOfCall
                                              write FTypeOfCall
                                              default blocking;
    property OnChangeInfo      : TNotifyEvent read FOnChange
                                              write FOnChange;
    property OnAsyncDone       : TNotifyEvent read FOnAsyncDone
                                              write FOnAsyncDone;
  end;

procedure Register;

procedure Register;

implementation
…
…
end.
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Under Unix, networking protocols are typically compiled directly into the operating systems kernel and are thus always initialized and available to applications. Under Windows, however, no such standard exists. Before an application uses any network services, the Winsock DLL requires that the application first make an initialization call. The WSock component performs this with its private method, **StartUp**. The constructor **TWSocket.Create** sets the properties to their default values, as shown in Listing 4.2, then calls **StartUp**.

Listing 4.2 The TWSocket.Create Constructor

```
constructor TWSocket.Create(AOwner : TComponent);
begin
 inherited Create(AOwner);
 FParent       := AOwner;
 FValidSocket  := INVALID_SOCKET;
 FSockType     := SockDefault;
 FAddrType     := AddrDefault;
 FService      := ServDefault;
 FProtocol     := ProtoDefault;
 with FSockAddress do
 begin
  sin_family := PF_INET;
  sin_addr.s_addr := INADDR_ANY;
  sin_port := 0;
 end;
 FSocketNo     := INVALID_SOCKET;
 FLocalName    := '';
 FInfo         := '';
 FTypeOfCall   := Blocking;
 FStarted      := StartUp;
 if not FStarted then
```

```
 begin
  inherited Destroy;
  Exit;
 end;
 FHomeHostName := 'local';
 FOKToDisplayErrors := TRUE;
 FConnected         := FALSE;
 FWSALastError      := 0;
 FTimeOut           := 0;
 FNoOfBlockingTasks := 0;
 PostInfo('Version ' + WSockVersionNo);
 FAsyncHWND         := AllocateHWND(AsyncOperation);
end;
```

StartUp determines whether the Winsock DLL is available as well as determining the status of the Winsock DLL. **StartUp** assigns values to the following properties: **FVendor**, **FWSVersion**, **FWSStatus**, **FMaxNoSocks**, and **FMaxUDPPSize**, as shown in Listing 4.3. These properties are for information only and have no effect on the running of the main application. You can display the data returned by **StartUp** if desired. If **StartUp** is unable to initialize the Winsock DLL, it displays an error message and exits.

Listing 4.3 The TWSocket.Startup Function

```
function TWSocket.StartUp : Boolean;
var
 VersionReqd : WordRec;
begin
 with VersionReqD do
 begin
  Hi := 1;
  Lo := 1;
 end;
 Result := WSAStartUp(Word(VersionReqD),@myWsaData) = 0;
 if not Result then
  PostInfo('Cannot start Winsock')
 else
 begin
  with myWsaData do
  begin
   FVendor       := StrPas(szDescription);
   FWSVersion    := Concat(IntToStr(Hi(wVersion)),
                           '.',(intToStr(Lo(wVersion))));
   FWSStatus     := StrPas(szSystemStatus);
   FMaxNoSocks   := IntToStr(iMaxSockets);
   FMaxUDPPSize  := IntToStr(iMaxUDPDg);
  end;
  PostInfo('Started Winsock');
 end;
end;
```

The “clean up” task is just as important as initialization. When the client application completes its task—and thus its need for Winsock services—it must inform the Winsock DLL to free the memory that was in use. WSock’s **CleanUp** procedure takes care of this automatically when closing the Winsock DLL, as shown in Listing 4.4.

Listing 4.4 The TWSocket.CleanUp Procedure

```
procedure TWSocket.CleanUp;
begin
 if FStarted then
 begin
  FStarted := False;
  if WSACleanUp = SOCKET_ERROR then
   MessageDlg(WSAErrorMsg, mtError,[mbOk],0)
 end;
end;
```

Finally, a call to the Winsock DLL can fail to complete for a variety of network-related reasons. When this happens, WSock, via the **WSAErrorMsg** function, calls Winsock’s **WSAGetLastError** function, reporting the error.

Running the Resolver Application

The RESOLVER program is a Winsock database conversion program that employs a few of the more interesting methods and properties provided by **TWSocket**. RESOLVER can resolve a host name to its Internet (IP) address, and vice versa. Given either a port number or a service name, it can derive the other. And it can translate between protocol numbers and protocol names. These are practical demonstrations, for the resolution of a host name and a service name are the most common operations that a Winsock application ever performs.

The complete application as it appears in the Delphi IDE is shown in Figure 4.1. Click on the **WSocket1** component and you’ll see its properties in the Object Inspector, as shown in Figure 4.2. The default values, shown here, are fine for performing resolutions using blocking functions. The **Service** property is set to its default value of **NoService**, as there is no specific service to perform resolution tasks in our application.

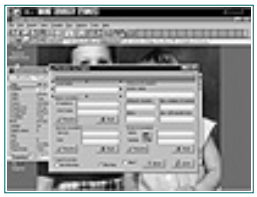

FIGURE 4.1 The RESOLVER application.

FIGURE 4.2 The WSocket properties display.

Figure 4.3 shows the Events page on which there are two event handlers. Whenever the status of the Winsock DLL changes, the **WSocket1ChangeInfo** event procedure passes the information from WSock to the application. Similarly, the **WSocket1AsyncDone** event procedure posts information whenever an asynchronous function completes its task.

FIGURE 4.3 The WSocket events display.

When you run the RESOLVER application, the **Application.CreateForm** procedure in RESOLVER.DPR calls the constructor **TWSocket.Create** to set WSock's properties to their default settings. After the constructor initializes the component and calls the Winsock DLL successfully, the **TMainForm.FormCreate** procedure performs several tasks, as shown in Listing 4.5.

Listing 4.5 The main form's FormCreate procedure

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
 with WSocket1 do
 begin
  MachineName.Text      := LocalName;
  VendorName.Text       := WSVendor;
  VersionNo.Text        := WSVersion;
  MaxNoSockets.Text     := WSMaxNoSocks;
  MaxUDPacketSize.Text  := WSMaxUDPPSize;
  WSStatusInfo.Text     := WSStatus;
  CallType              := Blocking;
 end;
 if WSocket1.CallType = Blocking then
 begin
```

```
  AbortAsyncHostBtn.Enabled  := FALSE;
  AbortAsyncServBtn.Enabled  := FALSE;
  AbortAsyncProtoBtn.Enabled := FALSE;
 end;
 HintBtn.Checked := TRUE;
 MainForm.ShowHint := TRUE;
end;
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

What's My Name?

RESOLVER reports the name that your machine presents to the network. It can do so because the **WSocket1.LocalName** property assigns the your machine's name to **Machine.Text**. The **TWSocket.GetLocalName** method acts as a wrapper for the Winsock API's **gethostname** routine. It retrieves your machine's name from the local host's file (typically located in the Windows directory) and returns it in the **LocalName** property.

Listing 4.6 shows the code for the **TWSocket.GetLocalName** method in WSOCK.PAS. It's important to note that **gethostname**, as with all Winsock routines, handles only ASCII strings. The **GetLocalName** method uses the **StrPas** function to convert the ASCII string to an Object Pascal. The machine name then appears in the **MachineName** edit control. If your machine is nameless, **GetLocalName** simply returns a blank string. Similarly, the miscellaneous information gathered by **TWSocket.StartUp** about the particular Winsock DLL in use is passed back to RESOLVER by the properties **FVendor**, **FWSVersion**, **FWSStatus**, **FMaxNoSocks**, and **FMaxUDPPSize** for display in the **WSInfoBox** group box.

Listing 4.6 The GetLocalName function

```
function TWSocket.GetLocalName : String;
var
 LocalName : array[0..MaxBufferSize] of Char;
begin
 if gethostname(LocalName, SizeOf(LocalName)) = 0 then
 begin
  Result := StrPas(LocalName);
 end else
 begin
  Result := '';
 end
end;
```

What's the Address?

The most common operation performed by a Winsock application is resolving a host name in

blocking mode. In this case, blocking means that the application is waiting for a response from a remote computer—a response that may never come. Unable to proceed or respond to input until it receives a response, a blocked application often appears “dead” to the user.

Under Unix, this type of operation poses little problem. Even if an application blocks, the pre-emptive nature of Unix allows other applications to operate normally. Windows 3.1, in contrast, implements only cooperative multitasking. Executing a blocking Winsock operation would lock the brakes on your Windows system.

To allow Windows to continue to execute in a blocking situation, Winsock replaces each blocking functions with a pseudo-blocking asynchronous equivalent. Instead of blocking, these routines enter a polling loop while waiting for the network event to complete. These non-blocking routines are identified by the **WSAAsync** prefix before the name. For example, **WSAAsyncGetHostByName** is the asynchronous version of **gethostbyname**.

Typically, an Internet host identifies itself over the network with a unique address in the form of a dotted decimal number quadruple such as 127.0.0.1. (Note that this is the special loopback address that you can use to test your Winsock applications on a non-networked machine.) Although highly convenient for computers, these addresses hold little appeal for humans. To reconcile this problem, a system was implemented that allows the creation of a unique human-readable name for each Internet address. For example, the name slipper109.iaccess.za is equivalent to the Internet address 196.7.7.109.

To resolve a host name, enter the name in RESOLVER’s Host edit control. After you press the Resolve button in the Name Resolution group box, RESOLVER assigns the name that you gave in the **Host.Text** control to the **Hostname** property. Then the property calls **TWSocket.SetRemote HostName**. Listing 4.7 shows how WSock handles this. If the **NameReqd** string is empty, **SetRemoteHostName** reports the error and exits. Otherwise, **StrpCopy** is used to convert **FRemoteName** from a Pascal string to an ASCII string.

Listing 4.7 Mapping a host name to its Internet address

```
procedure TWSocket.SetRemoteHostName(NameReqd : String);
var
 P : Pointer;
 IPAddress : LongInt;
 IAddr : TIN_ADDR;
begin
 FRemoteName := NameReqd;
 if Length(FRemoteName) = 0 then
 begin
  FStatus := Failure;
  MessageDlg('No host name given!', mtError,[mbOk],0);
  Exit;
 end;
 PostInfo('Resolving host');
 StrPCopy(FpHostName, FRemoteName);
 { check what type of address has been entered }
 IPAddress := inet_addr(FpHostName);
 if IPAddress <>INADDR_NONE then {this is a dotted address}
 begin
  FAddress := IPAddr;
  P := addr(IPAddress);
  case AddrType of
   AFINET : FHost := GetHostByAddr(P, 4, AF_INET);
```

```
 end;
end
else {no, it looks like a human readable hostname}
begin
 FAddress := HostAddr;
 FHost    := GetHostByName(FpHostName);
end;
if FHost = NIL then
begin{Unknown host, so aborting…}
 if OkToDisplayErrors then
  MessageDlg('Unknown host', mtError,[mbOk],0);
 FStatus := Failure;
 Exit;
end;
PostInfo('Host found');
FStatus := Success;
Move(FHost^.h_addr_list^, Fh_addr, SizeOf(FHost^.h_addr_list^));
if FAddress = HostAddr then
begin
 SetUpAddress;
 FRemoteName := StrPas(inet_ntoa(FSockAddress.sin_addr));
end
else
if FAddress = IPAddr then
begin
 FRemoteName := StrPas(FHost^.h_name);
 PostInfo('Host found…');
end;
end;
```

Next, the **SetRemoteHostName** method checks whether the string already contains a numeric Internet address using the **inet_addr** function. If not, the method assumes that the string contains a host name and calls the **gethostbyname** function to resolve it to an IP address. If the host name is not present in the local hosts file, **gethostbyname** looks for the name in a foreign hosts file elsewhere on the network.

If the name is not found, the lookup process times out and sets the private property **FHost**, which is a **pHostent** structure, to **NIL**. Then, **SetRemoteHostName** posts an error message, sets the **FStatus** flag to **Failure**, and exits back to the calling application. If the IP address is found, however, the **gethostbyname** function returns a pointer to **FHost**, which contains the IP address. The **SetUpAddress** procedure then extracts the IP address from the **FHost** structure. Finally, **SetRemoteHostName** sends back the dotted address as a Pascal string using the following statement:

```
FRemoteName := StrPas(inet_ntoa(FSockAddress.sin_addr));
```

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

The **inet_ntoa** function converts the returned IP address to an ASCII string in dotted format. The **StrPas** function finishes the conversion to a Pascal string. The address information for the socket is placed in **FSockAddress**, where it will later be used to set up a connection with a host machine. Setting the **Hostname** property writes the IP address to the **IPName** edit control, as shown in Figure 4.4.

FIGURE 4.4 RESOLVER after resolving a host name.

What's Your Name?

RESOLVER can also derive the name of a host from its numeric Internet address. The process begins when you enter an address into the **IPName** edit control, as shown in Figure 4.5. When you click the Resolve button, RESOLVER passes the address string in **IPName.Text** to the **SetRemoteHostName** method via the **Hostname** property.

FIGURE 4.5 A dotted IP address ready to resolve.

As before, the **SetRemoteHostName** method uses the **inet_addr** function to check whether the string is in valid Internet address form. Before calling this function, however, the method assigns the address of the **IPAddress** string to a pointer, **P**, which **gethostbyaddr** requires as one of its parameters.

If **inet_addr** returns a result other than **INADDR_NONE** (meaning that the string is a numeric Internet address), **SetRemoteHostName** calls **gethostbyaddr**. Like the call to **gethostbyname**, this call may also block. If **gethostbyaddr** is successful, it returns a pointer to the **pHostent** structure. If no corresponding name is found for the IP address, **FHost** is set to **NIL** and **SetRemoteHostName** reports the error, sets the **FStatus** flag, and exits. The **Hostname** property writes the host name obtained, using the statement below, back to the Host edit control:

```
FRemoteName := StrPas(FHost^.h_name);
```

Getting the Name Asynchronously

Using the blocking lookup functions **gethostbyname** and **gethostbyaddr** is fairly straightforward. Employing the asynchronous versions of these functions, **WSAAyncGetHostByName** and **WSAAsyncGetHostByAddr**, is a little more complex. To understand the asynchronous process, let's go through the steps of calling **WSAAsyncGetHostByName** from the RESOLVER program.

First, change the **CallType** property from **Blocking** to **NonBlocking** by selecting the **NonBlocking** radio button in the **TypeOfLookup** group box as shown in Figure 4.6. Pressing the Resolve button now assigns the name to the **AsyncHostName** property and passes it to the **SetAsyncHostName** procedure as shown in Listing 4.8.

FIGURE 4.6 Changing from blocking to non-blocking.

Listing 4.8 Resolving the host name.

```
procedure TWSocket.SetAsyncHostName(ReqdHostName : String);
var
 Size : PInteger;
 P : Pointer;
 IPAddress : TIn_addr;
 SAddress: array[0..31] of char;
 sa : Tin_addr;
begin
 FAsyncRemoteName := ReqdHostName;
 if Length(FAsyncRemoteName) = 0 then
 begin
  FStatus := Failure;
  MessageDlg('No host name given!', mtError,[mbOk],0);
  Exit;
 end;
 StrPcopy(SAddress, FAsyncRemoteName);
 IPAddress.s_addr := inet_addr(SAddress);
 if IPAddress.s_addr <> INADDR_NONE then {this is a dotted address}
 begin
  FAddress := IPAddr;
  FAsyncType := AsyncAddr;
  if IPAddress.s_addr <> 0 then
  FTaskHandle := WSAAsyncGetHostByAddr(FAsyncHWND, ASYNC_EVENT,
                   pChar(@IPAddress), 4, PF_INET,
                   @FAsyncBuff, SizeOf(FAsyncBuff));
  if FTaskHandle = 0 then
  begin
   MessageDlg(WSAErrorMsg,mtError,[mbOk], 0);
   FStatus := Failure;
   if FNoOfBlockingTasks > 0 then
    dec(FNoOfBlockingTasks);
   Exit;
```

```
  end else FStatus := Success;
 end
 else {no, it looks like a human readable hostname}
 begin
  FAddress := HostAddr;
  FAsyncType := AsyncName;
  Inc(FNoOfBlockingTasks);
  FTaskHandle := WSAAsyncGetHostByName(FAsyncHWND, ASYNC_EVENT,
                                       @FpHostName, @FAsyncBuff,
                                       MAXGETHOSTSTRUCT);
  if FTaskHandle = 0 then
  begin
   MessageDlg(WSAErrorMsg,mtError,[mbOk], 0);
   FStatus := Failure;
   if FNoOfBlockingTasks > 0 then
    dec(FNoOfBlockingTasks);
   Exit;
  end else FStatus := Success;
 end;
end;
```

SetAsyncHostName calls the **WSAAsyncGetHostByName** procedure with five important arguments. **FASyncHWND** is a handle to the window in which the asynchronous function will post its message on completion of the lookup operation. This window handle is initialized in the **TWSocket.Create** constructor by a call to **AllocateHWND** with **AsyncOperation** as its procedural parameter. **ASYNC_EVENT** is the event notification constant used by **WSAAsyncGetHostByName. FAsyncBuff** is an array of characters that holds the result of the operation. Finally, **MAXGETHOSTSTRUCT** is a Winsock constant representing the maximum size of the **FAsyncBuff** buffer. The **WSAAsyncGetHostByName** procedure returns the task number of the call as a **TaskHandle** type that is assigned to **FTaskHandle**.

WSAAyncGetHostByName returns immediately with a value of 0 if the call was unsuccessful or greater than 0 if successful. However, a non-zero value for **FTaskhandle** means only that the call to **WSAAyncGetHostByName** succeeded, not that the subsequent lookup operation (which continues to execute in the background) will be successful.

When the lookup does complete, the Winsock DLL triggers a **ASYNC_EVENT** event, notifying the **AsyncOperation** procedure that it should examine the **ASYNC_EVENT** message, as shown in Listing 4.9.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Listing 4.9 The AsyncOperation procedure

```
procedure TWSocket.AsyncOperation(var Mess : TMessage);
var
 MsgErr : Word;
begin
 if Mess.Msg = ASYNC_EVENT then
 begin
  MsgErr := WSAGetAsyncError(Mess.lparam);
  if (MsgErr <> 0) and (StrLen(FpHostName) > 0) then
  begin
   FStatus := Failure;
   MessageDlg(WSAErrorMsg,mtError,[mbOk], 0);
   Exit;
  end
  else
  begin
    FStatus := Success;
    PostInfo('WSAAsync operation succeeded!');
    case FAsyncType of
     AsyncName,
     AsyncAddr : begin
                   FHost := pHostent(@FAsyncBuff);
                   if FHost = NIL then
                     begin{Unknown host, so aborting…}
                       if OkToDisplayErrors then
                         MessageDlg('Unknown host', mtError,[mbOk],0);
                       FStatus := Failure;
                       Exit;
                     end;
                   case FAddress of
                     IPAddr :
                        begin
                          Move(FHost^.h_addr_list^, Fh_addr,
                               SizeOf(FHost^.h_addr_list^));
                          FAsyncRemoteName :=
                            StrPas(FHost^.h_name);
```

```
                    end;
                  HostAddr :
                    begin
                      Move(FHost^.h_addr_list^, Fh_addr,
                           SizeOf(FHost^.h_addr_list^));
                      SetUpAddress;
                      FAsyncRemoteName:=
                        StrPas(inet_ntoa(FSockAddress.sin_addr));
                    end;
                 end;{case}
                 AsyncChange;
               end;
AsyncServ : begin
              FServ := pServent(@FAsyncBuff);
              if FServ = NIL then
              begin { No service available }
                MessageDlg(WSAErrorMsg, mtError,[mbOk],0);
                FStatus := Failure;
                Exit;
              end;
              FAsyncPort := IntToStr(ntohs(FServ^.s_port));
              AsyncChange;
            end;
AsyncPort  : begin
               FServ := pServent(@FAsyncBuff);
               if FServ = NIL then
                 begin { No service available }
                   MessageDlg(WSAErrorMsg, mtError,[mbOk],0);
                   FStatus := Failure;
                   Exit;
                 end;
               FAsyncService := StrPas(FServ^.s_name);
               AsyncChange;
             end;
AsyncProtoName :
             begin
               FProto := pProtoEnt(@FAsyncBuff);
               if FProto = NIL then
                 begin
                   MessageDlg(WSAErrorMsg, mtError,[mbOk],0);
                   FStatus := Failure;
                   Exit;
                 end;
               FAsyncProtoNo := IntToStr(FProto^.p_proto);
               AsyncChange;
             end;
AsyncProtoNumber :
             begin
               FProto := pProtoEnt(@FAsyncBuff);
               if FProto = NIL then
                 begin
                   MessageDlg(WSAErrorMsg, mtError,[mbOk],0);
                   FStatus := Failure;
                   Exit;
                 end;
               FAsyncProtocol := StrPas(FProto^.p_name);
               AsyncChange;
```

```
                end;
   end; {case}
   if FNoOfBlockingTasks > 0 then
    dec(FNoOfBlockingTasks);
  end;
 end;
end;
```

The **WSAGetAsyncError** macro checks the **Mess** variable. If it indicates that an error occurred, **AsyncOperation** calls **WSAErrorMsg** to display the cause of the error, then exits with the **FStatus** flag set to **Failure**. If no error has occurred, we parse the **FAsyncType** variable.

When we called **WSAAyncGetHostByName**, we set the **FAsyncType** to **AsyncName** to indicate that we were performing an asynchronous name lookup. The **case** statement now branches based on the value of **FAsyncType** to the **AsyncName** clause. There, the character array **FAsyncBuff**, containing the result of the lookup, is typecast to a **pHostent** structure and stored in **FHost**. The address structure for the resolved host is read by **SetUpAddress** to get the corresponding IP address. **AsyncChange** calls **GetAsyncHostName** to return the IP address back to RESOLVER.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Who's at This Address?

To further illustrate the use of asynchronous mode, we'll examine how the **WSAAyncGetHostByAddr** function (shown in Listing 4.8) retrieves a host name when given only an Internet address. To use this function in the RESOLVER application, set the **CallType** property to **NonBlocking** in the **TypeOfLookUp** group box, and enter an Internet address in the **IPName** edit control.

As before, we assign the name to the **AsyncHostName** property for handling by the **TWSocket.SetAsyncHostName** method. If the name we passed is an empty string, **SetAsyncHostName** sets the **FStatus** flag to **Failure**, posts an error message, and exits. After establishing that the **FAsync RemoteName** is not empty, we call the **inet_addr** function to determine whether the string is a dotted decimal Internet address or host name. A return value different than **INADDR_NONE** indicates the string is in Internet address format.

This string is then passed to **WSAAyncGetHostByAddr** to get the host information for the Internet address. A successful call to **WSAAync GetHostByAddr** sets the **FTaskHandle** to a number greater than zero, but doesn't ensure that we will get a valid result from **WSAAyncGetHostByAddr** on completion. The method exits back to the RESOLVER application, and the lookup continues in the background.

When the lookup operation completes, the Winsock DLL posts a message to WSock by triggering the **ASYNC_EVENT** event. This trigger wakes up the **TWSocket.AsyncOperation** method, which examines the **Mess** variable. If **Mess** contains an error, the **AsyncOperation** method calls **WSAErrorMsg** to determine the error, sets the **FStatus** flag to **Failure**, and exits.

If the **Mess** variable contains no error, a case statement parses **FAsyncType**. In this example, **FAsyncType** has the value **AsyncAddr**, so the same portion of code executes that handled the **AsyncName** case. Next, we parse **FAddress** to execute the section of code that handles the result of **WSAAyncGetHostByAddr**. This setting is automatically determined by the **SetAsyncHostName** method by using the result of the **inet_addr** operation. That is, **FAddress** is set to **IPAddr** when a dotted decimal address is found, otherwise it is set to **HostAddr** for a host name. The host name is then extracted by the following code:

```
Move(FHost^.h_addr_list^, Fh_addr, SizeOf(FHost^.h_addr_list^));
FAsyncRemoteName := StrPas(FHost^.h_name);
```

This result is posted back to the application via the **AsyncChange** procedure.

Canceling a WSAAsync Operation

Because asynchronous operations run outside of normal program flow, canceling them poses a unique problem. To cancel any currently executing asynchronous operations, the Winsock API provides the **WSACancelAsyncRequest** function. (Note, however, that this function cannot cancel operations started by the **WSAAsyncSelect** function.) Listing 4.10 shows the **WSACancelAsyncRequest** function wrapped up inside the **FAsyncType** method.

Listing 4.10 Canceling an asynchronous operation

```
procedure TWSocket.CancelAsyncOperation(CancelOP : Boolean);
begin
 if WSACancelAsyncRequest(THandle(FTaskHandle)) =
    SOCKET_ERROR then
 begin
  MessageDlg(WSAErrorMsg, mtError,[mbOk],0);
  FStatus := Failure;
 end
 else
 begin
  FStatus := Success;
  PostInfo('WSAAsync lookup cancelled!');
 end;
end;
```

But the **WSACancelAsyncRequest** method is not visible to the RESOLVER application. So, how does RESOLVER cancel a **WSAAsyncGetHostByName** or **WSAAsyncGetHostByAddr** call? By accessing the **CancelAsyncOperation** method using the public boolean property, **CancelAsyncOp**.

Listing 4.11 shows what happens when you press the **Abort** button in the **NameResBox** group box in RESOLVER. Because the call type is non-blocking, we assign the **CancelAsyncOp** to be **True**. This signals WSock, via the **CancelAsyncOperation**, to call the **WSACancelAsyncRequest** and thus to kill the asynchronous operation.

Listing 4.11 Signaling a cancel operation.

```
procedure TMainForm.AbortAsyncHostBtnClick(Sender: TObject);
begin
 with WSocket1 do
 begin
  if CallType = NonBlocking then
   CancelAsyncOp := TRUE;
  NameResBtn.Enabled := TRUE;
 end;
end;
```

Resolving Ports and Services

As with name and address resolution, we can resolve a service name and a port using either a blocking or non-blocking (asynchronous) call. The Winsock API functions **getservbyname** or **getservbyport** provide these services in blocking mode.

Looking up the port associated with a particular service type is quite similar to the process of getting the host name. For example, if we wish to find the corresponding port number for FTP, we enter FTP in the **ServiceName** edit control, and then assign it to the **WSService** property. This passes the service name to the **TWSocket.SetServiceName** method for conversion. This code is shown in Listing 4.12.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Listing 4.12 Looking up a port.

```
procedure TMainForm.ServResBtnClick(Sender: TObject);
begin
 if ServiceName.AutoSelect then
 begin
  ServiceName.AutoSelect := FALSE;
  PortName.AutoSelect := TRUE;
  with WSocket1 do
  begin
   if CallType = NonBlocking then
   begin
    AsyncService := ServiceName.Text;
    PortName.Text := 'Pending…';
   end
   else
   WSService := ServiceName.Text;
   if Status = Success then
    PortName.Text := WSService
   else
    PortName.Text := 'unknown service';
  end;
 end else
 begin
  PortName.AutoSelect := FALSE;
  ServiceName.AutoSelect := TRUE;
  with WSocket1 do
  begin
   if CallType = NonBlocking then
   begin
    AsyncPortNo := PortName.Text;
```

```
    ServiceName.Text := 'Pending…';
   end
   else
   WSPort := PortName.Text;
   if Status = Success then
   begin
    ServiceName.Text := WSPort;
   end
   else
    PortName.Text := '0';
  end;
 end;
end;
```

After copying the **ReqdServiceName** to **ServName** using the Object Pascal function **StrPCopy**, the protocol string is set to "TCP", one of the required parameters for **getservbyname**. The protocol is set to "TCP" by default, which means that if you try to determine a port number for a service based on a different protocol, the **getservbyname** function would return a **NIL** pointer. The only way to use different protocols is to set the **Protocol** property in the Object Inspector to, say, "UDP", and then recompile and re-run RESOLVER. Alternately, you can modify RESOLVER to add a protocol option.

The **SetServiceName** method calls the **getservbyname** function to get the corresponding port number. If the service is found, the **getservbyname** function returns a pointer to **FServ**, a **pServent** structure. This structure now holds the port number. Otherwise, the function returns a null pointer, in which case the method calls **WSAErrorMsg** to determine and report the cause of the error, sets the **FStatus** flag to **Failure**, and exits back to the calling application. The port number is extracted using the following statement:

```
FServiceName := IntToStr(ntohs(FServ^.s_port));
```

The string is passed back to the application by the **WSService** property using the following code:

```
if Status = Success then
 PortName.Text := WSService
else
 PortName.Text := 'unknown service';
```

Figure 4.7 shows the result of the conversion.

FIGURE 4.7 The result of the service conversion.

Finding the Service

The process of translating a port number into its corresponding service is quite similar to the service lookup we just examined, except we use the blocking Winsock function **getservbyport**. Instead of tracing through that process, we'll look at the use of **WSAAsyncGetServByPort**, the asynchronous version of **getservbyport**.

To use the asynchronous mode, first change the **CallType** property from blocking to non-blocking by selecting the Non-blocking radio button in the **TypeOfLookup** group box. Then enter a port number in the **PortName** edit control and press the Resolve button in the Service resolution group box.

The port number, in **PortName.Text**, is passed to the **TWSocket.SetAsyncPort** method as **ReqdPort** when we assign it to the **AsyncPortNo** property. **SetAsyncPort** verifies that the port number string is not empty, then copies it to **FPortNo**, a null terminated string. Calling **WSAAsyncGetServByPort** then fetches the corresponding port number.

The result of the call is stored by **FTaskHandle**. If **FTaskHandle** is zero, then the call has failed. Otherwise, the call was successful, and the **SetAsyncPort** exits back to the application, leaving the lookup process to continue in the background. When the lookup process is complete, **AsyncOperation** is invoked by a message from the Winsock DLL. The **Mess** variable is examined for an error. If there is no error, the method returns the port number. Otherwise, the method sets the **FStatus** flag to **Failure**, reports the error, and exits back to the application.

Resolving Protocols

Getting the protocol name and number is not as common an operation as the other conversion functions, but the functions are supported in WSock for completeness. The Winsock API functions that perform these conversions are **getprotobyname**, **getprotobyno**, **WSAAsyncGetProtoByName** and **WSAAsyncGetProtoByNo**. The use and operation of these functions is similar to the previously discussed functions.

To Block or Not

If your application will use a local DNS, and the target host is on a local network, using blocking calls introduces far less overhead into your applications. However, if your application will connect to hosts beyond the local network, and the remote DNS is heavily used, asynchronous calls have a distinct advantage: Your application can perform other useful work while its waiting.

WSock is not a perfect Winsock component, but it provides a reasonable framework upon which to build other Internet components. Having covered the WSock component in some detail, we can now go on to build more interesting applications using our daughter components based on WSock. In the chapter that follows, we'll build an FTP client application. After that, well, I leave it to your imagination.

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Previous | Table of Contents | Next

CHAPTER 5
Shopper: An FTP Client Component

JOHN PENMAN

- The FTP Protocol
- Encapsulating The FTP Protocol As A Component
- Triggering Display Output
- The Shopper Component In Operation

Go shopping for Free Stuff on the Internet with a Winsock FTP client component.

Much of the Internet's popularity is an outgrowth of its ability to share information among computers. The protocol that makes much of this sharing possible is the File Transfer Protocol (FTP), one of the oldest protocols in use on the Internet. Today's FTP is formally defined in the Internet Request for Comment document RFC959.

As with other Internet protocols, FTP is cast in the classic client/server transaction model. I like to picture an FTP server as the old time shopkeeper that gets stuff from the shelves and hands them to the shopper, the FTP client. In this chapter, we'll implement an FTP client component for Delphi named, appropriately enough, Shopper.

Shopper depends on WSock, the simple component wrapper for Winsock API calls that we developed in Chapter 4. WSock sets up the basic functionality required by FTP to communicate using TCP/IP. With those details under control, we can begin immediately to examine the FTP process.

Are You Being Served?

By convention, an FTP server always listens for a client to set up a connection on TCP port number 21. This connection, known as the control connection, remains open until either the client or server closes its side of the connection. The client and server exchange FTP commands and reply codes, respectively, over this link. Internet protocols generally use plain English text—usually in uppercase—for their commands. This holds true even for interactions between programs.

In general, for every command that is sent by the client, the server sends a 3-digit reply code, followed by either a dash or a space, and then some text. The following two lines represent typical messages:

```
200 PORT command successful.
230-Welcome to your I-SITE Internet server!
```

The dash or space following the numeric code contains important information for the client. A dash following the code tells the client that the current message is a comment destined for human eyes, and may safely be ignored. A space after the code tells the client to proceed to the next step of the action. The text that follows is usually a status message or user instruction to the user.

Figure 5.1 shows a finite state diagram representation of the interaction between the client and server during the login sequence. The FTP session begins with the client sending the USER command, followed by the user name to the server and listening for a 3-digit reply code. If the user name is valid, the server responds with code 331 or 230. An invalid user name generates a response code of 4*xx* or 5*xx*, where “xx” is the subcode for the specific error in question

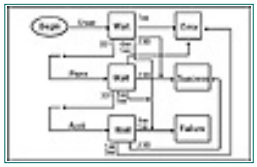

FIGURE 5.1 How the client logs in to the server

A response of 230 indicates that the user name is valid and no further information is needed to access the system. This server typically issues this code in response to the famous “anonymous” FTP login. A response of 331 indicates that the user name is valid, but a password is required. In this case, the client next sends the PASS command, followed by the password.

An incorrect password generates a 4*xx* or 5*xx* response, indicating an error. If the password is accepted by the server, the server may send a 230 code to indicate that the login sequence is complete. Alternately, if an account is required for login, the server sends another 331 code to tell the client to send the ACCT command and information.

Once it establishes a successful connection, the client can continue to issue commands. However, if there is a problem, such as a bad command syntax, or there are too many users logged on to the system, the server sends a 4*xx* or 5*xx*

code and severs the connection.

The Shopper Component

Shopper descends from the **WSock** VCL component developed in Chapter 4, and uses WSock's **TWSocket** class for handling mundane housekeeping chores such as loading of the Winsock DLL, filling the data structures for setting up a connection with a host, transfer of data, ending the connection with the server, and closing down Winsock.

The foundation WSock VCL has its **Service** property set to "NoService". As Shopper will always be an FTP client component, we set the **Service** property to "FTP" in the **TShopper.Create** constructor. The FTP protocol uses the same WSock default settings—great things, these components! As shown in Figure 5.2, Shopper has eight other properties in addition to **Service**: **AddrType**, **Protocol**, **SockType**, **CallType**, **LogOn**, **UserName**, **Password**, and **HomeServer**.

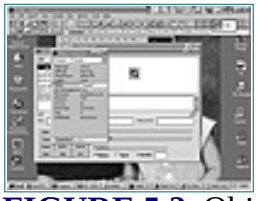

FIGURE 5.2 Object Inspector showing the properties of Shopper.

Shopper implements the more useful FTP commands, including USER, PASSWORD, RETR, and PUT, as individual properties. These FTP properties are in the public section of **TShopper**, where they are available to the component user. Each of these methods relies on the **FTPCommand** procedure, the heart of the **Shopper** component. **FTPCommand** is a simple parser implemented as a large **case** statement. What the expression lacks in elegance, however, it makes up for in simplicity.

Because the SHOPPER.PAS source file for the component in its entirety is over 1600 lines long, we won't be publishing it here within the chapter. Selected excerpts will be used to clarify certain points about its operation, and you can always print the full file from the CD-ROM for more leisurely perusal.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Displaying Output

Although Shopper is a non-visual component, it must occasionally communicate with the user's application to display messages passed between the client and the server. **TShopper**'s published property **OnChangeInfo** (descended from the **TWSocket** class in WSock) and private access procedure **Change** meet this need. The following code fragment illustrates the **Change** procedure:

```
procedure TWSocket.Change;
begin
 if Assigned(FOnChange) then
  FOnChange(Self);
end;
```

When a message is sent or received over the control connection, **TempStr** in **FTPCommand** sets the **Info** property, and then **FTPCommand** calls the **Change** procedure. Inside **Change**, **Assigned** returns **True** and the **Shopper1Change** procedure in the application displays **Info**.

To enable this communication between Shopper and the client application, I created the **Shopper1ChangeInfo** procedure by using the Events page in the Object Inspector. The **StatusMemo** window, in which these messages are displayed, gets updated whenever the **FOnChange** event occurs. **Shopper1ChangeInfo** contains the following code:

```
procedure TMainForm.Shopper1ChangeInfo(Sender: TObject);
begin
 StatusMemo.Lines.Add(Shopper1.Info);
end;
```

Putting Shopper to Work

FTPCLI, a basic FTP application created using the Shopper component, is shown in Figure 5.3. To create this application, start a new project, FTPCLI, call the main form **MainForm**, and store this in MAIN.PAS, shown in Listing 5.1. Be sure to install the WSock and Shopper components in the palette, and then pluck the Shopper component from the palette onto the main form. Add a button for each FTP command to the form. For example, the Open button calls up **Shopper1.Start** as shown here:

```
procedure TMainForm.ConnectBtnClick(Sender: TObject);
var
 Counter : Integer;
begin
 Shopper1.HostName   := HostInput.Text;{RemoteHostName;}
 { rest of code }
 Shopper1.Start;
 { more code not shown here }
end;
```

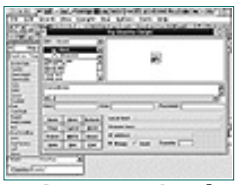

FIGURE 5.3 The finished FTP client application, FTPCLI

Listing 5.1 The FTP client demo main program unit

```
(*
  Developed for KickAss Delphi Programming
  by John Penman
  Ftp Client demo program
  Copyright (c) 1996 The Coriolis Group, Inc.
  Last Updated 7/5/96
*)


unit Main;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, About, MakeDir, StdCtrls, Wsock,
  Shopper, ExtCtrls, FileCtrl;

type
  TMainForm = class(TForm)
    ConnectBtn: TButton;
    AboutBtn: TButton;
    DisConBtn: TButton;
    ExitBtn: TButton;
    RemFilesList: TListBox;
    ViewBtn: TButton;
    MkDirBtn: TButton;
    CancelBtn: TButton;
    DelBtn: TButton;
    StatusMemo: TMemo;
    RefreshBtn: TButton;
    Progress: TEdit;
    Label1: TLabel;
    HelpBtn: TButton;
    BinaryRBtn: TRadioButton;
```

```
    AsciiRBtn: TRadioButton;
    LocalFileList: TFileListBox;
    LocalDirList: TDirectoryListBox;
    LocalDrive: TDriveComboBox;
    RemoteHostP: TPanel;
    RemoteIPP: TPanel;
    LocalHostP: TPanel;
    Shopper1: TShopper;
    AbortBtn: TButton;
    HostInput: TEdit;
    UserInput: TEdit;
    PassWordInput: TEdit;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    SiteBtn: TButton;
    procedure ConnectBtnClick(Sender: TObject);
    procedure AboutBtnClick(Sender: TObject);
    procedure DisConBtnClick(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure ViewBtnClick(Sender: TObject);
    procedure MkDirBtnClick(Sender: TObject);
    procedure Shopper1Change(Sender: TObject);
    procedure RemFilesListClick(Sender: TObject);
    procedure RemFilesListDblClick(Sender: TObject);
    procedure RefreshBtnClick(Sender: TObject);
    procedure Shopper1UpDateProgress(Sender: TObject);
    procedure Shopper1UpDateList(Sender: TObject);
    procedure HelpBtnClick(Sender: TObject);
    procedure LocalFileListChange(Sender: TObject);
    procedure LocalDriveChange(Sender: TObject);
    procedure LocalFileListDblClick(Sender: TObject);
    procedure LocalDirListChange(Sender: TObject);
    procedure Shopper1UpdateFileType(Sender: TObject);
    procedure Shopper1UpDateTimeTaken(Sender: TObject);
    procedure Shopper1UpDateBusyFlag(Sender: TObject);
    procedure DelBtnClick(Sender: TObject);
    procedure AbortBtnClick(Sender: TObject);
    procedure CancelBtnClick(Sender: TObject);
    procedure Shopper1ChangeInfo(Sender: TObject);
    procedure SiteBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }

  end;
var
  MainForm: TMainForm;

implementation

{$R *.DFM}
```

```
procedure TMainForm.ConnectBtnClick(Sender: TObject);
var
 Counter : Integer;
begin
 Shopper1.HostName   := HostInput.Text;{RemoteHostName;}
 if Shopper1.Status = Success then
 begin
  RemoteHostP.Caption :=
    Concat('Remote host : ',HostInput.Text);
  RemoteIPP.Caption   :=
    Concat('Remote IP   : ',Shopper1.HostName);
  with Shopper1 do
  begin
   UserName := UserInput.Text;
   Password := PasswordInput.Text;
  end;
  Shopper1.Start;
  if Shopper1.Status = Success then
  begin
   ConnectBtn.Enabled := FALSE;
   DisConBtn.Enabled  := TRUE;
   if Shopper1.Connected then
   begin
    RemFilesList.Clear;{951030JCP}
    {populate the remote files list box}
    for Counter := 0 to Shopper1.RemoteFiles.Count-1 do
     RemFilesList.Items.Add(Shopper1.RemoteFiles.Strings[Counter]);
    {Now enable buttons}
    AbortBtn.Enabled   := TRUE;
    SiteBtn.Enabled    := TRUE;
    ViewBtn.Enabled    := TRUE;
    HelpBtn.Enabled    := TRUE;
    DelBtn.Enabled     := TRUE;
    CancelBtn.Enabled  := TRUE;
    MkDirBtn.Enabled   := TRUE;
    RefreshBtn.Enabled := TRUE;
    ExitBtn.Enabled := FALSE;
   end;
  end;
 end;
end;

procedure TMainForm.AboutBtnClick(Sender: TObject);
begin
 AboutBox.Show;
end;

procedure TMainForm.DisConBtnClick(Sender: TObject);
begin
 try
  Shopper1.Finish;                {Quit server}
 finally
  AbortBtn.Enabled   := FALSE;
  SiteBtn.Enabled    := FALSE;
  DisConBtn.Enabled  := FALSE;
```

```
  ViewBtn.Enabled    := FALSE;
  HelpBtn.Enabled    := FALSE;
  DelBtn.Enabled     := FALSE;
  CancelBtn.Enabled  := FALSE;
  MkDirBtn.Enabled   := FALSE;
  RefreshBtn.Enabled := FALSE;
  ConnectBtn.Enabled := TRUE;
  ExitBtn.Enabled    := TRUE;
  RemFilesList.Clear;
  RemoteHostP.Caption := 'Remote host : ';
  RemoteIPP.Caption   := 'Remote IP   : ';
 end;

end;

procedure TMainForm.ExitBtnClick(Sender: TObject);
begin
 Shopper1.QuitSession;
 Close;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
 {Output name of local machine}
 LocalHostP.Caption :=
   Concat('Local host    : ',Shopper1.LocalName);
 {assign default login info from published properties}
 HostInput.Text := Shopper1.HomeServer;
 UserInput.Text := Shopper1.UserName;
 PasswordInput.Text := Shopper1.Password;
 {Disable certain controls}
 AbortBtn.Enabled   := FALSE;
 SiteBtn.Enabled    := FALSE;
 ConnectBtn.Enabled := TRUE;
 DisConBtn.Enabled  := FALSE;
 ViewBtn.Enabled    := FALSE;
 HelpBtn.Enabled    := FALSE;
 DelBtn.Enabled     := FALSE;
 CancelBtn.Enabled  := FALSE;
 MkDirBtn.Enabled   := FALSE;
 RefreshBtn.Enabled := FALSE;
 RemFilesList.Clear;
 StatusMemo.Clear;
end;

procedure TMainForm.ViewBtnClick(Sender: TObject);
begin
 Shopper1.View :=
   RemFilesList.Items.Strings[RemFilesList.ItemIndex];
end;

procedure TMainForm.MkDirBtnClick(Sender: TObject);
 var
 Counter : Integer;
```

```
begin
 MkDirDlg := TMkDirDlg.Create(Application);
 try
  if MkDirDlg.ShowModal = IDOK then;
  if NewDirName <> '' then
  begin
   Shopper1.FilesList;
   RemFilesList.Clear;
   for Counter := 0 to Shopper1.RemoteFiles.Count-1 do
    RemFilesList.Items.Add(Shopper1.RemoteFiles.Strings[Counter]);
   RemFilesList.UpDate;
  end;
  finally
   MkDirDlg.Free;
  end;
end;

procedure TMainForm.Shopper1Change(Sender: TObject);
begin
 StatusMemo.Lines.Add(Shopper1.Info);
end;

procedure TMainForm.RemFilesListClick(Sender: TObject);
begin
 if RemFilesList.ItemIndex = -1 then Exit;
  Shopper1.Selection := RemFilesList.ItemIndex;
end;

procedure TMainForm.RemFilesListDblClick(Sender: TObject);
var
 Counter : Integer;
begin
 if RemFilesList.ItemIndex = -1 then Exit;
 Shopper1.Get :=
   RemFilesList.Items.Strings[RemFilesList.ItemIndex];
 RemFilesList.Clear;
 for Counter := 0 to Shopper1.RemoteFiles.Count-1 do
   RemFilesList.Items.Add(Shopper1.RemoteFiles.Strings[Counter]);
 RemFilesList.UpDate;
 LocalFileList.UpDate;
end;

procedure TMainForm.RefreshBtnClick(Sender: TObject);
var
 Counter : Integer;
begin
 Shopper1.FilesList;
 RemFilesList.Clear;
 for Counter := 0 to Shopper1.RemoteFiles.Count-1 do
  RemFilesList.Items.Add(Shopper1.RemoteFiles.Strings[Counter]);
 RemFilesList.UpDate;
end;

procedure TMainForm.Shopper1UpDateProgress(Sender: TObject);
begin
```

```
 Progress.Text := Concat(IntToStr(Shopper1.Progress),'%');
end;

procedure TMainForm.Shopper1UpDateList(Sender: TObject);
var
 Counter : Integer;
begin
 RemFilesList.Clear;
 for Counter := 0 to Shopper1.RemoteFiles.Count-1 do
  RemFilesList.Items.Add(Shopper1.RemoteFiles.Strings[Counter]);
 RemFilesList.UpDate;
end;

procedure TMainForm.HelpBtnClick(Sender: TObject);
begin
 Shopper1.HelpFtp;
end;

procedure TMainForm.LocalFileListChange(Sender: TObject);
begin
 LocalFileList.Drive     := LocalDirList.Drive;
 LocalFileList.Directory := LocalDirList.Directory;
end;

procedure TMainForm.LocalDriveChange(Sender: TObject);
begin
 LocalDirList.Drive     := LocalDrive.Drive;
 LocalFileList.Drive    := LocalDirList.Drive;
end;

procedure TMainForm.LocalFileListDblClick(Sender: TObject);
var
 Counter : Integer;
begin
 if LocalFileList.ItemIndex = -1 then Exit;
 Shopper1.Put := LocalFileList.Items[LocalFileList.ItemIndex];
 LocalFileList.Clear;
 for Counter := 0 to Shopper1.RemoteFiles.Count-1 do
  LocalFileList.Items.Add(Shopper1.RemoteFiles.Strings[Counter]);
 LocalFileList.UpDate;
 Shopper1.FilesList;
 RemFilesList.Clear;
 for Counter := 0 to Shopper1.RemoteFiles.Count-1 do
  RemFilesList.Items.Add(Shopper1.RemoteFiles.Strings[Counter]);
 RemFilesList.UpDate;
end;

procedure TMainForm.LocalDirListChange(Sender: TObject);
begin
 LocalFileList.Directory := LocalDirList.Directory;
end;

procedure TMainForm.Shopper1UpdateFileType(Sender: TObject);
begin
 if Shopper1.FType = IMAGE then
```

```
 begin
  BinaryRBtn.Checked := TRUE;
  AsciiRBtn.Checked  := FALSE;
 end;
 if Shopper1.FType = ASCII then
 begin
  BinaryRBtn.Checked := FALSE;
  AsciiRBtn.Checked  := TRUE;
 end;
end;

procedure TMainForm.Shopper1UpDateTimeTaken(Sender: TObject);
var
 TimeStr : String;
begin
 TimeStr := 'Time taken is (secs) = ' +
            IntToStr(Shopper1.TimePassed);
 StatusMemo.Lines.Add(TimeStr);
end;

procedure TMainForm.Shopper1UpDateBusyFlag(Sender: TObject);
begin
 if Shopper1.Busy = TRUE then
 begin
  MkDirBtn.Enabled := FALSE;
  DelBtn.Enabled := FALSE;
  RefreshBtn.Enabled := FALSE;
  HelpBtn.Enabled := FALSE;
  DisConBtn.Enabled := FALSE;
  RemFilesList.Enabled := FALSE;
 end else
 begin
  MkDirBtn.Enabled := TRUE;
  DelBtn.Enabled := TRUE;
  RefreshBtn.Enabled := TRUE;
  HelpBtn.Enabled := TRUE;
  DisConBtn.Enabled := TRUE;
  RemFilesList.Enabled := TRUE;
 end;
end;

procedure TMainForm.DelBtnClick(Sender: TObject);
var
 Counter : Integer;
begin
 Shopper1.RmDirName :=
   RemFilesList.Items.Strings[RemFilesList.ItemIndex];
 Shopper1.FilesList;
 RemFilesList.Clear;
 for Counter := 0 to Shopper1.RemoteFiles.Count-1 do
  RemFilesList.Items.Add(Shopper1.RemoteFiles.Strings[Counter]);
 RemFilesList.UpDate;
end;

procedure TMainForm.AbortBtnClick(Sender: TObject);
```

```
begin
 Shopper1.Abort;
end;

procedure TMainForm.CancelBtnClick(Sender: TObject);
begin
 Shopper1.Cancel;
end;

procedure TMainForm.Shopper1ChangeInfo(Sender: TObject);
begin
 StatusMemo.Lines.Add(Shopper1.Info);
end;

procedure TMainForm.SiteBtnClick(Sender: TObject);
begin
 Shopper1.SiteFtp;
end;

end.
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Before you connect to an FTP server, FTPCLI requires that you enter the name of the FTP server, your user name, and your password into the **HostInput**, **UserInput**, and **PasswordInput** editboxes, respectively. In the case of anonymous login, enter “anonymous” in the **UserInput** editbox and your email address into the **PasswordInput** editbox. If you want to access a site that requires an account name, you’ll have to add an extra editbox and modify Shopper to send the ACCT command. To avoid having to type in the same information repeatedly, use the published properties to set the login information as shown previously in Figure 5.2.

Using the information you enter, the **Shopper1.Start** method calls **GetHost** to open the connection to the remote host. If this fails, **WSAErrorMsg** displays the possible cause of the problem, and **Status** is set to **Failure**. Otherwise, **Status** is set to **Success**. If the connection succeeds, **Start** calls **FTPCommand** to send USER, PASS, SYST, and PWD commands—in that sequence—with their appropriate arguments. **Start** then starts a data connection to transfer the listing of the remote host’s directories and files, using **GetPort** to set up the data port for the connection.

To retrieve the directory listing, **Start** sends the LIST command through **FTPCommand**. The result is stored for a later call to **Decode** to parse for directories and files. The mechanics of parsing are simple, but the mapping of the directories and files varies between systems. The parser in FTPCLI works well with servers that run on Unix and look-alikes. For other systems, the parser may occasionally return garbled directory listings.

Decode checks the first character of each line in FTPFILE.TMP for a ‘d’, which denotes a directory, or the first 2 characters ‘-’ and ‘r’ which denote a file. When a ‘d’ character is found, it is stripped, and the line is scanned for the directory name and then converted to the familiar \ddd format. The backslash tells FTPCLI that this is a directory. Likewise, in the case of a file, the ‘-’ and ‘r’ characters are removed, and the line is scanned for the name, time, date, and file size, which are chopped into substrings. These substrings are rearranged to form an acceptable string for viewing in FTPCLI’s listbox, as shown in Figure 5.4.

FIGURE 5.4 FTPCLI application with the directory listing of files from the remote FTP host.

The **FRemFiles.Add** method in Decode reads each line, formats it into a string, and adds it to the **FRemFiles** property. **FRemFiles** is a string list, derived from the **TStringList** class, created by the **TShopper.Create** constructor. FTPCLI accesses this string list through the **RemoteFiles** public property. After **Shopper1.Start** has executed successfully, FTPCLI calls the following code to add the list of directories and files to **RemFilesList** listbox:

```
for Counter := 0 to Shopper1.RemoteFiles.Count-1 do
  RemFilesList.Items.Add(Shopper1.RemoteFiles.Strings[Counter]);
```

To end our business with the FTP server, we need to kill the connection by sending a QUIT command. Clicking on the Close button calls **Shopper1.Quit** and terminates the session as shown here:

```
procedure TMainForm.DisConBtnClick(Sender: TObject);
begin
 try
  Shopper1.Finish;                {Quit server}
 finally
  AbortBtn.Enabled   := FALSE;
  SiteBtn.Enabled    := FALSE;
  DisConBtn.Enabled  := FALSE;
  ViewBtn.Enabled    := FALSE;
  HelpBtn.Enabled    := FALSE;
  DelBtn.Enabled     := FALSE;
  CancelBtn.Enabled  := FALSE;
  MkDirBtn.Enabled   := FALSE;
  RefreshBtn.Enabled := FALSE;
  ConnectBtn.Enabled := TRUE;
  ExitBtn.Enabled    := TRUE;
  RemFilesList.Clear;
  RemoteHostP.Caption := 'Remote host : ';
  RemoteIPP.Caption   := 'Remote IP   : ';
 end;
end;
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

To keep Shopper's methods direct and simple, I've chosen not to use buttons for downloading and uploading files. To download a file, for example, simply click on the desired file. The key to making this technique work is to create a new event. When you add the **RemFilesList** listbox to the main form, define a new **OnDblClick** event using the Events page in the Object Inspector. This event is handled by the **TMainForm.Rem FilesListDblClick** procedure. The code fragment shows that in response to the event, **Shopper1.Get** is assigned a file name.

```
procedure TMainForm.RemFilesListDblClick(Sender: TObject);
begin
 if RemFilesList.ItemIndex = -1 then Exit;
 Shopper1.Get :=
   RemFilesList.Items.Strings[RemFilesList.ItemIndex];
end;
```

Inside the Shopper component, the **Get** property passes the file name as **Name** to **Retrieve.** To ensure that the file is sent and stored correctly, **SetUpFileTransfer** checks the file's extension. For a binary file (such as .EXE, .DLL, and ZIP), **SetUpFileTransfer** tells **FTPCommand** to send the TYPE IMAGE command, instructing the server to send the file as a stream of contiguous bytes. For non-binary files, **SetUpFileTransfer** employs the TYPE A command. After the FTP server acknowledges the TYPE command, **SetUpFileTransfer** sends the RETR file name command through **FTPCommand**.

When you double click on a directory name such as '\DELPHI', no download occurs. Instead, **SetUpFileTransfer** calls **ChangeDir** to handle a directory change. **ChangeDir** then calls **FTPCommand**, sending the CWD command, followed by the '\DELPHI' string to the FTP server. If the server accepts the command, it returns a reply code of 250. **ChangeDir** next sends the LIST command, via **FTPCommand**, to update the remote files directory listbox. Finally, **Decode** creates a new directory listing for the changed directory.

Internally, the process of uploading a file is similar to downloading, and the **Shopper1.Put** property accomplishes this task using its **PutFile** method. On the main form, to facilitate uploading a file from the client to the server, I derived the following listboxes: **LocalDrive** from **TDriveComboBox**, **LocalDirList** from **TDirectoryListBox**, and **LocalFileList** from **TFileListBox**. Each of these listboxes are synchronized. When you select a different drive in **LocalDrive**, the contents of **LocalDirList** and **LocalFileList** are updated transparently. As with the **RemFilesList** listbox, I used the Events page in the Object Inspector to create a new OnDblClick event handler, **TMainForm.LocalFileListDblClick**, to handle a double click on a file in the **LocalFileList** listbox. A double click on the file you wish to upload tells **TMainForm.LocalFileListDblClick** to call the **Shopper1.Put** method.

As written, FTPCLI uploads or downloads only a single file at a time. However, you can remedy this shortcoming by enabling the listboxes to handle multiple selections in the Object Inspector, and modifying the **TShopper.SetUpFileTransfer** and **TShopper.PutFile** methods.

Odds and Ends

Shopper is an FTP client and a non-visual component, and has no provision for storing and retrieving host names, user names, passwords, and account information. Handling these is best left to the programmer, who can design these visible features to meet the needs of a specific application. Shopper's use of published events makes it easy to use within a wrapper application.

Shopper has one additional published property, **LogOn**, that you can use to tell Shopper to track FTP calls during a session. To use this feature, set **LogOn** to **True** and go to the Events page of the Object Inspector. Double click on the **OnUpDateProgInfo** event to derive a **Shopper1UpDateLogInfo** event handler. Then add the following code to this handler:

```
Writeln(LogFile, LogInfo);
```

The file variable, **LogFile**, is a text file which you would need to declare in MAIN.PAS. Note that using this option somewhat slows the interaction between the client and server, a result due to the increased overhead of writing every transaction to the file.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

CHAPTER 6
3D Fractal Landscapes

JON SHEMITZ

- The Nature Of Fractals
- Efficient Arrays To Store Fractal Vertices
- Fractal Randomness
- Fractal Generation
- Fractal Display

Big Sur too expensive for you? Create your own Virtual Sur, and make it as big as you like, using fractal techniques and that old Delphi magic.

The first time I heard the word “fractal” was a little over thirteen years ago, when I was still a mainframe programmer. A coworker and I were talking about our spiffy new IBM PC’s, and he asked me if I had any fractal software.

“No,” I said. “what are fractals?”

He explained that you made a fractal by applying some geometric operation to a simple figure, and then repeatedly applying the same operation to the result. While this explanation doesn’t even begin to touch on the sort of mathematical properties of fractals that interest mathematicians, it remains a good summary of how most fractals are generated.

It certainly describes 3D fractal landscape generation.

Bending and Dividing

To generate a landscape, simply assign random heights to the three vertices of an equilateral triangle, then ‘bend’ each of the edges by raising or lowering the midpoint by some random amount. If you draw lines between each of the three midpoints, you have carved the initial triangle into four triangles. If you apply this bending and dividing operation to each new triangle as you create it, before long you will end up with something that looks surprisingly like a real landscape. (See Figures 6.1, 6.2, and 6.3.)

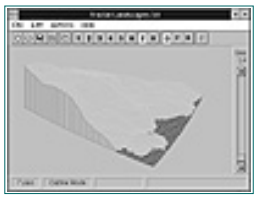

FIGURE 6.1 An outline fractal landscape.

FIGURE 6.2 A filled fractal landscape.

FIGURE 6.3 A rendered fractal landscape.

The Shared Edges Problem

Now, of course, fractal landscape generation’s not *quite* as simple as “bend, divide, repeat”—you have to make sure that you only bend a given line once, and of course, you do generally want to display the landscape—but those are the details.

The first and most important detail is that you have to keep track of what you’re doing. If you straightforwardly write the **FractureTriangle()** routine to bend each edge, you will end up with something like that shown in Figure 6.4. Your triangles won’t form a smooth mesh; each group of four triangles will “float” with each vertex at a different height than its neighbors’.

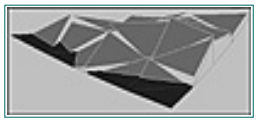

FIGURE 6.4 When edges don’t meet.

A look at Figure 6.5 may help make it clear what’s going on. Interior edges are shared by two triangles, which do not necessarily agree on where to set shared edges. Vertex I is the midpoint of the line DF, which is shared by triangles

CDF and DEF. If both triangles try to set the height of I, triangles 1, 2, and 3 will be given a different height for I than triangles 4, 5, and 6!

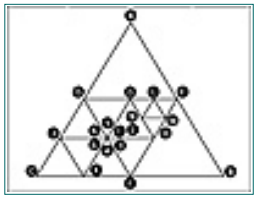

FIGURE 6.5 How triangles “fight” over edges.

Clearly, we need to maintain a database of vertices, so that we can set vertex I when we fracture triangle CDF—and use the same value for vertex I when we fracture triangle DEF. We might try to say that triangle DEF is an “inside” triangle, and that therefore we will fracture it last and use the “outer” triangles’ values for vertices G, H, and I—but look at triangles GEH and LMN. Lines GE and EH are shared with “outside” triangles, so we should use the outside triangles’ values for vertices L and M, but line GH is “all inside”, so we need to bend it. We could undoubtedly elaborate the scheme of inside and outside triangles to handle these “outside-inside subtriangles” properly, but we’d end up with code that was hard to read and liable to break whenever it was changed.

It’s far simpler to define a special value that only uninitialized vertices will have and make **FractureTriangle()** look to see if any given midpoint has already been set. If a midpoint has been set, **FractureTriangle()** uses that height; if no midpoint has been set, **FractureTriangle()** gives it a new height. It may be a *bit* slower to calculate and lookup the midpoints of inside triangles than to simply pass in the right arguments, but the code is smaller and clearer—and displaying a landscape certainly takes *much* longer than generating it.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

A Triangular Array

Now, when we bend a line, we are only changing the midpoint's *z* value, so in principle, we could somehow use the [*x*, *y*] coordinate pair as an index into a table of *z* values. However, this would turn out to be a *very* sparse array, and the code will run a lot faster if we can use a normal, contiguous array than if we have to keep walking a sparse array's lists. This is why Listing 6.1 defines a system of 2D logical addresses—the **TVertex** data type—which "name" the actual 3D coordinates in the **TTriple** data type.

Listing 6.1 GLOBAL.PAS

```
unit Global; {Fractal Landscapes 3.0 -- Copyright _ 1987..1996, Jon
  Shemitz}

interface

uses WinTypes;

type
  Int16 = {$ifdef Ver80} integer {$else} SmallInt {$endif} ;

const
  MaxPlys             = 8;
  MaxEdgeLength       = 1 shl (MaxPlys - 1);
  UnitLength: LongInt = 5000;
  ShadesOfGray        = 64;

type
  TCoordinate = -30000..30000;
  TTriple = record
            X,              { Width:  0 (left) to UnitLength (right)      }
            Y,              { Depth:  0 (front) to VanishingPoint.Y (back)}
            Z: TCoordinate; { Height: 0 (bottom) to UnitLength (top)      }
            end;

function Triple(X, Y, Z: TCoordinate): TTriple;

type
  TPixel = TPoint;

type
  GridCoordinate = 0..MaxEdgeLength; { Triangular grid space }
```

```
  TVertex = record
            AB, BC, CA: GridCoordinate;
            end;

function Vertex(AB, BC, CA: GridCoordinate): TVertex;

type
  DrawModes  = (dmOutline, dmFill, dmRender);
  DrawRates  = (drLow, drMedium, drHigh);

const
  Envelope = 3000;
  SeaLevel: word = 100;     { 0 (bottom) to UnitLength (top) }
  VanishingPoint: TTriple = ( X:  1500 ;
                              Y: 25000 ; { Apparent depth of vanishing
                              point }
                              Z: 15000 );
  LightSource: TTriple = ( X:  2500;
                           Y: +7500;
                           Z:  25000 );
  DrawMode: DrawModes   = dmOutline;
  DrawRate: DrawRates   = drHigh;

const
  Uninitialized = -30000;

var
  A, B, C: TVertex;
  Plys:       1..MaxPlys;
  EdgeLength: Int16;

  DisplayHeight,
  DisplayWidth: Int16;

implementation

function Triple(X, Y, Z: TCoordinate): TTriple;
begin
  Result.X := X;
  Result.Y := Y;
  Result.Z := Z;
end;

function Vertex(AB, BC, CA: GridCoordinate): TVertex;
begin
  Result.AB := AB;
  Result.BC := BC;
  Result.CA := CA;
end;

end.
```

Perhaps the simplest naming scheme is to simply number each vertex along the outermost triangle's three edges, as in the left half of Figure 6.6, and use all three coordinates to refer to each edge. While we really only need two coordinates and the third is redundant, I find it a bit clearer to be able to refer to the three outermost vertices as [1, 0, 0], [0, 1, 0], and [0, 0, 1] rather than as [1, 0], [0, 1], and [0, 0]. That's why a **TVertex** is defined as a coordinate triple, even though the third coordinate is not really necessary and does slow things down somewhat.

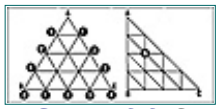

FIGURE 6.6 Storing vertices in a “square” array.

When it comes to the database of vertices, though, the third coordinate *is* ignored. As you can see from the right half of Figure 6.6, each vertex has the same coordinates if we make the equilateral triangle into an isosceles triangle. This lets us use the AB and BC coordinates much as if they were the normal row and column coordinates of a square array.

However, if we *did* store our triangular array in a square array, we would be wasting nearly half the space. This wouldn’t be such a big deal, except that in 16-bit segmented environments we run smack into the 64K segment size barrier. Each **TTriple** consists of three 16-bit fixed-point numbers, so a square array for an eight-ply database (Figure 6.7) would take $(2^{8-1}+1)^2$ vertices, or 99,846 bytes. If we store only the cells on and below the diagonal, we can pare this to 50,310 bytes. This lets us use simple array indexing instead of arrays of arrays or huge pointers. Similarly, at least in this demo program, the database will fit in the data segment, which does make for faster access than if we had to use heap blocks and pointers.

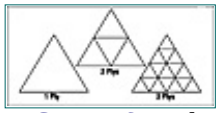

FIGURE 6.7 Plies versus number of triangles.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Since eight plies is hardly too fine a resolution for, say, a 1280 by 1024 screen, the demo program for this chapter, Fractal Landscapes 3.0, (a.k.a. FL3, a Delphi port of a Windows port of a DOS program, which was originally written as a “cute hack” in Turbo Pascal 4.0) uses the triangular database code in Listing 6.2. The basic idea is that each row of vertices follows right after the previous row. Since there is only one vertex on the first row, the second row starts in **DB**’s second “cell.” Since the second row has two vertices, the third row starts in **DB**’s fourth cell, and so on.

Listing 6.2 DATABASE.PAS

```
unit Database; {Fractal Landscapes 3.0 -- Copyright _ 1987..1996, Jon
  Shemitz}

{Database and landscape generation}

interface

uses SysUtils, Global;

{ Misc math fns and such }
function IDIV(Numerator: LongInt; Denominator: Int16): Int16;
{$ifdef Ver80} {Delphi 1.0 still supported InLine fns}
InLine(
         $5B /                  { POP   BX ; Denominator    }
         $58 /                  { POP   AX ; Lo word of Num }
         $5A /                  { POP   DX ; Hi word of Num }
         $F7 / $FB              { IDIV  BX ; Mixed DIV      }
      );
{$endif}

function IMUL(A, B: Int16): LongInt;
{$ifdef Ver80} {Delphi 1.0 still supported InLine fns}
InLine(
         $5B /                  { POP   BX }
         $58 /                  { POP   AX }
         $F7 / $EB              { IMUL  BX }
      );
{$endif}

function Rand(Envelope: integer): integer;

{database}
```

```
procedure ResetDB;

function GetTriple(const V: TVertex): TTriple; { DB[V] }

procedure SwapTriples(var A, B: TTriple);

function Midpoint(A, B: TVertex): TVertex;

function LoadLandscape(const FileName: TFileName): boolean;

function SaveLandscape(const FileName: TFileName): boolean;

{calculation}

procedure FractureTriangle(const A, B, C: TVertex; Plys: word);

function Unscale(ScaledCoordinate: LongInt): TCoordinate;
{$ifdef Ver80} {Delphi 1.0 still supported InLine fns}
InLine(
        $58 /                    { POP  AX              ; Lo word of SC   }
        $5A /                    { POP  DX              ; Hi word of SC   }
        $8B / $1E / UnitLength / { MOV  BX,[UnitLength] ; Lo word of scale}
        $F7 / $FB                { IDIV BX              ; Unscale         }
      );
{$endif}

implementation

{ Misc math fns and such }

{$ifNdef Ver80} {Delphi 2.0 doesn't support InLine fns}
function IDIV(Numerator: LongInt; Denominator: Int16): Int16;
begin
  Result := Numerator div Denominator;
end;
{$endif}

{$ifNdef Ver80} {Delphi 2.0 doesn't support InLine fns}
function IMUL(A, B: Int16): LongInt;
begin
  Result := Longint(A) * B;
end;
{$endif}

function Rand(Envelope: integer): integer;
{ Pseudonormal distribution, in range _Envelope }
begin
  Rand := integer(Random(Envelope)) + integer(Random(Envelope)) - Envelope;
end;

{$ifNdef Ver80} {Delphi 1.0 doesn't support InLine fns}
function Unscale(ScaledCoordinate: LongInt): TCoordinate;
begin
  Result := ScaledCoordinate div UnitLength;
end;
{$endif}

{ Database }
```

```
var
  DB: array[0..8384] of TTriple; { Triangular array: Vertices(MEL+1)
  entries }

  NumberOfVertices, TopRow: word;

  Envelopes:  array[1..MaxPlys] of word;

function Vertices(N: word): word;
{ Vertices in an equilateral triangle with edgelength = N-1 }
begin
  Vertices := (Sqr(N) + N) shr 1;
end;

function Midpoint(A, B: TVertex): TVertex;
begin
  Result := Vertex( (A.AB + B.AB) shr 1, { Average }
                    (A.BC + B.BC) shr 1,
                    (A.CA + B.CA) shr 1 );
end;

function Loc(const V: TVertex): word;
begin
  Loc := NumberOfVertices - Vertices(TopRow - V.AB) + V.BC;
  {      ^^^^^^^^^^^^^^^^^^ This is actually NOT necessary and just
                            wastes cycles, but I have retained it
                            for compatability with FL2 .FL files. }

end;

procedure SetTriple(var V: TVertex; var T: TTriple);     { DB[V] := T }
begin
  DB[Loc(V)] := T;
end;

function GetTriple(const V: TVertex): TTriple; { DB[V] }
begin
  Result := DB[Loc(V)];
end;

procedure SwapTriples(var A, B: TTriple);
var
  Tmp: TTriple;
begin
  Tmp := A; A := B; B := Tmp;
end;

procedure SwapZ(var A, B: TTriple);
var
  C: TCoordinate;
begin
  C := A.Z; A.Z := B.Z; B.Z := C;
end;

const
  Uninitialized = -30000;
procedure ResetDB;
var
  T:          TTriple;
  R, Theta:   double;
```

```
  I, Offset:  integer;
  tA, tB, tC: TTriple;
const
  Base_Rotation = - Pi / 2.1; {Rotate point counterclockwise a bit}
  RotateBy      = Pi * 2 / 3; {120|}
begin
  { Set Plys dependent stuff }
  EdgeLength := 1 shl (Plys - 1);
  TopRow := EdgeLength + 1; { A "fencepost" situation }
  NumberOfVertices := Vertices(TopRow);
  for I := Plys downto 1 do
    Envelopes[I] := Envelope shr Succ(Plys - I);
  { Then reset NumberOfVertices vertices in DB }
  T.X := Uninitialized;
  T.Y := Uninitialized;
  T.Z := Uninitialized;
  for I := Low(DB) to High(DB) do DB[I] := T;
  { Now, set "defining" (outside) points A, B, and C }
  A.AB := 0;          A.BC := EdgeLength; A.CA := 0;
  B.AB := 0;          B.BC := 0;          B.CA := EdgeLength;
  C.AB := EdgeLength; C.BC := 0;          C.CA := 0;
  { Then, assign them triples }
  Offset := UnitLength div 2;
  R      := UnitLength / 2;

  Theta := Base_Rotation;
  tA := Triple( Round(R * Cos(Theta)) + Offset,
                Round(R * Sin(Theta)) + Offset,
                SeaLevel + Rand(Envelope) );

  Theta := Theta + RotateBy;
  tB := Triple( Round(R * Cos(Theta)) + Offset,
                Round(R * Sin(Theta)) + Offset,
                SeaLevel + Rand(Envelope) );

  Theta := Theta + RotateBy;
  tC := Triple( Round(R * Cos(Theta)) + Offset,
                Round(R * Sin(Theta)) + Offset,
                SeaLevel + Rand(Envelope) );

  { At least one point above sealevel }
  if (tA.Z < SeaLevel) AND (tB.Z < SeaLevel) AND (tC.Z < SeaLevel) then
    repeat
      tB.Z := SeaLevel + Rand(Envelope);
    until tB.Z > SeaLevel;
  { Force A the lowest ... }
  if tA.Z > tB.Z then SwapZ(tA, tB);
  if tA.Z > tC.Z then SwapZ(tA, tC);

  SetTriple(A, tA);
  SetTriple(B, tB);
  SetTriple(C, tC);
end;

function SaveLandscape(const FileName: TFileName): boolean;
var
  Handle: integer;
begin
  Result := False;
  try
```

```
    Handle := FileCreate(FileName);
    try
      Result := (FileWrite(Handle, Plys, SizeOf(Plys)) = SizeOf(Plys))
                  and
                (FileWrite(Handle, DB, NumberOfVertices *
                  SizeOf(TTriple))
                 = NumberOfVertices * SizeOf(TTriple));
    finally
      FileClose(Handle);
    end;
  except
    on {any} Exception do Result := False;
  end;
end;

function LoadLandscape(const FileName: TFileName): boolean;
var
  Handle:  integer;
begin
  Result := False;
  try
    Handle := SysUtils.FileOpen(FileName, fmOpenRead);
    try
      if FileRead(Handle, Plys, SizeOf(Plys)) = SizeOf(Plys) then
        begin
        ResetDB;
        LoadLandscape := FileRead( Handle, DB,
                                   NumberOfVertices * SizeOf(TTriple))
                         = NumberOfVertices * SizeOf(TTriple);
        end;
    finally
      FileClose(Handle);
    end;
  except
    on {any} Exception do Result := False;
  end;
end;

{ Action }

procedure FractureLine( var vM: TVertex;
                        const vA, vB: TVertex;
                        Envelope: integer );
var
  A, B, M: TTriple;
begin
  vM := Midpoint(vA, vB);
  M  := GetTriple(vM);
  if M.X = Uninitialized then { Not set yet }
    begin
    A := GetTriple(vA); B := GetTriple(vB);
    M := Triple( A.X + (B.X - A.X) div 2,
                 A.Y + (B.Y - A.Y) div 2,
                 A.Z + (B.Z - A.Z) div 2 + Rand(Envelope) );
    { Mean height _ Random(Envelope) }
    SetTriple(vM, M);
    end;
end;

procedure FractureTriangle(const A, B, C: TVertex; Plys: word);
```

```
var
  Envelope: word;
  AB, BC, CA: TVertex;
begin
  if Plys > 1 then
    begin
    Envelope := Envelopes[Plys];
    FractureLine(AB, A, B, Envelope);
    FractureLine(BC, B, C, Envelope);
    FractureLine(CA, C, A, Envelope);
    Dec(Plys);
    FractureTriangle(CA, BC, C, Plys);
    FractureTriangle(AB, B, BC, Plys);
    FractureTriangle(BC, CA, AB, Plys);
    FractureTriangle(A, AB, CA, Plys);
    end;
end;

end.
```

Previous | Table of Contents | Next

Products | **Contact Us** | **About Us** | **Privacy** | **Ad Info** | **Home**

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Bending

Another subtlety that I didn't discover until I actually wrote the code is that you shouldn't apply the same amount of randomness when you bend the larger scale lines as when you bend the smaller scale lines. If you do, you either end up with a bumpy plane or a spiky landscape. You need to apply more randomness to the large outer triangles, which produce the overall shape of the landscape, and to apply less randomness to the smaller inner triangles, which basically control the smoothness of your landscape.

What I ended up using is a function that generates something vaguely like a normal distribution:

```
function Rand(Envelope: integer): integer;
{ Pseudonormal (sawtooth) distribution,
  in range ±Envelope }
begin
  Rand := integer(Random(Envelope)) +
          integer(Random(Envelope)) -
          Envelope;
end;
```

Here, the **Envelope** value for each ply is half that of the next larger ply. This certainly produces plausible-looking landscapes, but real landscapes aren't always as smooth as FL3's. Real landscapes do have the occasional sharp edge—cliffs, mesas, canyons, and so on—while FL3 never really produces anything more abrupt than a steep slope.

One approach you may want to experiment with is to replace **Rand**'s pseudonormal distribution within a constricting envelope with an exponential function. On smaller scales, the function would be more *likely* to produce a number close to 0 than on larger scales, but it might throw out a large number on any scale.

Draw, Then Display

In the first incarnation of this program, the same recursive routine that built the landscape was responsible for drawing it. If the **Plys** argument was greater than 1, it broke its input triangle into four new triangles, and then decremented **Plys** and applied itself to each new triangle. When the **Plys** argument was equal to 1, it called a routine that drew the triangle.

This was certainly simple enough, but it meant that changing from a "wire mesh" rendering to a filled-triangle rendering required generating a whole new landscape. Similarly, using this simple design in a Windows version would mean that changing the window size also generates a whole new landscape. Clearly, a better approach is to generate the landscape first and then draw it. This requires two parallel recursions from the outermost triangle to the innermost ones (which are the only ones which are actually drawn), but the second recursion doesn't cost much compared to actually drawing the rectangles, so the price of flexibility is fairly low.

Generating and Displaying the Landscape

After all that prolog, the actual generation code may seem refreshingly simple. **FractureTriangle()** (present in Listing 6.2) takes a triangle and the number of **Plys** remaining. If **Plys** is greater than 1, **FractureTriangle()** calls **FractureLine()** to create (or retrieve) a midpoint value, then calls itself on each of the four triangles that these midpoints define. **FractureLine()** calls **Midpoint()** (both in Listing 6.2) to calculate the vertex between its two input vertices, and then checks to see if it has been set yet. If the midpoint is still uninitialized, **FractureLine()** bends the line between the endpoints by raising or lowering its midpoint.

Once the landscape has been generated, FL3 uses the code in Listing 6.3 to display it in the current window, in the current display mode. If the user changes the window size or the display mode, FL3 redraws the landscape.

Listing 6.3 DISPLAY.PAS

```
unit Display; {Fractal Landscapes 3.0 -- Copyright _ 1987..1996, Jon
  Shemitz}

interface
uses WinTypes, WinProcs,
     SysUtils, Graphics, Forms,
     Global, Database;

const
  DrawingNow: boolean = False;
  AbortDraw:  boolean = False;

type
  EAbortedDrawing = class (Exception) end;

procedure ScreenColors;

procedure PrinterColors;

procedure DrawTriangle(       Canvas:    TCanvas;
                        const A, B, C:   TVertex;
                              Plys:      word;
                              PointDn:   boolean);

procedure DrawVerticals(Canvas: TCanvas);

{$ifdef Debug}
const DebugString: string = '';
{$endif}
implementation

uses Main;

type
  Surfaces = record
               Outline, Fill: TColor;
               end;

const
  scrnLand:     Surfaces = (Outline: clLime;  Fill: clGreen);
  scrnWater:    Surfaces = (Outline: clBlue;  Fill: clNavy);
  scrnVertical: Surfaces = (Outline: clGray;  Fill: clSilver);

  prnLand:     Surfaces = (Outline: clBlack; Fill: clWhite);
  prnWater:    Surfaces = (Outline: clBlack; Fill: clWhite);
  prnVertical: Surfaces = (Outline: clBlack; Fill: clWhite);
```

```
var
  Land, Water, Vertical: Surfaces;
procedure ScreenColors;
begin
  Land     := scrnLand;
  Water    := scrnWater;
  Vertical := scrnVertical;
end;

procedure PrinterColors;
begin
  Land     := prnLand;
  Water    := prnWater;
  Vertical := prnVertical;
end;

function Surface(Outline, Fill: TColor): Surfaces;
begin
  Result.Outline := Outline;
  Result.Fill    := Fill;
end;

{ $define Pascal} {$define Float}
{$ifdef Pascal}
  {$ifdef Float}
  type
    TFloatTriple = record X, Y, Z: double; end;

  function FloatTriple(T: TTriple): TFloatTriple;
  begin
    Result.X := T.X / UnitLength;
    Result.Y := T.Y / UnitLength;
    Result.Z := T.Z / UnitLength;
  end;

  function Project(const P: TTriple): TPixel; { 3D transform a point }
  var
    Delta_Y: double;
    Tr, V:    TFloatTriple;
  begin
    Tr := FloatTriple(P);
    V  := FloatTriple(VanishingPoint);

    Delta_Y  := Tr.Y / V.Y;
    Result.X := Round( DisplayWidth *
                       ((V.X - Tr.X) * Delta_Y + Tr.X));
    Result.Y := DisplayHeight -
                Round( DisplayHeight *
                       ((V.Z - Tr.Z) * Delta_Y + Tr.Z));
  end;
  {$else}
  function Project(const Tr: TTriple): TPixel; { 3D transform a point }
  var
    Delta_Y: integer;
  begin
    Delta_Y := MulDiv(Tr.Y, UnitLength, VanishingPoint.Y);
    Result.X := MulDiv( MulDiv( VanishingPoint.X - Tr.X,
                                Delta_Y, UnitLength) + Tr.X,
                                DisplayWidth, UnitLength);
```

```
    Result.Y := DisplayHeight -
                MulDiv( MulDiv( VanishingPoint.Z - Tr.Z,
                                Delta_Y, UnitLength) + Tr.Z,
                                DisplayHeight, UnitLength );
  end;
  {$endif}
{$else}
function Project(const Tr: TTriple): TPixel; assembler; {3D transform a
                                                          point}
asm
{$ifdef Ver80} {Delphi 1.0; 16-bit}
        les     di,[Tr]
        mov     si,word ptr UnitLength  { Scaling factor }

        mov     ax,[TTriple ptr es:di].Y{ Tr.Y }
        imul    si                      { Scale by LoWord(UnitLength) }
        idiv    VanishingPoint.Y        { Scaled(depth/vanishing.depth) }
{DeltaY         equ     bx }
        mov     bx,ax                   { preserve Delta.Y }

        mov     ax,VanishingPoint.Z
        sub     ax,[TTriple ptr es:di].Z{ Delta.Z }
        imul    bx                      { Delta.Z * Delta.Y }
        idiv    si                      { Unscale(Delta.Z * Delta.Y) }
        add     ax,[TTriple ptr es:di].Z{ Tr.Z + Unscale(Delta.Z * Delta.Y)
        }
        mov     cx,[DisplayHeight]      { We'll use it twice here ... }
        imul    cx                  { (Tr.Z+Delta.Z*Delta.Y)*Screen.Row }
        idiv    si                      { Unscale }
        sub     cx,ax                   { Px.Y }

        mov     ax,VanishingPoint.X
        sub     ax,[TTriple ptr es:di].X{ Delta.X }
        imul    bx                      { Delta.X * Delta.Y }
        idiv    si                      { Unscale(Delta.X * Delta.Y) }
        add     ax,[TTriple ptr es:di].X{ Tr.X + Unscale(Delta.X * Delta.Y)
        }
        imul    [DisplayWidth]          { (Tr.X+Delta.X*Delta.Y)*Screen.Col
        }
        idiv    si                      { Px.X := Unscale(above) }

        mov     dx,cx                    {Return (X,Y) in ax:dx}
{$else} {Delphi 2.0 or better; 32-bit}
        push    ebx                     { Delphi 2.0 requires that we }
        push    esi                     { preserve these registers    }
        push    edi
        mov     edi,eax                 { lea edi,[Tr]}
        push    edx                     { Save @ Result }
        mov     si,word ptr UnitLength  { Scaling factor }

        mov     ax,TTriple[edi].Y       { Tr.Y }
        imul    si                      { Scale by LoWord(UnitLength) }
        idiv    VanishingPoint.Y        { Scaled(depth/vanishing.depth) }
{DeltaY equ  bx }
        mov     bx,ax                   { preserve Delta.Y }

        mov     ax,VanishingPoint.Z
        sub     ax,TTriple[edi].Z       { Delta.Z }
        imul    bx                      { Delta.Z * Delta.Y }
        idiv    si                      { Unscale(Delta.Z * Delta.Y) }
```

```
        add     ax,TTriple[edi].Z       { Tr.Z + Unscale(Delta.Z *
                                            Delta.Y) }
        mov     cx,[DisplayHeight]      { We'll use it twice here ... }
        imul    cx                      {
                                        (Tr.Z+Delta.Z*Delta.Y)*Screen.Row }
        idiv    si                      { Unscale }
        sub     cx,ax                   { Px.Y }

        mov     ax,VanishingPoint.X
        sub     ax,TTriple[edi].X       { Delta.X }
        imul    bx                      { Delta.X * Delta.Y }
        idiv    si                      { Unscale(Delta.X * Delta.Y) }
        add     ax,TTriple[edi].X       { Tr.X + Unscale(Delta.X *
                                          Delta.Y) }
        imul    [DisplayWidth]          {
                                        (Tr.X+Delta.X*Delta.Y)*Screen.Col }
        idiv    si                      { Px.X := Unscale(above) }
// Now ax=x, cx=y; we want to make them longints and save them to Result
        mov     ebx,$0000FFFF
        and     eax,ebx                 {clear the high word}
        and     ecx,ebx
        pop     edx                     { restore @ Result }
        mov     TPixel[edx].X,eax
        mov     TPixel[edx].Y,ecx
        pop     edi
        pop     esi
        pop     ebx
{$endif}
end;
{$endif}

procedure DrawPixels(const Canvas:              TCanvas;
                     const A, B, C, D:          TPixel;
                     const N:                   word;
                     const Surface:             Surfaces);
begin
  if AbortDraw then raise EAbortedDrawing.Create('');

  Canvas.Pen.Color := Surface.Outline;
  if DrawMode = dmOutline
    then if N = 3
      then Canvas.PolyLine( [A, B, C, A] )
      else Canvas.PolyLine( [A, B, C, D, A] )
    else begin
         Canvas.Brush.Color := Surface.Fill;
         if N = 3
           then Canvas.Polygon( [A, B, C] )
           else Canvas.Polygon( [A, B, C, D] )
         end;
end;

procedure CalcCrossing(var Low, High, Crossing: TTriple;
  SetLow: boolean);
var
  CrossOverRatio: LongInt;
begin
CrossOverRatio := (SeaLevel - Low.Z) *
  UnitLength div (High.Z - Low.Z);
{ Distance of crossing point from A, as ratio of total line AB length, }
   { times UnitLength                                                    }
```

```
   Crossing := Triple( Low.X + Unscale((High.X - Low.X) *
                         CrossOverRatio),
                       Low.Y + Unscale((High.Y - Low.Y) *
                         CrossOverRatio),
                       SeaLevel );
   if SetLow then Low.Z := SeaLevel;
end;

procedure DrawVertical(Canvas: TCanvas; const A, B: TTriple;
                                        var pA, pB: TPixel);
var
  pC, pD: TPixel;
  tC, tD: TTriple;
begin
  tC   := A;
  tC.Z := SeaLevel;
  pC   := Project(tC);

  tD   := B;
  tD.Z := SeaLevel;
  pD   := Project(tD);

  DrawPixels(Canvas, pA, pB, pD, pC, 4, Vertical);
end;

procedure DrawVerticals(Canvas: TCanvas);
type
  Triad = record
          T: TTriple;
          V: TVertex;
          P: TPixel;
          end;
var
  Work: Triad;

  procedure Step( const Start: TVertex;
                  var Front:   Triad;
                  var StepDn:  GridCoordinate
                );
  var
    Idx: word;
    Back, Interpolate: Triad;
  begin
    Back.V := Start;
    Back.T := GetTriple(Back.V);
    if Back.T.Z > SeaLevel then Back.P := Project(Back.T);
    for Idx := 1 to EdgeLength do
      begin
      Front.V := Back.V;
      Inc(Work.V.BC);
      Dec(StepDn);
      Front.T := GetTriple(Front.V);
      if Front.T.Z > SeaLevel then Front.P := Project(Front.T);
      case (ord(Back.T.Z > SeaLevel) shl 1) + ord(Front.T.Z > SeaLevel)
        of
        1: begin { Back below, front above }
           CalcCrossing(Back.T, Front.T, Interpolate.T, False);
           Interpolate.P := Project(Interpolate.T);
           DrawVertical(Canvas, Interpolate.T, Front.T, Interpolate.P,
             Front.P);
```

```
            end;
          2: begin { Back above, front below }
             CalcCrossing(Front.T, Back.T, Interpolate.T, False);
             Interpolate.P := Project(Interpolate.T);
             DrawVertical(Canvas, Back.T, Interpolate.T, Back.P,
               Interpolate.P);
             end;
          3: DrawVertical(Canvas, Back.T, Front.T, Back.P, Front.P);
                  { Both above }
          end;
        Back := Front;
        end;
    end;

begin
  Step(C, Work, Work.V.AB );
  Step(B, Work, Work.V.CA );
end;

  function InnerProduct({const} A, B: TTriple): LongInt;
  begin
    InnerProduct := IMUL(A.X, B.X) + IMUL(A.Y, B.Y) + IMUL(A.Z, B.Z) ;
    { Damn but this >should< be UnScaled ...                          }
  end;

  function Delta(A, B: TTriple): TTriple;
  begin
    Result := Triple(A.X - B.X, A.Y - B.Y, A.Z - B.Z);
  end;

function LandColor(const A, B, C: TTriple): TColor;
var
  Center, ToA, ToLight: TTriple;
  Cos, Angle:           double;
  GrayLevel:            integer;
begin
  Center := Triple( (A.X + B.X + C.X) div 3,
                    (A.Y + B.Y + C.Y) div 3,
                    (A.Z + B.Z + C.Z) div 3 );
  ToA := Delta(A, Center);
  ToLight := Delta(Center, LightSource);

  {$ifopt R-} {$define ResetR} {$endif}
  {$R+}
    try
      Cos := InnerProduct(ToA, ToLight) /
             (Sqrt({Abs(}InnerProduct(ToA, ToA){)}) *
              Sqrt({Abs(}InnerProduct(ToLight, ToLight){)}) );
      try
        Angle := ArcTan (Sqrt (1 - Sqr (Cos)) / Cos);
      except
        on Exception do Angle := Pi / 2; {ArcCos(0)}
      end;
      {$ifdef HighContrast}
      GrayLevel := 255 - Round(255 * (Abs(Angle) / (Pi / 2)));
      {$else}
      GrayLevel := 235 - Round(180 * (Abs(Angle) / (Pi / 2)));
      {$endif}
    except
      on {any} Exception do GrayLevel := 255; {division by 0 ...}
```

```
    end;
  {$ifdef ResetR} {$R-} {$undef ResetR} {$endif}

  Result := PaletteRGB(GrayLevel, GrayLevel, GrayLevel);
end;

procedure Draw3Vertices( Canvas: TCanvas;
                         const A, B, C: TVertex; Display: boolean);
var
  Color: TColor;
  pA, pB, pC, pD, pE: TPixel;
  tA, tB, tC, tD, tE: TTriple;
  aBelow, bBelow, cBelow: boolean;
begin
  tA := GetTriple(A); tB := GetTriple(B); tC := GetTriple(C);
  {$ifdef FloatingTriangles}
  ta.z := ta.z + random(Envelope shr Plys) - random(Envelope shr Plys);
  tb.z := tb.z + random(Envelope shr Plys) - random(Envelope shr Plys);
  tc.z := tc.z + random(Envelope shr Plys) - random(Envelope shr Plys);
  {$endif}
  aBelow := tA.Z <= SeaLevel;
  bBelow := tB.Z <= SeaLevel;
  cBelow := tC.Z <= SeaLevel;
  case ord(aBelow) + ord(bBelow) + ord(cBelow) of
    0:       if Display then {All above}
               begin
               pA := Project(tA); pB := Project(tB); pC := Project(tC);
               if DrawMode = dmRender
                 then begin
                      Color := LandColor(tA, tB, tC);
                      DrawPixels( Canvas,
                                  pA, pB, pC, pC, 3, Surface(Color,
                                    Color));
                      end
                 else DrawPixels( Canvas,
                                  pA, pB, pC, pC, 3, Land);
               end;
    3:       if Display then {All below}
               begin
               tA.Z := SeaLevel; tB.Z := SeaLevel; tC.Z := SeaLevel;
               pA := Project(tA); pB := Project(tB); pC := Project(tC);
               DrawPixels( Canvas, pA, pB, pC, pC, 3, Water);
               end;
    2:       begin {One vertex above water}
             { First ensure it's tA }
             if aBelow then
               if bBelow
                 then SwapTriples(tA, tC)
                 else SwapTriples(tA, tB);
             CalcCrossing(tB, tA, tD, True);
             CalcCrossing(tC, tA, tE, True);
             pA := Project(tA); pB := Project(tB); pC := Project(tC);
             pD := Project(tD); pE := Project(tE);
             DrawPixels( Canvas, pD, pB, pC, pE, 4, Water);
             if Drawmode = dmRender
               then begin
                    Color := LandColor(tD, tA, tE);
                    DrawPixels( Canvas, pD, pA, pE, pE, 3,
                                Surface(Color, Color));
                    end
```

```
              else DrawPixels( Canvas, pD, pA, pE, pE, 3, Land);
            end;
    1:      begin {One vertex below water}
            { First ensure it's tA }
            if bBelow
              then SwapTriples(tA, tB)
              else if cBelow then SwapTriples(tA, tC);
            CalcCrossing(tA, tB, tD, False);
            CalcCrossing(tA, tC, tE, True);
            pA := Project(tA); pB := Project(tB); pC := Project(tC);
            pD := Project(tD); pE := Project(tE);
            DrawPixels( Canvas, pD, pA, pE, pE, 3, Water);
            if DrawMode = dmRender
              then begin
                   Color := LandColor(tD, tB, tC);
                   DrawPixels( Canvas,
                               pD, pB, pC, pE, 4, Surface(Color, Color));
                   end
              else DrawPixels( Canvas, pD, pB, pC, pE, 4, Land);
            end;
   end;
end;

procedure DrawTriangle(       Canvas:   TCanvas;
                        const A, B, C:  TVertex;
                              Plys:     word;
                              PointDn:  boolean);
var
  AB, BC, CA: TVertex;
begin
  if Plys = 1
then Draw3Vertices(Canvas, A, B, C, (DrawMode <> dmOutline) OR
                                    PointDn)
    else
      begin
      AB := Midpoint(A, B);
      BC := Midpoint(B, C);
      CA := Midpoint(C, A);
      if Plys = 3 then FractalLandscape.DrewSomeTriangles(16);
      Dec(Plys);
      if PointDn
        then begin
             DrawTriangle(Canvas, CA, BC, C, Plys, True);
             DrawTriangle(Canvas, AB, B, BC, Plys, True);
             DrawTriangle(Canvas, BC, CA, AB, Plys, False);
             DrawTriangle(Canvas, A, AB, CA, Plys, True);
             end
        else begin
             DrawTriangle(Canvas, A, CA, AB, Plys, False);
             DrawTriangle(Canvas, BC, CA, AB, Plys, True);
             DrawTriangle(Canvas, CA, C, BC, Plys, False);
             DrawTriangle(Canvas, AB, BC, B, Plys, False);
             end;
      end;
end;

begin
  ScreenColors;
end.
```

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

There are three display modes: Outline, Filled, and Rendered. All three display the landscape as a collection of triangles by doing a single-point perspective transform of the triangle's three **TTriple**s to screen **TPixel**s, then doing either a **PolyLine** or **Polygon** draw. The only difference between the Outline mode and Filled and Rendered modes is that where Outline mode draws a simple "wire mesh" picture of the landscape with no hidden line removal, Filled and Rendered mode rely on drawing order and polygon filling to do a brute force hidden line removal. (This is the so-called "Painter's Algorithm.") Rendered mode in turn differs from Filled mode in basing each triangle's color on the angle it presents to a ray from the "sun."

Draw3Vertices() implements a simplistic concept of "sea level" to lend a bit of verisimilitude to the display. Any triangle that's completely above sea level is drawn normally, while any triangle that's completely below sea level, is drawn in blue, at sea level. If the triangle crosses sea level, FL3 interpolates the crossing points, and draws part of the triangle above water and part below. While this is plausible enough for "coastlines," it's a bit less plausible for lakes: FL3 doesn't detect the lowest lip of a basin, but colors as water only the part(s) of a basin that are under sea level.

Once all the triangles are drawn, FL3 draws vertical lines along the two front edges from sea level to any vertices which are above sea level. This is particularly effective in the two filled modes, where the filled vertical quadrilaterals obscure the undersides of the landscape surface.

The Project() Routine

The **Project()** routine is the workhorse on which all drawing operations depend. It converts a 3D **TTriple** to a 2D **TPixel** using single-point perspective and the current window dimensions.

In effect, it draws a line through the point and a vanishing point, and marks the

intersection of that line and the screen. Most of the complexity in Listing 6.3 comes from the use of fixed-point math. (This is primarily a legacy of FL3's origins in the days when floating-point math was very slow, but it *does* also serve to reduce the vertex database's size to the point where it fits in the data segment.) If we strip away the fixed point math, we end up with this:

```
{ 3D transform a point: }
function Project(const P: TTriple): TPixel;
var
  Ratio: double;
  Pt, V: TFloatTriple;
begin
  Pt := FloatTriple(P);
  V  := FloatTriple(VanishingPoint);

  Ratio    := Pt.Y / V.Y;
  Result.X := Round( DisplayWidth *
                  ((V.X - Pt.X) * Ratio + Pt.X));
  Result.Y := DisplayHeight -
              Round( DisplayHeight *
                  ((V.Z - Pt.Z) * Ratio + Pt.Z));
end;
```

That is, it calculates the ratio of the point's depth to the vanishing point's depth, and applies this to the difference between the point's *x* and *z* coordinates and the vanishing point's *x* and *z* coordinates. Since the **TTriple** coordinate system ranges from 0 to 1, we can simply multiply the projected coordinates by the window size to obtain the proper screen coordinates.

Outline Mode

Outline mode is the simplest of all the drawing modes. It simply outlines 'land' triangles in green and water triangles in blue. (See Figure 6.1; alas, we don't have color here.) About all that's worth really noting here is how simple and clear Delphi makes **DrawPixels()**. The API version of **DrawPixels** (for either C or Borland Pascal) would replace the single, simple statement **Canvas.PolyLine([A, B, C, A])** with a local array declaration and four assignments—not to mention all the bother creating and destroying Windows device contexts (DCs) and pens.

Filled Mode

Outline mode is relatively fast and does a pretty good job of suggesting texture, but it has one big problem: You can see right through it. This means that the mesh on the backside of a hill, say, shows through to the front side.

Sophisticated graphics applications have complicated algorithms for "hidden line removal," but FL3 is *not* sophisticated, and relies on brute force techniques to remove hidden lines by drawing over them. (See Figure 6.2.)

That is, **DrawTriangle()** takes care to draw the rearmost triangles first, so that any triangles in front will be drawn later, obscuring the triangles in back. On

the initial call to **DrawTriangle()**, the triangle is “point down.” Vertex A is closest to the front and the bottom of the window, and B and C are in back, towards the top of the window. (See Figure 6.8.) Thus

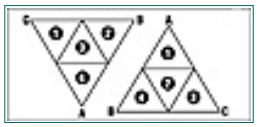

FIGURE 6.8 Drawing Order and Triangle Orientation.

```
DrawTriangle(Canvas, CA, BC, C, Plys, True);
DrawTriangle(Canvas, AB, B, BC, Plys, True);
DrawTriangle(Canvas, BC, CA, AB, Plys, False);
DrawTriangle(Canvas, A, AB, CA, Plys, True);
```

draws the leftmost subtriangle first, then the rightmost. Both of these “outside” subtriangles face the same way as triangle ABC, and the order of the parameters to the recursive call to **DrawTriangle()** ensures that the new A will also be in front, with the new B and C in back.

The third call draws the “inside” subtriangle, as this is visually in front of the second, rightmost triangle. In every triangle, the inside subtriangle is upside down to its outer triangle, so on this initial call, the inside triangle is “point up.” The parameter ordering ensures that on point-up calls, A is still the “point” facing up, with B and C in front, towards the bottom of the screen. If you step through **DrawTriangle()**’s **not PointDn** set of recursive calls, you’ll see that point-up triangles are drawn back-to-front, right-to-left.

The fourth call draws the last, frontmost subtriangle.

Rendered Mode

Rendered mode attempts to look more realistic than filled mode by coloring each triangle based on the angle between the triangle and a ray from the “sun,” so that triangles orthogonal to the rays from the sun are lighter than triangles that present a more oblique angle to the sun’s rays. (See Figure 6.3.) While it *does* seem to do a good job of showing, say, the curvature of a basin, overall it’s a bit...gray. You may wish to experiment with a palette and some sort of **LandColor()** function that use a little false color.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Create Your Own Worlds

In summary, FL3 is a simple demo of fractal landscape generation techniques. You may wish to try to improve the random number distribution or the rendering code, or to modify the algorithm so as to generate entire fractal planets.

It may not be obvious that “plasma” routines are basically just a rectangular version of a fractal landscape generation algorithm: They fracture rectangles instead of triangles, and use color to show the 3rd dimension instead of perspective.

The full source code suite, which includes all Delphi files necessary to build and run Fractal Landscapes 3., is present on the CD-ROM in the directory for this chapter.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

CHAPTER 7
Problems with Persistents, and Other Advice

JON SHEMITZ and ED JORDAN

- Component Read And Write Methods
- Pentium RDTSC Instruction
- Converting Comp Values To Strings
- Changing A Drag And Drop Cursor
- Delphi apps That Set Up Themselves
- Using “Inherited” With Redeclared Properties
- Disabling Individual Radio Buttons

Sometimes you’ll discover that Delphi sets a component property’s value with its **read** method and not its **write** method. Creating these methods carelessly means big trouble! Jon and Ed share their thoughts on avoiding hassles like this and making the most of what Delphi offers.

Delphi properties have a simple, powerful interface: They look like variables, but you have full control over how (and whether) your class’s users can read and write your class’s properties. You can allow direct read access, just as if the property were a variable, or you can specify a **read** method that will be called whenever the property’s value is read. You can allow direct write access, or you can specify a **write** method that will be called whenever the property’s value is set.

Right?

Wrong.

Reading to Write?

What actually happens is this: While the simple model described above does apply in most cases, things get weird when the property is a **TPersistent** descendent like a **TBitmap** or a **TFont**. For **TPersistent** descendents, the **write** method is called when you set the property at design time or change it at runtime—but *not* when a component is created and loaded from its form’s .DFM

stream. Instead, the runtime library calls the property's **read** method to get a pointer to the property's private field variable, then uses that pointer to call the property's read-from-stream method. The **write** method is not called when the component is loaded!

Of course, in most cases this doesn't matter—the property gets loaded, and it has the same value at runtime as it had at design time. However, there *are* a couple of ways this can affect you.

First, the **read** method should never return **Nil**. While it may seem sensible to use a strategy of not actually creating a private field variable until the **write** method supplies an actual value to copy, Delphi's component loading code is not quite smart enough to notice that it has no **TPersistent** to tell to load. If your **read** method returns **Nil**, you will get GPF's when the component is loaded. (For what it's worth, this behavior is what alerted me to the problem, though I'll confess that it took me a while to find out what was going on.)

Second, don't rely on your **write** method to extract information from the field variable and save it in runtime-only fields elsewhere in your component. The **write** method will be called when you set the property directly at design time or at runtime, but *not* when you indirectly set the property at component load time. If you expect your **write** method to update your component's internal state, your component will not load properly.

Reasonable Workarounds

Now, I think there's a very good chance that Borland will conclude that read-to-write *is* a bug, and will change this behavior in future releases of Delphi. That is, you should avoid any solutions to the problems of GPF-on-load or partial-loading that will fail if the **write** method *does* get called at component load time.

It's easy to prevent GPF-on-load in a "forward-compatible" way. We now know that if a **TPersistent** property is stored, Delphi will call its **read** method when the object is loaded from a stream. So, as shown in Listing 7.1, the object's **Create** constructor must create an object of the appropriate type, and set the property's private field variable. This *is* a bit wasteful if the property is not always set or stored, but a few hundred bytes of storage or a few hundred instructions of **Create** code are just plain insignificant to a sixteen or thirty-two megabyte Pentium.

Listing 7.1 PERSIST.SRC

```
{interface}
type DemoComponent =
  class(TComponent)

    private
      fGlyph:        TBitmap;
      fGlyphWritten: boolean;

      procedure SetGlyph(Glyph: TBitmap); {not shown}

    protected
      constructor Create(Owner: TComponent); override;
      procedure Loaded; override;

    public

    published
      property Glyph: TBitmap read fGlyph write SetGlyph;

  end;
```

```
{implementation}

constructor DemoComponent.Create(Owner: TComponent);
begin
  inherited Create(Owner);
  fGlyph := TBitmap.Create;
  {Be sure to fill in the field with an empty object}
end;

procedure DemoComponent.SetGlyph(Glyph: TBitmap);
begin
  if fGlyph <> Glyph then    { fGlyph = Glyph when SetGlyph is    }
     begin                   { called by the Loaded procedure     }
     fGlyph.Free;    {Assign can fail if the target is not  Empty:}
     fGlyph := TBitmap.Create; {Free/Create/Assign is a lot  safer}
     fGlyph.Assign(Glyph);
     end;
   {Extract any necessary data ... set the PropertyWritten flag}
   fGlyphWritten := True;
end;

procedure DemoComponent.Loaded;
begin
  inherited Loaded; {Don't forget to do this!}
  if (not fGlyphWritten) and (not fGlyph.Empty) then
    SetGlyph(fGlyph); {Extract any necessary data from the bitmap}
end;
```

Partial-loading is a little more complex. Fortunately, Delphi components have a **Loaded** method which you can override to perform any necessary post-processing. You can overcome partial-loading by taking advantage of the **Loaded** method, and by making a few small changes to your code.

The first thing to do is to add an **fPropertyWritten** boolean flag for every **TPersistent** property that might be stored. (See Listing 7.1.) The flag will be set to **False** when the object is created, and should be set to **True** only by the **write** method.

Second, you must override (using the **override** directive) your component's **Loaded** method, and add a line like this

```
if not fPropertyWritten then
  SetProperty(fProperty);
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

so that **Loaded** calls your **write** method if (and *only* if) the component-loading code didn't call it.

Third, you generally *don't* want to **Assign** a **TPersistent** to itself, and you *certainly* don't want to **Free** it, **Create** a new instance, then **Assign** the old (freed) instance to the new instance. It's best to use code like that shown in Listing 7.2. This way, you only reset the private field variable if the new value does not equal the existing field. Adding this test ensures that **SetProperty(fProperty)** will not cause a GPF now, and won't constitute much overhead if read-to-write *does* go away.

Listing 7.2 PERSIST2.SRC

```
if fProperty <> NewPropertyValue then
  begin
    fProperty.Free;                    {Assigning 'over' a }
    fProperty := TPropertyType.Create; {TPersistent can fail:}
    fProperty.Assign(NewPropertyValue) {Free/Create/Assign's safer}
  end;
{Extract any necessary data from NewPropertyValue}
fPropertyWritten := True;
```

A Little Perspective

My personal suspicion is that read-to-write is the result of a bit of overzealous optimization by the Delphi team. While it seems like it would be pretty hard to make a case for it's not being a bug, whenever I run into a bug or bit of poor design in Delphi, I like to ask myself how many apps I've ever used or written that are more stable than Delphi, or have a higher ratio of good to bad design decisions. The answer is always "Few, if any."

It's also worth bearing in mind that the **write** method *is* called at load time for simple types like integers, enums, and strings—and that read-to-write for **TPersistent** objects and descendents is pretty easy to deal with.

Using RDTSC for Pentium Benchmarking

Back in ancient times, writing fast code wasn't just a matter of choosing the right algorithm; you had to know your instruction times, and you had to benchmark different sequences. Of course, because the system timer only ticked every 55 milliseconds, benchmarking meant either repeating your code hundreds of thousands of times, or pulling hackerish tricks like reading the timer chip's internal registers to get the time in 838 nanosecond increments.

Nowadays, of course, compilers are so good and processors are so fast that it's gotten pretty hard to come up with a piece of 'brain-damaged' brute force code that's going to noticeably slow your programs. It seems more than a little ironic, then, that Intel waited for the Pentium to introduce a benchmarking instruction. RDTSC—Read Time Stamp Counter—returns the number of clock cycles since the CPU was powered up or reset. Where was RDTSC back when we *really* needed it?

Still, better late than never. RDTSC is a two byte instruction: $0F 31. It returns a 64-bit count in EDX:EAX. Since the 8087 **comp** datatype is a 64-bit integer, we can use the Delphi code in Listing 7.3 to read the current value.

Listing 7.3 RDTSC.SRC

```
const
   D32 = $66;

function RDTSC: comp;
var
   TimeStamp: record
                  case byte of
                     1: (Whole:  comp);
                     2: (Lo, Hi: LongInt);
                  end;
begin
   asm
db $0F; db $31;// BASM doesn't suppprt RDTSC
            Pentium RDTSC - Read Time Stamp Counter - instruction
$ifdef Cpu386
mov[TimeStamp.Lo],eax// the low dword
             mov[TimeStamp.Hi],edx// the high dword
$else
db D32
movword ptr TimeStamp.Lo,AX
            mov [TimeStamp.Lo],eax - the low dword
db D32
movword ptr TimeStamp.Hi,DX
            mov [TimeStamp.Hi],edx - the high dword
$endif
   end;
   Result := TimeStamp.Whole;
end;
```

One problem you may run into when you use RDTSC is that both **IntToStr** and **Format('%d')** can only handle **LongInt** values, not **comp** values. While you can pass a **comp** value to one of these functions, it cannot be any larger than **High(LongInt)**, which is 2,147,483,647. While that would be a mighty satisfying number of dollars to see on your brokerage statement, it's only a little over 16 seconds of clock ticks to a 133 MHz Pentium. If you need to compare two long-running processes, the difference between the start ticks and the stop ticks can easily exceed **High(LongInt)**. In these cases, you can use the **CompToStr** function shown in Listing 7.4.

Listing 7.4 COMP2STR.SRC

```
function CompToStr(N: comp): CompStr;
var
   Low3: string[3];
   N1:   extended;
begin
   if N < 0
      then Result := '-' + CompToStr(-N)
```

```
      else begin
              N1 := N / 1000;
              Str(Round(Frac(N1) * 1000), Low3);
              N := Int(N1);
              if N > 0
                 then begin
                        while Length(Low3) < 3 do Low3 := '0 ' + Low3;
                        Result := CompToStr(N) +  ThousandSeparator +
                        Low3;
                        end
                 else Result := Low3
              end;
end;
```

In the end, I guess you could say this is just another case of “The rich get richer”—we need to do a lot less benchmarking than we ever did before, while at the same time, the RDTSC instruction makes benchmarking easy and reliable.

And on that note, I turn control of this chapter over to my collaborator, Ed Jordan. Take it, Ed.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Drag-and-Drop Rectangles for a Delphi Listbox

Thanks, Jon. When a user drags an item in Delphi, the mouse cursor changes; by default, it becomes an arrow with a small box clinging to its tail. Signifying drag and drop like this works with no fuss—after all, the mouse was going to be there anyway.

It is possible, however, to give users more assurance that something is happening. For example, when dragging an item from a list box, the mouse could carry around the outline of a rectangle the same size and shape as one line of list box text. As it turns out, dragging this rectangle is easy. Listing 7.5 presents the complete source code for a ListBox component that adds the feature.

The first question is where to draw. A rectangle drawn on the list box canvas will disappear if the mouse moves off the edge of the list box—not ideal. And on the parent canvas of the list box, the rectangle we draw will be clipped by any sibling windowed controls.

The one place where the rectangle will always show is the “canvas” of the screen. Actually, the screen doesn’t come with a built-in canvas property, but you can gain access to the screen’s device context with a call to **GetDC** and a parameter of 0. The rectangle can then be drawn using the Windows API function **DrawFocusRect**. **DrawOutline** in Listing 7.5 illustrates getting the DC, drawing on it, and releasing it.

Listing 7.5 OUTLNLBX.PAS

```
unit OutlnLBx;

interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;

type
  TOutlineListBox = class( TListBox )
  private
```

```
    DrawX, DrawY: Integer; { Mouse position }
    XOfs, YOfs: Integer;   { Drawing offset from mouse pos.}
    OutlineDrawn: Boolean; { TRUE if outline drawn }
    procedure DrawOutline;
  protected
    procedure MouseDown( Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer ); override;
    procedure MouseMove( Shift: TShiftState; X, Y: Integer );
      override;
    procedure MouseUp( Button: TMouseButton; Shift: TShiftState;
      X, Y: Integer ); override;
  end;

implementation

procedure TOutlineListBox.DrawOutline;
var
  ScreenDC: HDC;
  Outline: TRect;
begin
  ScreenDC := GetDC( 0 );
  try
    with ClientToScreen( Point( DrawX - XOfs, DrawY - YOfs )) do
    begin
      Outline := Bounds( X, Y, Width, ItemHeight );
      DrawFocusRect( ScreenDC, Outline );
    end;
  finally
    ReleaseDC( 0, ScreenDC );
  end;
end;

procedure TOutlineListBox.MouseDown;
begin
  inherited MouseDown( Button, Shift, X, Y );
  if ( Button = mbLeft ) then
  begin
    XOfs := X;
    YOfs := Y - ( Y div ItemHeight ) * ItemHeight;
  end;
end;

procedure TOutlineListBox.MouseMove;
begin
  inherited MouseMove( Shift, X, Y );
  if Dragging then
  begin
    if OutlineDrawn then DrawOutline; { Erases rectangle }
    DrawX := X;
    DrawY := Y;
    DrawOutline;                      { Draws rectangle }
    OutlineDrawn := True;
  end;
```

```
end;

procedure TOutlineListBox.MouseUp;
begin
  inherited MouseUp( Button, Shift, X, Y );
  if ( Button = mbLeft ) and OutlineDrawn then
  begin
    DrawOutline;         { Erases rectangle for last time }
    OutlineDrawn := False;
  end;
end;

end.
```

Three other methods finish off the draggable outline code. **MouseDown** makes a note of the offset of the mouse from the top-left corner of the rectangle. Remembering the offset ensures that if the mouse "grabbed" the rectangle by its middle, it keeps hold of the middle as it moves around.

MouseMove erases the rectangle if it has already been drawn at least once, then draws it in the new position. In the process, it remembers the top-left corner for erasing next time.

MouseUp erases the rectangle one last time.

One note about the drawing mechanism: **DrawFocusRect** draws rectangles via XOR. Drawn this way, a rectangle shows up against almost any background; and drawn twice in the same place, it disappears.

Making String Collections More List-Like

When I moved to Delphi from Borland Pascal, I wished that string lists were a little more like string collections—where were my **ForEach** iterators?

Now, though, as I port a Delphi application *back* to Turbo Vision, I'm wishing that string collections were a lot more like string lists.

Moving strings into and out of collections is like mixing concrete compared to the same smooth operations with lists—mainly because the new Object Pascal syntax is so much cleaner.

Compare this code, for adding an item to Turbo Vision collection

```
AStringColl^.AtInsert(AStringColl^.Count, NewStr(S));
S := PString(AStringColl^.At(Index))^;
```

to this code, which does the same thing for a Delphi string list

```
StringList.Add(S);
S := StringList[Index];
```

and you'll see what I mean. Not only are the string collection operations practically unreadable, the second line of code above is simply wrong. If the **PString** being indexed is **NIL**, signifying that we put an empty string into the collection, we'll read garbage into our string variable **S**.

Fortunately, a descendant of **TStringCollection** can make strings and collections mix more gracefully. The new object type shown in Listing 7.6 creates a pointer-safe **StrAt** method and a simple **Add** method. Now we can cleanly code things like this:

```
StrList^.Add(S);
S := StrList^.StrAt(Index);
```

And this similar syntax will ease our trips back and forth between the new and old worlds—as long as that old DOS world still matters.

Listing 7.6 STRLIST.PAS

```
{ Create a friendlier, TStringList-like string colleciton.}
unit StrList;
interface
uses Objects;

type
  PStrListCollection = ^TStrListCollection;
  TStrListCollection = object(TStringCollection)
    function StrAt(Index: Integer): string;
    procedure Add(const S: string);
  end;

implementation

{ Translate a pointer into a string, handling the nil case. }
function PtrToStr(P: Pointer): string;
begin
  if P = nil then PtrToStr := '' else PtrToStr := PString(P)^;
end;

{ Safely return a string from the string collection. }
function TStrListCollection.StrAt(Index: Integer): string;
begin
  StrAt := PtrToStr(At(Index));
end;

{ Add a string to the end of the string collection. }
procedure TStrListCollection.Add(const S: string);
begin
  AtInsert(Count, NewStr(S));
end;

end.
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Letting Delphi Applications Set Up Themselves

Since I develop shareware in Delphi, I wanted to use it to create a simple setup program for people who download my software from online services or bulletin boards. Unfortunately, because of the considerable resources Delphi provides nearly automatically, even a simple setup program can run nearly 200k (of course, the increase in program size levels off very quickly after that). Now, this size is unremarkable for a normal Windows application, but setup programs should be as small as possible—especially when the user is paying for every second of download time, or when I am paying to email a registered version to a subscriber.

Happily, though, I have hit on a way to use the full resources of Delphi for my setup program, with a minimal increase in download size: I use my main application as its own setup program. The application is initially named SETUP.EXE, and when run under that name, it installs itself. As far as users can tell, they are running a separate setup program. After installation, the program renames itself, and ceases to be an installer.

Here's how it works. Listing 7.7 shows the main block of a typical unmodified Delphi application's project (.DPR) file. Listing 7.8 shows the same main block with additions that make it function as a setup program. Note that we test to see whether the application's .EXE name is SETUP.EXE—if it is, we run a form, or a series of forms, to let the user choose the program directory, program group, and other setup options.

Listing 7.7 BEFORE.SRC

```
{ The main block of an application's .DPR file, before
  being changed to function as a setup program. }
begin
  Application.Title := 'Program Title';
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

Listing 7.8 AFTER.SRC

```
{ The main block of an application's .DPR file, after being
```

```
   changed to function as a setup program. }
{ Note that SYSUTILS.PAS must be added this unit's USES clause. }

begin
  Application.Title := 'Program Title';
  if UpperCase(ExtractFileName(Application.ExeName)) =
     'SETUP.EXE' then
    Application.CreateForm(TSetupForm, SetupForm)
  else
    Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

Before I zip up my software (.EXE file, help file, read-me file, and so on) to upload it, I rename my .EXE file SETUP.EXE. After the user downloads the software, unzips it, and runs SETUP.EXE, the application copies itself and the auxilliary files to the final directory, renaming itself to its proper name. The next time it is run, it will find that it is *not* named SETUP.EXE, and so will behave normally.

In exchange for an inconsequentially small increase in program size and download time, users get a helpful setup program, and I get (I hope) a few more sales.

Using INHERITED with Redeclared Properties in Delphi

Suppose you're developing a Delphi VCL component—a descendant of **TDrawGrid**, let's say—and you need to take some special action when the component user (in this case, a programmer) changes the **ColCount** property. There are two ways to do this; the best way for you depends on whether you want mere notification of the change, or whether you want to control what the value of **ColCount** can be.

The **ColCount** property determines the number of columns in a grid. Like most properties, its value is stored in a private field (**FColCount**, in this case), and it is changed by way of a private access method (**SetColCount**). Thus, when a programmer writes

```
ColCount := AValue;
```

in his code, or when he changes **ColCount** in the Object Inspector at design time, **SetColCount** is called, and with the help of other private methods, changes the value of the **FColCount** field, and makes adjustments in the grid as needed. All this is encapsulated, out of reach.

But the original developers of **TDrawGrid** anticipated that developers of **TDrawGrid** descendants might want notification of a change in the number of columns—so after the change is made, but before it is shown, the **SizeChanged** method is called. **SizeChanged** is a dynamic method, which means we can override it, and our own **SizeChanged** will be called every time the number of columns (or the number of rows) changes. See Listing 7.9.

Listing 7.9 SIZECHAN.SRC

```
{ A descendant of TDrawGrid that overrides the SizeChanged
   method.  This lets the descendant component be aware of
   when its number of columns or rows has changed. }

{ In unit's interface... }

type
```

```
  TMyGrid = class(TDrawGrid)
  protected
    procedure SizeChanged(OldColCount, OldRowCount: Longint);
      override;
  end;

{ In unit's implementation... }

procedure TMyGrid.SizeChanged(OldColCount, OldRowCount: Longint);
begin
  { Take whatever actions necessary }
end;
```

Overriding **SizeChanged** provides all the notification we need, but if we want control over the number of columns—for example, if we're developing a grid that should have no more than three columns—**SizeChanged** is not a satisfactory place to shoehorn in our veto. By the time **SizeChanged** is called (note the past tense in the name), the change has already been made. The best we can do then, if **ColCount** was set to 4, is change it to 3, requiring the whole private process of change to be repeated.

What we want is front end control, and we can get this by redeclaring the **ColCount** property with our own access methods (see the **TMyGrid** declaration in Listing 7.10). Our redeclaration of the property will hide our ancestor's **ColCount**, so that when a programmer writes

```
ColCount := AValue;
```

our own nonvirtual **SetColCount** access method will be called. As you can see from the **SetColCount** method in Listing 7.10, we first check to see whether the suggested value for the number of columns is less than or equal to 3. If it is, we make the change.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Listing 7.10 SETCOLCT.SRC

```
{ A descendant of TDrawGrid that redeclares the ColCount
  property with new access methods.  This lets the descendant
  component control the number of columns. }

{ In unit's interface... }

type
  TMyGrid = class(TDrawGrid)
  private
    function GetColCount: LongInt;
    procedure SetColCount(Value: LongInt);
  published
    property ColCount: LongInt read GetColCount
      write SetColCount default 0;
  end;

{ In unit's implementation... }

function TMyGrid.GetColCount: LongInt;
begin
  Result := inherited ColCount;
end;

procedure SetColCount(Value: LongInt);
begin
  if Value <= 3 then inherited ColCount := Value;
end;
```

But how we make the change is perhaps the most interesting part of redeclaring a property. We can't directly change the **FColCount** field—and we wouldn't want to if we

could—because the other adjustments needed in the grid would not get made. We can't call our ancestor's **SetColCount** method—that's private. And if we write

```
ColCount := Value;
```

within our own **SetColCount** method, we will create endless recursion and crash the stack.

The answer is to use the **INHERITED** keyword with the property name:

```
inherited ColCount := Value;
```

The ability to use **INHERITED** with an ancestor's property name is not as well documented as its use with an inherited public or protected method. It's a pleasant surprise, but it is entirely consistent with Object Pascal's other pleasant surprises.

Taking Snapshots of the Screen with Delphi

In Delphi, if we want to capture the image of a form's client area, we can call **GetFormImage**. But sometimes we want a snapshot of the entire form—title bar, frame, and all. Or we want a snapshot of the entire screen. If we were desperate, we might pop up a message box that says, "Press Print Screen button NOW!" and then figure out a way to get the screen's mug off the clipboard.

But we're not *that* desperate. Combining Delphi canvases with a few GDI functions makes capturing the screen in code a snap. **CaptureScreenRect**, in Listing 7.11, demonstrates this. It gets the screen's device context with **GetDC(0)**, and then it copies a rectangular area from that DC to a bitmap's canvas. To do the copying, it uses **BitBlt**. The key to using **BitBlt**—or any GDI function—with Delphi is remembering that a canvas' Handle is the DC that Windows needs.

Listing 7.11 SCRNCAP.PAS

```
{ Screen capture functions for Delphi }
unit ScrnCap;

interface
uses WinTypes, WinProcs, Forms, Classes, Graphics;

function CaptureScreenRect( ARect: TRect ): TBitmap;
function CaptureScreen: TBitmap;
function CaptureClientImage( Control: TControl ): TBitmap;
function CaptureControlImage( Control: TControl ): TBitmap;

implementation

{ Use this to capture a rectangle on the screen... }
function CaptureScreenRect( ARect: TRect ): TBitmap;
var
  ScreenDC: HDC;
begin
  Result := TBitmap.Create;
  with Result, ARect do
```

```
  begin
    Width := Right - Left;
    Height := Bottom - Top;

    ScreenDC := GetDC( 0 );
    try
      BitBlt( Canvas.Handle, 0, 0, Width, Height, ScreenDC,
        Left, Top, SRCCOPY );
    finally
      ReleaseDC( 0, ScreenDC );
    end;
  end;
end;

{ Use this to capture the entire screen... }
function CaptureScreen: TBitmap;
begin
  with Screen do
    Result := CaptureScreenRect( Rect( 0, 0, Width, Height ));
end;

{ Use this to capture just the client area of a form
  or control... }
function CaptureClientImage( Control: TControl ): TBitmap;
begin
  with Control, Control.ClientOrigin do
    Result := CaptureScreenRect( Bounds( X, Y, ClientWidth,
                ClientHeight ));
end;

{ Use this to capture an entire form or control... }
function CaptureControlImage( Control: TControl ): TBitmap;
begin
  with Control do
    if Parent = nil then
      Result := CaptureScreenRect( Bounds( Left, Top, Width,
                  Height ))
    else
      with Parent.ClientToScreen( Point( Left, Top )) do
        Result := CaptureScreenRect( Bounds( X, Y, Width,
                    Height ));
end;

end.
```

The remaining screen capture functions in Listing 7.11 cobble up rectangles and farm out the real work to **CaptureScreenRect**. **CaptureScreen** throws together a rectangle for the whole screen. **CaptureClientImage** and **CaptureControlImage** throw together rectangles for the client area and for the entire area of a control, respectively.

These four functions can be used to capture any arbitrary screen area, as well as the screen images of forms, buttons, memos, combo boxes, and so on. We just tell the little buggers to

say cheese—and free the bitmaps when we’re done.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Delphi RadioGroup Buttons You Can Disable

When I am designing forms, nothing is quite so comforting as controls that know how to snap into place, spring out to size, and arrange their contents into columns: Hey, I feel as though I'm not fighting the battle alone. And the benefit of these size-wise controls is not just emotional; how many lines of code has a panel's **Align** property saved me? Tens? Hundreds? That's why, when I need to disable individual radio buttons, I am reluctant to heave the springy little RadioGroup control over the side. **TRadioGroup** automatically arranges its child buttons into columns, spaces them evenly, and lets them be named with one string list.

What it does *not* do is allow access to individual buttons—and for good reason, I'm sure. But since I trust myself to disable responsibly, I derived an enhanced component from **TRadioGroup**, as shown in Listing 7.12. **TRadioBtnGroup** has a new property, **ItemEnabled**, that lets us get and set the enabled status of individual buttons.

Listing 7.12 RBTNGRPS.PAS

```
{ A radio group with buttons that can be disabled }
unit RBtnGrps;
interface
uses StdCtrls, ExtCtrls;

type
  TRadioBtnGroup = class( TRadioGroup )
  private
    function GetItemEnabled( Index: Integer ): Boolean;
    procedure SetItemEnabled( Index: Integer; Value: Boolean );
    function GetButtons( Index: Integer ): TRadioButton;
  protected
    function CheckAnyBut( NotThisIndex: Integer ): Boolean;
    property Buttons[ Index: Integer ]: TRadioButton
      read GetButtons;
  public
```

```
    property ItemEnabled[ Index: Integer ]: Boolean
      read GetItemEnabled write SetItemEnabled;
  end;

implementation

function TRadioBtnGroup.CheckAnyBut;
var
  Index: Integer;
begin
  Result := True;
  for Index := NotThisIndex + 1 to Items.Count - 1 do
    if Buttons[ Index ].Enabled then
    begin
      Buttons[ Index ].Checked := True;
      Exit;
    end;
  for Index := 0 to NotThisIndex - 1 do
    if Buttons[ Index ].Enabled then
    begin
      Buttons[ Index ].Checked := True;
      Exit;
    end;
  Result := False;
end;

function TRadioBtnGroup.GetItemEnabled;
begin
  Result := Buttons[ Index ].Enabled;
end;

procedure TRadioBtnGroup.SetItemEnabled;
begin
  if ( not Value ) and ( Index = ItemIndex ) and
      Buttons[ Index ].Checked and
      ( not CheckAnyBut( Index )) then
    ItemIndex := -1;
  Buttons[ Index ].Enabled := Value;
end;

function TRadioBtnGroup.GetButtons;
begin
  Result := Components[ Index ] as TRadioButton;
end;

end.
```

Internally, **TRadioBtnGroup** uses the **GetButtons** method to gain access to its radio buttons. **GetButtons** relies on the fact that, since a radio group owns its child buttons, it holds them in its **Components** array. All **GetButtons** does is index into the **Components** array and safely typecast the result.

The new control tries its best to be an intelligent helper. If a checked button is disabled, the

control tries to check another button; if all the buttons are disabled, it unchecks all the buttons. Depending on your needs, you might want to change this latter behavior.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

CHAPTER 8
Animated Screen Savers in Delphi

RAY KONOPKA

- Hooking the Application's OnIdle Event
- Windows 95 Callback Functions
- A Combo Box for Color Selection
- Special Project File Modifications

Anything you can do with the Windows API, you can do in Delphi. How about a screen saver? Fear not the Windows Callback, and kiss MakeProcInstance goodbye!

Unlike many other visual development tools, Delphi provides *complete* access to the Windows API. Yes, the VCL does encapsulate much of this API, but not all of it. This chapter investigates how Delphi applications can take advantage of the Windows API. To demonstrate the process, we're going to create a screen saver.

Figure 8.1 shows a picture of the screen saver in action. The basic scheme is that a laser beam (okay, a line) draws out the Def Leppard logo and then erases it. I selected the logo of my favorite band, but you could use any logo you want. All you need to do is determine the coordinates of the line segments that define the path of the laser.

FIGURE 8.1 The Def Leppard screen saver.

Secrets of the Main Form

The heart of the screen saver lies in the **LaserFrm** form file. Listing 8.1 shows the source code for the unit embodying the form file. This form file defines the **TFrmLaser** form class, which

represents the main window on which the laser beam will draw the logo.

Listing 8.1 LASERFRM.PAS

```
unit LaserFrm;

interface

uses
  Messages, WinTypes, WinProcs, Classes, Graphics, Controls, Forms,
  DLPoints, SvrUtils;

type
  TFrmLaser = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormClose( Sender: TObject;
                         var Action: TCloseAction);
    procedure FormKeyDown( Sender: TObject; var Key: Word;
                           Shift: TShiftState);
    procedure FormMouseDown( Sender: TObject; Button: TMouseButton;
                             Shift: TShiftState; X, Y: Integer);
    procedure FormMouseMove( Sender: TObject; Shift: TShiftState;
                             X, Y: Integer);

  private
    FMousePt : TPoint;
    FDone : Boolean;
    FOriginPoint : Longint;
    FOrigins : array[ TLaserOrigin ] of Longint;
    FCurrPoint : Integer;
    FFirstPass : Boolean;
    FGettingPassword : Boolean;
    FOrigX : Integer;
    FOrigY : Integer;
    procedure IdleProc( Sender : TObject; var Done : Boolean );
    procedure Animate;
    procedure CursorOff;
    procedure CursorOn;
    procedure PopulateOrigins;
    procedure GetPassword;
    procedure SetTopMost( TopMost : Boolean );
  end;

var
  FrmLaser: TFrmLaser;

implementation

{$R *.DFM}

uses
  GetPWFrm, MMSystem;

const
  Offset = 25;     { Distance from Screen Edge of Laser Source }
```

```
{======================}
{== TFrmLaser Methods ==}
{======================}

procedure TFrmLaser.FormCreate( Sender : TObject );
begin
  Left := 0;                      { Resize Form to fit screen }
  Top := 0;
  Height := Screen.Height;
  Width := Screen.Width;

  Randomize;              { Initialize Random Number Generator }
  FCurrPoint := 1;
  FFirstPass := True;
  FDone := False;

  FMousePt.X := -1;
  FMousePt.Y := -1;
  LoadSettings;          { Load Settings from Control.ini file }

  Canvas.Pen.Color := LaserColor;
  Canvas.Pen.Mode := pmXor;

  PopulateOrigins; { Set Source Locations based on Screen Size }
  FOriginPoint := FOrigins[ LaserOrigin ];

  SetTopMost( True );
  CursorOff;
  Application.OnIdle := IdleProc;
end; {= TFrmLaser.FormCreate =}


procedure TFrmLaser.FormClose( Sender : TObject;
                               var Action : TCloseAction );
begin
  Application.OnIdle := nil;
  CursorOn;                  { Don't forget to turn on cursor }
end;

procedure TFrmLaser.FormKeyDown( Sender : TObject; var Key : Word;
                                 Shift : TShiftState );
begin
  GetPassword;
end;


procedure TFrmLaser.FormMouseDown( Sender : TObject;
                                   Button : TMouseButton;
                                   Shift : TShiftState;
                                   X, Y : Integer );
begin
  GetPassword;
end;
```

```
procedure TFrmLaser.FormMouseMove( Sender : TObject;
                                   Shift : TShiftState;
                                   X, Y : Integer );
begin
  if FDone then
    Exit;

  if ( FMousePt.X = -1 ) and ( FMousePt.Y = -1 ) then
  begin
    FMousePt.X := X;                    { Record first point }
    FMousePt.Y := Y;
    Exit;
  end;

  if ( Abs( X - FMousePt.X ) > 3 ) or
     ( Abs( Y - FMousePt.Y ) > 3 ) then
  begin
    GetPassword;      { If mouse moved > 3 pixels, GetPassword }
    FMousePt.X := -1;                  { Reset mouse position }
    FMousePt.Y := -1;
  end
  else
  begin
    FMousePt.X := X;               { Record new mouse position }
    FMousePt.Y := Y;
  end;
end;


procedure TFrmLaser.IdleProc(Sender : TObject; var Done : Boolean);
begin
  Animate;
  Done := False;                { Prevent call to WaitMessage }
end;

{= LaserLine is a Windows callback procedure that is called   =}
{= by the LineDDA procedure in the Animate method below.      =}

procedure LaserLine( X, Y : Integer; Origin : Longint ); export;
begin
  Application.ProcessMessages;      { Process pending messages }

  { If screen saver shut down, then don't draw any more lines  }
  if FrmLaser.FDone then
    Exit;

  with FrmLaser.Canvas do
  begin
    MoveTo( LoWord( Origin ), HiWord( Origin ) );
    LineTo( X, Y );
    WinDelay( LaserSpeed );
    MoveTo( LoWord( Origin ), HiWord( Origin ) );
    LineTo( X, Y );
```

```
    Pixels[ X, Y ] := LaserColor;
  end;
end;


procedure TFrmLaser.Animate;
begin
  if ( FCurrPoint = 1 ) and FFirstPass then
  begin
    FOrigX := Random( Screen.Width - 300 );
    FOrigY := Random( Screen.Height - 120 );
    if RandomColors then
    begin
      LaserColor := Colors[ Random( 15 ) + 1 ];
      Canvas.Pen.Color := LaserColor;
    end;
    if RandomOrigin then
    begin
      LaserOrigin := TLaserOrigin( Random( 9 ) + 1 );
      FOriginPoint := FOrigins[ LaserOrigin ];
    end;
  end;

  { LaserPoints of -1 indicate breaks in connecting the dots }
  if LaserPoints[ FCurrPoint + 1 ].X <> -1 then
  begin
    if LaserPoints[ FCurrPoint ].X = -1 then
       Canvas.MoveTo( LaserPoints[ FCurrPoint + 1 ].X + FOrigX,
                      LaserPoints[ FCurrPoint + 1 ].Y + FOrigY )
    else
    begin
      { LaserLine called for each point between }
      { ( X1, Y1 ) and ( X2, Y2 ) }
      LineDda( LaserPoints[ FCurrPoint ].X + FOrigX,
               LaserPoints[ FCurrPoint ].Y + FOrigY,
               LaserPoints[ FCurrPoint + 1 ].X + FOrigX,
               LaserPoints[ FCurrPoint + 1 ].Y + FOrigY,
               @LaserLine, Pointer( FOriginPoint ) );
    end;
  end;

  Inc( FCurrPoint );          { Get Ready for Next Laser Point }

  if FCurrPoint = MaxLaserPoints then
  begin
    WinDelay( PauseDelay * 100 );       { Pause before Erasing }
    FCurrPoint := 1;                     { Setup for next pass }
    FFirstPass := not FFirstPass;
  end;
end; {= TFrmLaser.Animate =}


procedure TFrmLaser.CursorOff;
var
  Count : Integer;
```

```
begin
  repeat
    Count := ShowCursor( False );
  until Count < 0;
end;


procedure TFrmLaser.CursorOn;
var
  Count : Integer;
begin
  repeat
    Count := ShowCursor( True );
  until Count >= 0;
end;


procedure TFrmLaser.PopulateOrigins;
begin
  FOrigins[ loUpperLeft ] := MakeLong( Offset, Offset );
  FOrigins[ loUpperCenter ] := MakeLong( Screen.Width div 2,
                                         Offset );
  FOrigins[ loUpperRight ] := MakeLong( Screen.Width - Offset,
                                        Offset );
  FOrigins[ loMidLeft ] := MakeLong( Offset,
                                     Screen.Height div 2 );
  FOrigins[ loMidCenter ] := MakeLong( Screen.Width div 2,
                                       Screen.Height div 2 );
  FOrigins[ loMidRight ] := MakeLong( Screen.Width - Offset,
                                      Screen.Height div 2 );
  FOrigins[ loLowerLeft ] := MakeLong( Offset,
                                       Screen.Height - Offset );
  FOrigins[ loLowerCenter ] := MakeLong( Screen.Width div 2,
                                         Screen.Height-Offset );
  FOrigins[ loLowerRight ] := MakeLong( Screen.Width - Offset,
                                        Screen.Height - Offset);
end;


procedure TFrmLaser.GetPassword;
var
  CanExit : Boolean;
begin
  CanExit := False;
  Application.OnIdle := nil;                { Stop Animating }

  if not PWProtected or ( Password = '' ) then
    CanExit := True
  else
  begin
    SetTopMost( False );
    CursorOn;
    FrmGetPassword := TFrmGetPassword.Create( Application );
    try
      FrmGetPassword.ShowModal;          { Display Password Form }
```

```
      if FrmGetPassword.ModalResult = mrOK then
        CanExit := True;
    finally
      FrmGetPassword.Free;
    end;
  end;

  if CanExit then
  begin
    FDone := True;
    Close;
  end
  else
  begin
    CursorOff;
    SetFocus;
    SetTopMost( True );
    Application.OnIdle := IdleProc;         { Resume Animating }
  end;
end; {= TFrmLaser.GetPassword =}


procedure TFrmLaser.SetTopMost( TopMost : Boolean );
begin
  { Use SetWindowPos rather than the FormStyle property to }
  { avoid erasing current laser image. }
  if TopMost then
    SetWindowPos( Handle, hwnd_TopMost, 0, 0, 0, 0,
                  swp_NoSize + swp_NoMove )
  else
    SetWindowPos( Handle, hwnd_NoTopMost, 0, 0, 0, 0,
                  swp_NoSize + swp_NoMove );
end;

end.
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Only two properties of the main form need modification. First, the **Color** property is set to **clBlack** to provide the characteristic black background. Second, the **BorderStyle** property is set to **bsNone**. You might be tempted to set the **WindowState** property to **wsMaximized** to force the form to occupy the entire screen, but this does not produce the desired effect. A change in **WindowState** will only alter the form's size if **BorderStyle** is set to **bsSizeable**. Unfortunately, when the border is sizable, a title bar appears on the form, which is not desirable for a screen saver.

Adapting to the Environment and the User

The form size must be adjusted in the **FormCreate** method. Since Windows can be used in several different screen resolutions, the **Screen** object is used to determine the current width and height of the screen. The **Screen** object is automatically created by the **Application** object and holds information pertaining to the current display device. For example, the **Screen** object also maintains a list of available display fonts.

After the form's size is adjusted, a number of variables that control the attributes of the laser beam are initialized. For example, the **Randomize** procedure is called to initialize Delphi's random number generator. As we will see shortly, the origin of the laser beam can be randomly chosen from nine possible locations. Likewise, the color of the beam can be randomly selected.

The **FCurrPoint** and **FFirstPass** private fields all govern how the laser beam proceeds through the set of points that define the beam's path. **FCurrPoint** denotes the current point along the path. **FFirstPass** is **True** when the logo is being drawn and **False** when it is being erased.

The **FMousePt** field is used to record changes in the position of the mouse. This is necessary because if the mouse is moved more than 4 pixels in any direction, the screen saver prompts the user to enter a password. If the

password option is not selected, the screen saver terminates. This same processing also occurs when a key is pressed or one of the mouse buttons is clicked.

Next, the **LoadSettings** procedure is called. This procedure is defined in the **SvrUtils** unit, which is shown in Listing 8.2. This unit defines a number of global variables which control the laser beam, including **LaserColor**, **LaserSource**, **LaserSpeed**, and **Password**. These variables are defined in the **SvrUtils** unit because the configuration form also needs access to these values. All of the screen saver settings are stored in the Control.ini file.

After the origin of the laser has been determined, the form's style is altered so that it remains on top of all other windows. This is accomplished by calling the **SetTopMost** method. This method uses the **SetWindowPos** API function to make this change rather than altering the form's **FormStyle** property. This is necessary because changing **FormStyle** from **fsNormal** to **fsStayOnTop** causes the form to erase its contents. This has the undesirable effect of clearing the screen when the password dialog box is displayed.

SetWindowPos is an example of a window management function. There are dozens of functions like this in the Windows API. All are similar in that they operate on the handle of a window. For instance, the first parameter to **SetWindowPos** must be the handle of the window to be manipulated. To use this function on a Delphi form, the form's **Handle** property is used.

Since there is no need for the mouse cursor to be visible, the **CursorOff** method is called next. This method uses the **ShowCursor** API function to turn off the cursor. The **ShowCursor** function takes a boolean parameter which indicates whether the cursor should be made visible or invisible. An interesting feature of **ShowCursor** is that Windows keeps track of the number of times **ShowCursor** is called. Therefore, if **ShowCursor(True)** is called twice, then **ShowCursor(False)** must be called twice before the cursor will be hidden. This is the reason for the repeat loops in both the **CursorOn** and **CursorOff** methods.

Idle Work

The final and most important initialization step performed by the **FormCreate** method is setting the application object's **OnIdle** event to the form's **IdleProc** method. The **OnIdle** event gets invoked when your application does not have to process any messages. This typically happens when your application is waiting for user input. A screen saver's main purpose is to execute until the user presses a key or moves the mouse—a perfect opportunity to handle the **OnIdle** event!

When the **OnIdle** event is generated, the **IdleProc** method is called. First, the **Animate** procedure, which is responsible for drawing the laser beam, is called. We will return to this method shortly. Next, the **IdleProc** method sets the **Done** parameter to **False**. The **Done** parameter is slightly misnamed. A more appropriate name would be *Wait*, because the parameter determines whether or not Delphi calls the **WaitMessage** API function.

By default, **Done** is set to **True,** which means that after the **OnIdle** event has

been handled, the **WaitMessage** function is called. This function does not return until a new message is received in the current application's message queue. For most idle processing, it makes sense to wait because another message is usually not too far behind. However, waiting for a new message defeats the purpose of a screen saver. The user is not interacting with the computer, otherwise, the screen saver would not be active! Therefore, **Done** is set to **False** so the **Animate** method will continue to be called.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Animation

Let's take a closer look at the **Animate** method. The most important feature of this method is that it does not draw the entire logo when it is called. Instead, it only draws a line segment between two points defined in the **LaserPoints** array.

Breaking the animation up into pieces allows the screen saver to be more responsive to user input. For example, it would be unacceptable for the screen saver to continue drawing the logo after the user pressed a key. Instead, the screen saver should respond immediately.

Since the animation of the logo is performed in two passes (drawing and erasing), the **Animate** method generates a random location for the logo at the beginning of the first pass. Random values for laser color and origin may also be calculated at this time. Then, the x-coordinate of the *next* laser point is checked. If its value equals -1, then the point represents a break in the path of the laser beam. This is how each letter in the logo is drawn individually.

When a -1 is encountered, the current pen position is moved to the next point in the array using the **Canvas.MoveTo** method. When the next point in the path does not represent a stopping point, the **LineDda** function is called. **LineDda** is an API function that computes all successive points in the line defined by the first four parameters. For each point on the line, the **LaserLine** procedure, specified as the fifth parameter, is called. Since Windows itself is responsible for making the call, **LaserLine** is appropriately called a *callback function*.

Callback Functions

There are many Windows API functions that require the use of callback functions. For example, all of the enumerating functions such as

EnumWindows and **EnumFonts** require a function that is called for each item in the enumeration.

Callback functions can be either functions or procedures in Delphi. For example, the callback function used by the **LineDda** procedure does not have a return value, so it can be declared as a procedure.

There are two additional rules that govern the use of callback functions in Delphi. First, the callback must be a stand-alone procedure or function. That is, a method cannot serve as a callback function. Second, the callback must be declared using the **export** directive. The **export** directive forces the procedure to use the **far** calling model, which is necessary because Windows invokes the actual procedure call. The **export** directive also causes the compiler to generate special entry and exit code for the procedure so that it may function properly as a callback.

Windows veterans may be wondering why there is no call to **MakeProcInstance**. In early Windows compilers, it was necessary to call **MakeProcInstance,** passing it the address of the exported function. This enabled the exported function to access variables and data that were located in the current application's data segment.

Under Delphi, and most newer compilers, it is usually no longer necessary to call **MakeProcInstance**; that is, as long as the correct compiler setting is selected. In the Options|Project|Compiler settings, make sure *Smart Callbacks* is checked. If it is not, you must call **MakeProcInstance**. (Note: there is a corresponding **FreeProcInstance**.)

Getting back to the screen saver, the **LaserLine** callback procedure is quite simple. The first two parameters to **LaserLine** represent the x- and y-coordinates of one of the points on the line specified in the **LineDda** call. The third parameter is a 4-byte, user-defined value that corresponds to the sixth parameter in the **LineDda** call. In this example, the origin of the laser beam is specified.

Inside **LaserLine**, the **ProcessMessages** method of the **Application** object is called so that background tasks will continue to execute. Next, a line is drawn connecting the origin to the point represented by (**X**, **Y**). After a brief delay, the line is redrawn. This erases the original line because the **Canvas.Pen.Mode** property was set to **pmXor** in the **FormCreate** method. The last step is to set the pixel at (**X**, **Y**) to the color of the laser. The effect of this is that the laser appears to draw the logo.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Configuration

As mentioned earlier, there are several ways to configure the screen saver. Figure 8.2 shows the configuration dialog box that is used to alter the settings. The nine radio buttons in the upper left are used to specify the origin of the laser beam. If the Random check box is checked, each time the logo is drawn, the beam will originate from a different location.

FIGURE 8.2 The configuration dialog.

Likewise, the color of the beam may be selected or chosen at random. The two track bars on the right hand side of the dialog are used to specify how fast the logo is drawn, and how long before it gets erased, respectively.

A quick note about the track bars. Although the screen captures show the screen saver running under Windows 95, the application is not a 32-bit application, and as such, does not have access to the standard Windows 95 track bar. The track bars used in this dialog box are instances of the RzTrackBar custom component. While I don't have the space to describe how this component works, the complete description can be found in my book, *Developing Custom Delphi Components*, which is available from Coriolis Group Books.

Figure 8.2 also shows that this screen saver supports a password. The screen saver satisfies this requirement by using the **EncryptString** function which is shown in Listing 8.2. Located in the **SvrUtils** unit, **EncryptString** takes a single string parameter and encrypts it using the same technique used by Windows for encrypting screen saver passwords. This enables the laser screen saver to use the same password that is used for the standard Windows screen savers (for example, *Starfield Simulation*).

Listing 8.2 SVRUTILS.PAS

```
unit SvrUtils;

interface

uses
```

```
  Graphics;

const
  Colors : array[ 0..15 ] of TColor =
    ( clBlack, clMaroon, clGreen, clOlive, clNavy, clPurple,
      clTeal, clGray, clSilver, clRed, clLime, clYellow, clBlue,
      clFuchsia, clAqua, clWhite );
type
  TLaserOrigin = ( loUpperLeft, loUpperCenter, loUpperRight,
                   loMidLeft, loMidCenter, loMidRight,
                   loLowerLeft, loLowerCenter, loLowerRight );

var
  LaserSpeed : Longint;
  PauseDelay : Longint;
  LaserColor : TColor;
  LaserOrigin : TLaserOrigin;
  Password : string;
  PWProtected : Boolean;
  RandomColors : Boolean;
  RandomOrigin : Boolean;

procedure LoadSettings;
procedure SaveSettings;
function EncryptString( const S : string ) : string;
procedure SetPassword( const S : string );
procedure WinDelay( Duration : Longint );

implementation

uses
  IniFiles, SysUtils, WinProcs, Forms, MMSystem;

const
  Section = 'Screen Saver.DLLaser';

procedure LoadSettings;
var
  IniFile : TIniFile;
  TempInt : Integer;
begin
  IniFile := TIniFile.Create( 'CONTROL.INI' );
  Password := IniFile.ReadString( 'ScreenSaver', 'Password', '' );
  LaserSpeed := IniFile.ReadInteger( Section, 'LaserSpeed', 10 );
  PauseDelay := IniFile.ReadInteger( Section, 'PauseDelay', 10 );
  LaserColor := IniFile.ReadInteger( Section, 'LaserColor',clGreen);
  TempInt := IniFile.ReadInteger( Section, 'LaserOrigin',
                                  Ord(loLowerCenter) );
  LaserOrigin := TLaserOrigin( TempInt );
  PWProtected := IniFile.ReadBool( Section, 'PWProtected', False );
  RandomColors := IniFile.ReadBool(Section, 'RandomColors', False);
  RandomOrigin := IniFile.ReadBool(Section, 'RandomOrigin', False);
  IniFile.Free;
end; {= LoadSettings =}
```

```
procedure SaveSettings;
var
  IniFile : TIniFile;
begin
  IniFile := TIniFile.Create( 'CONTROL.INI' );
  IniFile.WriteInteger( Section, 'LaserSpeed', LaserSpeed );
  IniFile.WriteInteger( Section, 'PauseDelay', PauseDelay );
  IniFile.WriteInteger( Section, 'LaserColor', LaserColor );
  IniFile.WriteInteger( Section, 'LaserOrigin',Ord(loLowerCenter));
  IniFile.WriteBool( Section, 'PWProtected', PWProtected );
  IniFile.WriteBool( Section, 'RandomColors', RandomColors );
  IniFile.WriteBool( Section, 'RandomOrigin', RandomOrigin );
  IniFile.Free;
end;


function EncryptString( const S : string ) : string;
var
  I, Len : Integer;
  B : Byte;
  Stz : array[ 0..255 ] of Char;

  procedure EXor( X : Byte; var Y : Byte );
  const                { '[]=' - not allowed in profile string }
    NotAllowed = [ 0..$20, $7f..$90, $93..$9F, $3D, $5B, $5D ];
  begin
   if not ( ( Y xor X ) in NotAllowed ) then
     Y := Y xor X;
  end;

begin {= EncryptString =}
  { Encryption method works for null-terminated strings }
  { Therefore, first copy S to a null-terminated string }
  StrPCopy( Stz, UpperCase( S ) );
  Len := StrLen( Stz );
  if Len = 0 then
  begin
    Result := '';
    Exit;
  end;

  for I := 0 to Len - 1 do                        { First Pass }
  begin
    B := Byte( Stz[ I ] );
    Exor( Len, B );
    if I = 0 then
      Exor( $2A, B )
    else
    begin
      Exor( I, B );
      Exor( Byte( Stz[ I - 1 ] ), B );
    end;
    Stz[ I ] := Char( B );              { Store Encrypted Byte }
  end;
```

```
  if Len > 1 then                                  { Second Pass }
  begin
    for I := Len - 1 downto 0 do
    begin
      B := Byte( Stz[ I ] );
      Exor( Len, B );
      if I = Len - 1 then
        Exor( $2A, B )
      else
      begin
        Exor( I, B );
        Exor( Byte( Stz[ I + 1 ] ), B );
      end;
      Stz[ I ] := Char( B );              { Store Encrypted Byte }
    end;
  end;

  Result := StrPas( Stz );              { Return a Pascal string }
end; {= EncryptString =}

procedure SetPassword( const S : string );
var
  IniFile : TIniFile;
begin
  IniFile := TIniFile.Create( 'CONTROL.INI' );
  if S = '' then
    Password := ''
  else
    Password := EncryptString( S );
  IniFile.WriteString( 'ScreenSaver', 'Password', Password );
  IniFile.Free;
end; {= SetPassword =}

procedure WinDelay( Duration : Longint );
var
  Start : Longint;
begin
  if Duration = 0 then
    Exit;
  Start := TimeGetTime;       { TimeGetTime from MMSystem Unit }
  repeat
    Application.ProcessMessages;
  until TimeGetTime - Start >= Duration;
end;

end.
```

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

A Color ComboBox

Take a closer look at the combo box in Figure 8.3. Rather than simply display the name of each color, this combo box displays a sample of each color next to the name. This is accomplished using the owner-draw features of the combo box.

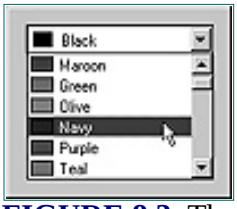

FIGURE 8.3 The TRzColorComboBox component.

Of course, this makes a good candidate for a custom Delphi component, which is exactly what it is. Listing 8.3 shows the source code for the RzColCbx unit. The component's declaration is roughly similar to that of the DrivesComboBox component. Both classes descend from CustomComboBox and operate as owner-draw controls. In addition, neither component surfaces any of the properties which affect owner-draw behavior. For example, the **Style** and **ItemHeight** properties are not redeclared as **published**.

Listing 8.3 RZCOLCBX.PAS

```
unit RzColCbx;
interface
uses
  Messages, WinTypes, WinProcs, Classes, Forms, Graphics, Controls,
  StdCtrls, Menus;

type
  TRzColorComboBox = class( TCustomComboBox )
  private
    FIncludeColor : Boolean;
    FShowSysColors : Boolean;
    procedure SetShowSysColors( Value : Boolean );
    function GetSelectedColor : TColor;
```

```
    procedure SetSelectedColor( Value : TColor );
    procedure DrawItem( Index : Integer; Rect : TRect;
                        State : TOwnerDrawState ); override;
  protected
    procedure CreateWnd; override;
    procedure AddColor( const S : string ); virtual;
  public
    constructor Create( AOwner : TComponent ); override;
  published
    property SelectedColor : TColor
      read GetSelectedColor
      write SetSelectedColor
      default clBlack;

    property ShowSysColors : Boolean
      read FShowSysColors
      write SetShowSysColors
      default True;

    { Inherited Properties & Events }
    property Color;
    property Ctl3D;
    property DragMode;
    property DragCursor;
    property Enabled;
    property Font;
    property ParentColor;
    property ParentCtl3D;
    property ParentFont;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property Sorted;
    property TabOrder;
    property TabStop;
    property Visible;
    property OnChange;
    property OnClick;
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnDropDown;
    property OnEndDrag;
    property OnEnter;
    property OnExit;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
  end;

procedure Register;

implementation

constructor TRzColorComboBox.Create( AOwner : TComponent );
```

```
begin
  inherited Create( AOwner );
  Style := csOwnerDrawFixed;   { Notice Style is not published }
  FShowSysColors := True;
end;

procedure TRzColorComboBox.CreateWnd;
begin
  inherited CreateWnd;
  Clear;                                { Clear items from list }
  FIncludeColor := True;
  GetColorValues( AddColor );   { Call AddColor for all Colors }
  SelectedColor := clBlack;
end;

procedure TRzColorComboBox.AddColor( const S : string );
var
  C : Longint;
begin
  { GetColorValues calls AddColor for all colors, including }
  { system colors.  Flag is cleared to skip system colors.  }
  if ( S = 'clScrollBar' ) and not FShowSysColors then
    FIncludeColor := False;

  if FIncludeColor then
  begin
    IdentToColor( S, C );      { Converts string to color value }
    { Rather than a pointer to an object, the Objects property }
    { is populated with the Longint value of the Color }
    Items.AddObject( Copy( S, 3, 20 ), TObject( C ) );
  end;
end;

procedure TRzColorComboBox.SetShowSysColors( Value : Boolean );
begin
  if Value <> FShowSysColors then
  begin
    FShowSysColors := Value;
    RecreateWnd;
  end;
end;

function TRzColorComboBox.GetSelectedColor : TColor;
begin
  if ItemIndex = -1 then
    Result := clBlack
  else
    Result := TColor( Items.Objects[ ItemIndex ] );
end;

procedure TRzColorComboBox.SetSelectedColor( Value : TColor );

  function IndexFromColor( C : TColor ) : Integer;
  var
    I : Integer;
```

```
  begin
    I := 0;
    while (I < Items.Count) and (C <> TColor(Items.Objects[I])) do
      Inc( I );

    if I = Items.Count then
      Result := -1
    else
      Result := I;
  end;

begin
  ItemIndex := IndexFromColor( Value );
end;

procedure TRzColorComboBox.DrawItem( Index : Integer; Rect : TRect;
                                     State : TOwnerDrawState );

var
  R : TRect;
  C : Longint;
begin
  with Canvas do
  begin
    R := Rect;
    InflateRect( R, -2, -2 );
    R.Right := 20;          { R represents size of color block }
    FillRect( Rect );

    { Color value (i.e. TColor) is stored in Objects property }
    Brush.Color := TColor( Items.Objects[ Index ] );
    Rectangle( R.Left, R.Top, R.Right, R.Bottom );
    if odSelected in State then
      Brush.Color := clHighlight
    else
      Brush.Color := Color;               { Display Color Name }
    TextOut( Rect.Left + 24, Rect.Top + 2, Items[ Index ] );
  end;
end;

procedure Register;
begin
  RegisterComponents( 'Raize', [ TRzColorComboBox ] );
end;

end.
```

The RzColorComboBox component defines two properties that make the component easier to use. First, **SelectedColor** is defined as a **TColor** property. This makes it easier to determine the currently selected color value. Likewise, the **SelectedColor** property can also be used to set the current color.

The second property is **ShowSysColors**. This property affects how many colors appear in the combo box. The **GetColorValues** procedure, defined in the **Graphics** unit, is used to populate the combo box with the available colors. The available colors include the standard 16 colors, plus the

standard system colors (for example, **clScrollBar** and **clHighlight**). If you do not want the system colors to be displayed in the combo box, then set **ShowSysColors** to **False**.

The RzColorComboBox component uses the **Items** list to manage the list of colors. The **Strings** list holds the color names while the **Objects** list holds the color values. Storing the color values in the **Objects** list makes it unnecessary to convert the selected color name to a **TColor** value.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

The Project File

In most normal application projects, the Delphi project (.DPR) file is rarely modified. However, for a screen saver, there are several modifications that need to be performed in order for the screen saver to work properly. Listing 8.4 shows the source code for the **DLLaser** main program project file. The first modification involves adding a description to the program. This is accomplished by using the {$D} directive. For screen savers, the description string *must* start with SCRNSAVE and be followed by a colon and the name of the screen saver. This string is used to identify the screen saver within the Desktop settings of the control panel.

The second modification prevents more than one instance of the screen saver from being executed. This is accomplished by checking the **HPrevInst** global variable. If this value is not zero, then another instance of the screen saver is currently active.

The final modification to the main program file involves processing command line arguments to determine which form to load. When Windows invokes a screen saver, a “/s” (or “-s”) parameter is passed to the screen saver program. When Windows needs to display the configuration dialog box, no command line parameters are passed. The **ParamStr** function can be used in a Delphi program to extract the command line arguments. In this example, the **FrmLaser** form is created if the screen saver is to be started, while the **FrmConfig** form is created if the configuration dialog box is to be displayed.

Listing 8.4 DLLASER.DPR

```
program DLLaser;
{================================================================
  The following description must start with the {$D SCRNSAVE
  and follow with the name to show up in control panel.
  Compile this program to an .EXE and copy it to your windows
  directory with the extension .SCR.
 ================================================================}
{$D SCRNSAVE : Def Leppard Laser}

uses
  Forms,
```

```
  SysUtils,
  Laserfrm in 'LASERFRM.PAS' {FrmLaser},
  Config in 'CONFIG.PAS' {FrmConfig},
  Setpwfrm in 'SETPWFRM.PAS' {FrmSetPassword},
  Svrutils in 'SVRUTILS.PAS',
  Getpwfrm in 'GETPWFRM.PAS' {FrmGetPassword},
  Dlpoints in 'DLPOINTS.PAS';

{$R *.RES}

var
  Params : string;
begin
  if HPrevInst <> 0 then
    Exit;                      { Only allow one instance of app }
  { Build a different "Main" form depending on Command Line }
  Params := UpperCase( ParamStr( 1 ) );
  if ( Params = '/S' ) or ( Params = '-S' ) then
  begin
    Application.Title := 'Def Leppard Laser Screen Saver';
    Application.CreateForm( TFrmLaser, FrmLaser );
  end
  else
  begin
    Application.CreateForm( TFrmConfig, FrmConfig );
  end;
  Application.Run;
end.
```

One final note: Once the **DLLaser** program is compiled to an executable file, you must rename the file, giving it an .SCR extension, and then move it to the Windows directory. Once it's there, you can select it using Control Panel.

NOTE:
Under Windows 95, a screen saver is selected and configured by accessing the Display Properties dialog through the Control Panel, or by right clicking on the desktop. The Screen Saver property sheet lists all installed screen savers using the description strings. However, the description string for a particular screen saver will only appear in the list if the actual file name contains all upper case characters.

Yes, you read that last sentence correctly. To make matters worse, when you try to rename a file in Windows 95 to upper case, the Explorer only displays the first character in upper case. Fortunately, the file does indeed get renamed correctly, even though it doesn't appear that way.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

CHAPTER 9
The Shadowy Math Unit

TERENCE GOGGIN

Explore the new Delphi unit no one knows about...and harness it to do all sorts of statistical dirty work on your data.

Delphi 2 contains a first-class utility unit that, so far, has received only second-class attention. It's in the documentation (sort of), and people seem to know it's there (some people, at least), but most of them don't have the first notion how to use it. It's called the Math unit, and it's an excellent collection of financial, statistical, and general arithmetic and trigonometric functions.

Using some of these functions, this chapter will show you how to build—and use—a data-aware statistical reporting component, TDBStatistics. This component will allow you to give your users a complete data profile of up to 13 different statistics.

Three Good Reasons to Use the Math Unit

There are actually three good reasons to use the Math unit. First and foremost

is speed. The Math unit's routines are *fast.* Many of them are coded in assembly language specifically optimized for the Pentium's Floating Point Unit. Unless you've got a P6 chip and a lot of free time, it probably doesn't get much faster than that!

Second, the alternative of using SQL's statistical routines is unacceptable. SQL only offers four or five statistical functions—far too few to provide a complete statistical picture.

Third, choosing the Math unit over SQL or BDE-based solutions ensures that the TDBStatistics component should work with replacement database engines such as Apollo, Titan, or Direct Access.

Dynamic Data and Static Declarations

The Math unit *is* fast and comprehensive, but there's a catch. You'll need a trick or two to make the most of the statistical functions. You see, many of the Math unit's routines take a parameter declared as:

```
const Data: array of Double
```

Functions like this can be difficult to use because arrays passed via these parameters must be *statically declared*. At first glance, then, there doesn't appear to be a way to pass dynamic data to these functions. Most programmers assume that they've got to use one of two partial solutions to solve this dynamic data dilemma:

1. They can hard-code values into the program; or
2. Make the array as large as possible and hope no user ever exceeds that limit.

Hard-coding values into a program is *occasionally* necessary, but most of the time it turns out to be a very bad idea. In fact, that's exactly what happens here. Consider this call to the **Mean** function:

```
Mean([3, 2, 1, 5, 6]);
```

Essentially, what we have in this line of code is a calculator that gives the same result each time, no matter what values are entered. That's not exactly a useful result, is it?

Obviously, the idea of hard-coding data just won't cut it. This leaves the option of "over-declaring" the size of the array. While this technique is typically very useful—if not downright necessary at times—in some situations, it can introduce unseen complications.

This is especially true in the case of the Math unit. Again, consider the **Mean** function. The "mean" is defined as the Sum of N terms / N. Let's assume we have a 10,000 element array that we're going to pass to the Mean function. If the user were to enter only 50 values, the denominator—that's N—would be off by 9,950! Whoops!

Over-declaring doesn't appear to be such a good idea either. So, how do we

pass dynamic data to these otherwise excellent routines? The answer lies in an all but unknown function buried in the System unit called **Slice**:

```
function Slice(var A: array; Count: Integer): array;
```

Slice takes an array of any size and type and returns **Count** elements as though they were a *separate and distinct array*. With **Slice**, we can declare a very large array and use as many or as few elements as we want.

The **Slice** function, then, allows us to revive the idea of "over-declaring" such that we no longer need to be concerned about the problem of the inaccurate denominator. This, in turn, makes it possible for us to pass dynamic data to these functions.

Armed with our newfound solution, we can now proceed with the creation of our quick and easy statistical reporting component.

Creating TDBStatistics

We've just discovered—thanks to **Slice**—how to pass dynamic data to the functions of the Math unit. So now, what we need is a way to make these routines work efficiently for database analysis. The simplest and friendliest way to do this is wrapping the routines up as a component; a component called, appropriately enough, DBStatistics.

Defining the Component's Tasks

When you're building components, it's usually a good idea to begin by defining the component's tasks. As you might expect, that's just what we'll do for DBStatistics.

DBStatistics' main task is to provide us with easy access to one, several, or all of the 13 possible statistics, given a field name and a DataSource. In order to do this, the component will need the following inner workings:

1. Access to data, preferably via a DataSource;
2. A place to store large amounts of data locally;
3. A way to extract the data from a DataSource; and
4. A way to make the 13 statistics easily available.

In the next four sections, we'll discuss these inner workings in detail.

Getting Access to the Data

To extract the data **TDBStatistics** is to analyze, it must first have a way to link to some **TTable** or **TQuery**. The easiest and simplest way to do this is to provide our component with a DataSource property. And that's exactly what we do. The **private** section contains the following declaration

```
fDataSource : TDataSource;
```

which, of course, is made available at design time via a **published** property.

We also need a way for DBStatistics to know which field it is to analyze. This can be easily provided via a DataField property. There's nothing new to this; you can see it in any component on the "Data Controls" palette tab. Because these two properties are so common, including them in DBStatistics helps to create a familiar design-time feel.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Storing Data Locally

Now that we have access to data, we need a place to store this data once we extract it. The Math unit's routines require a statically declared array; therefore, we must have just such an array. Our array is called, appropriately enough, **Data**.

The question that remains, then, is what the maximum number of elements in **Data** should be. When choosing this value, there are two factors to consider. We must first consider the number of records in our average table. If our tables generally don't exceed 4000 records, then 4500 would be a good maximum elements value.

The other factor to take into account is the amount of available memory. If memory is not an issue, then we can make the array as large as we like. If memory is an issue, however, trial and error is the way to find the optimum size for the array.

For the purposes of illustration, though, let's declare our **Data** array to have 10,000 elements. (Naturally, the 10,000 value will be declared as a constant, **MaxValues**.)This should give us plenty of space for most everyday applications.

Extracting data

The next two pieces of **TDBStatistics,** are very tightly interrelated, so we'll try to develop both of them at the same time. The first of these two areas of interest is the error checking routine, **GetRange**. When the component extracts data, the error checking routine must ensure that everything is done "legally." In most cases, "legally" means nothing more than preventing the component from, say, reading beyond the last record.

However, in the case of **TDBStatistics,** it's a little more subtle than that. Because users might want to analyze a set of records that is larger than our array, we should allow them to select a subset of their data. This is done through two properties: **UpperBound** and **LowerBound**. They provide the component with starting and ending record numbers. The job of the error checking routine then, is to monitor these two values. The simplest way to accomplish this is through a function that:

1. Checks the values for possible errors;
2. Makes any necessary adjustments; and
3. Returns the (adjusted) difference between the two values.

The first two tests ensure that **LowerBound** is greater than zero, and that **UpperBound** is at least **LowerBound + 1**:

```
if (LowerBound < 1) then
    LowerBound := 1;
if (UpperBound < 1) then
    UpperBound := LowerBound + 1;
```

The next test ensures that **UpperBound** is greater than **LowerBound.** If it's found that **LowerBound** is greater than **UpperBound**, the two values are swapped:

```
if (LowerBound > UpperBound) then
  begin
    TempInt := UpperBound;
    UpperBound := LowerBound;
    LowerBound := TempInt;
  end;
```

Then, we test and adjust the values of **UpperBound** and **LowerBound** to make sure that they do not exceed the number of records in the DataSource. (The **DataSource.DataSet.RecordCount** value has already been retrieved and stored in the **Records** variable):

```
if (LowerBound > Records) then
    LowerBound := 1;
if (UpperBound > Records) then
    UpperBound := Records;
```

The last test makes sure that the difference between **UpperBound** and **LowerBound** doesn't exceed the number of elements in the **Data** array. In other words, we can't store more values than we have elements in the array:

```
if (UpperBound - LowerBound > MaxValues) then
  UpperBound := LowerBound + MaxValues;
```

Finally, the **GetRange** function returns the difference between the now adjusted **UpperBound** and **LowerBound:**

```
Result := UpperBound - LowerBound;
```

And that's our error checking routine.

Now that we have our error checking done, we can proceed to the task of extracting the data from the DataSource and storing it in our **Data** array. This is done in a procedure called **FillArray**.

The real work of **FillArray** begins with a call to **GetRange**. Then, once the bounds have been adjusted (as described above), we can extract the data and store it locally. First, we open the DataSource and go to the record number specified by **LowerBound:**

```
fDataSource.DataSet.Open;
fDataSource.DataSet.MoveBy(LowerBound);
```

Next, we test the type of **fDataField**. If the field contains numerical values, then we just extract the data, one record at a time, placing each value into the **Data** array:

```
if ((fDataSource.DataSet.FieldByName(fDataField) is TCurrencyField) or
(fDataSource.DataSet.FieldByName(fDataField) is TFloatField) or
(fDataSource.DataSet.FieldByName(fDataField) is TIntegerField) or
(fDataSource.DataSet.FieldByName(fDataField) is TSmallIntField))then
begin
  for I := LowerBound to UpperBound do
     begin
       if not (fDataSource.DataSet.FieldByName(fDataField).IsNull) then
```

```
          Data[Index] :=
             fDataSource.DataSet.FieldByName(fDataField).Value
             else
          Data[Index] := 0;
        Inc(Index);
        fDataSource.DataSet.Next;
      end;
end;
```

If the field is a character field, though, we should extract the data in a slightly different manner. The only character field data that the component expects to deal with are ZIP codes. There are two possible types of ZIP codes: the older five digit kind, and the newer five “plus four” kind.

As far as **TDBStatistics** is concerned, if the field contains five digit ZIP codes, the value can be converted to a numeric type without additional handling. If the value is a nine-digit, hyphenated ZIP code, though, the hyphen must first be changed to a ‘.’ character so that the value can be converted to a numeric type:

```
else if (fDataSource.DataSet.FieldByName(fDataField) is TStringField)
  then
   begin
     for I := LowerBound to UpperBound do
       begin
         TempString :=
           fDataSource.DataSet.FieldByName(fDataField).Value;
         if (Pos('-', TempString) > 0) then
           TempString[Pos('-', TempString)] := '.';
         Data[Index] := StrToFloat(TempString);
         Inc(Index);
         fDataSource.DataSet.Next;
       end;
   end;
```

Finally, we close the DataSource and reset two boolean flags:

```
fDataSource.DataSet.Close;
IsArrayFilled := True;
DidGetAll := False;
```

The **IsArrayFilled** variable lets the other methods of the component know if the data has been extracted from the DataSource. If it is set to **False**, the other routines can call **FillArray** before completing their work. The **DidGetAll** variable is another boolean flag used by access methods. (Its purpose will become clear momentarily.)

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Making the Data Available

Now that we've done all the background work, the only thing left is to provide a way for us to retrieve the statistics. There are two ways to do this:

1. A method that retrieves all of the 13 results at once;
2. Individual access methods for each of the 13 results, made available via properties.

In this component we use both ideas.

To retrieve all of the statistics at once, we have the **GetAllStats** procedure. **GetAllStats** passes the **Data** array to all 13 statistical functions and stores the results in variables defined in the **private** section of our component. In addition, it sets the boolean variable **DidGetAll** to **True**, indicating to other methods that all of the statistics have already been retrieved.

The individual access methods, of course, can then check the value of the **DidGetAll** variable. If it is set to **True**, the access method can simply return the already stored result. On the other hand, if **DidGetAll** is set to **False**, the access method can call its related Math function directly and return the result from the Math unit. As an example of a typical access method, let's look at **GetMean**, which returns the mean of the selected DataField's values.

First, we make sure that the data has been extracted from the DataSource and stored in the **Data** array:

```
if not (IsArrayFilled) then
  FillArray;
```

Now, another test—this time to make sure that the statistical result we're looking for hasn't already been retrieved. The idea here is that if the result has already been obtained and stored, there's no reason to get it again; the access method should simply return the already stored value to save time.

On the other hand, if it hasn't yet been retrieved, we call the relevant Math unit function, using **Slice** and the **GetRange** function. Lastly, we return the statistic given to us by the Math unit:

```
if not (DidGetAll) then
   fMean := Math.Mean(Slice(Data,GetRange));
Result := fMean;
```

Now that we've provided a quick and easy way to access the statistical results, **TDBStatistics** is ready to be plugged into any project.

Test Driving the DBStatistics Component

Well, now that we've built this wonderful component, let's put it to work. In this section, we'll create a program that allows the user to select any field in any table, and get a complete statistical report displayed neatly in a Memo component. Figure 16.1 shows the **StatsProject** at design time.

FIGURE 9.1 StatsProject at design time.

You'll notice that everything on the form is pretty standard—there are a few ordinary visual controls, a **TTable**, a **TDataSource**, a **TOpenDialog**, and, of course, a **TDBStatistics**. When the user starts the **StatsProject**, he/she must first select a table. This is done through **BtnTableSelect**, the **TButton** control marked "1. Select a table & field." In the **BtnTableSelect**'s **OnClick** event, a table name is retrieved via the TOpenDialog component, **OpenDialog1**.

First, the **Execute** method is called. If a valid file name was selected, then we proceed:

```
with OpenDialog1 do
   begin
     Execute;
     if FileName = '' then
       exit;
```

We now set the properties of the **TTable** component based on the user's file selection. In addition, we display the relevant information via two **TLabel** controls:

```
Table1.DatabaseName := ExtractFilePath(FileName);
LblDatabase.Caption := ExtractFilePath(FileName);
Table1.TableName := ExtractFileName(FileName);
LblTable.Caption := ExtractFileName(FileName);
```

Because **TDBStatistics** only operates on one field at a time, we must provide a way for the user to select a field. This is done by retrieving the field names of the **TTable** and placing them into a ComboBox component:

```
CBFields.Items.Clear; {reset TComboBox}
CBFields.Text := '';
Table1.Open;
for I := 0 to Table1.FieldDefs.Count-1  do
  begin
    Application.ProcessMessages;
    CBFields.Items.Add(Table1.Fields[I].FieldName);
  end;
Table1.Close;
```

And that's all there is to selecting a table and field name.

Once the user has selected a field to be analyzed, he/she can generate a statistical report in the Memo control by clicking the **BtnReports** button. (It's the one marked "2. Generate a report.") In the **BtnReports.OnClick** event, then, we first set the appropriate properties for the DBStatistics component, **DBStatistics1:**

```
DBStatistics1.LowerBound := 1;
```

```
Table1.Open;
DBStatistics1.UpperBound := Table1.RecordCount;
Table1.Close;
DataSource1.DataSet := Table1;
DBStatistics1.DataSource := DataSource1;
DBStatistics1.DataField := CBFields.Text;{user-selected field}
```

Then, we call **DBStatistics1.GetAllStats** and print the results to the Memo component:

```
DBStatistics1.GetAllStats;
Memo1.Text := '';
Memo1.Lines.Add('Mean: ' + #09 + #09 + FloatToStr(DBStatistics1.Mean));

{etc......}

Memo1.Lines.Add('Kurtosis: ' + #09 + #09 +
  FloatToStr(DBStatistics1.Kurtosis));
```

And that's about all there is to it. We now have a complete statistical report generator.

Bugs in the Math Unit

Yes, believe it or not, there is a bug in the Math unit. But it's probably better you hear it now than find out the hard way, right?

So here goes: **MinValue** and **MaxValue** are reversed. **MinValue** actually returns the largest value in the array, while **MaxValue** returns the smallest value in the array. While this bug isn't a showstopper, it does deserve mentioning. (Of course, the DBStatistics component fixes this problem with regards to its own **MaxValue** and **MinValue** properties.)

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Poly: The Function That Got Away

While I was preparing the complete list of the Math unit's functions presented at the end of this chapter, there was one function that was intentionally omitted. Why? Because Borland didn't document it either! So it isn't clear whether it's "really" supposed to be in the Math unit at all. (In fact, the only way anyone would ever find it is if they were studying the source for the Math unit while preparing a chapter for *KickAss Delphi*....)

What's this mysterious function all about? Here it is:

```
function Poly(X: Extended; const Coefficients: array of Double):
  Extended;
```

Basically, **Poly** solves polynomial equations for you. The only limitation is that they must be polynomials in *one* variable. You pass **Poly** two pieces of information: a value, X, at which you want the polynomial evaluated, and the coefficients of the polynomial's terms. The coefficients should be ordered in increasing powers of X.

What this means is that if you have a polynomial such as

$$4x^4 [+ 0x^3] - x^2 + 3x + 34$$

the values in your coefficients array should be ordered like this:

```
34, 3, -1, 0, 4.
```

If you were to put an interface on the **Poly** function, you'd probably end up with something like Figure 16.2, which shows PolyProject running.

FIGURE 9.2 Graphing a function with Poly.

PolyProject (on the CD-ROM for this chapter) is just such an application. It allows the user to enter a polynomial, and then actually graphs it. You'll notice that the PolyProject's main window has a few Edit controls, along with some Labels, set up so that the user can enter coefficients for a generic polynomial. However, all of the significant work of the PolyProject is done in the OnClick event of the "Solve!" Button:

```
procedure TForm1.SolveButtonClick(Sender: TObject);
var
    i : Integer;
    XCoes : array[0..4] of double;
    X,Y, OffsetX, OffsetY : Integer;
    NewRect: TRect;
```

Our first job is to fill the **XCoes** array with the coefficient values the user has entered:

```
begin
  XCoes[0] := StrToFloat(TxtX0.Text);
  XCoes[1] := StrToFloat(TxtX1.Text);
  XCoes[2] := StrToFloat(TxtX2.Text);
  XCoes[3] := StrToFloat(TxtX3.Text);
  XCoes[4] := StrToFloat(TxtX4.Text);
...
```

That done, we now have to set the center for our graphing area. I prefer to use the center of the **TImage** control, but you could just as easily use any point you like:

```
OffsetX := Image1.Width div 2;
OffsetY := Image1.Height div 2;
```

Next, we initialize our X-coordinate value and clear the graph by essentially filling **Image1** with a solid white rectangle. Then, we reset **Image1.Canvas.Brush.Color** so that our graph will be drawn in black:

```
X := 0; {Initialize X just to be safe}
NewRect := Rect(0, 0, Image1.Width, Image1.Height);
Image1.Canvas.Brush.Color := clWhite;
Image1.Canvas.FillRect(NewRect);
Image1.Canvas.Brush.Color := clBlack;
```

Here's where the calculations are done. We first generate a starting point for the line. Note that we don't actually draw anything until we're inside the **for** loop. This is because we need a point at which to position the "pen" before we do any serious drawing. So we call the **Poly** function, passing the value contained in the lower bound range edit box (**TxtRangeStart**) and the **XCoes** array:

```
with Image1.Canvas do
  begin
    Y := Trunc(Poly(StrToInt(TxtRangeStart.Text), XCoes));
...
```

You may be surprised to see that I've truncated the result of the **Poly** function. This is for graphing purposes only; although the **Poly** function returns a floating point (**Extended** type) value, the Windows API only understands integer coordinates.

Next, we convert the result we just got to an (X, Y) pair that corresponds to our center, and we move the "pen" to that point:

```
X := StrToInt(TxtRangeStart.Text) + OffsetX;
Y := OffsetY - Y;
MoveTo(X,Y);
```

We then use a **for** loop to iterate through the X-range values, one at a time, starting with the lower bound plus 1 (we just did the lower bound value a moment ago), and continuing through to the value contained in the upper bound edit box (**TxtRangeEnd**):

```
for i := StrToInt(TxtRangeStart.Text) + 1 to
StrToInt(TxtRangeEnd.Text) do
  begin
    Y := Trunc(Poly(I, XCoes));
    X := I + OffsetX;
    Y := OffsetY - Y;
    LineTo(X,Y);
    MoveTo(X,Y);
  end;
```

While **Poly** appears to be the only *publicly* available, undocumented function in the Math unit, there are a few routines declared in the **implementation** section that look interesting. (One actually determines if value A is "small" relative to value B!) While they can't be discussed here due to space constraints, I'd recommend reading them over if you've got the source code—and the patience, of course!

Filling the Pascal Power Gap

The Math unit introduces something Pascal has never had: an official power (exponent) function. In fact, the Math unit contains *two* power functions.

The first power function, **Power**, takes two **Extended** type values; one is the base, the other is the exponent. The second power function, **IntPower**, takes one **Extended** type parameter and one integer parameter; the **Extended** value is the base, the integer is the exponent.

The difference between the two routines is that **IntPower**, like many of the Math unit's routines, is coded entirely in Pentium FPU optimized assembler, making it quite fast.

If you're concerned about which one your app should use, don't be.

Although it's not explained or documented, **Power** itself will decide whether the exponent is an integer. If it is, **Power** calls **IntPower**. If, on the other hand, the exponent is a true floating point value, **Power** uses the natural log/base *e* method of calculating its result.

Math Unit Function Summary

To close out this chapter, I'll present a complete list of the functions and procedures presented in the Math unit. Most of them are reasonably self-explanatory. Those that aren't—well, sit yourself down and do some sleuthing. Not all Delphi knowledge will be handed to you on a silver-plated Help screen!

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Trigonometric Functions and Procedures

ArcCos

Inverse Cosine.

ArcCosh

Inverse Hyperbolic Cosine.

ArcSin

Inverse Sine.

ArcSinh

Inverse Hyperbolic Sine.

ArcTanh

Inverse Hyperbolic Tangent.

ArcTan2

Inverse Tangent with corrections for the proper quadrant. (The function can be found in the System unit.)

Cosh

Hyperbolic Cosine.

Cotan

Cotangent.

CycleToRad

Converts Cycles to Radians.

DegToRad

Converts a measure in degrees to radians..

GradToRad

Converts Grads to Radians.

Hypot

Calculates the length of the hypotenuse of a triangle given the length of the two sides.

RadToDeg

Converts a measure in radians to degrees.

RadToCycle

Converts Radians to Cycles.

RadToGrad

Converts Radians to Grads.

SinCos

Calculates both the Sine and Cosine for a given angle. Just as with **SumAndSquares** and **MeanAndStdDev**, it is faster to generate both values at once.

Sinh

Hyperbolic Sine.

Tan

Tangent.

Tanh

Hyperbolic Tangent.

Arithmetic Functions and Procedures

Ceil

Rounds up.

Floor

Rounds down.

Frexp

Calculates the Mantissa and exponent of a given value.

IntPower

Raises a number to an integral power. If you’re not going to use Float-based exponents, this function is to be highly recommended because of its speed.

Ldexp

Calculates X times 2 to a given power.

LnXP1

Calculates the natural log of X + 1. This is intended for estimating Xs close to zero.

Log10

Calculates the natural log of X.

Log2

Calculates the log in base two of X.

LogN

Given a base and an X value, calculates the base N log of X.

Power

Raises a number to an exponent. This one isn't as fast as IntPower, but if you need Float-type exponents, it'll do quite well.

Financial Functions and Procedures

DoubleDecliningBalance

Calculates depreciation using the double-declining balance formula.

FutureValue

Calculates the future value of an investment.

InterestPayment

Calculates the amount of the loan payment that is interest.

InterestRate

Calculates the interest rate necessary to earn a specified amount.

InternalRateOfReturn

Calculates a rate of return based on cash flow data.

NumberOfPeriods

Calculates the number of periods for an investment to reach a specified value.

NetPresentValue

Calculates the current value of a set of cash flow values, taking the interest rate into account.

Payment

Calculates the amount of Loan payments, given the amount borrowed, the interest rate, and present and future values of the investment.

PeriodPayment

Calculates the amount of the loan payment that is principal.

PresentValue

Calculates the present value of an investment.

SLNDepreciation

Calculates the Straight Line Depreciation of an investment.

SYDDepreciation

Calculates the Sum-of-Years-Digits Depreciation.

Statistical Functions and Procedures

MaxValue

Is supposed to find the largest value in a set of data, but instead finds *smallest* value.

Mean

Calculates the Mean for a set of values.

MeanAndStdDev

Calculates *both* the Mean and standard deviation for a set of data. This is faster than calculating each value separately.

MinValue

Is supposed to find the smallest value in a set of data, but instead finds *largest* value.

MomentSkewKurtosis

Calculates the skew and kurtosis for a given set of data.

Norm

Calculates the norm for a given set of data. The norm is the square root of the sum of squares.

PopnStdDev

Calculates the population standard deviation, which differs from the standard deviation in that it uses the population variance (**PopnVariance**), defined below.

PopnVariance

Calculates the population variance for a given set of data. This routine uses the TotalVariance / n or “biased” formula.

RandG

Generates “fake” data points given a mean and a standard deviation.

StdDev

Calculates the standard deviation for a set of data.

Sum

Calculates the sum of a set of values.

SumsAndSquares

Calculates *both* the sum and the sum of squares for a given set of data. As with several of the Math unit’s routines, it is faster to generate these two results at once, rather than first generating the Sum and then the sum of squares.

SumOfSquares

Calculates the sum of the squares of a set of values.

TotalVariance

Calculates the total variance for a given set of data. Total variance is the sum of every value’s distance from the mean squared.

Variance

Calculates the sample variance for a given set of data. This routine uses the TotalVariance / (n-1) or “unbiased” formula.

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Previous | Table of Contents | Next

CHAPTER 10
Dynamic User Interfaces

TERENCE GOGGIN

- Moving Controls At Runtime
- Re-sizing Controls At Runtime
- Changing Tab Order At Runtime
- Changing Control Fonts At Runtime
- Building In A Runtime Object Inspector
- Saving Runtime Component Changes

If your users don't like the user interface you hand them, well, let them eat icons! No, wait, let them cook up an interface they *can* swallow....

Let's face it: No two people look at data in the same way. After all, if everybody had the same idea of how data should be presented, there would only be one "Personal Information Manager" program. But that's not the case; the world is virtually littered with PIMs of all shapes and sizes.

Some PIMs are lucky accidents in interface design and receive immediate widespread acceptance. Others are difficult to use and probably only intuitive to their creators. The problem seems to be that there's no in-between.

Occasionally, users will find one of these difficult-to-use programs so beneficial that they will force themselves to use it, no matter how difficult a task learning it might be. However, you can't count on this when you're designing a program. Mostly, you can count on complaints.

A perfect example of this is the Word 6.0 toolbar. Maybe the purpose of those curvy arrow buttons has always been clear to you. On the other hand, you might think the whole thing is far too cluttered and confusing. Again, there's really no middle ground: It's either intuitive or it isn't.

Since the goal of any software company is, in the end, to sell as many units as possible, the designers of graphical interfaces can't ignore these "everything is in the wrong place" customers—yet they can't alter the entire design to suit the quirks of individual end users.

Until now, this problem hasn't really been properly addressed. *No one has attempted a do-it-yourself design for end users*. But, with Delphi 2.0, a little bit of creative coding, and a runtime Object Inspector, you can!

First, we'll take a look at what a simple database application with dynamic design capabilities might look like. Then, we can examine some of the mechanics that make this on-the-fly user interface design possible in the first place.

An Example "UI-It-Yourself" Application

Figure 10.1 is a "composite" screen shot of a sample database application that shows all of the options you can offer to the end user.

FIGURE 10.1 Custom UI options.

The selected item from the form's main menu offers three options:

- Adjust All Fonts (pick a new font for all of the controls;
- Tab Order (set the Tab Orders of the controls); and
- Show Properties (show the Object Inspector).

There's also a popup menu that offers the ability to change the form's background color.

In addition, there is another popup menu that offers four choices:

- Escape/No changes (cancels possible changes);
- Adjust Size & Position (re-size and move the control);
- Change Font (change the individual control's font); and
- View Properties (show the Object Inspector).

Every control's **PopupMenu** property points to this popup menu.

On the left side of the screen is a runtime Object Inspector that allows users to view and change some of the additional properties of the controls.

The greatest part about this dynamic, do-it-yourself interface is that there's a simple "jumping-off point" boilerplate project included on the CD-ROM (Starter.Dpr). You can add this project to your Object Repository and simply

draw on it when you need to. It’s that easy!

As you can see from this first example UI, we’ll end up capturing many of Delphi’s design-time features and making them available to the end user at runtime.

Building-in a “Delphi” for Your Users

When you design a program using Delphi, you need a few tools to define the look and feel you want. Similarly, users will need the following tools to help them define the look and feel *they* want:

- A way to move controls at runtime;
- A way to resize controls at runtime;
- A way to change the Tab Order of the controls as they move them around;
- A way to change additional properties of controls, such as color or border style, and
- A way to automatically save & load the changes they’ve made.

Of course, it’s vital that we make each of these tools fast, simple, and easy-to-use. What we want is for our users to have much (if not all) of the same flexibility that we programmers have at design-time. In a sense, we’re giving them their own, somewhat limited version of Delphi in our apps, to be used at runtime. Explaining how to accomplish each of the points listed above is what this chapter is about.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Moving Controls

While there are several ways to move controls at runtime, the best method for our purposes involves an almost undocumented **WM_SYSCOMMAND** trick. In order to move a **TWinControl**, you call **ReleaseCapture** and send the control a **WM_SYSCOMMAND** message, specifying the literal $F012 as the **wParam** parameter. Or, in code terms:

```
ReleaseCapture;
SendMessage(SomeWinControl.Handle, WM_SYSCOMMAND, $F012, 0);
```

The results of this code, as seen, on the screen, are shown in Figure 10.2.

FIGURE 10.2 Moving a Windows button.

Visually, the effect is the same as moving a modal dialog box—a thin, dotted outline of the control follows the cursor until the mouse button is released.

As you may have noticed, however, this method does have a limitation: It requires a window handle to work. WinControl components have a window handle, GraphicControl components do not. Therefore, components of type **TGraphicControl** (such as **TLabel**) are not supported. Obviously, for our dynamic forms solution to be truly useful and complete, we must find a way to support GraphicControl components.

In order to get around this, then, we must enhance the **WM_SYS COMMAND** mechanism just described. Of course, **WM_SYSCOMMAND** can't be used on GraphicControl components directly, but there *is* a way to fake it by creating a transparent **TWinControl** that will sit above the control being moved.

When a user selects the Adjust Size & Position popup menu option in the sample application, we'll place our transparent **TWinControl** above the selected control. The user will be able to drag the transparent control around (via **WM_SYSCOMMAND** using the

$F012 parameter) as if it were the “selected” control. In other words, when the user clicks the “selected” control and begins to drag it somewhere, they’ll really be dragging our transparent **TWinControl**. Then, when the user opts to keep the changes they’ve made (again by selecting the Adjust Size & Position option), we’ll hide the transparent **TWinControl** and programmatically move the “selected” control to the new location.

In fact, this is exactly what Delphi does at design-time. If you look closely, you can see that when you drag a control around in Delphi, you’re really dragging a thick-bordered transparent rectangle. This is shown in Figure 10.3.

FIGURE 10.3 Dragging a control within Delphi at design-time.

In essence, this transparent rectangle appears above the control you wish to move. From the time you click the “selected” control until you let the mouse button up, the transparent rectangle follows your cursor. As soon as you let the mouse button up, though, the transparent rectangle disappears, and the control you intended to move jumps to the new location.

We will create our own transparent **TWinControl** derivative, called the SizingRect component, of class **TSizingRect.** Again, the **TSizingRect** class will serve as a control’s stand-in while that control is being dragged around.

The important methods of class **TSizingRect** are **CreateParams** and **Paint.** The **CreateParams** method is used to set up certain behavioral aspects of the control before it’s actually created. We’ll use the **CreateParams** method to make our control transparent, as shown in Listing 10.1.

Listing 10.1 The TSizingRect.CreateParams Method

```
procedure TSizingRect.CreateParams(var Params: TCreateParams);
begin
      inherited CreateParams(Params);
      Params.ExStyle := Params.ExStyle + WS_EX_TRANSPARENT;
end;
```

The **Paint** method, shown in Listing 10.2, draws the thick rectangle border our users will see when a SizingRect is being used. When drawing this 3-pixel wide rectangle, we set the canvas’s **Pen.Mode** property to **pmNot**. This ensures that, just as with Delphi’s re-sizing mechanism, the rectangle drawn will always be a different color than the form.

Listing 10.2 The TSizingRect.Paint Method

```
procedure TSizingRect.Paint;
begin
      inherited Paint;
      if fVisible = False then
         Exit;
      Canvas.Pen.Mode := pmNot;
      Canvas.Pen.Width := 3;
      Canvas.Brush.Style := bsClear;
      Canvas.Rectangle(0, 0, Width, Height);
```

```
end;
```

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Re-sizing Controls

Re-sizing controls is even easier than moving them. Again, we'll use Delphi's design-time mechanisms as a guide. Delphi allows you to re-size a selected control by clicking on one of the six black handles at the control's edges, and dragging that handle until the altered size of the control suits you.

We'll take the same approach to our runtime re-sizing. The only difference is that for the sake of simplicity (and to reduce the amount of coding needed) we'll limit ourselves to just one of the six possible re-sizing handles.

Since we're already using the **TSizingRect** class to manage the movement of the control, we'll use it to manage the re-sizing of controls as well. We'll simply designate the lower right-hand corner of **TSizingRect** to be a "hot spot" where users can click to re-size the control.

Also, to make this design easier to use, we'll indicate which corner is "hot" by drawing a small white box there, and by changing the cursor every time the mouse passes over the white box. The action happens in the handler for the MouseMove event, which is shown in its entirety in Listing 10.3. I'll be discussing the handler piece by piece in the text that follows.

Listing 10.3 SizingRect's MouseMove Event Handler

```
procedure TFrmMain.SizingRect1MouseMove(Sender: TObject;
                                        Shift: TShiftState;
                                        X, Y: Integer);
begin
{ControlDC and ControlRect are globally declared variables used
 by several procedures. These two lines, then, are just setup.}
  ControlDC := GetDC(TWinControl(Sender).Handle);
  GetWindowRect(TWinControl(Sender).Handle, ControlRect);

  if ((X > TControl(Sender).Width -SizeVal) and
     (Y > TControl(Sender).Height -SizeVal)) then
  begin
    TWinControl(Sender).Cursor := crSizeNWSE;
    Rectangle(ControlDC, TWinControl(Sender).Width - SizeVal,
              TControl(Sender).Height -SizeVal,
```

```
                TControl(Sender).Width, TControl(Sender).Height);

  end
    else
  begin
    TWinControl(Sender).Cursor := crDefault;
  end;
  if ((TWinControl(Sender).Cursor = crSizeNWSE) and
     (ssLeft in Shift)) then
  begin
    TWinControl(Sender).Width := X;
    TWinControl(Sender).Height := Y;
  end;
end;
```

After some initial setup, the handler tests to see if the mouse is within the sizing area. The **SizeVal** constant, which determines the area of the white sizing box, is defined in the **TSizingRect** unit. If the mouse is in the sizing area, the handler changes the cursor to the appropriate "sizing" cursor, and, of course, draws the rectangle.

```
if ((X > TControl(Sender).Width -SizeVal) and
    (Y > TControl(Sender).Height -SizeVal)) then
  begin
    TWinControl(Sender).Cursor := crSizeNWSE;
    Rectangle(ControlDC,
              TWinControl(Sender).Width - SizeVal,
              TControl(Sender).Height - SizeVal,
              TControl(Sender).Width, Control(Sender).Height);
  end
```

If the mouse is outside the sizing area, the routine simply changes the cursor back to the default style.

```
else
  begin
    TWinControl(Sender).Cursor := crDefault;
  end;
```

Finally, we check to see if the user is still re-sizing the control: If the cursor is equal to **crSizeNWSE** and the left mouse button is depressed, then the user is still re-sizing. The routine then sets the control's bottom right corner to the mouse position.

```
if ((TWinControl(Sender).Cursor = crSizeNWSE) and
(ssLeft in Shift)) then
  begin
    TWinControl(Sender).Width := X;
    TWinControl(Sender).Height := Y;
  end;
end;
```

The overall effect here is that as long as the mouse is depressed and the cursor is in the hot spot, the sizing corner will follow the mouse's movements.

Responding to the Popup Menu

In our sample app, the **TSizingRect** component is activated by **PopupMenu1**, which is set as the PopupMenu for each of the controls on the form. Figure 10.4 shows **PopupMenu1** at runtime, after a

user has right-clicked the DBImage.

FIGURE 10.4 The popup menu for a right mouse click.

At this point, the user can make the following choices;

- Do nothing (by selecting the Escape/No changes item);
- Re-size or move the control (the Adjust Size & Position item);
- Change the control's font (Change Font); or
- Display the MiniInspector (View Properties).

Selecting the Adjust Size & Position item invokes the **TFrmMain.AdjustClick** procedure, shown in Listing 10.4.

Listing 10.4 The Adjust Size & Position OnClick Handler

```
procedure TFrmMain.AdjustClick(Sender: TObject);
begin
  if (Adjust.Checked = True) then
    begin
      if ((PopupMenu1.PopupComponent <> ComponentBeingAdjusted) and
          (PopupMenu1.PopupComponent <> SizingRect1)) then
        begin
           MessageDlg('You can only adjust one element at a time.' +
           #13#10 +
           'Please unselect the current element before continuing.',
           mtWarning, [mbOK], 0);
           Exit;
        end;
      Adjust.Checked := False;
      With TWinControl(ComponentBeingAdjusted) do
        begin
          Top := SizingRect1.Top;
          Left := SizingRect1.Left;
          Width := SizingRect1.Width;
          Height := SizingRect1.Height;
        end;
      SizingRect1.Cursor := crDefault;
      SizingRect1.Visible := False;
      SizingRect1.Top := -40;
      SizingRect1.Left := -40;
      SizingRect1.Width := 40;
      SizingRect1.Height := 40;
      MiniInspector1.ShowThisComponent(ComponentBeingAdjusted);
      ComponentBeingAdjusted := Self; { i.e., no control is }
    end                               { currently selected. )
  else
    begin
      if ((ComponentBeingAdjusted <> Self) and
          (PopupMenu1.PopupComponent <> ComponentBeingAdjusted))
```

```
          then
         begin
          MessageDlg('You can only adjust one element at a time.'
          + #13#10 +
          'Please unselect the current element before continuing.',
          mtWarning, [mbOK], 0);
          Exit;
         end;
       Adjust.Checked := True;
       ComponentBeingAdjusted := PopupMenu1.PopupComponent;
       With TWinControl(PopupMenu1.PopupComponent) do
         begin
          SizingRect1.Top := Top;
          SizingRect1.Left := Left;
          SizingRect1.Width := Width;
          SizingRect1.Height := Height;
         end;
       SizingRect1.Visible := True;
       MiniInspector1.ShowThisComponent(ComponentBeingAdjusted);
    end;
end;
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Once various checks are performed to be sure the operation is legal, the **TSizingRect** is moved to the same position as the control being adjusted. (The variable **ComponentBeingAdjusted** exists for the benefit of additional routines that may not be able to rely on **PopupMenu1.PopupComponent**.) It's done like this:

```
ComponentBeingAdjusted := PopupMenu1.PopupComponent;
With TWinControl(PopupMenu1.PopupComponent) do
  begin
    SizingRect1.Top := Top;
    SizingRect1.Left := Left;
    SizingRect1.Width := Width;
    SizingRect1.Height := Height;
  end;
SizingRect1.Visible := True;
MiniInspector1.ShowThisComponent(ComponentBeingAdjusted);
```

At this point, the SizingRect component is active and can be moved and re-sized by the mouse, as shown in Figure 10.5.

FIGURE 10.5 The SizingRect's rectangle.

When the user has finished adjusting the control, they can again right click the control to save or discard their changes, as shown in Figure 10.6.

FIGURE 10.6 Saving or discarding changes.

If the user does wish to move ahead with the adjustment, and thus selects the second option, the control being adjusted is moved and re-sized to the new specifications, after which the SizingRect is again hidden. (This code is also part of **TFrmMain.AdjustClick**).

```
With TWinControl(ComponentBeingAdjusted) do
    begin
      Top := SizingRect1.Top;
      Left := SizingRect1.Left;
      Width := SizingRect1.Width;
      Height := SizingRect1.Height;
    end;
SizingRect1.Cursor := crDefault;
SizingRect1.Visible := False;
SizingRect1.Top := -40;
SizingRect1.Left := -40;
{…}
```

Abandoning Changes

If the user doesn't want to keep any changes, and thus chooses the first popup menu option, the **TSizingRect** is hidden and the selected control remains unchanged. This is done in **TFrmMain.Escape1Click**, as shown in Listing 10.5.

Listing 10.5 The Escape/No Changes OnClick Handler

```
procedure TFrmMain.Escape1Click(Sender: TObject);
begin
  {if we were just now resizing a component, then...}
  if (Adjust.Checked = True) then
    begin
      Adjust.Checked := False;
      SizingRect1.Cursor := crDefault;
      SizingRect1.Visible := False;
      SizingRect1.Top := -40;
      SizingRect1.Left := -40;
      SizingRect1.Width := 40;
      SizingRect1.Height := 40;
      ComponentBeingAdjusted := Self; {i.e., no control is  }
    end;                              { currently selected. }
end;
```

NOTE:
In the Starter.Dpr project, the SizingRect component is hidden off to the top right of the main form to prevent it from being accidentally shown at the wrong time. When using the project as a starting point for your own apps, be sure to locate the SizingRect component and, from Delphi's main menu, select Edit|Bring To Front after you have added all of your own controls. Also, be sure that all of the controls' PopupMenu properties point to **PopupMenu1**.

Changing the Tab Order at Runtime

If users are going to move controls around, chances are they'll want to be able to adjust the tab order as well. After all, our do-it-yourself design would completely fail the ease-of-use test if users were locked into the original tab order. Navigating from one control to another would be a very confusing task.

In Delphi, you set the tab order of the controls via the Tab Order dialog box, which consists primarily of a ListBox component, an up-arrow button, and a down-arrow button. Since this seems to work well for Delphi, we'll use the same kind of dialog box in our system. Figure 10.7 shows our FrmTabOrder component at runtime.

FIGURE 10.7 The FrmTabOrder component at runtime.

FrmTabOrder itself, however, is just a friendly interface. What actually manages the tab order is the code that shows FrmTabOrder. This is the **TFrmMain.TabOrder1Click** method, shown in Listing 10.6. I'll discuss this handler in detail next.

Listing 10.6 The tab order menu item's OnClick Handler

```
procedure TFrmMain.TabOrder1Click(Sender: TObject);
var
   i : Integer;
begin
  FrmTabOrder.LBControls.Items.Clear;
  for i := 0 to ComponentCount -1 do
  begin
    if ((Components[i] is TWinControl) and
       not (Components[i] is TSizingRect)) then
       FrmTabOrder.LBControls.Items.Add(Components[i].Name);
  end;
  FrmTabOrder.ShowModal;
  if FrmTabOrder.ModalResult = mrOK then
  begin
    for i := 0 to FrmTabOrder.LbControls.Items.Count -1 do
      TWinControl(FindComponent(
                  FrmTabOrder.LbControls.Items[i])).TabOrder := i;
  end;
end;
```

Now, in detail. First, the routine clears the list box. Then it iterates through all of the form's controls, adding every control that's a **TWinControl** to the list box, except for the SizingRect.

```
FrmTabOrder.LBControls.Items.Clear;
for I := 0 to ComponentCount -1 do
  begin
    if ((Components[I] is TWinControl) and
    not (Components[I] is TSizingRect)) then
       FrmTabOrder.LBControls.Items.Add(Components[I].Name);
  end;
```

Next, the routine shows the form. (**FrmTabOrder.LBControls** handles the re-ordering of the items itself.) If the user clicks the OK button, our program goes through **FrmTabOrder.LBControls.Items**, extracting the index of each item, assigning that value to the related control's **TabOrder** property.

```
FrmTabOrder.ShowModal;
if FrmTabOrder.ModalResult = mrOK then
   begin
     for I := 0 to FrmTabOrder.LbControls.Items.Count -1 do
       TWinControl(FindComponent(
                   FrmTabOrder.LbControls.Items[I])).TabOrder := I;
   end;
```

Pretty simple, isn't it? That's all there is to runtime control of component tab order.

Changing Other Properties

Now we've got to tackle the issue of changing some of the other properties of the controls on the form. For instance, what if a user wanted to change the font or color of some DBEdit components to indicate that they were required fields? Well, there are a few simple things that we can do for some of these additional properties. As we've just discovered, it's relatively easy to change the tab order for all of the controls. It's also quite simple to change other individual properties of a control.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Changing Control Fonts at Runtime

For instance, in our sample app, we offer the user the ability to change every control's character font via a main menu item, Adjust All Fonts. The code to do this is relatively simple, as shown in Listing 10.7.

Listing 10.7 Changing fonts of all controls on a form

```
procedure TFrmMain.AdjustMenu2Click(Sender: TObject);
var
  i : Integer;
begin
  {multiple Font changes}
  if FontDialog1.Execute then
    begin
      for i := 0 to ComponentCount - 1 do
        begin
          try
            if ((Components[i] is TWinControl) or
               (Components[i] is TGraphicControl)) and
                not ((Components[i] is TMenu) and
               (Components[i] is TMenuItem)) then
                  TMagic(Components[i]).Font := FontDialog1.Font;
          except
            Continue;
          end;
        end;
    end;
end;
```

Something interesting is going on here. Note the **TMagic** typecast in the actual assignment statement. **TMagic** is a control class defined in the **TSizingRect** unit that does absolutely nothing from a code action standpoint. The only reason it exists at all is to publish certain protected properties; namely, the **Font** property. Since the **Font** property of most controls is protected, we can't change it at runtime without performing just such a typecast. Therefore,

TMagic makes it possible to change a control's font at runtime.

Our sample application also offers users the ability to change a single control's font via the Change Font option of the popup menu. This, too, is a relatively simple process, as shown in Listing 10.8.

Listing 10.8 Changing the font for a single control at runtime

```
procedure TFrmMain.ChangeFont1Click(Sender: TObject);
begin
  if FontDialog1.Execute then
    try
      TMagic(PopupMenu1.PopupComponent).Font := FontDialog1.Font;
    except
      Exit;
    end;
end;
```

NOTE:
Even with the help of **TMagic**, there are some control classes—for instance, **TMenu**—that will raise an exception if you try to change their fonts, "magic" typecast or not. Therefore, it's usually best to test the type of the control before attempting to change its font. In the above example, though, there was no need to filter out the "no-font" controls. This is because the code is launched via a popup menu. The controls that have a popup menu property do not object if you change their fonts—even if they don't display any text. (An example of this would be a ScrollBar component.)

Changing Properties in an Object Inspector

We must now provide the users with a way to edit the additional properties **Caption**, **CharCase**, or **Color**, for instance. It's not unreasonable for a user to expect the ability to change any of these items, considering that we're letting them change everything else.

How does Delphi allow us to change these properties at design-time? Through the Object Inspector. So, for our project, we'll use an Object Inspector of our own.

NOTE:
Because the Object Inspector presented in this chapter has been previously available as a commercial product, only a demo version (that is, without source code) is provided on the CD-ROM. Basically, this version limits the type of properties and controls available. However, it is otherwise fully functional, and there are no "nag screens". Please consult the licensing agreement for more details. Information about the full version of the **TMiniInspector** class, which includes all source code, can be found on the CD-ROM accompanying this book, or by clicking the MiniInspector component's **About_This_Component** property at design-time.

To install **TMiniInspector** to your component palette, choose Components|Install and add **MiniOI.DCU**. You will also want to ensure that the following three files are in the same directory as **MiniOI.DCU**:

- OICompDemo.DCU
- OICompDemo.DFM
- MiniOI.DCR

The Object Inspector in our project works exactly like the Object Inspector supplied with Delphi. The user selects a component via the combo box at the top. He or she can then edit the properties by either typing in a value or by clicking on a button to select a separate property editing form, where one is available. Figure 10.8 shows **TMiniInspector** at runtime.

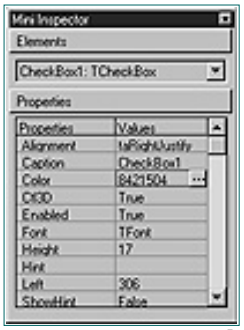

FIGURE 10.8 The MiniInspector component at runtime.

In our sample app, when the user selects either the **Show Properties** option from the main menu, or the **View Properties** option from the popup menu, we display the Object Inspector with a simple call to its **Show** method.

```
MiniInspector1.Show;
```

Then, if the user activated the Object Inspector via the popup menu, we'll tell the **TMiniInspector** component to show the control that was right-clicked in the first place.

```
if PopupMenu1.PopupComponent <> nil then
 MiniInspector1.ShowThisComponent(PopupMenu1.PopupComponent);
```

The **ShowThisComponenent** method is a function that takes a **TComponent** parameter and returns a boolean value. If the component type passed to the function can be found in the combo box, it will be displayed and the function will return a value of **True**. If the component can't be found, or if the MiniInspector is not visible, the function will return a value of **False**.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Saving Component Changes Made at Runtime

Now that we have methods for changing just about all of the elements of a user interface, we'd better come up with a way to save the changes so that they can become "persistent" across sessions. Users will *not* be amused if they're forced to alter the UI each time they run the application! While we might be tempted to just plunge in and use .INI files—or maybe the Windows 95 Registry if we're brave enough—there are some serious drawbacks to both of these approaches. The problem is that since each component has numerous properties of different types, there's no straightforward way to write a generic "**Save_This_Component**" method.

If we tried, we'd probably end up testing each component's type, and then saving certain properties based on that type. This isn't very efficient, you'll agree. On the other hand, we could decide to save only those properties which are common to all components. Since the **TComponent** type—the ancestor of all other component types—has only nine properties (not including the **Left**, **Top**, **Width,** and **Height** properties), this, too, would be useless.

However, all is not lost! There are several very good mechanisms for saving and loading the properties of components. All you have to do is dig through Borland's documentation and experiment a little bit.

In this expedition into the documentation, our final destination is the **TFiler** / **TWriter** / **TReader** family of objects. According to the Delphi 2 help files, **TFiler** is "the abstract base object for the reader objects and writer objects that Delphi uses for saving (and loading) forms and components in form files."

That definition says something very important to us; namely, that the **TWriter** and **TReader** objects can be used to save and load a component's properties from a file. By associating an instance of the **TWriter** or **TReader** class with an instance of **TFileStream**, we can, in fact, use the **WriteRootComponent** and **ReadRootComponent** methods to do exactly what we want.

Snag: Components with Components as Properties

The only limitation of these methods is that there are a few types of components that cannot be saved directly. The components in question are those that have *other* components as properties.

The problem occurs when trying to load these component properties from the file. Since these components are saved on their own, loading them as *properties of another component* raises an exception with a message to the effect of "A component named Widget1 already exists."

Thankfully, this behavior seems to be limited to only four types of components: **TMainMenu**, **TMenuItem, TPopupMenu**, and **TForm**.

For our purposes, the first three don't really matter. However, we probably should give users the power to save some of the form's properties. Since there aren't likely to be that many properties we really need to be concerned with, it's simpler to save the ones we *do* care about.

In Listing 10.9 is the code to save a form's properties, as executed in the FormCloseQuery event. As is my style, I'll discuss pertinent portions of the event handler in detail shortly.

Listing 10.9 The event handler for the FormCloseQuery event

```
procedure TFrmMain.FormCloseQuery(Sender: TObject;
                                  var CanClose: Boolean);
var
  Writer : TWriter;
  FileStream : TFileStream;
  i : Integer;
  TempRect : TRect;

begin
  {Just to be on the safe side, we delete the old *.KAD file:}
  DeleteFile(ExtractFilePath(Application.ExeName) +
             TObject(Self).ClassName + '.KAD');
  {Then, go ahead & re-write it:}
  FileStream :=
    TFileStream.Create(ExtractFilePath(Application.ExeName)
    + TObject(Self).ClassName + '.KAD',fmOpenWrite or fmCreate);
  for i := 0 to ComponentCount-1 do
    begin
      {There are some controls that should not (or cannot) be
      saved in this manner. Luckily, they're ones you're not likely to
      want to save anyhow:}
      if ((Components[i] is TSizingRect) or
          (Components[i] is TMenu) or
          (Components[i] is TMenuItem) or
          (Components[i] is TPopupMenu) or
          (not(Components[i] is TControl))) then
        Continue;
      Writer := TWriter.Create(FileStream, SizeOf(Components[i]));
      Writer.WriteRootComponent(Components[i]);
      Writer.Free;
    end;
  {Save & Load form properties here:}
  TempRect.Top := Self.Top;
  TempRect.Left := Self.Left;
  TempRect.Bottom := TempRect.Top + Self.Height;
  TempRect.Right := TempRect.Left + Self.Width;
  FileStream.Write(TempRect, SizeOf(TRect));
  FileStream.Write(Self.Color, SizeOf(TColor));
  FileStream.Free;
  {Don't forget to 'OK' the closing of the form!}
  CanClose := True;
end;
```

Now, let's take a detailed look at how all this works. First, just to be on the safe side, we delete the old *.KAD file. Next, we re-create it.

```
FileStream := TFileStream.Create(ExtractFilePath
                                 (Application.ExeName) +
```

```
                                        TObject(Self).ClassName +
                                        '.KAD',fmOpenWrite or fmCreate);
```

Next, we filter out the types that can't be saved, taking no action when those types are encountered.

```
for i := 0 to ComponentCount-1 do
  begin
    if ((Components[i] is TSizingRect) or
        (Components[i] is TMenu) or
        (Components[i] is TMenuItem) or
        (Components[i] is TPopupMenu) or
        (not(Components[i] is TControl))) then
        Continue;
```

Each time we discover a component that can be saved, we save it out to the stream.

```
Writer := TWriter.Create(FileStream, SizeOf(Components[i]));
Writer.WriteRootComponent(Components[i]);
Writer.Free;
```

After iterating through the form's components and saving any pertinent ones, we save any of the form's own properties that are important to the application.

```
TempRect.Top := Self.Top;
TempRect.Left := Self.Left;
TempRect.Bottom := TempRect.Top + Self.Height;
TempRect.Right := TempRect.Left + Self.Width;
FileStream.Write(TempRect, SizeOf(TRect));
FileStream.Write(Self.Color, SizeOf(TColor));
FileStream.Free;
```

Finally, we set the flag allowing the form to be closed.

```
CanClose := True;
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Alternate Paths to a Stream

You may have noticed that the **TFileStream** class also appears to have methods that will save and restore a component's properties. While **TFileStream** has, in fact, two different sets of methods for saving and loading components to and from files, these routines do some additional processing that makes them somewhat less efficient than the **TReader/TWriter** implementation we've chosen.

The **WriteComponentRes** and **ReadComponentRes** methods read and write the component as a standard windows resource. This adds some processing overhead to the equation. In addition, these routines save data that is simply of no use to us, making our properties files larger than they really need to be.

The **WriteComponent** and **ReadComponent** methods also achieve the same end result as we do, but with an additional function call or two. Our method is more efficient and slightly faster.

Toward More Flexible User Interfaces

Runtime-definable user interfaces really deserve more than a single book chapter to treat them in full detail. We've looked at giving users control over the most common UI elements, including control position and size, fonts, and tab order, as well as other properties that may be displayed in a commercial property editor component. That's only scratching the surface, especially when you consider that a really good user interface customization system should itself have a careful UI, as well as guidelines and checks to ensure that the user does not somehow render the application unusable. Consider this chapter as a jumping-off point from which you can explore the matter as far as you (and your users) need to take it.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

CHAPTER 11
Hierarchical Data in Relational Databases

RICHARD HAVEN

Database data isn't always expressed as rows and columns. Here are some pointers on handling hierarchical data within Delphi databases, with some VCL components to shoulder the bulk of the load.

The real world contains lots of hierarchical data. This broad category includes companies, and within companies, divisions, departments, and teams; parts which make up parts which make up parts; specialties, sub-specialties, and skills; employees and supervisors; and on and on. A group of items where one item can own or be the "parent" of any number of other items is organized in a *tree hierarchy*. The VCL object hierarchy is an immediate example: A **TEdit** is a **TControl** because **TEdit** descends from **TControl**. **TEdit** is also a

TWinControl and a **TCustomControl** because it descends from those as well.

Such relationships are not intuitively represented by the relational database model. Often, these hierarchical relationships are recursive (in that any row can own any other row) and arbitrarily nested, which means that any row can own another row no matter who owns it, the prohibitions against circular links notwithstanding. Simple navigation and display of a tree hierarchy can be difficult in the two-dimensional landscape of rows and columns, and querying is especially confusing. You often want to extract an item's *lineage* (the list of its owner and its owner's owner, and so on) or *progeny* (the list of all its descendants and its descendants' descendants) to use as criteria for a query. This chapter will provide some mechanisms for dealing with such hierarchical relationships within the relational database idea familiar to Delphi programmers.

One-to-Many Hierarchies

Delphi deals well with traditional relational databases: These are tables (sometimes called *relations*) with rows (records) and columns (fields) that link to one another by matching common values in columns of different rows. (See Figure 11.1.) To link rows this way is also called *relating* the rows in question. The relational model is not the only model used in database management. The *hierarchical* and *network* database models were standard before the relational model, and the *object-oriented* database model is a modern alternative.

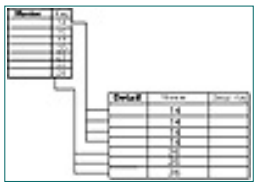

FIGURE 11.1 Master/detail tables.

The value of any of these models lies in how they help the database designer to create databases to fit the data. The relational model fits many kinds of real-world data structures: many invoices to each customer; many parts to many vendors; many members each having many characteristics; and so on. Delphi's **TTable.MasterSource** and **TQuery.DataSource** properties let you show related subsets of a table or query based on linking values in another table. This is one way to establish a *master/detail* relationship within Delphi, where "master" is the "one" in "one-to-many" and "detail" is the "many."

Simple Recursive Hierarchical Data

The relational model works well for master/detail data in the same table as long as there is only a single level of ownership: that is, when any single row is either owned by one other row or owns another row. The boss and staff rows shown in Table 11.1 might as well be in two separate tables at this single level of recursion.

TABLE 11.1 Simple recursive hierarchical data.

Emp_ID	Boss_ID	Emp_Name
Boss 1	<nil>	Frank Eng
Boss 2	<nil>	Sharon Oakstein
Boss 3	<nil>	Charles Willings
Staff 1	Boss 1	Roger Otkin
Staff 2	Boss 1	Marilyn Fionne
Staff 3	Boss 1	Judy Czeglarek
Staff 4	Boss 2	Sean O'Donhail
Staff 5	Boss 3	Karol Klauss
Staff 6	Boss 3	James Riordan

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Table 11.2 summarizes the property values necessary to have two sets of Table, DataSource and DBGrid components, where both sets point to the same physical table. One set will show parent rows while the other set will show the child rows that belong to the currently selected parent row. The **MasterSource** and **MasterFields** properties of the child Table automatically restrict it to the rows belonging to the current row in the parent Table.

TABLE 11.2 Property setup for parent or child row display.

Parent component set	Child Component Set
Table1.TableName = ‘employees’	Table2.TableName = ‘employees’
Table1.IndexFieldName = ‘Boss_ID;Emp_ID’	Table2.MasterSource = ‘DataSource1’
DataSource1.DataSet = ‘Table1’	Table2.MasterFields = ‘Emp_ID’
DBGrid1.DataSource = ‘DataSource1’	Table2.IndexFieldName = ‘Boss_ID;Emp_ID’
Table1.SetRange([‘’],[‘’]);	DataSource2.DataSet = ‘Table2’ DBGrid2.DataSource = ‘DataSource2’

To restrict the parent Table so that it does not show any child rows, place an explicit filter on it so it only shows rows that have nothing in their **Boss_ID** column. These are parent rows.

All properties called out in Table 11.2 are design-time properties, except for **Table1.SetRange**. This is a method that you call to make **Table1** display only boss rows (that is, those where **Boss_ID** = **Nil**). A good place to put the call to **Table1.SetRange** is in the **Table1.AfterOpen** handler, so it executes whether the table is left open at design-time or opened at runtime.

Figure 11.2 shows a Delphi form containing two dbGrid components, set up according to the properties summarized in Table 11.2. The boss rows are shown in one grid, and the staff rows in the other. All rows are, in fact, present in one physical table.

FIGURE 11.2 A Master/Detail relationship between rows of the same table.

Whenever **DataSource1** (connected to **Table1**) changes, **Table2** will automatically set its range just as though you had executed the code in Listing 11.1.

Listing 11.1 Equivalent code for automatic range updates

```
procedure TForm1.DataSource1DataChange(Sender : TObject;
                                       Field : TField);
begin
  if (Field = nil) or (Field.FieldName = 'Emp_ID') then
    Table2.SetRange([Table1.FieldByName('Emp_ID').AsString],
                    [Table1.FieldByName('Emp_ID').AsString]);
end;
```

NOTE:
It can only do this if there is an index on **Table2** for **Boss_ID** so it can filter all the rows where **Table1.Boss_ID** for one grid contains the null string and **Table2.Boss_ID = Table1.Emp_ID** for the other. This index can have extra columns that are not part of the range, which determine the sequence of the rows in the filtered set. In this case, only staff for one boss will appear in **Table2**, and those rows will be in **Emp_ID** sequence. If the table is on a SQL Server, all non-BLOB (Binary Large Objects) columns are considered indexed, although performance will suffer if you use columns that don't actually have physical indexes.

Class TQuery as a Detail DataSet

A **TQuery** can be a detail DataSet as well by letting the master data set (of class **TTable** or **TQuery**) send its values to the **TQuery** as parameters to a dynamic query. In the example above, the **SQL** property of the detail **TQuery** would look something like this:

```
SELECT *
FROM Employee T1
WHERE T1.Boss_ID = :Emp_ID
```

The **TQuery.DataSource** property indicates where the parameter value comes from. In the SQL query shown above, the parameter named **Emp_ID**, gets its value from **TQuery.DataSource.DataSet. FieldByName('Emp_ID')**. Anytime the master field changes, the query re-executes with its new parameter value.

Detail **TQuery** datasets have to use dynamic SQL and the **DataSource** property together. The query is *dynamic* because it uses a parameter instead of recreating the full SQL text each time. However, if there are several possible statement structures for the query and you wish to use more than one, you will have to create the entire SQL statement textually. Parameters will only take you so far.

You can use code instead of the **MasterSource** property if you want more control over how ranges appear or you want to send customized SQL through a **TQuery** instead of letting **TQuery.DataSource** do it for you. For example, you might want to include certain rows besides the ones specified by the range. This is best done when an OnDataChange event handler is attached to the *master* DataSource, as shown in Listing 11.2.

Listing 11.2 Special handling in range updates for values outside the range

```
procedure TForm1.DataSource1DataChange(Sender : TObject; Field : TField);
begin
  if (Field = nil) or (Field.FieldName = 'Emp_ID') then
    begin
      Query2.DisableControls;
      Query2.Close;
      with Query2.SQL do
        begin
          Clear;
          Add('SELECT *');
          Add('FROM employees T1');
          Add('WHERE T1.Boss_ID = ' + Table1.Field
ByName('Emp_ID').AsString);
          Add('OR T1.Boss_ID IS NULL');  {// extra code }
        end;
      Query2.Open;
      Query2.EnableControls;
```

```
    end;
end;
```

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Nested Recursive Hierarchical Data

Recursively hierarchical data means that the owner and the owned rows are all in the same table: One non-key column of a row contains the key value of another row, indicating that that row is owned by the other row. This non-key column is called a *foreign key*, even if it links the row with another row in the same table. The examples described earlier reflect one level of ownership, in that each row is either a boss or a staffer. To allow a staffer to also be a boss, the table becomes fully recursive: Any row can be both *be* a boss and *have* a boss. Note that the table's key is **Emp_ID**. You do not have to add the **Boss_ID** column as part of the key in order to manage the relationship; a secondary index starting with **Boss_ID** will work fine. See Table 11.3.

TABLE 11.3 A recursive table.

Emp_ID	Boss_ID	Emp_Name
	Boss 1	<nil>
	Boss 2	<nil>
	Boss 3	<nil>
	Manager 1	Boss 1
	Manager 2	Boss 1
	Manager 3	Boss 2
	Staff 1	Boss 1
	Staff 2	Manager 1
	Staff 3	Manager 1
	Staff 4	Manager 3
	Staff 5	Boss 3
	Staff 6	Boss 3

This data has three levels: Boss, Manager, and Staff. Rather than adding another DBGrid for

the new level, Form2 (see Figure 11.3) shows two levels at a time. By showing two levels, the data can be arbitrarily nested without changing the visual interface at all.

FIGURE 11.3 A recursive Master/Detail relationship between rows of the same table.

You can change the range of **Table1** so it only shows employees with **Boss_ID** of some chosen value, and **Table2** will dutifully show only details of those rows (for example, the employees working for that manager). This "step" type of interface is fine for shallow hierarchy trees, but when there are many paths to follow, it's easy for a user to get lost. The least you can do is show a label with the lineage, as Figure 11.3 does. The label at the top of the form (**Label1**) shows the lineage of the current record in **Table1**.

Hierarchy Navigation

To navigate up and down the hierarchy, you need to offer two additional utilities. In the example we've been following, the user goes down the branch when they double-click on a detail row of the right-hand (child) grid. This changes the filter on the table connected to the left-hand (parent) grid so that it shows the same rows as the right-hand grid. This new filter then changes the rows in the right-hand grid: it now shows child rows of the current parent row. This is a difficult business to describe in plain text, so Listing 11.3 shows an event handler for the OnDoubleClick event that demonstrates this action. Read the code closely to make sure you understand how it works.

Listing 11.3 Double-click handler for recursive relationship navigation

```
procedure TForm2.DBGrid2DblClick(Sender : TObject);
var
  NewRangeID, SelectedEmployeeID : String;

begin
{ show the user the current range }
  if Table1.FieldByName('Boss_ID').AsString = '' then
    Label1.Caption := Table2.FieldByName('Boss_ID').AsString
  else
    Label1.Caption := Label1.Caption + ':' +
                      Table2.FieldByName('Boss_ID').AsString;

{ Assume that Table1.IndexFieldNames is still Boss_ID;Emp_ID }

  SelectedEmployeeID := Table2.FieldByName('Emp_ID').AsString;
  NewRangeID := Table2.FieldByName('Boss_ID').AsString;
  Table1.SetRange([NewRangeID],[NewRangeID]);
  Table1.FindKey([NewRangeID,SelectedEmployeeID]);
end;

procedure TForm2.UpOneLevelButtonClick(Sender : TObject);
var
  PrevPos : Integer;
```

```
  NewRangeID : String;

begin
{We want to filter on the selected employee's Boss_ID }
  NewRangeID := Table1.FieldByName('Boss_ID').AsString;
  Table1.CancelRange;
  Table1.IndexFieldNames := 'Emp_ID';
  Table1.FindKey([NewRangeID]);
  NewRangeID := Table1.FieldByName('Boss_ID').AsString;
  Table1.IndexFieldNames := 'Boss_ID';

{ This will resynchronize Table2 }
  Table1.SetRange([NewRangeID],[NewRangeID]);

  if Table1.FieldByName('Boss_ID').AsString = '' then
    Label1.Caption := '<Top level>'
  else
    begin
      PrevPos := 0;
      while Pos(':',Copy(Label1.Caption,PrevPos + 1,999))<>0 do
         PrevPos :=
            Pos(':',Copy(Label1.Caption,PrevPos + 1,999)) +
            PrevPos;
      Label1.Caption := Copy(Label1.Caption,1,(PrevPos - 1));
    end;
end;
```

The left-hand grid's table is filtered on the current filter's **Boss_ID** value when the user clicks the Up One Level button. While this allows infinite recursion, it does not allow any easy way to get a list of *all* members of a boss' staff's staff, and so on down the hierarchy. It also gives you no way to get a staff member's boss' boss. You could iterate through the linking values for each level in either direction, but the problem is that it is arbitrary: You don't know how many levels of ownership you will have to go through to get to either end. (In this, of course, it's just like any other tree-shaped data structure.)

But hierarchies like this are useful in allowing a user to pick an item from any level of detail while giving an application the right level of data. For example, you could have geographical regions broken down into smaller and smaller areas, but the application would still want to know what region those areas are in. You might also want to have general categories broken into specialties, but still allow the choice of a general category to include all the specialized ones. For example, choosing all Painters automatically includes Painters:Decorative, Painters:Commercial, Painters:Marine, and so on. This way, you don't have to separately list members of every specialty in a general category to get all of the members of the general category.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Displaying the Data

While there is no clever way to avoid walking up and down the hierarchy tree, you can use tools that do all that walking for you. Think about how your users want to use the data. They may not care if it is hierarchical or not, but they may want to find an item by looking for its ancestor. They may want to search for items by name wherever they appear in the hierarchy, or only in the descendents of the current item. They may want the ID of the item they choose, or the entire lineage or progeny of that item.

For example, a basic question you will confront in building applications is what to do when the user asks for the “next” item. This could be the next sibling of the current item’s parent; it could be the first child; it could be the next parent if there are no siblings; it could even be the first child of the next sibling. With a visual interface, the user’s expectations are going to be based on position, display, and their own actions, not necessarily on a logical protocol based on the application’s abstract data design.

Besides the one-level relationship described earlier involving two DBGrid components, the obvious controls to use with hierarchical data are the Outline and TreeView components, because they were specifically created to show trees of items rather than simple linear lists. They tend to take up a lot of screen area, so you cannot use them everywhere you want the user to choose an item from a hierarchy. They also encourage you to load the entire structure into memory at once (which can be memory wasteful). You *can* configure them to load branches as the user opens those branches, but that flexibility comes with a performance penalty.

An example of how such controls may be loaded is shown in Listing 11.4. You should have a working familiarity with the generation operation of the Outline component before trying to follow this code.

Listing 11.4 Loading an Outline component from a list of items

```
procedure LoadItemStringsFromTop(ListOfItems : TListOfItems);
var
  Counter : Integer;
```

```
procedure LoadOutline(StartIndex : Integer; StartItem : TItem);
var
  NewIndex : Integer;
begin
  NewIndex := MyOutline.AddChildObject(StartIndex,
                                       StartItem.Description,
                                       StartItem);
  if StartItem.FirstChildItem <> nil then
    LoadOutline(NewIndex,StartItem.FirstChildItem);
  if StartItem.FirstSiblingItem <> nil then
    LoadOutline(StartIndex,StartItem.FirstSiblingItem);
end;

begin
  MyOutline.Clear;
  for Counter := 0 to ListOfItems.Count - 1 do
    if ListOfItems[Counter].Level = 1 then
      LoadOutline(0,ListOfItems[Counter]);
end;
```

To load an Outline, you can go from the top to the bottom adding children to each *node*, assuming that each node knows whether it is a top-level node and what its children are. **TListOfItems** and **TItem** (present in Listing 11.4) are fictitious classes that contain this information. (See the TreeData Components below.)

Unfortunately the standard hierarchical model does not keep an explicit list of children, but rather determines the set of children as those items that have the same item as a parent. Unless you pre-load the entire set into memory (like a **TListOfItems**) and establish the parent-to-child links, you have to load from the bottom to the top. That is, as you add each item's parent item, you have to check if a sibling has added the parent already, and if the parent is loaded, tell the control that your new item is part of that parent's branch.

Using the Data

The point of any user interface should be to communicate effectively with the user. The user needs to see enough of the data to make a choice (or see enough to know that a choice cannot be made), and then somehow, needs to indicate that choice to the application. With graphical trees, it's easy enough to use the double-click or spacebar keypress to indicate that the current item is the chosen one.

Once the user makes a choice, your application somehow has to identify the item chosen. The text loaded in the control for an item may not uniquely identify an item and may be duplicated in different branches, so most strategies incorporate additional unique ID code columns that contain no intrinsic information (like dates) in them. These IDs should be short and are usually numerical. This makes it easy to ensure that they are unique by adding one (1) to the highest existing value.

NOTE:
Just because you use the ID value to set up the hierarchy in the control does not mean that you can automatically get the ID value back out again. The **Index** property of the **TOutline** class changes for each item as you modify the Outline's contents; it is a relative value and not specifically associated with any particular item.

To be sure, you must somehow link the ID with the item itself. An ID class is one way to do this.

```
type
  TMyIDClass = class(TObject)
  public
    ID : Integer;
  end;
begin
…
  NewIDObject := TMyIDClass.Create;
  NewIDObject.ID := ItemTable.FieldByName('ID').AsInteger;
  MyOutline.AddChildItem(0, Item
Table.FieldByName('Description').AsString;
```

To link the ID directly to the item, you can use the **Object** pointer that is already associated with most Windows controls that store lists of items. The **TOutline** component offers this pointer as **Items[Index].Data** instead of calling it **Object** (another anomaly: the **Index** value starts from one (1) instead of zero (0), as most lists do). This pointer will link an object descended from **TObject** (that is to say, any class instance) to an item. You have to define a new class that holds your ID value, create an instance of it for every item you load, put the ID in that instance, and set the pointer.

To get your ID back, your code might look like this:

```
with MyOutline do
  ThisID := (Items[SelectedItem].Data as TMyIDClass).ID;
```

Your application may need more information than just the ID. You can use the ID to look up that other information in a table, and you can also store that information in the same structure that holds the ID associated with the item simply by expanding the **TMyIDClass** definition.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Finding Rows

You need to identify which item in a control the user chooses, but you will also need to locate an item within a control if the control is going to interact with other controls and components. Depending on your control, you may have to iterate through the entire structure until you find the item you are looking for. If your hierarchy subdivides a sorted list into sorted groups (for example, a parent item for each letter of the alphabet and all the items with descriptions starting with that letter as its children), you can take advantage of the grouping to narrow down your search.

One pitfall to avoid if possible is examining the same items more than once during the search. As with the problem of loading items, if you search starting from an item up through its parent, you will examine the same parent item for every child it has. Unless the contents of the items create some groupings that narrow your search, checking the list of items linearly is always the safest way to find an item.

Using Hierarchical Data in Queries

A hard place to reconcile the hierarchical and relational models is in the area of queries. The relational model is good at identifying rows by their attributes (columns) or common values in other tables (joins). Such a query is often a short, self-evident string in SQL.

SQL, however, is *not* good at expressing concepts like “somewhere in your lineage, there is a green box.” It is good at finding green boxes, but has no idea what a “lineage” is. The section discussing SQL later on in this chapter has some solutions for finding rows iteratively, but if you have the hierarchy in memory, you can get the lineage as a set of IDs and use it in a query with the **IN** criteria operator. It looks for a field’s value in a delimited list of alternatives. Listing 11.5 demonstrates how this is done by creating a SQL query programatically in a **TQuery**’s **SQL** property. Notice here that SQL does not do all the work itself; the hierarchy object first identifies the lineage using its own machinery.

Listing 11.5 Using SQL to search a lineage

```
procedure TForm1.FindColoredBoxes(ColorName : String;
                                  StartingID : Integer);
```

```
var
  DescendantList : String;

begin
  DescendantString := HierarchyObject.GetDescendants(StartingID);
  with Query1 do
    begin
      DisableControls;
      Close;
      with SQL do
        begin
          Clear;
          Add('SELECT *');
          Add('FROM BoxList T1');
          Add('WHERE T1.BoxColor = "' + ColorName + '"');
          { This assumes that the IDs in DescendantString
            are delimited with commas }
          if DescendantString <> '' then
          Add('AND T1.BoxID IN (' + DescendantString ')');
        end;
      Open;
      EnableControls;
    end;
end;
```

As an example: If your query is looking for all painters, you can go to your hierarchy, find the parent of all painters, programmatically extract all the descendent IDs of that item, and use those ID values as criteria. This allows you to use a single ID to identify a row, allow that ID to be very specialized, and still retrieve that row when you ask for the general category. Because you use the hierarchy, you don't have to know how many descendants the Painter item has—you will get them all.

If you have a hierarchy in an Outline or TreeView component, you can use the navigational properties of those components to iterate through the descendants of any given item. Otherwise, you have to load it into memory and set pointers between parents and children, or use the iterative or recursive query techniques described below.

Referential Integrity and Circular References

Ironically, a recursive hierarchy in a single table makes referential integrity easy. Within the table, every parent ID must be another row's item ID or else null (indicating a top-level item). This protects all of the descendants of an item because no item ID can change if any other row depends upon it.

When the hierarchy table contains lookup values, the dependent column needs only to relate to the Item_ID column in the single table. If the lookup values are in several columns or tables to allow multi-level groupings or relationships, referential integrity becomes more complicated.

The most dangerous thing for code that controls a hierarchy is a circular reference. If one item refers to a non-existent ancestor item, that is a detectable problem; however, if an item refers to an ancestor item that is also a descendent item (which is hard to discern if the items are separated by several generations), the code will loop endlessly, looking for an end to the succession of linked items.

The answer? Apart from checking each candidate ancestor item to see if any of its ancestors are

already in the family (which can be expensive in performance terms), your code can have a burn-out counter that throws an exception if the code processes through an excessive number of search cycles. One comfort is that if you use a graphical control to manage the hierarchy on a form, there is no way the end user can set up a circular reference.

Using SQL

If your hierarchies are huge and cannot operate entirely within memory, a SQL implementation may be a better solution. If you know how many levels of recursion are involved, you can use SQL subqueries to bridge generations, as shown here in Listing 11.6.

Listing 11.6 Using SQL to bridge a known number of generations

```
SELECT *
FROM Items T1
WHERE T1.Parent_ID IN
       (SELECT T2.Item_ID
       FROM Items T2
       WHERE T2.Parent_ID IN
           {SELECT T3.Item_ID
           FROM Items T3
           WHERE T3.Parent_ID = 'Fred'))
```

This requires exactly three generations: The only rows returned are the grandchildren of Fred. To get children and grandchildren, run two queries, and either use **UNION** or add the result sets together using **INSERT INTO** or temporary tables.

To find an item's parent item, simply query for the one row where the **Item_ID** is equal to the **Parent_ID**. To find all the children of an item, query for all the rows that have the **Item_ID** value as their **Parent_ID**. To find all the siblings, find all the items that have the same **Parent_ID**. (Note here that the original item will also be a part of the set unless you specifically exclude it.) Once you have a child or sibling set, to find all the descendants, you have to iterate through each row of that set and run a query to find its children. Iterate through *that* set to find any children of the children, and so on.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Solving the Problem of Arbitrary Nesting

The big problem comes in when the nesting is arbitrary and you don't *know* how many levels a lineage might have. There is no conditional branching in SQL; either a subquery is there or it's not. Joe Celko devotes two chapters of his book *SQL for Smarties*[1] to Trees and Graphs; that is, the data that goes into visual graphs. His techniques are advanced and demonstrate what you have to do to correctly associate one item (or *node*) to another.

[1] *SQL for Smarties, Advanced SQL Programming*, Morgan Kaufmann, 1995, 1-55860-323-9

If you want a simpler, though possibly inefficient and certainly inelegant solution, consider using a temporary table to hold the cumulative results of many queries (the Final table), and another temporary table (the Working table) to hold the results of the most recent one. You may have to use two Working tables and alternate between them, depending on which SQL server you are using. The algorithm looks like this:

1. Query for children of the starting item;
2. Copy the item IDs to the Working table;
3. Query for children of any of the item IDs in the Working table;
4. If there are no child rows, exit;
5. Add the Working table to the Final table;
6. Erase the Working table and add all the query's item IDs to it;
7. Go back to step 3.

Each cycle retrieves the next generation, and the Final table will contain all the child rows in generation sequence.

Using Stored Procedures

Stored procedures resemble SQL with conditional and looping logic, which SQL itself

lacks. InterBase allows stored procedures, called *select procedures*, which return any number of rows just as a SQL query would. Its procedure language allows you to iterate over a result set (using a query or another select procedure inside the procedure) and operate on it, as shown in Listing 11.7, which is in the InterBase command language. (The line numbers are significant in the discussion that follows.)

Listing 11.7 InterBase Select Procedures

```
1.  CREATE PROCEDURE GETCHILDREN (STARTING_ITEM_ID SMALLINT,
                                  THISLEVEL SMALLINT)
2.  RETURNS(ITEM_ID SMALLINT, DESCRIPTION CHAR(30),
            ITEMLEVEL SMALLINT) AS
3.  BEGIN
4.    FOR
5.      SELECT T1.ITEM_ID, T1.DESCRIPTION
6.      FROM ITEMS T1
7.      WHERE T1.PARENT_ID = :STARTING_ITEM_ID
8.      INTO :ITEM_ID, :DESCRIPTION
9.    DO BEGIN
10.       ITEMLEVEL = THISLEVEL + 1;
11.       SUSPEND;
12.       FOR
13.          SELECT T1.ITEM_ID, T1.DESCRIPTION, T1.ITEMLEVEL
14.          FROM  GETCHILDREN(:ITEM_ID, :ITEMLEVEL) T1
15.          INTO :ITEM_ID,  :DESCRIPTION, :ITEMLEVEL
16.       DO BEGIN
17.            SUSPEND;
18.       END
19.   END
20. END;
```

This sort of recursive iteration is exactly what you need to walk up and down the branches of hierarchical data because these stored procedures are recursive: You can call them from within themselves to get children of children, and so on. Instead of getting all the rows in a generation, and then getting the next generation, this strategy gets the first child of the first child, then the first grandchild of the first child, and so on until the last descendant is found.

In the InterBase language, **SUSPEND** means to return the **RETURNS** variables as the next row in the result set. The first **SUSPEND** (line 11) returns the values from the first row of the query on the immediate children of **STARTING_ITEM_ID** (lines 5 through 8). The next **SUSPEND** (line 17) returns whatever comes back from a recursive call to the **GETCHILDREN** select procedure. As long as that second call returns rows (that is, as long as it still has grandchildren and descendants to report), the second **SUSPEND** will keep passing the return values back to the original caller. When it has no more return values, the calling code continues and uses the first **SUSPEND** to return the second row of the original query. If you don't reset the **ITEMLEVEL** in the outer loop (line 10), it will contain the value from the last iteration of the inner loop (line 15).

Use a **TQuery** to call InterBase select procedures, not a **TStoredProc**. The syntax is simple:

```
with Query1 do
begin
  SQL.Clear;
  SQL.Add('SELECT * FROM
          GetChildren(' + IntToStr(CurrentItemID) + ',0)');
  Open;
end;
```

The result set will have all the children of the current item and will identify their level.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

The TreeData Components

The TreeData components are components I've created that allow you to view, navigate, and manage hierarchical data. They will display the rows in a graphical tree form such that each level is indented to show its ownership, show the entire lineage of an item, and give your application a list of all the ancestor or descendent IDs. There are several components in the family, and I've summarized them in Table 11.4.

TABLE 11.4 The TreeData Family Of Components

Control	Description	Use
TreeDataComboBox	Displays the items in an indented tree layout on the drop-down list; displays the lineage in the edit control. Allows incremental searching on both the edit and the drop-down. Data-aware to link to chosen IDs to a DataSource.	Get a single choice from the user: get the ID or all the Ancestor or Descendent IDs.
TreeDataOutline	Displays the entire tree graphically and allows expansion and contraction of various branches. Data-aware to link to chosen IDs to a DataSource.	Get a single choice from the user: get the ID or all the Ancestor or of Descendent IDs.

TreeDataList	Combines a TreeDataComboBox with a data-aware LlstBox. Data-aware to link all choseb ID to a DataSource.	Allows user to pick any number of tree items and save them to a table as a set or load them as a set.
TreeDataUpdate	Combines a TreeOutline. with editing functions to edit and update the rows that make up the hierarchy. Immediate or cached updates to DataSource.	Maintains a hierarchical dataset.

The TreeData controls crystallize a lot of what I've been discussing in this chapter so far. Unfortunately, they represent thousands of lines of source code and cannot be fully reproduced here on the printed page. All the controls are available in source code form, however, on the CD-ROM accompanying this book.

They all require a table name, SQL query text, or a DataSource, as well as the field names with the item's ID, the item's parent's ID, and display values to display in the tree. They load all the available rows and then fill the control's display with the values in the correct hierarchy.

TreeData Property Management

Certain properties are very important in this operation. All the TreeData controls take the design-time properties: **LookupDatabaseName**, **LookupTableName**, **LookupSQL** (mutually exclusive with **LookupTableName**), **LookupDisplayField**, **LookupIDField**, **LookupParentIDField,** and if used with Delphi 2.0, **LookupSessionName**. (See Figure 11.4.) Using these properties, a TreeData control loads all the data into memory and displays it as a hierarchical tree. The connection to the data table is then closed.

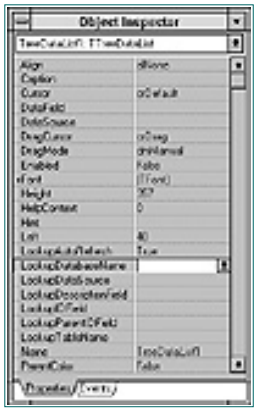

FIGURE 11.4 Properties of the TreeData components.

The TreeData controls also have a **LookupSource** property that can use an open DataSource to get the data. This lets you filter, range, or query the **DataSource.DataSet** property to control what items go into the control. The

LookupAutoRefresh property specifies whether or not you want to reload the lookup data if the **LookupSource** changes.

TreeData Component Internals

All these components share a base code unit TREEUTIL.PAS, which holds the definition of the various internal classes that manage the data. TREEUTIL.PAS defines a class named **TTreeDataItem,** which holds information about an item, and a class named **TTreeDataItems**, which is a **TList** descendent that keeps track of all the **TTreeDataItem** objects. Each of the controls expose a **TTreeDataItems** object as the property **ItemList**. You can call the public methods of this object to load, save, find, move, or delete an item. You can also get all ancestor or descendant IDs for an item, or get the top-level ID in an item's lineage.

TTreeDataItems actually descends from class **TStringList**, which holds all the item IDs. The **Object** property of each **TStringList** item points to the **TTreeDataItem** object for that item. The user object pointers are in a separate **TList** and synchronized by the index into both lists; this index does not change after the data is loaded, so there is no chance of de-synchronization. Sorted **TStrngLists** will use binary searching within the **IndexOf** method, so they can easily find IDs. After it loads all the items and sorts the IDs, class **TTreeDataItems** iterates through all of them and puts the index for the first child and the next sibling in each item's data structure. This allows easy iteration down and across the hierarchy.

With all that said of the TreeData family as a whole, I'll briefly discuss each control in the family separately.

TreeDataComboBox

This control is data-aware, so you can use it to save the user's choices to a table. It stores the single value to **LookupIDField**. The drop-down portion of the display shows the graphical tree, and the edit box shows the hierarchy of descriptions separated by colons. The edit box also allows incremental type-in that displays the first match to text already entered. When a match is found, typing the colon (or semi-colon) locks that item and moves the search point to text entered after the colon or semicolon. This allows you to continue typing to search that item's children. This is shown in Figure 11.5.

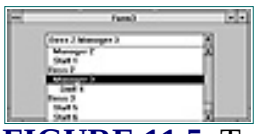

FIGURE 11.5 TreeDataComboBox.

The TreeDataComboBox component has properties for the description, ID, and **Item.FullDescription**, which contains the delimited descriptions as displayed in the edit control. It has additional properties that return a string with all the parent or child IDs concatenated together.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

TreeDataListBox

This control incorporates both a **TTreeDataComboBox** at the top and a data-aware **TListBox** beneath it. (See Figure 11.6.) The **TListBox** acts on all the rows in its DataSource instead of just the current row. You can use the combo box to pick items and then add them to the list. When you call **SaveIDs** or exit the control with the **SaveOnExit** flag set, the control writes all the IDs to the DataSource, one per row. The DataSource can incorporate a range (for example, a MasterSource) or filter so it presents detail rows for only master items in the MasterSource or filter.

FIGURE 11.6 TTreeDataListBox.

TreeDataOutline and TreeDataUpdate

TreeDataOutline displays the hierarchy in a graphical structure resembling the Windows 95 Explorer interface. As with the other controls, you can get the current item's ID, description, and **Item.FullDescription**, as well as a string list of all the parent or child IDs.

The TreeDataUpdate control (see Figure 11.7) combines this functionality with some additional behavior to manage a hierarchical data structure from a table.

FIGURE 11.7 TreeDataUpdate.

End Note

Hierarchies use the terminology of family relationships (parents, grandchildren, ancestors, descendants) because families are the most universal example of a gathering of individual entities—here, human beings—relating to other entities in hierarchical ways. Ironically, it is also a reminder that while you may build systems to handle recursive hierarchies generically, each item's value lies in the information that it contains; its *place* in the family tree is not valuable in and of itself. Hierarchical structuring is a means of finding the correct item; it's up to you to ensure that the item is worth finding.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

CHAPTER 12
The Oracle Vanishes

DON TAYLOR

- Utilities For Linking Memos And Pictures To A Delphi Database
- Delphi Drag And Drop
- Packing Paradox And DBASE Tables
- Keystroke Translation
- Playing WAV Files

Ace Breakpoint has returned—but his casebook is missing, and larceny is suspected. Ace begins the hunt for the Mysterious Stranger, who has in turn been hunting for Delphi Wisdom. The plot thickens!

You're about to embark on a journey that some might call strange. I guess I would be among the first to agree with them.

It was only a year ago that both Delphi and Ace Breakpoint first came upon the scene. Delphi was, of course, the product of the extremely creative team of developers at Borland International. Ace was—well, the product of necessity.

While learning Delphi, it became obvious that no amount of words could ever describe every aspect of this incredible product. One could never just *read* about Delphi and learn it—one had to *discover* it firsthand. My assignment: To write the tutorial map, to lead the expedition and to create a detailed example, using Delphi in a real-world situation. And oh, yes —to hold the readers' attention for some 200+ pages!

My proposed solution was to write an adventure story that carried the reader through the entire tutorial, featuring one of the most unusual programming consultants of all time. I pitched the idea to Jeff Duntemann, then held my breath. In case you're not aware, Jeff is just as daring and fun-loving as he is brilliant. He gave me the thumbs-up, and I created Ace Breakpoint, hard-hitting fictional Private Investigator-turned-programmer....

As a boy growing up in Hackensack, Ace dreamed of becoming a P.I., just like his heroes in the classic 1940 movies—Phil Marlowe, Sam Spade, and Ellery Queen. But after years of study and hard work learning investigative techniques and snappy dialog, Ace discovered there was little demand for a 1940's-style P.I. in today's world.

Undaunted, Ace decided to make a serious career change. This time, Ace would choose a profession that would be in demand for a *lifetime*—he would become a Windows programmer. But in addition to becoming today's professional, he also wanted to become a "person of the 90's." So he moved to Poulsbo, Washington, and for two long years he attended night classes in computer programming at the Suquamish School of Art. On graduation day, he was awarded an Associate of Arts in Programming Appropriateness. He promptly rented an office and hung out his shingle.

As with most heroes, Ace has his minor flaws. In spite of all his formal education, he is still a little rough around the edges. Although he does his best to care about the needs of others, there are times when sensitivity hangs on him like a cheap suit. He frequently makes mistakes. But he has a tenacious quality about him, and you've got to love him for that. When faced with problems, he will doggedly pursue them until he digs out the answers.

Oh, yes—just one little matter. Although the Breakpoint adventures are fictional, one thing is true. The location—Poulsbo—is a real town, situated about 15 miles (as the crow swims) west of Seattle. At its inception, it was a fishing village, founded by a hardy group of Norwegian immigrants. Today Poulsbo is largely a tourist town, with marinas and a winding street full of quaint shops and restaurants. (You don't want to move there, though. It's too crowded already. And it rains nearly all the time. Honest.)

Very little else in the Breakpoint adventures bears any connection to reality. And of course, unless stated otherwise, characters in the stories are purely fictional, and are not meant to represent any person, living or dead.

So don't touch that dial! Grab your favorite snack, turn down the lights, slide up close to your screen, and prepare yourself for the adventure I call...

Ace Breakpoint in...

"The Case of the Missing Ink!"

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Previous | Table of Contents | Next

An Evening at the Office

I stepped inside my office and slammed the door behind me so hard that I could hear a thud outside as my shingle fell to the ground. It had been pouring all day, and now it seemed to be raining even harder. The excessive rain of the past few days had turned the field next to my office complex into a thick, muddy paste that was now relentlessly encroaching on the asphalt parking lot like a television news crew heading for a disaster scene. I scraped my shoes on the mat inside the door in a futile attempt to remove the sticky brown goo.

I stripped off my hat and coat and tossed the soaking armload on the couch, missing Mewnix by a whisker. The fatigued feline didn't so much as flinch, but he did manage to pop open one eye and fire a dirty look in my direction. I plunked down in the chair in front of my computer and switched it on. Having just returned from the regular Tuesday night meeting of my Win95 codependency group, I needed to lighten up a little. I decided to go online and pick up my mail.

The only thing waiting for me was a note from Gene Fowler. He said he had enjoyed the adventures in *Delphi Programming Explorer*, and he had extended the capabilities of the quick and dirty utilities I created to handle the editing and pasting of memos and graphics into tables. His two utilities and some other stuff were zipped together in a file attached to the note. I unzipped the file and examined his work.

The first was called MEMOEDIT.EXE, a generalized text editor for working with memo fields in a table. I gave it a try and then took a shot of the screen, which is shown in Figure 12.1. It was self-explanatory and very easy to use. The other utility was PICTEDIT.EXE, a generalized picture editor for use with BLOB fields in a table. A picture of its interface is shown in Figure 12.2. As with the memo editor, the picture editor was intuitive. Of course, Delphi 2.0's Database Explorer provides an editor for memo fields. Personally, I liked

Gene's memo editor a little better. And I sure wished both of these puppies had been around when I created that original demo using Delphi 1.0! I copied both programs into my utilities directory, and shot off a thank-you reply to Gene.

FIGURE 12.1 Fowler's MemoEdit utility.

FIGURE 12.2 Fowler's PictEdit utility.

NOTE:
Gene Fowler is a real person who actually wrote MEMOEDIT.EXE and PICTEDIT.EXE. Those utilities, along with several others, can be found in the \CHAP1\FOWLER subdirectory on the CD-ROM that accompanies this book.

I found myself reminiscing about what Gene had said. Although that adventure had taken place only a year ago, it already seemed like a lifetime. It had started out innocently enough, with Melvin Bohacker and myself competing for a potential contract. Bohacker was a tall, pencil-necked Geek with a propensity for snack food and programming in C and C++, using a multitude of libraries. Up to that time he had beaten me out of every consulting job I had gone after.

The competition sounded straightforward. But when the smoke cleared just 24 hours later, I was a physical wreck—beaten, bloodied and bewildered. If it hadn't been for my friends, I don't know what I would have done.

We had all gotten to know each other at the Art School, and we had stuck together like glue ever since. As close friends, even they had been to some extent subjected to the heat of the competition. Bruiser Buckendorf-Rabinowicz (the former pro football player) was taken in for psychological evaluation. Biff Murphy, the Business Ethics major, was in fear for my life. Muffy Katz, the professional part-time psychic and Fashion Therapy major, was in fear of—well, to tell the truth, she was probably most in fear of breaking a nail.

And then there was Helen Highwater. For nearly five years, Helen had stuck

by me, through thick and thin. She was the love of my life. But in the midst of the adventure, I almost lost her. Although I had been devastated as a result of the competition (I still bear the coffee stains on my hands as testimony), I managed to get in my licks with Bohacker the following afternoon. In fact, I had hit him so hard that some of the blood from his pummeled nose splattered on the trenchcoat I was wearing. That bloodstained memento still hangs in the back of my closet.

I had to chuckle. A lot of water had gone under the bridge in the past year. I was now using Delphi 2.0 almost exclusively, and my consulting practice was steadily growing. Biff was still working the takeout window at Buck McGawk's Norwegian Fried Chicken Haus. Helen was now the day manager at the Hardanger World outlet at the local mall. Muffy left her job to start her own line of designer clothes and accessories, under the *Pure Prophet* label. And then there's Bruiser's tragic story. He was diagnosed as a codependent megalomaniac and committed to a state institution. Two months later, after intense study, he had learned to tune a guitar. Then one night he and four other inmates escaped and started a grunge band called *Not Ourselves Today*. Their first CD went platinum. Their second album is scheduled for release this week. (I don't plan to buy either of them. There are, after all, limits to every friendship.)

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

An Urgent Plea

I wandered over to the TV and flipped to the Shopping Channel for some background noise, then picked up the latest volume of my Casebook and began dutifully recording the details of the day's events. I had written only two paragraphs when the telephone rang.

"Breakpoint. What can I do for ya?"

"Mr. Breakpoint—thank goodness you're there!" said the voice coming from the receiver. "I truly don't know what I'd have done if you hadn't answered your phone."

It was an unusual voice, to say the least. Certainly, it was desperate. And it was definitely female. Hot and cold, breathy and chilling all at the same time. Sort of like Lauren Bacall, with ice cubes. She was in a near panic.

"Slow down," I said. "Just take it easy, and explain your situation in detail."

"I'm an heiress," she replied, her voice still breathless. "For the past week I've been followed at a distance by a very large man whom I believe is intending to kidnap me and hold me for ransom. A few moments ago when I stopped at a traffic light, he jumped out of his car and tried to force open my door. I sped off and managed to lose him temporarily, but he may catch up with me any minute. I found your name in the phone book. I have no one else to turn to."

"Okay. Keep calm," I said reassuringly. "Can you describe this man? Did he have a beard or mustache?"

"I couldn't tell," she replied. "His head was covered with a nylon stocking."

"Seamed or seamless?" I queried.

"Seamless, I think."

“What shade?”

“Natural. No, no—it may have been sand. Oh, I don’t know! Listen, Mr. Breakpoint,” she pleaded, “you’re my only hope. He’s liable to show up at any moment. I need you right now.”

“Just hold on,” I said. “Where are you?”

“I’m at a pay phone on the highway, at a place called Ole’s Espresso Bar and Tire Emporium. I desperately need your protect—”

My heart fell as I heard the unmistakable sounds of a struggle, followed by a crash, a muffled scream, and the hiss of steam escaping from an espresso machine. Then the line went dead. I wondered why nobody ever called the cops any more.

I glanced at my watch. It was 10:13 P.M. I tossed my Casebook on the desk, grabbed my keys and raingear, and raced for the car. The pouring rain mingled with a fog that had begun to roll in, making it difficult to find my car, parked only 50 feet away. I flipped on my headlights, but even their powerful Halogen beams could barely penetrate the curtain in front of me. The row of shrubs planted alongside my office looked like a bunch of Angry White Males standing in the shadows, waiting for a bus. I backed out of my space and managed to find my way to the street. Then I slammed the accelerator to the floor.

The Disappearance

At 10:57 P.M., I pulled back into the parking lot at my office. The mud had become so thick that the car skidded across the pavement like an elephant on ice skates, stopped only by the tires smashing into the curbing in front of space #132.

I had been stood up by the woman on the phone. When I arrived at the phone booth, there was no one in sight. Ole said he hadn’t seen anyone near the pay phone all evening. It was a perplexing situation. The only thing I had gotten out of this wild goose chase was a free tire rotation while I waited for my latté order.

I made it to the edge of the parking lot, slipping and sliding all the way. I turned the corner and headed down the walkway.

And then my heart stopped.

The door to my office was standing wide open. As I ran the last few steps, I specifically remembered having locked the door when I left. Nevertheless, it was open, and all the lights were on inside. As I crossed the threshold, my eyes scanned the entire room at lightning speed. *Nothing missing,* I thought. No—Mewnix was nowhere in sight, but he probably just wandered out the open doorway. Then my eyes fell on my desk, and a lump the size of a large fur ball suddenly stuck in my throat: My Casebook was gone!

Moving quickly to the desk, I frantically thrashed through the papers on top, then searched each of the drawers, and in total desperation, I examined the

floor around the desk. But the Casebook was nowhere to be found.

I slumped down in my chair. The phone call had been merely a ruse to get me out of the office. Thanks to the woman on the phone (and quite likely an accomplice) I had just graduated to Chump First Class.

I had been so engrossed in looking for the missing Casebook that I hadn't noticed the thick carpet of carbon dioxide fog that had rolled into the room. It now covered the entire floor, spilling over the tops of my running shoes. The room had gone dim, the only lighting in the room now seemingly coming through the fog from the floor beneath. My eyes were drawn like a magnet to a mysterious figure framed in the doorway. It was a wispy creature, definitely humanoid, but not clearly male or female. Sensing I had noticed it, it spoke out in an announcer-like voice, with entirely too much reverb to suit my comfort.

Welcome back, Mr. Breakpoint. You have been expected.

"Wait a second," Ace demanded. "Who—or what—are you?"

First things first. As you have discovered, your Casebook has indeed been stolen. The entire contents will be read by someone else. Not just your technical notes, but all your personal writings as well. The Author felt you needed some help in conveying the remainder of this story. He has therefore provided you with a narrator.

Ace removed his Fedora and scratched the top of his head. "A narrator?"

The Third Person.

"A politically correct, 1990's version of Harry Lime, the famous detective called The Third Man?" Ace asked.

No. The Third Person. The one you studied in English Comp. Remember?

"Sort of," Ace replied. "I don't know your name, Mr. or Mrs.... Hey—are you a man or a woman?"

Third Persons don't have names. Being merely a literary device, a Third Person has no gender. Nor does a Third Person have a life of its own, for that matter.

"I can kinda identify," Ace observed. "But answer me this: How come I have lost control of the narration of this story?"

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

It is only a temporary situation. Control will automatically revert to you near the end of the story. Until then, it is the job of the narrator to describe the flow of events. Pretend you're watching an episode of "The Outer Limits."

"I don't know if I'm gonna like this."

At the "Sleeveless Arms"

In a shabby, brick-faced building across town, a shadowy figure slipped from the hallway into a recently rented second-story apartment and locked the door. Inching through the darkness, the tall, lanky form found its way to a small desk in the corner, where it snapped on the familiar banker's lamp, bathing the desktop in a warm glow. The figure carefully removed a hardbound leather notebook from the inside pocket of a full-length raincoat, and pushing aside several unopened packages of snack food, laid it on the desk. The light reflecting from the desktop illuminated the small portion of face exposed between an upturned collar and the wide brim of the dirt-brown hat: two blazing, close set eyes and a sharp nose that barely extended beyond a thick, perfectly-manicured mustache.

"So this is it, eh?" said the rain-soaked figure, in a voice dripping with anticipation. "Let's just see what we have here."

The mysterious figure sat down at the desk, paused for just a moment, and then turned to the first page and began to read from Ace Breakpoint's Casebook.

Doing the Old 'Drag/Drop'

Casebook No. 16, March 19: Sometimes the things that look the simplest can be the most difficult. Then again, the opposite can be true. At least that's the case with intra-application drag and drop operations in Delphi applications.

I was creating an application in which I wanted to select an item and relate it to a date by dragging it over and dropping it on a calendar on the same form. First, I had to

investigate what that would entail. I found there to be four primary steps to any drag and drop operation:

1. Initiating the **BeginDrag** method of the originating (source) component, by handling mouse-related events occurring within that component;

2. Creating a handler for the destination component's OnDragOver event, to indicate where within the component a drop is permissible;

3. Creating a handler for the destination component's OnDragDrop event, to determine what action is taken as a result of the dropping action; and

4. Optionally, creating a handler for the originating component's OnDragEnd event. While the other three steps are required in any drag/drop, this step is needed only if some housekeeping is required on the originating component when the drag ends—housekeeping that will be performed even if the operation was aborted.

Kind of a Drag

My first challenge was the calendar component. Although a good one is provided with Delphi, it did not provide an easy way to relate a position on the calendar to a date. Because the sample calendar component shows only the days in the current month (leaving blank entries everywhere else in its grid), I would have to write code that computed the dates relating to those blanks. Not a pretty situation.

My solution turned out to be the OvcCalendar component provided with TurboPower Software's Orpheus VCL package. This powerful little component relates a date to every cell in the grid, displaying days from the previous and/or following month. Since the title row containing the names of the days is the same size as the rows holding the dates, an OvcCalendar makes relating absolute date and position a straightforward process.

NOTE:
The OvcCalendar component, along with the rest of the components in the Orpheus collection, have all been included in a special trial version on the CD-ROM that accompanies this book. The collection can be found in the \CHAP1\ORPHEUS directory.

For my investigation, I created an application with a single form that contained an edit box, a string grid, and a calendar (and the ever-present button to exit the application). The idea would be to enter a string in the edit box, and then drag and drop the string on the calendar, where it would be associated with a date. A string containing the date and the entered string would then be added to the string grid. A snapshot of the running version of the experiment is shown in Figure 12.3.

FIGURE 12.3 Ace's drag and drop experiment.

I started with Step 1. It seemed logical to look for a mouse-down message arriving in the edit box, which turned out to be a good idea—more or less. By setting up a drag

from an edit box, I had unknowingly put myself at cross purposes: The OnMouseDown event already had a standard meaning in the context of an edit box, that of selecting text. By using that event to enable a drag, I would lose the ability to select a portion of the text by dragging the I-beam cursor over it.

The OnMouseDown event handler is provided with several pieces of information—a reference to the object originating the message; a parameter that indicates which mouse button is down; another parameter that indicates the state of the shift, control, and alt keys; and finally, x-y position information on the cursor. In this case, the originator (sender) would be the edit box, because that is where I wanted the drag to begin. I could ignore the positioning information, since I didn't care where the drag could begin within the simple rectangle of an edit box. And I wanted to start a drag only when the left mouse button was pressed. (I later discovered I had to filter out double-clicks, too, because allowing them to initiate drags caused some strange side effects.)

Pretty simple. My final code appears in Listing 12.1.

Listing 12.1 An event handler to begin the drag

```
procedure TDDDemoForm.EditBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
 if (Button = mbLeft)
   and (EditBox.Text <> '')
   and not (ssDouble in Shift)
  then TEdit(Sender).BeginDrag(False);
end;
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

While testing my original version of the handler, I discovered there are two ways to initiate the drag. Using **True** as the argument of the **BeginDrag** method starts the drag as soon as the mouse is pressed, while selecting **False** delays until the mouse is actually moved a few pixels. The second choice felt far more natural. After some bumbles, I also added the line of code that blocks out any attempt to drag an empty string. The typecast used to call the **BeginDrag** method is typically required when working with the **Sender** and **Source** object references provided to event handlers.

Step 2 was next. The handler for OnDragOver events brings with it several parameters. **Source** specifies the object where the drag started (in this case, the edit box). **Sender** is the object sending the message—the potential target for the drop. In this case it is the calendar. The **X** and **Y** parameters give the relative coordinates of the mouse cursor within the **Sender**, and **State** specifies whether the object being dragged is entering, leaving or moving across the **Sender**. Although default cursors are provided for the drag/drop process, this drag state information makes it easy to specify your own cursors for each part of the process. Finally, there is the **Accept** parameter, a boolean passed by reference.

The object of the game is to decide, based on the information provided, whether or not it is acceptable to complete the drop. The process is sort of like a pilot of a small plane (the **Source**) radioing to a controller on the ground, saying “Here are my coordinates over the field in which you’re standing. Is it okay to drop my payload?”

It turned out that by selecting the OvcCalendar, I had made my work exceedingly simple. With the exception of the day titles, everything within the calendar’s client area was fair game for a drop. Since all the rows are the same height, I just had to figure out how tall a row was, and then accept a drop only when the X,Y position was past the first row. The code appears in Listing 12.2.

Listing 12.2 Checking for the acceptability of a drop

```
procedure TDDDemoForm.CalendarDragOver(Sender, Source: TObject;
  X,  Y: Integer; State: TDragState; var Accept: Boolean);
 var
  RHeight : Integer;
```

```
  RNum : Integer;
 begin
  RHeight := Calendar.ClientHeight div 7;
  RNum := Y div RHeight + 1;
  Accept := RNum > 1;
 end;
```

Dropping the Payload

I proceeded to Step 3. The strategy was to figure out the row and column locations of the first day of the month and the cursor position. From that, a little arithmetic would give me the number of days difference between the two dates, which I could then add to the first day's date to obtain the *absolute* date.

That first day will always appear in the second row (under the day titles), no matter which day of the week it falls on. So much for the row. Since I hadn't changed the calendar from the standard Sunday-Saturday format, getting the day of the week for the first day gave me a column reference position. From there, I just had to advance the number of days into the calendar. Fortunately, the Orpheus package comes with some powerful date conversion and arithmetic routines that helped greatly. My calculation routine for the date pointed at is shown in Listing 12.3.

Listing 12.3 Calculating the date from the mouse position

```
function DatePointedTo : TOvcDate;
  var
   Idx : Longint;
   DOW : Integer;
   Day1 : TOvcDate;
  begin
   { Compute first day as Row = 2, Col = DOW, then
     calculate an offset to the date pointed to. Then
     add the two. }
   Day1 := DMYToDate(1, Calendar.Month, Calendar.Year);
   DOW := Ord(DayOfWeek(Day1)) + 1;
   Idx := (RNum - 2) * 7;
   if CNum < DOW
    then Idx := Idx - (DOW - CNum)
    else if CNum > DOW
          then Idx := Idx + (CNum - DOW);
   Result := IncDate(Day1, Idx, 0, 0);
  end; { DatePointedTo }
```

All that was left was some simple housekeeping—converting the date and edit box text to a string, and adding that string to the string grid. I also found it handy to clear out the edit box once the drop was complete; with the editing capabilities of the edit box somewhat hampered by the dragging support, it became cumbersome to clear it manually.

Note to myself: In this case, it was appropriate to clear the edit box as part of the dragdrop event handler, because I wanted it performed only if the drop was performed. Had I wanted to clear the edit box whether or not the drop succeeded, I would have done so in an OnEndDrag handler for the edit box.

Reminder to myself: Make sure the OvcCalendar's **Initialize** property is set to **True**.

Otherwise, it comes up in an undefined state!

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Listing 12.4 shows the code for the entire unit.

Listing 12.4 Code for the drag/drop demo

```
{————————————————————————————————————————}
{              Drag/Drop Demonstration                 }
{         DRAGMAIN.PAS : Drag/Drop Main Unit           }
{             By Ace Breakpoint, N.T.P.                }
{              Assisted by Don Taylor                  }
{                                                      }
{ Application that demonstrates dragging and           }
{ dropping within the single application.              }
{                                                      }
{ Written for *Kick-Ass Delphi Programming*            }
{ Copyright (c) 1996 The Coriolis Group, Inc.          }
{               Last Updated 3/19/96                   }
{————————————————————————————————————————}
unit DragMain;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, StdCtrls, OvcBase, OvcCal, OvcData, OvcDT,
  ExtCtrls;

type
  TDDDemoForm = class(TForm)
    Calendar: TOvcCalendar;
    OvcController1: TOvcController;
    EditBox: TEdit;
    StringGrid: TStringGrid;
    Label1: TLabel;
    Bevel1: TBevel;
    QuitBtn: TButton;
    Panel1: TPanel;
    Label2: TLabel;
    Label3: TLabel;
```

```
    Label4: TLabel;
    Label5: TLabel;
    procedure QuitBtnClick(Sender: TObject);
    procedure EditBoxMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure CalendarDragOver(Sender, Source: TObject; X, Y: Integer;
      State: TDragState; var Accept: Boolean);
    procedure CalendarDragDrop(Sender, Source: TObject; X, Y: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  DDDemoForm: TDDDemoForm;

implementation

{$R *.DFM}
procedure TDDDemoForm.QuitBtnClick(Sender: TObject);
begin
 Close;
end;

procedure TDDDemoForm.EditBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
 if (Button = mbLeft)
   and (EditBox.Text <> '')
   and not (ssDouble in Shift)
  then TEdit(Sender).BeginDrag(False);
end;

procedure TDDDemoForm.CalendarDragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
var
 RHeight : Integer;
 RNum : Integer;
begin
 RHeight := Calendar.ClientHeight div 7;
 RNum := Y div RHeight + 1;
 Accept := RNum > 1;
end;

procedure TDDDemoForm.CalendarDragDrop(Sender, Source: TObject; X,
  Y: Integer);
var
 RHeight : Integer;
 CWidth  : Integer;
 RNum    : Integer;
 CNum    : Integer;
 s       : String;

  function DatePointedTo : TOvcDate;
  var
   Idx : Longint;
```

```
   DOW : Integer;
   Day1 : TOvcDate;
  begin
   { Compute first day as Row = 2, Col = DOW, then
     calculate an offset to the date pointed to. Then
     add the two. }
   Day1 := DMYToDate(1, Calendar.Month, Calendar.Year);
   DOW := Ord(DayOfWeek(Day1)) + 1;
   Idx := (RNum - 2) * 7;
   if CNum < DOW
    then Idx := Idx - (DOW - CNum)
    else if CNum > DOW
          then Idx := Idx + (CNum - DOW);
   Result := IncDate(Day1, Idx, 0, 0);
  end; { DatePointedTo }

begin
 RHeight := Calendar.ClientHeight div 7;
 RNum := Y div RHeight + 1;
 CWidth := Calendar.ClientWidth div 7;
 CNum := X div CWidth + 1;

 { Put the date and task in the string list }
 s := DateTimeToStr(OvcDateToDateTime(DatePointedTo))
      + ' - ' + EditBox.Text;
 StringGrid.Cells[0, StringGrid.RowCount - 1] := s;

 { Add a blank row to the string list }
 StringGrid.RowCount := StringGrid.RowCount + 1;

 EditBox.Text := '';

end;

end.
```

End of entry, March 19.

The sinister figure leaned forward, intently reading the purloined Casebook. The dark eyes swept across to the next page.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Packing Paradox and dBASE Tables

Casebook No. 16, March 20: When I was growing up, my mother constantly hounded me to put away bits and pieces that had not gotten incorporated into various projects I was working on. “Clean up your mess!” was a phrase I heard at least once a day. Perhaps the Bag Man had a mother like that, too—maybe that’s why he approached me with a request for an easy way to delete the excess space in Paradox and dBASE tables from a Delphi application.

I guess, like my strange client, the Bag Man, I had just assumed there would be a **TTable** method for accomplishing this. After all, both dBASE and Paradox offer that functionality. I suppose Delphi’s design team was looking at a bigger picture, one that included large client/server databases that wouldn’t recognize such commands from an application.

Although Delphi’s designers didn’t provide a direct way to pack tables, they *did* provide an easy way to “get under the hood” and access lots of low-level stuff. Not just Windows’ internals. We’re talking any old application program interface (API) that wants to make itself available—including the Borland Database Engine (BDE).

The BDE provides a lot of low-level services to programs. It is the basis for the methods associated with Delphi’s data-aware components. And the BDE units are available to any Delphi program.

A little searching on the Internet rewarded me with a routine that used the BDE routines to pack both Paradox and dBASE tables. Unfortunately, the routine appeared to have been written anonymously, so I can’t credit the original author here. I modified the code just a bit, to place it in a unit and incorporate some exception handling. I saved the modified version in a file named PAKTABLE.PAS, which can be seen in Listing 12.5.

Listing 12.5 A unit for packing Paradox and dBASE tables

```
{————————————————————————————————————————}
{             Data Table Packing Demo                    }
{          PAKTABLE.PAS : Packing Routine                }
{             By Ace Breakpoint, N.T.P.                  }
{             Assisted by Don Taylor                     }
{                                                        }
{ A unit containing a specialized procedure for          }
{ "packing" Paradox and dBASE tables to eliminate        }
{ wasted space.                                          }
{                                                        }
{ Written for *Kick-Ass Delphi Programming*              }
{ Copyright (c) 1996 The Coriolis Group, Inc.            }
```

```
{              Last Updated 3/20/96                }
{──────────────────────────────────────────────────}

unit PakTable;

interface

uses
 SysUtils, Dialogs, DBTables, DBiTypes, DBiProcs, DBiErrs;

function PackTable(var ATable : TTable) : Boolean;

implementation

type
 EDBPackMisc = class(Exception);

var
 ActiveStatus : Boolean;
 ExclusiveStatus : Boolean;
 Error : DBiResult;
 ErrorMsg : DBiMsg;
 pTableDesc : pCRTblDesc;
 AHandle : hDBiDB;

{ PackTable packs the records (and in the case of dBASE
  tables, actually removes records previously marked
  for deletion) in Paradox and dBASE tables. The
  TableType property of the table to be packed must
  be either ttParadox or ttDBase; ttDefault will not
  work. Also, the table must not be in use by anyone
  else, because it has to be put in Exclusive mode. }
function PackTable(var ATable : TTable) : Boolean;
begin
 Result := False;
 try
  with ATable do
   begin
    { Save the current status of the table }
    ActiveStatus := Active;
    ExclusiveStatus := Exclusive;

    { Disconnect the table from controls and make it exclusive }
    DisableControls;
    Active := False;
    Exclusive := True;
   end; { with }

  try
   { Pack the table, depending on the table's type }
   case ATable.TableType of
    ttParadox :
     begin
      { Create a description table and prepare it for use }
      GetMem(pTableDesc, SizeOf(CRTblDesc));
      FillChar(pTableDesc^, SizeOf(CRTblDesc), 0);

      with pTableDesc^ do
       begin
```

```
          StrPCopy(szTblName, ATable.TableName);
          StrPCopy(szTblType, szParadox);
          bPack := True;
         end; { with }

        { Get the table's database handle }
        with ATable do
         begin
          Active := True;
          AHandle := ATable.DBHandle;
          Active := False;
         end; { with }

        try
         { Attempt a restructure/pack and handle any error }
         Error := DBiDoRestructure(AHandle, 1, pTableDesc, nil, nil, nil,
           False);
         if Error = DBIERR_NONE
          then Result := True
          else begin
                DBiGetErrorString(Error, ErrorMsg);
                raise EDBPackMisc.Create(ErrorMsg);
               end;
        finally
         FreeMem(pTableDesc, SizeOf(CRTblDesc));
        end; { try }
       end;

      ttDBase :
       with ATable do
        begin
         Active := True;
         Error := DBiPackTable(DBHandle, Handle, nil, nil, True);
         if Error = DBIERR_NONE
          then Result := True
          else raise EDBPackMisc.Create('Could not pack this dBASE table');
        end;
      else raise EDBPackMisc.Create('Cannot pack this table type');
     end; { case }
    except
     on E:EDBPackMisc do
      MessageDlg(E.Message, mtError, [mbOK], 0);
    end; { try }

   finally
    { Restore the original condition of the table }
    with ATable do
     begin
      Active := False;
      Exclusive := ExclusiveStatus;
      Active := ActiveStatus;
      EnableControls;
     end; { with }
    end; { try }
  end;

  end.
```

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Deleted records are treated differently by Paradox and dBASE. When dBASE “deletes” a record, it doesn’t physically remove it. It just marks the record for deletion, by placing an asterisk in the first byte of the record. The advantage is that records can be easily “undeleted”. The disadvantage is that zero space is recovered when a record is deleted. On the other hand, Paradox really does delete the record, and it reuses the space when new records are added.

The mechanics of packing the two flavors of tables are different, as well. dBASE tables are packed with a **DBiPackTable** command. In contrast, Paradox tables are packed as part of a restructuring process (a relationship that may explain why the pack option was placed on Paradox’s restructure dialog).

Most of the machinations in **PackTable** center around capturing the state of the table (so it can be restored on exit) and getting the table in the proper state prior to making the appropriate call to the BDE’s API. The only way **PackTable** can differentiate between the two table styles is by the value given to the **TableType** property. When setting the table’s properties, either **ttParadox** or **ttDBase** must be selected; the standard **ttDefault** won’t cut the mustard. And no matter which type of table is being packed, it *must* be operated on in the Exclusive mode. No one else can be accessing the table while the packing operation is taking place.

The Packing Demo

I needed a simple program to demonstrate the functionality of the PakTable unit. The running version of what I came up with is shown in Figure 12.4. The code for the program can be seen in Listing 12.6.

FIGURE 12.4 The Packing Demo program.

Listing 12.6 Code for the packing demo

```
{————————————————————————————————————————————}
{                Data Table Packing Demo                }
{              PackMain.PAS : Main Form                 }
```

```
{            By Ace Breakpoint, N.T.P.              }
{             Assisted by Don Taylor                }
{                                                   }
{ A program demonstrating the use of the PakTable   }
{ unit for packing Paradox and dBASE tables.        }
{                                                   }
{ Written for *Kick-Ass Delphi Programming*         }
{ Copyright (c) 1996 The Coriolis Group, Inc.       }
{              Last Updated 3/20/96                 }
{———————————————————————————————————————————————}
unit PackMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs,
  DB, DBTables, StdCtrls, Grids, DBGrids, PakTable, ExtCtrls;

type
  TForm1 = class(TForm)
    AddBtn: TButton;
    RemoveBtn: TButton;
    PackBtn: TButton;
    QuitBtn: TButton;
    Table1: TTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    Label1: TLabel;
    TableNameLabel: TLabel;
    Label2: TLabel;
    FileSizeLabel: TLabel;
    Label3: TLabel;
    NumRecsLabel: TLabel;
    Bevel1: TBevel;
    procedure QuitBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure AddBtnClick(Sender: TObject);
    procedure RemoveBtnClick(Sender: TObject);
    procedure PackBtnClick(Sender: TObject);
    procedure FormActivate(Sender: TObject);
  private
    TablePathName : ShortString;
    procedure UpdateFileLabels;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
```

```
procedure TForm1.QuitBtnClick(Sender: TObject);
begin
 Close;
end;

procedure TForm1.FormCreate(Sender: TObject);
var
 s : ShortString;
begin
 Table1.Active := True;
 s := Application.ExeName;
 TablePathName := Copy(s, 1, pos('.', s)) + 'DB';
 TableNameLabel.Caption := TablePathName;
end;

procedure TForm1.UpdateFileLabels;
var
  f : File of Byte;
begin
 { Can't have the table open during the Reset }
 Table1.Close;
 AssignFile(f, TablePathName);
 {$I-}
 Reset(f);
 {$I+}
 if IOResult = 0
  then begin
        FileSizeLabel.Caption := IntToStr(FileSize(f));
        CloseFile(f);
       end
  else FileSizeLabel.Caption := 'I/O error!';

 { Re-open the table }
 Table1.Open;
 NumRecsLabel.Caption := IntToStr(Table1.RecordCount);
end;

procedure TForm1.AddBtnClick(Sender: TObject);
var
 i : Integer;
begin
 with Table1 do
  begin
   for i := 1 to 100 do
    begin
     Append;
     Table1.FieldByName('MessageString').AsString
      := IntToStr(i) + ': Hello. My name is Mister Ed.';
     Post;
    end; { for }
  end; { with }

 UpdateFileLabels;
```

```
end;

procedure TForm1.RemoveBtnClick(Sender: TObject);
begin
 with Table1 do
  begin
   First;
   while not EOF do
    begin
     Edit;
     Delete;
     MoveBy(3);
    end; { while }
  end; { with }

 UpdateFileLabels;
end;

procedure TForm1.PackBtnClick(Sender: TObject);
begin
 if not PackTable(Table1)
  then MessageDlg('Error packing the table', mtError, [mbOK], 0);

 UpdateFileLabels;
end;

procedure TForm1.FormActivate(Sender: TObject);
begin
 UpdateFileLabels;
end;

end.
```

This simple application demonstrates the packing of Paradox files. Clicking the Add button adds 100 new records to the table; clicking Remove deletes every third record. By hitting the Add and Remove buttons a few times while monitoring the display, it becomes obvious that not all the space is being recovered by the delete operation. Clicking the Pack button doesn't change the number of records, but it can certainly reduce the total file size.

End of entry, March 20.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Back at Ace's Office

He had been so engrossed with the loss of his Casebook, Ace hadn't even noticed the morning light streaming in through the venetian blinds. Although he had paced the small room until he had nearly worn a hole in the carpet, he was no closer to a solution to the whereabouts of his Casebook than when he started.

Defeated, he flopped down in his chair. *If I'm such a great detective, why can't I solve this simple mystery*? he thought. During the past nine hours, he had gone over the scenario a thousand times, but still there were no answers. Only questions. He had even dusted the room for fingerprints, and found only his own. There didn't seem to be a single clue.

Ace sat there for nearly an hour, drowning in the gloom. His Casebook, the repository of all technical knowledge he had gained in working with Delphi, was gone. Someone out there was reading his secret aspirations. His innermost thoughts. He felt naked and exposed. And in all likelihood he would never see the Casebook again. *If only there were even a glimmer of hope,* he thought.

His eyes popped wide open. There *was* a glimmer. He had seen it last night, as his headlights swept across the shrubbery in the hasty departure from his office. He remembered seeing the two tiny reflections from one of those trees that could have come from the eyes of a human being!

The former detective leaped to his feet. *If there was a man, he must have left footprints in the soft, soggy soil!* He rushed outside, and was in the midst of painstaking examination of the ground around the shrubbery when he was startled by the sound of a familiar voice.

"Hi, neighbor."

Ace whirled around. "Uh—Hi, Marge," he replied. "I didn't hear you."

“Looking for anything in particular?” she asked, her squinty eyes peering out from underneath a camouflage-colored sou’wester.

Marge Reynolds was a frumpy, middle-aged widow and a socially responsible member of the Polyester Knit movement. She had been Ace’s neighbor for nearly two years, and her cat, Charmin, was one of Mewnix’s most ardent admirers. Marge was someone who kept her ear to the ground and her eyes fixed in the gap between the drapes she never quite closed. Very little escaped her notice. Ace decided to take her into his confidence, and told her the entire sordid tale. She was particularly interested in the part where Ace thought he saw a man in the bushes.

“You know, it must have been the same man I saw,” she said thoughtfully. “When I opened the door to let Charmin out last night, he was moving about in the bushes near the door. When I asked him what he was doing out there at that time of night, he took off like a shot.”

“What time was that?” Ace queried.

“Oh, I would say it was about ten or ten-thirty,” she replied.

“Can you describe him?”

“He was rather tall and slender,” she said. “He was wearing a long raincoat and a hat that totally hid his face. He was holding something in his hand. Could have been a tool. Maybe even a weapon. You know, someone else might have seen him. I think we should ask the manager.”

“Great idea!”

The two would-be sleuths strode across the parking lot to the manager’s office. But when they got there, they found the door standing wide open and no one in sight! They exchanged questioning glances. Then Marge expressed the concern that weighed heavily on both their minds:

“I wish Frank and Joe Hardy were here right now,” she said grimly.

Different Strokes

Across town, the sounds of the last bus leaving for Bayport drifted in through a broken pane in the only window in the shabby, one-room apartment. Oblivious to the noise from the street below, the mysterious figure reached up to the corner of the hardbound Casebook and turned the page.

Casebook No. 16, March 21: Got a new client today, a man by the name of Barry Mountebank. As the chief Spin Doctor for a prominent politician, Barry wants me to develop a word processor that can create different descriptions of the same events, depending on which way the wind is blowing on a given day.

I thought it might be profitable to create a demo that would show how you can filter and translate the keystrokes arriving at a Delphi application, turning keystrokes received into whatever keystrokes you please. When I proposed this to the client, he got very excited.

The technique is exceedingly simple. The Delphi Application object recognizes an OnMessage event that lets anyone tie directly into the message chain for all components in that application.

I set out to create a simple application that would demonstrate three points:

1. It is an easy process to exchange one keystroke for another;
2. All key swaps performed in this manner are automatically sent to all components within the application, even other forms; and
3. Swapping can be turned on and off on the fly.

Figure 1.5 shows the example I created, with two forms up on the screen. Selecting the appropriate radio button switches filtering off and on. When filtering is on, the capital and lowercase “a” are reversed, the backspace key acts as the delete key, and the delete key and shift-F5 combination both take on the former role of the backspace key. The code is included in KSMAIN.PAS and KSFORM2.PAS, shown as Listings 12.7 and 12.8.

FIGURE 12.5 Filtering Delphi’s keystrokes.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Listing 12.7 Main form of the key swapping demo

```
{————————————————————————————————————————}
{                Key Swapping Demo                    }
{             KSMAIN.PAS : Main Form                  }
{             By Ace Breakpoint, N.T.P.               }
{              Assisted by Don Taylor                 }
{                                                     }
{ A simple application that demonstrates the          }
{ selective filtering and substitution of one         }
{ key for another, throughout an application.         }
{                                                     }
{ Written for *Kick-Ass Delphi Programming*           }
{ Copyright (c) 1996 The Coriolis Group, Inc.         }
{               Last Updated 3/21/96                  }
{————————————————————————————————————————}

unit KsMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, KSForm2, ExtCtrls;

type
  TForm1 = class(TForm)
    ExitBtn: TButton;
    ShowBtn: TButton;
    Form1Memo: TMemo;
    Bevel1: TBevel;
    KeyHandlerRBGroup: TRadioGroup;
    procedure FormCreate(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
    procedure ShowBtnClick(Sender: TObject);
   private
    procedure OnAppMessage(var Msg : TMsg; var Handled : Boolean);
   public
```

```
    { Public declarations }
   end;

const
 Shifted : Boolean = False;

var
  Form1: TForm1;
implementation

{$R *.DFM}

procedure TForm1.OnAppMessage(var Msg : TMsg; var Handled : Boolean);
begin
 if KeyHandlerRBGroup.ItemIndex = 1
 then with Msg do
  begin
   case Message of
    WM_KEYDOWN :
     begin
      case WParam of
       VK_SHIFT : Shifted := True;
       VK_F5 : if Shifted then WParam := VK_BACK;
       VK_DELETE : WParam := VK_BACK;
       VK_BACK : WParam := VK_DELETE;
      end; { case }
     end;

    WM_CHAR :
     begin
      case chr(WParam) of
       'a' : WParam := ord('A');
       'A' : WParam := ord('a');
      end; { case }
     end;

    WM_KEYUP :
     begin
      case WParam of
       VK_SHIFT : Shifted := False;
      end; { case }
     end;
   end; { case }
  end; { with }
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 Application.OnMessage := OnAppMessage;
 KeyHandlerRBGroup.ItemIndex := 0;
end;

procedure TForm1.ExitBtnClick(Sender: TObject);
begin
 Close;
end;
```

```
procedure TForm1.ShowBtnClick(Sender: TObject);
begin
 Form2.Show;
end;

end.
```

Listing 1.8 Secondary entry form for the swapping demo

```
{————————————————————————————————————————————————————}
{                Key Swapping Demo                     }
{         KSFORM2.PAS : Secondary Entry Form           }
{             By Ace Breakpoint, N.T.P.                }
{              Assisted by Don Taylor                  }
{                                                      }
{ A simple application that demonstrates the           }
{ selective filtering and substitution of one          }
{ key for another, throughout an application.          }
{                                                      }
{ Written for *Kick-Ass Delphi Programming*            }
{ Copyright (c) 1996 The Coriolis Group, Inc.          }
{               Last Updated 3/21/96                   }
{————————————————————————————————————————————————————}

unit KsForm2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dia-
    logs, StdCtrls, ExtCtrls;

type
  TForm2 = class(TForm)
    CloseBtn: TButton;
    Bevel1: TBevel;
    Form2Memo: TMemo;
    procedure CloseBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form2: TForm2;

implementation

{$R *.DFM}

procedure TForm2.CloseBtnClick(Sender: TObject);
begin
 Close;
end;

end.
```

It was helpful to dig into the source code for the **TApplication** object, to see just how messages were processed by default, and how any OnMessage event handler I might create should participate in the overall process. Specifically, what should I do (if anything) with the **Handled** variable passed to the event handler? Listing 12.9 reveals the source for the **TApplication**'s **ProcessMessage** method, which is called as part of any application's endless message processing loop.

Listing 12.9 Source code for TApplication's ProcessMessage method

```
function TApplication.ProcessMessage: Boolean;
var
  Handled: Boolean;
  Msg: TMsg;
begin
  Result := False;
  if PeekMessage(Msg, 0, 0, 0, PM_REMOVE) then
  begin
    Result := True;
    if Msg.Message <> WM_QUIT then
    begin
      Handled := False;
      if Assigned(FOnMessage) then FOnMessage(Msg, Handled);
      if not IsHintMsg(Msg) and not Handled and not IsMDIMsg(Msg) and
        not IsKeyMsg(Msg) and not IsDlgMsg(Msg) then
      begin
        TranslateMessage(Msg);
        DispatchMessage(Msg);
      end;
    end
    else
      FTerminate := True;
  end;
end;
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Previous | Table of Contents | Next

Listing 12.9 reveals the source of the **Handled** variable. As can be seen, the **ProcessMessage** method is called to detect and process any message waiting in the queue. If a message is present, it is removed from the queue. If the message is **WM_QUIT**, the **FTerminate** field is set **True**; otherwise, **Handled** is set to **False**. and if it is defined, the OnMessage handler is called. If the message comes back with **Handled** set to **False**, and it is not one of several message types (a hint message, an MDI-related message, a control notification message or a dialog message), then the standard **TranslateMessage** and **DispatchMessage** routines are called to process the message. It is evident that setting **Handled** to **True** in the OnMessage event handler would halt any further processing of the message. I just wanted to substitute one keystroke for another, and let processing go on as usual. Therefore, I did not touch the **Handled** variable.

My OnMessage handler is straightforward. If the Filtered radio button is selected, the case statement picks off selected messages and substitutes key values, using Windows-defined virtual key constants for the control keys. One item of note: Detecting a shifted character is a two-step process. Since this routine only receives single keystrokes, it doesn't know whether any of the shift keys (Alt, Ctrl and Shift) are being held down when other keys are pressed. I had to detect the shift presses and releases by first looking for a **VK_SHIFT** as the **wParam** value passed during **WM_KEYDOWN** and **WM_KEYUP** messages, and then on detection of a **VK_SHIFT**, storing the shift status in a boolean variable.

The OnMessage handler belongs to the application and not the main form, so there is no provision for setting it as a property during program design. Instead, it gets installed at runtime as part of the main form's OnCreate handler.

End of entry, March 21.

Playing a .WAV File

The sinister character in the long raincoat began to quiver, as at last it looked up from the Casebook. A high-pitched cackle began to emanate from the twisted mouth, daring the mustache to stay in synchronization with the lips below it.

"Now I can absorb all the material in this Casebook—information so powerful, it can only be used for Good or for Evil. I will become the most respected and powerful Windows programmer on the face of the earth. Thanks to Ace Breakpoint, I will no longer be known merely as 'Bohacker' or 'Hey—you.' Instead, I will become known far and wide by another name: *The Delphi Avenger*.

With my newly found programming knowledge *I'll rule the world!* Ha ha ha ha!...."

The fit of uncontrolled laughter lasted for nearly ten minutes. Then the Avenger opened a fresh pack of HoHos and smugly turned to the next page.

Some Sound Advice

Casebook No. 16, March 22: Today, I discovered how to play a .WAV file from my Delphi applications. Not that it's at all difficult. But I was thinking how great it would be to have one of my biggest heroes—Humphrey Bogart—speak when I clicked a button in an application.

Figure 12.6 shows an executing version of the form I created to test my experiments. Listing 12.10 contains the code.

FIGURE 12.6 Ace's WAV form.

Listing 12.10 A demo program that plays .WAV files

```
{————————————————————————————————————————}
{            The .WAV File Player Demo                  }
{             PLAYMAIN.PAS : Main Unit                  }
{            By Ace Breakpoint, N.T.P.                  }
{             Assisted by Don Taylor                    }
{                                                       }
{ Application that demonstrates how to play a .WAV      }
{ file in a Delphi application.                         }
{                                                       }
{ Written for *Kick-Ass Delphi Programming*             }
{ Copyright (c) 1996 The Coriolis Group, Inc.           }
{              Last Updated 3/22/96                     }
{————————————————————————————————————————}

unit playmain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, MMSystem;

type
  TForm1 = class(TForm)
    BadgeBtn: TButton;
    ExitBtn: TButton;
    procedure BadgeBtnClick(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.BadgeBtnClick(Sender: TObject);
begin
 if not PlaySound('badges.wav', 0, SND_FILENAME)
  then MessageDlg('Problem playing sound file', mtError,
         [mbOK], 0);
end;

procedure TForm1.ExitBtnClick(Sender: TObject);
begin
 Close;
end;

end.
```

At first, I had assumed I would have to use a MediaPlayer component to play a file. I soon discovered an alternative low-level function called **PlaySound** in the MMSystem unit. I simply gave it the name of the file and the constant **SND_FILENAME**, which indicated I wanted the routine to play a sound stored in a file. It doesn't get much easier than that.

Note to myself: The file used for my experiment (BADGES.WAV) is an edited excerpt from the famous "Stinking Badges" dialog in the classic 1948 Bogie movie, "The Treasure of the Sierra Madre." It's one of my faves.

End of entry, March 22.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

A Disconcerting Discovery

The Avenger closed the Casebook and leaned back, considering the events of the last several hours. It had been a long night, and one fraught with peril. But the ruse had worked perfectly. Just as expected, Breakpoint had been lured away by the sound of a helpless female voice. As soon as the former private detective had left his office, the Delphi Avenger was waiting in perfect coordination to break in and steal the precious Casebook.

Breakpoint was probably a basket case by now. But what became of him was of little importance. The only thing that mattered was the Casebook and the intelligence it contained. The end most certainly justified the means. If that meant the end of Ace Breakpoint, so much the better.

The Avenger began to pat the leather-covered oracle, then suddenly stopped. It was now apparent that all had not gone perfectly, and that something was missing. A frantic search through the pockets of the raincoat came up empty.

Beads of nervous sweat began to pop out on the Avenger's forehead. *Maybe it had merely fallen out of a pocket, and it was still in the car. Perhaps it had been left on the top of the vehicle an3d had slipped off, and was now floating in a gutter somewhere.*

But there remained the likelihood that it was lying somewhere near Breakpoint's office. And that would mean it was...evidence. Very *important* evidence.

So important, it would be worth the risk of being caught to get it back.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

CHAPTER 13
A Revelation in the Mud

DON TAYLOR

- Setting a minimum size for a resizeable form
- Splash screens
- Global data modules
- Preemptive miltitasking

With Ace hard in pursuit, the nefarious Delphi Avenger soaks up Ace's hard-won programming expertise in every area from splash screens to preemptive multitasking.

Helen rushed into Ace's office. "I'm so sorry you lost your Casebook. It's gonna be all right, Baby," she said, slipping her arms around Ace and pressing her cheek against his. "I would have been here sooner, but it's so slippery out there."

Helen Highwater had come from a well-to-do family, but insisted on making her own way in the world. To look at her feminine, five-foot-five slender frame and the strawberry blond hair falling just above her shoulders, few would suspect just how determined Helen could be. So far, she had made it through college to a management position at an outlet store at the mall. But her bigger dream—as yet unfulfilled—was to spend the rest of her life as the wife of Ace Breakpoint.

"Not lost, Helen. That Casebook was *stolen*. It was a con from start to finish, and I fell for it like some rube from Bayport."

Ace related the entire story of what had happened the night before, and of his

conversation earlier that morning with Marge Reynolds.

“So at least we have a fairly good description of the perpetrator,” he concluded. “I’m just not sure it leads anywhere. We left a note on the Manager’s door, asking him to call when he returns.”

“Don’t you see?” Helen asked incredulously. “It’s got to be Melvin Bohacker. The description fits. I’m sure he is just being vindictive about what happened in *The Case of the Duplicitous Demo*, and he’s trying to get back at both of us. He probably hired that woman to make the bogus phone call for him. I’ll bet he’s sitting at home right now, gloating over the whole situation.”

“I don’t think it’s that simple, Helen,” Ace replied. “You didn’t see Bohacker’s face when I—”

Ace’s attention had been grabbed by something outside. Through the blinds in the side window, he could make out the shape of a man moving about in the bushes. He was a tallish man, wearing a brimmed hat and a long, khaki-colored raincoat. He moved closer to the window and peered inside.

“That’s him!” Ace shouted. “That’s the man Marge described, and undoubtedly the man who stole my Casebook. He has returned, just as sure as another sequel to *Nightmare on Elm Street*—and I’ve got some *very* pointed questions to ask him!”

The intruder’s eyes shot wide open when Ace cried out, but as soon as Ace moved toward the door, the mysterious stranger bolted from the shrubbery, heading diagonally across the parking lot. Ace followed in hot pursuit, hampered greatly by the slippery mud that now completely covered the asphalt.

“Stop!” Ace commanded. But the raincoated figure just scrambled faster. Ace managed to give an extra surge, catching up with the stranger and tackling him in front of the Manager’s office. The tall interloper hit the ground hard, letting out a loud groan. Together, they slid a full six feet in the mud before coming to a stop.

“Hey—what’s goin’ on here?” came a voice from behind. Ace wrenched his head around far enough to see Marvin Gardens, the property manager, walking up behind him.

“I’ve caught the intruder Marge saw last night,” Ace replied, vainly trying to keep the rain out of his eyes. “The guy who lifted my Casebook.” With some effort, Ace managed to stand up, still maintaining an iron grip on the man in the raincoat. Both men were covered with mud from head to toe.

“Yeah, I got yer note,” said Gardens, chewing on the stogie clamped between his crooked, yellow teeth. “I was gonna call ya. But dis is no intruder, Breakpoint. Say ‘Hello’ ta my new groundskeeper, Sergei Stakupopov. He don’t speak much English, but dat much he’ll understand.”

“Wait a second,” Ace protested. “Last night this man was seen holding some kind of weapon. He was confronted twice, and each time he ran away. Gardener or not, that says he’s guilty.”

“Where he’s from, people live in fear of da secret police,” Gardens replied, after taking a puff off the cheap stogie. “In his country, a challenge yelled by someone could be da last thing ya ever hear. He’s here onna Green Card, and he’s deathly afraid he’s gonna lose it, which would mean his family would have ta return to da homeland. That’s why he’s willing ta work long, hard hours. He was trimmin’ da bushes by your apartment last night—Marge probably saw him holdin’ a pairra shears, dat’s all.”

Ace loosened his grip on the immigrant, then sheepishly shook hands with the groundskeeper and apologized profusely. Sergei watched him guardedly, then smiled politely and said “Hello.”

The former P.I. trudged back to his office, where Helen was waiting with a million questions. He related his experiences of the past few minutes, then just stood there, hanging his head.

“Hey, it’s going to be okay,” Helen affirmed once again. “This is just a temporary setback. Now get out of that muddy trenchcoat before you catch a cold.”

Reluctantly, Ace complied. “I wish I could believe it’s only temporary,” he said, choosing a fresh trenchcoat and hat from the row of identical outfits in his closet.

“Of course it is, Sweetheart,” she replied. “Listen, my lunch hour is over, and I’ve got to get back to the shop. But give me a call this afternoon. I’ll stop by after work, to see how things are going.”

She gave him a kiss on the cheek and then slipped out into the pouring rain.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Resizing Forms

The Avenger popped open another pack of HoHos and took a big bite. The decision not to return to Breakpoint's apartment office had been a gamble, but hopefully it would pay off. After all, the chances of the has-been detective finding any incriminating evidence accidentally left behind were slim to none. It was better to absorb as much information as quickly as possible.

Casebook No. 16, March 25: I had often wondered how some applications were able to limit the minimum size of a resizable window, so I determined to find out how it was done. I didn't have any idea how easy that task would be. I did suspect the process had something to do with messages, however. By this time, I had realized that nearly everything that takes place within the Windows environment is somehow related to messaging.

I was also intrigued by the ability of some rather complex forms to maintain an integrity of their component parts, even if those forms were resized. That would also be part of today's independent investigations.

I wanted a form with a minimum of four major components. I created a rough form containing a speedbutton, a memo, and two regular buttons. It's shown in Figure 13.1. My first goal was to limit the resizing process to a minimum form size specified by the programmer. The answer came in a message called **WM_GETMINMAXINFO**.

FIGURE 13.1 The Form Resize Demo Form at design time.

WM_GETMINMAXINFO is a Windows message that notifies an application when the system is checking the size of a window, so it has an opportunity to change default values. Among these values are the minimum and maximum "tracking sizes," which determine the horizontal and vertical ranges for the width and height of a window. The default minimum is the size of an icon; the default maximum is the size of the entire screen.

The actual parameter passed to the message handler for **WM_ GETMINMAXINFO** is a point that represents the X,Y offset from the upper-left corner of the window, measured in pixels.

The challenge, then, would be to write a replacement message handler, defining points that would describe minimum and maximum window sizes. I first declared variables for the minimum height and

width, so I could easily manipulate these values.

The next step was to determine the point representing the lower-right corner of the form. The **lParam** of the default **WM_GETMINMAXINFO** handler is a pointer to an array of five point structures. Thankfully, the wizards at Borland thoughtfully provided a **TWMGetMinMaxInfo** message type that hides most of the complexity.

Listing 13.1 contains the complete source for my resizing experiment. The listing includes the handler that resulted after several head-scratching attempts (I was surprised by the "interesting" effects that could be attained by forgetting such frivolities as calling the inherited handler). As can be seen in the listing, simply dereferencing the **MinMaxInfo** portion of the message gives quick and easy access to the points defined by **ptMinTrackSize** and **ptMaxTrackSize**. I added some code in the form's OnCreate handler to compute values for **MinWidth** and **MinHeight**, based on the sizes of the form's components at startup.

Listing 13.1 Source code for the Form Resize Demo

```
{————————————————————————————————————————————————————}
{                  Form Resize Demo                    }
{                RS.PAS : Main Form                    }
{             By Ace Breakpoint, N.T.P.                }
{              Assisted by Don Taylor                  }
{                                                      }
{ This application demonstrates how a combination      }
{ of panels with alignments set, along with gaining    }
{ control of Windows' resizing messaging, can create   }
{ flexible forms that follow and limit resizing        }
{ commands.                                            }
{                                                      }
{ Written for *Kick-Ass Delphi Programming*            }
{ Copyright (c) 1996 The Coriolis Group, Inc.          }
{               Last Updated 3/25/96                   }
{————————————————————————————————————————————————————}

unit Rs;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Buttons;

type
  TRSMainForm = class(TForm)
    ControlPanel: TPanel;
    RSMemoPanel: TPanel;
    RSMemo: TMemo;
    BtnPanel: TPanel;
    SBPanel: TPanel;
    QuitSB: TSpeedButton;
    QuitBtn: TButton;
    SBComboPanel: TPanel;
    ComboBox1: TComboBox;
    SpeedButton1: TSpeedButton;
    Button1: TButton;
    procedure QuitBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormResize(Sender: TObject);
```

```
  private
    { Private declarations }
    procedure WMGetMinMaxInfo(var Msg: TWMGetMinMaxInfo);
      message WM_GET MINMAXINFO;
  public
    { Public declarations }
  end;

var
  RSMainForm: TRSMainForm;
  MinWidth : Integer;
  MinHeight : Integer;

implementation

{$R *.DFM}

procedure TRSMainForm.QuitBtnClick(Sender: TObject);
begin
 Close;
end;

procedure TRSMainForm.FormCreate(Sender: TObject);
begin
 MinWidth := RSMemoPanel.Width + BtnPanel.Width + 10;
 MinHeight := RSMainForm.Height
   - (RSMainForm.ClientHeight - (RSMemo.Top + RSMemo.Height)) + 10;
end;

procedure TRSMainForm.WMGetMinMaxInfo(var Msg: TWMGetMinMaxInfo);
begin
 inherited;
  with Msg.MinMaxInfo^ do
  begin
    with ptMinTrackSize do
    begin
      X := MinWidth;
      Y := MinHeight;
    end;  { with }

    with ptMaxTrackSize do
    begin
      X := Screen.Width;
      Y := Screen.Height;
    end;  { with }
  end;  { with }
end;

procedure TRSMainForm.FormResize(Sender: TObject);
begin
 RSMemo.Height := RSMemoPanel.Height - (2 * RSMemo.Top);
 RSMemoPanel.Width := RSMainForm.ClientWidth - BtnPanel.Width;
 RSMemo.Width := RSMemoPanel.Width - (2 * RSMemo.Left);
end;

end.
```

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Now it was time to play with the arrangement of components on the form, so their positioning would be consistent even if the form containing them was resized.

When I first started working with panels, they gave me headaches. But as I got to know them better, I fell in love with three cool characteristics they share. First, they can, with the **Alignment** property, be forced into absolute correspondence with their parents. If you set a panel's **Alignment** to **alTop**, it will take over the entire top of the form it's placed on. Second, other components (memos, buttons, etc.) placed on a panel will stay in a fixed position relative to that panel, as long as it is not resized. Finally, a panel placed on another panel has the same characteristics as a panel placed on a form: If the **Alignment** is set to **alBottom**, it will spread out and glue itself across the bottom of the parent panel.

All of this creates a neat little set of behaviors that make consistent form resizing possible. I had several goals for the form shown in Figure 13.1:

- Panel3 would have a fixed height and occupy the entire top of the form, so its length would always track the width of the form;
- SB1 would remain in a fixed position relative to the left side of Panel3 (and thus the upper-left corner of the window);
- Panel4 would be aligned with the right side of Panel3, so Panel4 would not change in size, but would track the right side of Panel3 as the length of Panel3 changed;
- Panel5 (which contains Memo1) would be aligned with the left side of the form, and its height would track with form resizing;
- Panel1 (which contains Panel2) would be aligned with the right side of the form, and its height would also track with form resizing;
- Panel2 (which contains Button1 and Button2) would be aligned with the bottom of Panel1, so it would remain the same size and follow the

bottom of the Panel1 (and thus the form) during a resize; and

- Button1 and Button2 would be placed in a fixed position on Panel2, so they would each remain fixed with respect to both the bottom and right side of the form.

Whew! I got to work setting the properties of the various panels. It's a little confusing working with panels, until you learn the rules of etiquette. Panel alignments set to **alTop** or **alBottom** will always take precedence over those set **alLeft** or **alRight**. Anyway, I set Panel3 to **alTop**, Panel1 and Panel4 to **alRight**, Panel5 to alLeft, and Panel2 to **alBottom**. I set the **BevelOuter** properties on Panel4 and Panel2 to **bvNone**, so they would "disappear," and I set the color of Panel3 and Panel4 to **clGray**, to set them off aesthetically from the rest of the form. I also added a combobox and another speedbutton to Panel4, so I could verify that tracking was working properly. I then renamed the panels and blanked out all their captions.

I decided I wanted the overall panel containing the buttons (formerly Panel1) to remain a constant width, while enabling the panel holding the memo to take up the rest of the available width. I would then automatically resize Memo1 to take up all of its parent panel, less some margins on each side. I was able to accomplish this with a little arithmetic in the form's OnResize event handler.

After several tries, I got everything just the way I wanted it. Figure 13.2 shows the demo form as it appears at the start of the program; Figure 13.3 is a shot of how the form looks when it has been resized to the specified limits.

End of entry, March 25.

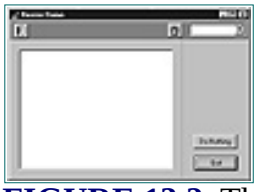

FIGURE 13.2 The Resize Demo form at program startup.

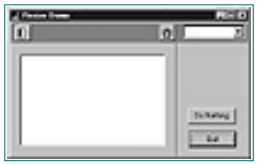

FIGURE 13.3 The Resize Demo form after resizing to the limits.

Making a Splash

The mysterious figure closed the Casebook and reached for the telephone. The instrument readily accepted the seven digits keyed in, then produced a series of soft buzzes that indicated the line was ringing at the other end. There was a click. Then the Avenger began to speak to an intimately familiar voice on the other end of the line.

"Hello, Baby Cakes... Yes, it's me. Just thought you'd like to know how things went last night. I broke into Ace Breakpoint's apartment as planned, and stole his Casebook right out from under his nose. The whole thing was almost flawless....Uh-huh.... I want you to know how much I appreciate how

invaluable your part was in all of this. I couldn't have done it without you.... What? Yes, it's been a long time coming—and believe me, revenge is sweet. Right. Listen, Baby Cakes—drop everything and meet me tonight at nine o'clock at the Gates Motel, just off Highway 101. Right—up on the hill, just outside of Norton City. Uh-huh... Tonight I'll show you the book that will change both our lives, and make me the world's preeminent Windows programmer.... That's right, Baby, *me*—the Delphi Avenger. 9:00 P.M. Don't be late. Bye."

The Avenger hung up the phone and picked up the Casebook, and chuckling loudly, began once again to peruse its pages.

Casebook No. 16, March 26: Today the Bag Man called me again. This time he wants me to create a "splash screen" component he can use in a variety of programs. He basically wants to drop a component on the main form of an application, set a few properties, and have it pop up a screen before the main form shows up.

I got to thinking about what I would like to see in a generic splash screen. Not a great deal. I would probably make it modal, so I could prevent the program from executing until the screen had properly displayed its contents to the user. I would want to kill the screen with a click of a button, a timeout, or both. And of course, I would want the ability to include a graphic on the screen. I sat down at the computer and created the initial design shown in Figure 13.4.

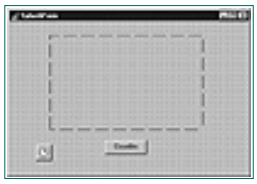

FIGURE 13.4 A first cut at the splash screen, shown at design time.

Now, how was I going to turn this into a component? After a couple of false starts, I decided the best approach would be to create a new component with a **TForm** object as its core, but with another object "wrapped" around the **TForm** to manage it. This wrapper object could manage only the properties associated with the splash screen object, giving the splash screen a simple user interface. The wrapper could also manage the creation and destruction of the form, all from one simple command issued by the owner of the wrapper object. I decided to call the wrapper a **TSplashDialog**.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

As part of my initial specification, I decided to write a simple-minded test application—a form containing only a button—that would eventually contain a **TSplashDialog**. The code for the test application is shown in Listing 13.2.

Listing 13.2 The test application for TSplashDialog

```
{————————————————————————————————————————}
{              Splash Screen Component                  }
{      SPLSHMN.PAS : Demo Program Main Form             }
{            By Ace Breakpoint, N.T.P.                  }
{             Assisted by Don Taylor                    }
{                                                       }
{ A simple program that demonstrates the use of         }
{ the TSplashDialog component. Try combinations         }
{ of time delays, sizes, graphics and such to see       }
{ the flexibility provided.                             }
{                                                       }
{ Written for *Kick-Ass Delphi Programming*             }
{ Copyright (c) 1996 The Coriolis Group, Inc.           }
{              Last Updated 3/26/96                     }
{————————————————————————————————————————}

unit SplshMn;

{.$define Test }

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, SplshDlg;

type
  TForm1 = class(TForm)
    QuitBtn: TButton;
```

```
    SplashDialog1: TSplashDialog;
    procedure QuitBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    {$ifdef Test }
    SplashDialog1 : TSplashDialog;
    {$endif}
  public
    { Public declarations }
  end;
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.QuitBtnClick(Sender: TObject);
begin
 Close;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 {$ifdef Test}
 SplashDialog1 := TSplashDialog.Create(Application);
 {$endif}
 SplashDialog1.Execute;
end;

end.
```

This brings up something noteworthy: Never test a component under construction by placing it in Delphi's library. Bad—very bad—stuff can happen if things aren't right. Plus, it's a slow, tedious process to recompile the library for each and every run. Instead, include the component's unit in a test program, and then create an instance of the component as the object it is. That is what is done here. Through the use of a conditional compiler directive and a constant I called **Test**, this simple program has been given two modes. As shown (with the period preceding the '$' in the directive), the $define becomes only a comment, and the program can be used to test the installed version of the component. Deleting the period enables the conditional code, which in this program declares a **TSplashDialog** with the same name (SplashDialog1) that the IDE will give the component when it is dropped on the form. Use of the conditional test in the OnCreate handler also creates an instance of SplashDialog1.

As can be seen, I decided to put the dialog into action with a method called **Execute**, in the proud tradition of the specialized system dialogs like **TOpenDialog**.

I had set the stage for a **TSplashDialog**. Now, what properties should it have? It must enable the programmer to specify its size, although I would assume it would always be displayed at the center of the user's screen. The screen needs to be told whether it has a button, and if so, the caption on that button. It needs to know if the dialog is to have a timeout, and if so, what the time delay will be. It needs a **TPicture** object to plug into the **TImage** component. And to give it some flexibility with that graphic, the programmer should be able to specify the alignment, whether or not the **TImage** is to be autosized to the graphic's dimensions, and whether or not the graphic is to be stretched to fit the **TImage** dimensions.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

A couple of hours later, I had a more-or-less finished component. The source code appears in Listing 2.3.

Listing 13.3 Source code for the TSplashDialog component

```
{————————————————————————————————————————}
{              Splash Screen Component                 }
{           SPLSHDLG.PAS : Component unit              }
{             By Ace Breakpoint, N.T.P.                }
{              Assisted by Don Taylor                  }
{                                                      }
{ This unit describes a specialized component that     }
{ pops up a "splash screen" whenever your program      }
{ wants to display one (usually at program startup)    }
{                                                      }
{ Written for *Kick-Ass Delphi Programming*            }
{ Copyright (c) 1996 The Coriolis Group, Inc.          }
{               Last Updated 3/26/96                   }
{————————————————————————————————————————}

unit SplshDlg;

{.$define Test }

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, ExtCtrls, StdCtrls;

type
  ESplashConflict = class(Exception);
  TImageAlign = (iaNone, iaTop, iaBottom, iaLeft, iaRight,
     iaClient, iaAllAboveButton);

  { TSplashForm is the actual form displayed to the user. It contains
    a TImage, a TButton and a TTimer, so the programmer can vary how
```

```
  the splash screen is used. }
TSplashForm = class(TForm)
  CloseBtn: TButton;
  Image: TImage;
  DelayTimer: TTimer;
  procedure CloseBtnClick(Sender: TObject);
  procedure Button1Click(Sender: TObject);
  procedure DelayTimerTimer(Sender: TObject);
private
  { Private declarations }
public
  { Private declarations }
end;

{ TSplashDialog is the wrapper that surrounds the TSplashForm.
  The form is owned by the TSplashDialog, so it can "automatically"
  be created, set up, executed and destroyed at any time. The
  TSplashDialog makes available only those properties used for the
  splash dialog, then passes them on to the TSplashForm when it
  is created. }
TSplashDialog = class(TComponent)
private
  FAlign : TImageAlign;
  FAutoSize : Boolean;
  FButtonCaption : String;
  FCaption : String;
  FDelay : Word;
  FHasButton : Boolean;
  FHasDelay : Boolean;
  FHeight : Word;
  FPicture : TPicture;
  FStretch : Boolean;
  FWidth : Word;
  procedure SetCaption(Value : String);
  procedure SetDelay(Value : Word);
  procedure SetHasButton(Value : Boolean);
  procedure SetHasDelay(Value : Boolean);
  procedure SetHeight(Value : Word);
  procedure SetPicture(Value : TPicture);
  procedure SetWidth(Value : Word);
public
  constructor Create(AOwner : TComponent); override;
  destructor Destroy;
  function Execute : Boolean; virtual;
published
  property Align : TImageAlign read FAlign write FAlign;
  property AutoSize : Boolean read FAutoSize write FAutoSize;
  property ButtonCaption : String read FButtonCaption
                                   write FButtonCaption;
  property Caption : String read FCaption write SetCaption;
  property Delay : Word read FDelay write SetDelay;
  property HasButton : Boolean read FHasButton write SetHasButton;
  property HasDelay : Boolean read FHasDelay write SetHasDelay;
  property Height : Word read FHeight write SetHeight;
  property Picture : TPicture read FPicture write SetPicture;
  property Stretch : Boolean read FStretch write FStretch;
```

```
    property Width : Word read FWidth write SetWidth;
  end;

procedure Register;

implementation

{$R *.DFM}

var
 SplashDlg: TSplashDialog;

procedure TSplashDialog.SetCaption(Value : String);
begin
 if Value <> FCaption
  then FCaption := Value;
end;

{ Set the value of FHasButton. If the user has specified he
  wants no button and no delay timer, raise an exception --
  without both, the screen has no way to be cleared! }
procedure TSplashDialog.SetHasButton(Value : Boolean);
begin
 if not Value and not FHasDelay
  then raise
    ESplashConflict.Create('Must have either a button or a delay!')
  else FHasButton := Value;
end;

{ Set the value of FHasDelay, protecting against the anomaly
  cited above. }
procedure TSplashDialog.SetHasDelay(Value : Boolean);
begin
 if not Value and not FHasButton
  then raise
    ESplashConflict.Create('Must have either a button or a delay!')
  else FHasDelay := Value;
end;

procedure TSplashDialog.SetHeight(Value : Word);
begin
 if (Value <> FHeight) and (Value > 10)
  then FHeight := Value;
end;

procedure TSplashDialog.SetWidth(Value : Word);
begin
 if (Value <> FWidth) and (Value > 20)
  then FWidth := Value;
end;

procedure TSplashDialog.SetDelay(Value : Word);
begin
 if (Value <> FDelay) and (Value > 0)
  then FDelay := Value;
end;
```

```
procedure TSplashDialog.SetPicture(Value : TPicture);
begin
 if Value <> nil then FPicture.Assign(Value);
end;

constructor TSplashDialog.Create(AOwner : TComponent);
begin
 inherited Create(AOwner);

 { Set defaults }
 FAlign := iaAllAboveButton;
 FAutoSize := False;
 FStretch := False;
 FButtonCaption := 'OK';
 FCaption := copy(ClassName, 2, Length(ClassName) - 1);
 FDelay := 3500;
 FHasButton := True;
 FHasDelay := True;
 FHeight := 200;
 FWidth := 300;
 FPicture := TPicture.Create;

 {$ifdef Test }
 FPicture.LoadFromFile('splash.bmp');
 FHasDelay := False;
 {$endif }

end;

destructor TSplashDialog.Destroy;
begin
 FPicture.Free;
 inherited Destroy;
end;

{ The Execute method is the primary focus. It's the method
  called by the owner of the TSplashDialog when the splash
  screen is desired. Execute creates a SplashForm and modifies
  it according to the parameters given to the SplashDialog.
  The SplashForm is destroyed when it has closed. }
function TSplashDialog.Execute : Boolean;
var
 SplashForm : TSplashForm;
begin
 try
  SplashForm := TSplashForm.Create(Application);
 except
  on E:Exception do
   begin
    MessageBeep(MB_ICONERROR);
    Exit;
   end;
 end; { try }

 with SplashForm do
```

```
  begin
   Position := poScreenCenter;
   Caption := FCaption;
   Height := FHeight;
   Width := FWidth;

   if FAlign = iaAllAboveButton
    then begin
          if FHasButton
           then begin
                 Image.Align := alTop;
                 Image.Height := ClientHeight - CloseBtn.Height - 15;
                end
           else Image.Align := alClient;
         end
    else Image.Align := TAlign(Ord(FAlign));
   Image.AutoSize := FAutoSize;
   Image.Stretch := FStretch;
   if Image.Picture <> nil
    then Image.Picture.Assign(FPicture);

   if FHasButton
    then begin
          CloseBtn.Caption := FButtonCaption;
          CloseBtn.Left := (ClientWidth - CloseBtn.Width) div 2;
          CloseBtn.Top := ClientHeight - CloseBtn.Height - 10;
         end
    else CloseBtn.Visible := False;

   if FHasDelay
    then begin
          DelayTimer.Interval := FDelay;
          DelayTimer.Enabled := True;
         end;

   try
    ShowModal;
   finally
    Free;
   end; { try }
  end; { with }
end;

procedure TSplashForm.CloseBtnClick(Sender: TObject);
begin
 Close;
end;

procedure Register;
begin
 RegisterComponents('Ace''s Stuff', [TSplashDialog]);
end;

procedure TSplashForm.Button1Click(Sender: TObject);
begin
 Close;
```

```
end;

procedure TSplashForm.DelayTimerTimer(Sender: TObject);
begin
 Enabled := False;
 Close;
end;

end.
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Several comments need to be made about this code. Here again, I used a conditional directive to give the component two modes. When in the test mode, it automatically loads a special test bitmap and disables the timer.

I added some code to prevent the condition where neither a button nor a timer was selected. (This would mean there would be no way to end the modal dialog!) I also declared an enumerated type (**TImageAlign**) to functionally extend the **TAlign** type by adding a value called **iaAllAboveButton**. This new value would indicate the user wanted to use the form's entire client area, but only that portion above the button. Oh, yes—I also declared a special exception class as a catch-all for problems detected when the programmer was setting properties.

By far, the most interesting part of this project was selecting the **TPicture** and placing it in the **TImage**. After triggering several exceptions for system access violations, I finally found myself combing through the VCL source code, looking for instances of selecting and assigning bitmaps. Once the answers were found, I realized how easy the process had been made, thanks to the foresight of Delphi's designers. When you declare a property of type **TPicture**, Delphi's IDE already knows how to work with it. You create an instance of a **TPicture**, and the IDE will call up the Picture Editor. Once a bitmap is selected by the Picture Editor, it will automatically be stored in the stream when the form file is closed, which means the bitmap will be there the next time the file is opened.

True to its design goals, the **TSplashDialog** wrapper manages the critical properties of the form. When its **Execute** method is called, **TSplashDialog** creates an instance of the form, sets its properties accordingly, and then calls **ShowModal** to halt all other processing in the program. When execution resumes, the form is destroyed. The test splash screen is shown in Figure 13.5.

FIGURE 13.5 The splash screen making a splash at runtime.

End of entry, March 26.

Ace Gets an Answer

"Hello, Helen? Yeah, Baby. Just calling to let you know there's nothing new. It's a brick wall. I've gone over everything a hundred more times, and I'm right back where I started. No clues whatsoever. I feel absolutely helpless. Some detective *I* am!..."

"Ace, you know you're still a good P.I.— one of the best. It just so happens you can't do this one on your own," Helen said.

"I suppose you're right," he admitted. "I could use a little help."

"Why don't you talk to The Author," Helen suggested. "Remember how he helped you before?"

"Yeah, I suppose you're right. Guess it's worth a try. Thanks, Baby. Love you."

"Love you, too," she replied.

Ace hung up the phone and grabbed the phone directory. Running his finger down the page, he found the number he was after. He quickly dialed, and the line was as quickly answered.

"Hello, Ace," said the voice.

"Uh, hello," Ace replied. "I guess you already know why I'm calling."

"You're looking for some answers regarding the theft of your Casebook."

"Yeah, I'm feeling totally helpless here. I decided to give you a call."

"I haven't heard from you in quite some time, Ace," said the voice. "Not since *The Case of the Duplicitous Demo*. I gave you some help that night, didn't I?"

"Oh, yes," Ace replied. "You reminded me how special I was, and you gave me the courage to keep going. I don't think I would have made it without your help."

"I wonder why you never thanked me."

"Uh, I suppose it was a combination of things," Ace replied, stalling for time. "Once I had the problem solved, I guess I thought I didn't need any help from anyone else." He paused momentarily. "And I didn't want to bug you. After all, I'm just one of your many characters."

"One of my *favorite* characters. I've missed hearing from you. Sometimes it

makes me sad when you try to do everything on your own. Remember, you can call on me anytime, day or night. It doesn't matter how big the problem is—or how small. But as to your question at the moment. You want to know who stole your Casebook."

"Right. Helen thinks it was Melvin Bohacker and a female accomplice. At first I didn't think so, but now I'm not so sure. *Was* it Bohacker who stole the Casebook?"

"There are three possible answers to that question: 'Yes,' 'No,' and the answer I have for you, 'Not right now.'"

"But I've got to know who took my most valuable possession. I don't know how I'm going to go on without it. And my reputation with Helen as a P.I. is on the line, as well."

"Ace, in your case, life is definitely *not* like a box of chocolates. For you, it's more like event-driven programming: You see, it's not only important that an event occurs, it's equally as important *when* it occurs. Life is a mystery. And in any mystery, the timing of events is critical. Besides, sometimes you learn more by *discovering* things than by just being given the answer."

"So what should I do?" Ace asked plaintively.

"Go to the parking lot and thoroughly search the area near where your car is parked. You'll find the key that will unlock this whole mystery."

"Thanks," Ace replied excitedly. "I won't forget this." He was barely able to get the phone back in the cradle before he was out the door.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Making Data Global to an Application

In the shabby little apartment, the Avenger stuffed down a whole Twinkie and continued to read Ace's purloined Casebook.

Casebook No. 16, March 27: With Delphi 1.0, sharing tables between forms was always bothersome. Although there was an alternative to placing tables and datasources on every form that needed data from a table, it was a clumsy process that involved temporarily creating extra datasources and then later deleting them. I had on many occasions wished for an easier way. When Delphi 2.0 was released, I heard that it included objects called Data Modules that greatly simplified the process. I decided to investigate.

A Data Module, as it turns out, is basically a specialized form that will accept only the standard items on Delphi's Data Access palette. You can build the entire database for an application, including tables, sources, queries and whatever—all in a data module. To make the whole kit and caboodle available to a form, you just include the data module in the form's *implementation* section (*not* the interface section). Components and fields on the data module then become available to data aware components on the form (and to the Object Inspector as well).

I decided to try a simple example using some data for a client, The Chichen Itza Pizza Company. The data is stored in a Paradox table in a file named PIZADATA.DB. The table contains three fields: the name, retail price and actual cost of the company's best-selling products. To this mix, I wanted to add a calculated field to display the percent profit generated by each item.

I first created a DatabaseName alias called "Pizza" to point to the table's directory. Then I created a new data module and gave it the name PizzaData. Onto that module, I dropped a Table and a DataSource, giving them the names ProductTable and ProductSource. I connected ProductSource to ProductTable, and set **AutoEdit** to **False**. I opened the Fields Editor in the table and added all the available fields. Next, I added a new calculated field to hold the percent profit, and then wrote a handler for ProductTable's OnCalc event. The data module can be seen in Figure 13.6. The code for PizzaData is shown in Listing 13.4.

FIGURE 13.6 A data module at design time.

Listing 13.4 Code for the Data Module Demo

```
{————————————————————————————————————————————————————}
{                  Data Module Demo                    }
{             PIZADAT.PAS : Data Module                }
{             By Ace Breakpoint, N.T.P.                }
{              Assisted by Don Taylor                  }
{                                                      }
{ This data module contains a simple Table/            }
{ Datasource combination that links to a Paradox       }
{ data table. A calculated field is provided to        }
{ users of this module.                                }
{                                                      }
{ Written for *Kick-Ass Delphi Programming*            }
{ Copyright (c) 1996 The Coriolis Group, Inc.          }
{               Last Updated 3/27/96                   }
{————————————————————————————————————————————————————}

unit PizaDat;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, DB, DBTables;

type
  TPizzaData = class(TDataModule)
    ProductTable: TTable;
    ProductSource: TDataSource;
    ProductTableName: TStringField;
    ProductTablePrice: TCurrencyField;
    ProductTableCost: TCurrencyField;
    ProductTablePctProfit: TFloatField;
    procedure ProductTableCalcFields(DataSet: TDataSet);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  PizzaData: TPizzaData;

implementation

{$R *.DFM}

procedure TPizzaData.ProductTableCalcFields(DataSet: TDataSet);
begin
 ProductTablePctProfit.Value := 100.0 *
  ((ProductTablePrice.Value - ProductTableCost.Value) /
```

```
    ProductTableCost.Value);
end;

end.
```

I had the data. Now it was time for the demo. I decided to play with two forms. The first form would simply access the data through ProductSource on the data module. This form would display the product's name and the calculated field, and I would give it a Navigator so I could browse through the data.

The other form (which would be the application's main form) would have its own Table and DataSource, plus a DBGrid and a Navigator. In addition, I would provide a set of radio buttons that would permit dynamic switching between the data module and the local source. By selecting the local source, the Navigators on the two forms would work independently, since they would be working with different table objects.

Figure 13.7 shows the two forms up and running. Listing 13.5 illustrates the code for the demo's main form, while Listing 13.6 contains the code for the secondary form.

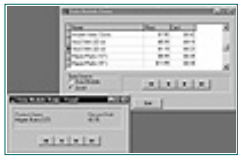

FIGURE 13.7 The Data Module Demo at runtime.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Listing 13.5 Code for the Data Module Demo's main form

```
{————————————————————————————————————————}
{                 Data Module Demo                       }
{              PIZAMAIN.PAS : Main Form                  }
{              By Ace Breakpoint, N.T.P.                 }
{               Assisted by Don Taylor                   }
{                                                        }
{ This demo shows how a form can be connected to         }
{ a data module constructed for a project. This          }
{ form provides a switch to select the source of         }
{ its data -- the module or a local Table/Datasource     }
{ combination.                                           }
{                                                        }
{ Written for *Kick-Ass Delphi Programming*              }
{ Copyright (c) 1996 The Coriolis Group, Inc.            }
{                Last Updated 3/27/96                    }
{————————————————————————————————————————}

unit PizaMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Grids, DBGrids, ExtCtrls, DBCtrls, DBTables, DB, StdCtrls;

type
  TForm1 = class(TForm)
    DBGrid: TDBGrid;
    Navigator: TDBNavigator;
    DataSourceRBGroup: TRadioGroup;
    QuitBtn: TButton;
    LocalTable: TTable;
    LocalDataSource: TDataSource;
    LocalTableName: TStringField;
    LocalTablePrice: TCurrencyField;
```

```
    LocalTableCost: TCurrencyField;
    Bevel1: TBevel;
    procedure FormShow(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure DataSourceRBGroupClick(Sender: TObject);
    procedure QuitBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

uses
 PizaDat, PizaFrm2;

{$R *.DFM}

procedure TForm1.FormShow(Sender: TObject);
begin
 Form2.Show;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 DataSourceRBGroup.ItemIndex := 0;
end;

procedure TForm1.DataSourceRBGroupClick(Sender: TObject);
begin
 if Tag > 0
 then case DataSourceRBGroup.ItemIndex of
       0 : begin
            DBGrid.DataSource := PizzaData.ProductSource;
            Navigator.DataSource := PizzaData.ProductSource;
           end;
       1 : begin
            DBGrid.DataSource := LocalDataSource;
            Navigator.DataSource := LocalDataSource;
           end;
      end { case }
 else Tag := 1;
end;

procedure TForm1.QuitBtnClick(Sender: TObject);
begin
 Close;
end;

end.
```

Listing 13.6 Code for the Data Module Demo's secondary form

```
{──────────────────────────────────────────────────────────}
{                 Data Module Demo                         }
{          PIZAFRM2.PAS : Secondary Form                   }
{            By Ace Breakpoint, N.T.P.                     }
{             Assisted by Don Taylor                       }
{                                                          }
{ This demo shows how a form can be connected to           }
{ a data module constructed for a project. This            }
{ form receives its data from the project's data           }
{ module.                                                  }
{                                                          }
{ Written for *Kick-Ass Delphi Programming*                }
{ Copyright (c) 1996 The Coriolis Group, Inc.              }
{              Last Updated 3/27/96                        }
{──────────────────────────────────────────────────────────}

unit PizaFrm2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, DBCtrls, ExtCtrls, DB;

type
  TForm2 = class(TForm)
    NameDBText: TDBText;
    PctDBText: TDBText;
    Label1: TLabel;
    Label2: TLabel;
    Navigator: TDBNavigator;
    Bevel1: TBevel;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form2: TForm2;

implementation

uses
 PizaDat;

{$R *.DFM}

end.
```

Before compilation, I set the **Active** property for all the table objects to **True**. I guess that's just force of habit.

Both forms declare PizaDat (the data module's unit name) in their implementation section. Once that is done, the fields from the data module become selectable through the Object Inspector for any data-aware component.

As can be seen, it was a straightforward exercise to make the switch between the two data sources with the radio buttons. Well, almost...

It seems that the OnClick event handler for radio buttons gets called during the creation of a form. In this case, the data sources weren't fully constructed and connected, which generated an exception. I chose to avert this situation by using the form's **Tag** property. On initialization, it is zero and there is no attempt to connect the sources. The next time through is a different story, because **Tag** has been set equal to one.

Sure enough, the demo works as designed. When the main form is connected to the data module, the main form displays the percent profit field, and both forms synchronize on the same record, no matter which form is controlling the browsing. When the main form is switched to the local data source, the two forms operate independently. Switching back to the data module instantly synchronizes the two forms.

Of course, a data module is not limited to something as simple as a calculated field. A data module is, after all, an Object Pascal unit that can contain newly defined objects, methods, and handlers for every event in a Table, DataSource, SQL Query, or whatever. In fact, a programmer can implement an entire set of business rules for a company's data. Pretty cool. And very, very powerful.

End of entry, March 27.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Previous | Table of Contents | Next

An Exciting Discovery!

Ace Breakpoint rapidly dialed Helen's work number. She picked up on the second ring.

"Hardanger World. May I help you?" she said.

"Helen—I've got some really exciting news. I wanted you to be the first to hear it," he said.

"That's wonderful, Sweetheart," Helen replied. "What's happened?"

"I've found an important clue—er, I mean The Author led me to an important piece of evidence."

"I'm so glad you got in touch with him, Ace. What did you find?"

"It's a single leather glove. It was lying on the ground in the parking space next to mine, nearly buried in the mud. Only the tip of the thumb sticking up. If I hadn't known to look there, I never would have found it."

"That means it was probably dropped by someone who parked next to you," Helen observed.

"Dang!" he exclaimed. "You know, there *was* a car parked there last night. I noticed it only because that spot is normally empty. But I was in such a hurry to get out of here, it didn't really register. In fact, I can't even remember what make the car was. I only remember it was large and kind of boxy—a dirty white color, I think. By now, there's not a Snowball's Chance of getting a tread imprint."

"How about the glove? Can you tell anything from it?" Helen asked.

Ace examined the glove closely. "Needless to say, it's covered with mud and a

complete mess. There's no possibility of lifting any fingerprints from it. Let's see the inside... Totally soaked with water... No lining... Wait! Wait just a second! Part way down inside, I can see a couple of short hairs. Probably from the back of the thief's hand."

"It's *Bohacker*, Ace," Helen said in a forced whisper.

"Sweetheart, it can't be. For a while I did think it might be him. But I tried to call him several times earlier today at his home, and there was no answer. Besides, now that I have the glove—"

"Call it 'woman's intuition,' or call it anything you like," Helen interrupted. "I just *know* that's Bohacker's glove," she insisted.

"If I had another hair or something from his body, we could know for sure by performing a DNA test," Ace observed.

"I suppose we may never know. It's just... Ace! Don't you still have that trenchcoat with some of Melvin Bohacker's blood on it?"

"Sure I do!" he exclaimed. "It's still in my closet. I can take it and the glove down to Crime City."

"Huh? What's that?" she asked.

"Crime City," he replied. It's one of those all-night criminal laboratory chain stores. They can do a DNA match test and fax the results to my office in a couple of hours. I think I may even have a coupon for two dollars off the regular price. Tell you what: I'll drop off the stuff and then pick you up when you get off work. We can catch a quick burger somewhere. By that time, the test results should be in. That okay with you?"

"Count on it," she said.

Taking Win95 for a Walk

Ace swallowed the last bite of his hamburger and sat deep in thought, staring unexpressively across the table at his dinner partner. Realizing she wasn't likely to get any more conversation out of him for awhile, Helen suggested they head back to his office to see if the test results had come in.

"While you're taking care of the check, I have a stop to make," she said, gracefully getting up from the table.

"Okay," he said absent-mindedly. His eyes automatically followed her as she meandered toward the front of the restaurant. The *Maison de Mort Rouge Viande* was one of those upscale, bovine-centric joints that virtually dripped with class. The kind of place Helen liked best. Of course, she had grown up eating in places like this.

Ace partially snapped out of his trance. *If there's a chance this guy really is Bohacker,* he thought, *I want to know it now. And with Helen out of earshot for the moment, this might be a good time to check in with my Man on the Street, to see what's happening.*

He pulled the trusty cell phone from his coat pocket and dialed the number for Buck McGawk's Norwegian Fried Chicken Haus. A place, he remembered, that dripped with something other than class.

"Welcome to Buck's," the voice said. "Our special tonight is the Pullet Surprise. May I take your order?"

"Is that you, Biff?" Ace asked.

"Ace—how's it going, Buddy?"

"I need some information, and I need it fast. Have you seen Melvin Bohacker recently?"

"You know, it's funny like you should mention him. Something really weird happened today."

"Tell me about it," Ace said.

"Well, I may have told you before that Bohacker has a standing order for a tub of Extra Greasy on Tuesdays and Fridays. So normally he would have come by tonight to pick up his dinner."

"Uh-huh."

"Well, he called in earlier this afternoon and canceled," Biff continued. "Said he was going out of town on an unexpected trip, and didn't know for sure when he'd be back."

"Was that all?" Ace prompted.

"It was kind of noisy, with cars pulling through here and all. But it sounded sorta like he said he was meeting some woman over in Norton City. Whaddya think? That little receptionist dumped him?"

"I wouldn't know, Biff," Ace replied. "Listen. Gotta go. I'll talk to you later." Ace switched off the phone, slipped it in his pocket, and headed for the register.

Meanwhile, across town, the missing Casebook was still being scrutinized....

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Casebook No. 16, March 28: Ever since its initial release, I have heard that one thing that set Delphi apart from other visual programming tools is its ability to make all the difficult Windows stuff easy to work with, while affording programmers the ability to work at the nuts-and-bolts level. I decided to investigate some of the inner workings of Windows 95, and how they could be accessed from a Delphi application.

One of the major differences between Windows 3.1 and Win95 is the changes resulting from the addition of preemptive multitasking. Under Win 3.1, the multitasking was cooperative, which meant only one task could execute at a time, and no other task could get control of the CPU until the current task yielded control in a cooperative manner. Among other things, this meant that one program could always prevent another program from accessing system data structures until it was good and ready to allow that access. But Win95, with its multithreading and *preemptive* multitasking, creates quite a different scenario, where the operating system is in absolute control, parceling out CPU time slices on a prioritized basis.

Microsoft quietly included a Dynamic Link Library called TOOLHELP.DLL with Win 3.1. Although it received little coverage in books or magazines (and the documentation for it was almost non-existent), Borland did provide an interface unit for it as part of Delphi 1.0. ToolHelp provided several interesting "under-the-hood" routines, including **TaskFirst** and **TaskNext** routines that enabled a programmer to "walk" the list of currently active tasks within the system. I was pleased to see a similar interface unit, called TLHELP32, included with Delphi 2.0 for the 32-bit world, and I decided to concentrate today's investigations in that area.

Just Say "Cheese"

At first, I was rather surprised by the difference between the 16-bit and 32-bit versions of ToolHelp. Several of the routines (including **TaskFirst** and

TaskNext) were missing in the 32-bit version. Was it an oversight?

As it turns out, it was not. The implementation of preemptive, multithreaded multitasking has created a situation so dynamic, it has established the software equivalent of the Heisenberg Uncertainty Principle: You can't let a system that is continuously managing tasks and creating, prioritizing, scheduling and destroying threads do its thing and also dynamically report on the system at the same time. By the time you would get a report, the system's state is quite likely to have changed.

My fondest desire would be that Win95 would hand over control for a few CPU cycles, so I could get an absolutely accurate report of the state of the system. But that would put the entire system at the mercy of one application, and that's in total disagreement with what preemptive multitasking is all about.

So what's the solution? It's to take a "snapshot" of the entire system, scheduled at Win95's discretion. We can then analyze that snapshot to our hearts' content without interfering with system operation. It's certainly not a perfect solution, but at least it's workable.

The snapshot capability is included in the 32-bit version of ToolHelp, in a function called **CreateToolHelp32Snapshot**. This function requires two parameters. The first is a mask that determines what kind of information to collect. Table 13.1 details the masks and their values. The second parameter is a handle to a process within the system. This handle—to an object called a *process ID*—gives us access to a single process; by examining that process, several pieces of information can be obtained. In general, it works like this:

- Make a call to get a list of all current processes, using the appropriate mask to collect the desired data. This returns a handle to a memory record maintained by KERNEL32, the source of all knowledge regarding system status.
- Use the **Process32First** and **Process32Next** routines to walk down the list of processes, examining various aspects of each process. The information about each process will be placed in a variable specified in the call to the walking routines.

TABLE 13.1 Masks for CreateToolHelp32Snapshot

Name	Value	Collects data for
TH32CS_SNAPHEAPLIST	1	Memory heaps within a process
TH32CS_SNAPPROCESS	2	All processes, system-wide
TH32CS_SNAPTHREAD	4	Threads within a specified process
TH32CS_SNAPMODULE	8	Modules within a specified process
TH32CS_SNAPALL	15	All of the above

Even though this was an investigation and not a formal exercise, it was time to make a few design decisions. First, I had to determine my overall goal. I

decided I wanted to create an application that would display the names of all active processes in the system. In addition, I wanted to show the names of every module (that is, collection of code, data, bitmaps, device drivers, etc. that make up a process) in the system. In addition, I wanted the option of listing only the modules that are associated with a specified process. Finally, I wanted to see the count of how many system-wide instances there were of every module displayed.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

The WalkStuf Unit

After several hours of thrashing and many false starts, I created the unit shown in Listing 13.7. This bit of code provides several general-purpose routines that make it pretty straightforward to get lists of modules and procedures from Win95.

Listing 13.7 Code for the WalkStuf unit

```
{————————————————————————————————————————————}
{                The Walking Demo                }
{          WALKSTUF.PAS : Utilities Unit         }
{            By Ace Breakpoint, N.T.P.           }
{             Assisted by Don Taylor             }
{                                                }
{ This unit contains some routines for returning }
{ specific results from the TlHelp32 unit.       }
{                                                }
{ Written for *Kick-Ass Delphi Programming*       }
{ Copyright (c) 1996 The Coriolis Group, Inc.    }
{              Last Updated 3/28/96              }
{————————————————————————————————————————————}

unit WalkStuf;

interface

uses
 Windows, Classes, Dialogs, SysUtils, TLHelp32;

const
 ws_FullPath = True;
 ws_NoDirectory = False;
 ws_Unique = True;
 ws_DupesOK = False;
 ws_InstanceCount = True;
 ws_NoInstanceCount = False;

 function GetSystemProcessList(FullPath : Boolean;
                               Unique : Boolean) : TStringList;
```

```
 function GetSystemModuleList(FullPath : Boolean;
                              Unique : Boolean;
                              IncludeData : Boolean) : TStringList;

 function GetProcessModules(ProcName : String;
                            FullPath : Boolean;
                            IncludeData : Boolean) : TStringList;

 function GetLocalModuleList : TStringList;
 function ModuleSysInstCount(ModuleName : String) : Integer;

implementation

{
Return a string with the path information removed.
}
function ChopPath(PathName : String) : String;
var
 s : String;
begin
 s := PathName;
 if Length(s) > 0
  then begin
        while Pos(':', s) > 0 do Delete(s, 1, Pos(':', s));
        while Pos('\', s) > 0 do Delete(s, 1, Pos('\', s));
        Result := s;
       end
  else Result := '';

end;

{
Return a String List containing the names of all active
processes in the system.
}
function GetSystemProcessList(FullPath : Boolean;
                              Unique : Boolean) : TStringList;
var
 AList : TStringList;
 ProcHandle : THandle;
 AProcEntry : TProcessEntry32;
begin
 AList := TStringList.Create;
 Result := AList;
 AList.Sorted := True;
 if Unique
  then AList.Duplicates := dupIgnore
  else Alist.Duplicates := dupAccept;

 ProcHandle := CreateToolHelp32Snapshot(TH32CS_SNAPPROCESS, 0);
 if ProcHandle = -1 then Exit;

 AProcEntry.dwSize := sizeof(TProcessEntry32);

 if Process32First(ProcHandle, AProcEntry)
  then begin
        { Add the first process }
        if FullPath
         then AList.Add(AProcEntry.szExeFile)
```

```
         else AList.Add(ChopPath(AProcEntry.szExeFile));

        { Add any other processes }
        while Process32Next(ProcHandle, AProcEntry) do
         if FullPath
         then AList.Add(AProcEntry.szExeFile)
         else AList.Add(ChopPath(AProcEntry.szExeFile));
       end;

 CloseHandle(ProcHandle);
end;

{
Return a String List containing the names of all active
modules in all processes in the system.
}
function GetSystemModuleList(FullPath : Boolean;
                             Unique : Boolean;
                             IncludeData : Boolean) : TStringList;
var
 s : String;
 AList : TStringList;
 ProcHandle : THandle;
 ModHandle : THandle;
 AProcEntry : TProcessEntry32;
 AModEntry : TModuleEntry32;
begin
 AList := TStringList.Create;
 Result := AList;
 AList.Sorted := True;
 if Unique
  then AList.Duplicates := dupIgnore
  else Alist.Duplicates := dupAccept;

 ProcHandle := CreateToolHelp32Snapshot(TH32CS_SNAPPROCESS, 0);
 if ProcHandle = -1 then Exit;

 AProcEntry.dwSize := sizeof(TProcessEntry32);
 AModEntry.dwSize := sizeof(TModuleEntry32);

 if Process32First(ProcHandle, AProcEntry)
  then begin
        { Work on the first process }
        ModHandle :=
          CreateToolHelp32Snapshot(TH32CS_SNAPMODULE,
          AProcEntry.th32ProcessID);
        if Module32First(ModHandle, AModEntry)
         then begin
               { Work on the first module in the first process }
               if IncludeData
               then s := '<' + IntToStr(AModEntry.GlblcntUsage)
               else s := '';

               if FullPath
                then s := AModEntry.szExePath + s
                else s := AModEntry.szModule + s;
               AList.Add(s);

               { Work on any remaining module in first process }
```

```
                    while Module32Next(ModHandle, AModEntry) do
                     begin
                      if IncludeData
                       then s := '<' + IntToStr(AModEntry.GlblcntUsage)
                       else s := '';

                      if FullPath
                       then s := AModEntry.szExePath + s
                       else s := AModEntry.szModule + s;
                      AList.Add(s);
                     end;
                    CloseHandle(ModHandle);

                    { Work on any remaining processes }
                    while Process32Next(ProcHandle, AProcEntry) do
                     begin
                      ModHandle :=
                        CreateToolHelp32Snapshot(TH32CS_SNAPMODULE,
                        AProcEntry.th32ProcessID);
                      if Module32First(ModHandle, AModEntry)
                       then begin
                             { Work on first module in this process }
                             if IncludeData
                             then s := '<' + IntToStr(AModEntry.GlblcntUsage)
                             else s := '';

                             if FullPath
                              then s := AModEntry.szExePath + s
                              else s := AModEntry.szModule + s;
                              AList.Add(s);

                             { Work on remaining modules in this process }
                             while Module32Next(ModHandle, AModEntry) do
                              begin
                               if IncludeData
                                then
                                  s := '<' + IntToStr(AModEntry.GlblcntUsage)
                                else s := '';

                               if FullPath
                                then s := AModEntry.szExePath + s
                                else s := AModEntry.szModule + s;
                                AList.Add(s);
                              end;
                            end;
                            CloseHandle(ModHandle);
                     end; { while }

                  end;
          end;

 CloseHandle(ProcHandle);
end;
{
Return a String List containing the names of all active
modules in the current process.
}
function GetLocalModuleList : TStringList;
var
```

```
 AList : TStringList;
 ModHandle : THandle;
 AModEntry : TModuleEntry32;
begin
 AList := TStringList.Create;
 AList.Sorted := True;
 Result := AList;
 ModHandle := CreateToolHelp32Snapshot(TH32CS_SNAPMODULE, 0);
 if ModHandle = -1 then Exit;

 AModEntry.dwSize := sizeof(TModuleEntry32);

 if Module32First(ModHandle, AModEntry)
  then begin
        { Add the first module }
        AList.Add(AModEntry.szModule);

        { Add any remaining modules }
        while Module32Next(ModHandle, AModEntry) do
         AList.Add(AModEntry.szModule);
       end;

 CloseHandle(ModHandle);
end;

{
Return a String List containing the names of all active
modules in the process specified by name.
}
function GetProcessModules(ProcName : String;
                           FullPath : Boolean;
                           IncludeData : Boolean) : TStringList;
var
 s : String;
 Found : Boolean;
 Done : Boolean;
 AList : TStringList;
 ProcHandle : THandle;
 ModHandle : THandle;
 AProcEntry : TProcessEntry32;
 AModEntry : TModuleEntry32;

begin
 AList := TStringList.Create;
 Result := AList;
 AList.Sorted := True;

 ProcHandle := CreateToolHelp32Snapshot(TH32CS_SNAPALL, 0);
 if ProcHandle = -1 then Exit;

 AProcEntry.dwSize := sizeof(TProcessEntry32);
 AModEntry.dwSize := sizeof(TModuleEntry32);

 Done := False;

 if Process32First(ProcHandle, AProcEntry)
  then begin
        { Examine processes until a match is found }
        Found := UpperCase(AProcEntry.szExeFile) = UpperCase(ProcName);
```

```
        if not Found
         then repeat
               Done := not Process32Next(ProcHandle, AProcEntry);
               if not Done
                then Found :=
                  UpperCase(AProcEntry.szExeFile) = UpperCase(ProcName);
              until Done or Found;

        if Found
         then begin
               ModHandle :=
                 CreateToolHelp32Snapshot(TH32CS_SNAPMODULE,
                 AProcEntry.th32ProcessID);
               if Module32First(ModHandle, AModEntry)
                then begin
                      { Work on the first module in the first process }
                      if IncludeData
                       then s := '<' + IntToStr(AModEntry.GlblcntUsage)
                       else s := '';

                      if FullPath
                       then s := AModEntry.szExePath + s
                       else s := AModEntry.szModule + s;
                      AList.Add(s);
                      { Work on any remaining module in first process }
                      while Module32Next(ModHandle, AModEntry) do
                       begin
                        if IncludeData
                         then s := '<' + IntToStr(AModEntry.GlblcntUsage)
                         else s := '';

                        if FullPath
                         then s := AModEntry.szExePath + s
                         else s := AModEntry.szModule + s;
                        AList.Add(s);
                       end;
                     end;
               CloseHandle(ModHandle);

              end;
       end;
 CloseHandle(ProcHandle);
end;

{
Return a count of the total number of times the specified
module appears in all processes in the system.
}
function ModuleSysInstCount(ModuleName : String) : Integer;
var
 Idx : Integer;
 p : Integer;
 s : String;
 ModList : TStringList;
 MatchFound : Boolean;
begin
 Result := -1;
 ModList := GetSystemModuleList
   (ws_NoDirectory, ws_DupesOK, ws_InstanceCount);
```

```
 if ModList = nil then Exit;

 Idx := 0;
 MatchFound := False;
 while (Idx < ModList.Count) and not MatchFound do
  begin
   s := ModList.Strings[Idx];
   p := pos('<', s);
   MatchFound := Uppercase(copy(s, 1, p - 1)) = Uppercase(ModuleName);
   if not MatchFound then Inc(Idx);
  end; { while }
 if MatchFound
  then Result := StrToInt(copy(s, p + 1, Length(s) - p))
  else Result := 0;
end;

end.
```

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

WalkStuf provides five useful functions that make exploring Win95's operations a lot easier. **GetSystemProcessList** returns a string list containing the names of all active processes in the system. An option is provided to list only the process name (without its full path) and to suppress multiple instances of the same process. **GetSystemModuleList** returns a string list of all modules from all processes in the system. The same options are provided for suppressing path information and multiple instances; in addition, there is a provision to append the number of system-wide instances of the module to each entry's string. **GetProcessModules** returns a string list of all modules for a specified process. **GetLocalModuleList** brings back a string list of only the modules in the current process. Finally, **ModuleSysInstCount** returns an integer representing the number of system-wide instances of a specified module.

There are a few items of interest in the WalkStuf routines that bear some explanation. **GetSystemProcessList** illustrates how to "walk" the list of processes. **ProcHandle** is assigned a handle to an area within **KERNEL32** that is set up to contain a list of all the processes. Next, the **dwSize** field of the **TProcessEntry32** (a record designed to handle process information) is set to the size of that data type. (Although this seems almost silly, it is critical to the operation!) **Process32First** is then called, using the **ProcHandle** to the **KERNEL32** information and the variable (**AProcEntry**) where the data is to be dumped.

If **Process32First** returns **True**, several pieces of information about the first process in the list have been copied to the fields of **AProcEntry**. Perhaps the two most interesting are **szExeFile** and **th32ProcessID**. The **szExeFile** field contains a string which identifies the complete path to the .EXE file the process was created from. The **th32ProcessID** field is a unique ID value for the process being examined; it can be passed on to other functions within ToolHelp. More on that in a moment.

After the **szExeFile** is added to the string list, a while loop is used to repeatedly call **Process32Next**. This routine takes the same parameters, and if it returns **True**, another process's data will be placed in **AProcEntry**. (If you've ever used **FindFirst** and **FindNext** under DOS, you already know this drill.) When all is finished, there is one last housekeeping chore to perform. The call to **CreateToolHelp32Snapshot** created a Win95 object that must be disposed. This is done with a call to **CloseHandle**.

GetSystemModuleList is a more extensive example of walking. For each process, the complete list of modules for that process is examined with **Module32First** and **Module32Next**. For each process, a handle is obtained using **CreateToolHelp32Snapshot**. This time, the unique process ID of the process currently being examined (**AProcEntry.th32ProcessID**) is used in the call, resulting in a handle referring to module information for that process only. Note the use of the mask **TH32CS_SNAPMODULE**, which limits the data retreived to that for modules.

TModuleEntry32 records contain several fields. For these purposes, the three most interesting are **szExePath**, a string containing the complete path to the module; **szModule**, a string containing the base name of the module; and **GlblcntUsage**, a double word field containing the number of system-wide instances of this module.

Note once again that the size of the **TModuleEntry32** record must be entered in **AModEntry**'s **dwSize** field, and that every call to **CreateToolHelp32Snapshot** must have a matching **CloseHandle** to dispose of the object created.

The remaining routines are, for the most part, variations on a theme. **GetLocalModuleList** walks the list of only the modules belonging to the current process, obtained by using a zero as the process ID number. **GetProcessModules** walks through the module list, seeking a match for the specified process. If found, it walks through the modules for that process. And **ModuleSysInstCount** uses a customized call to **GetSystemModuleList** to get a system-wide list of modules, from which it searches for a match for the specified module. From the string entry for the matching module, it converts the number of instances and returns that number as an integer.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Stepping Out

They say when the going gets tough, the tough make coffee. I was at that point, so I brewed a pot of Scandinavian blend and set to designing a demo application that would use several of the WalkStuf routines. A running version of what I came up with is shown in Figure 13.8. The code appears in Listing 13.8.

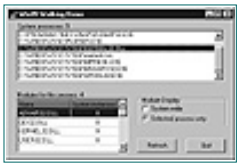

FIGURE 13.8 The Windows 95 Walking Demo taking a walk.

Listing 13.8 Code for the Walking Demo

```
{──────────────────────────────────────────────────────}
{                 The Walking Demo                     }
{              WALKMAIN.PAS : Main Unit                }
{             By Ace Breakpoint, N.T.P.                }
{              Assisted by Don Taylor                  }
{                                                      }
{ This program demonstrates some techniques for        }
{ "walking" through the Win95 system fo get            }
{ specific system-wide information.                    }
{                                                      }
{ Written for *Kick-Ass Delphi Programming*            }
{ Copyright (c) 1996 The Coriolis Group, Inc.          }
{               Last Updated 3/28/96                   }
{──────────────────────────────────────────────────────}
unit WalkMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
```

```
    Dialogs, WalkStuf, Grids, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    ModuleGrid: TStringGrid;
    RefreshBtn: TButton;
    QuitBtn: TButton;
    ModuleRBGroup: TRadioGroup;
    ProcessesLabel: TLabel;
    ProcessListBox: TListBox;
    ModulesLabel: TLabel;
    procedure QuitBtnClick(Sender: TObject);
    procedure RefreshBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure ModuleRBGroupClick(Sender: TObject);
    procedure ProcessListBoxClick(Sender: TObject);
  private
    TheList : TStringList;
    procedure RefreshForm;
    procedure DisplayProcessModules;
    procedure ClearModuleGrid;
    procedure FillProcessList;
    procedure FillModuleGrid;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

{
Return a string of spaces of the specified length.
}
function Spaces(Size : Integer) : String;
begin
 Result := '';
 while Length(Result) < Size do Result := Result + ' ';
end;
{
Clear the screen elements, retrieve data and
update the screen.
}
procedure TForm1.RefreshForm;
begin
 ClearModuleGrid;
 ProcessListBox.Clear;
 TheList := GetSystemProcessList(ws_FullPath, ws_DupesOK);
 FillProcessList;
 ProcessesLabel.Caption :=
   'System processes: ' + IntToStr(TheList.Count);
```

```
 TheList.Free;

 case ModuleRBGroup.ItemIndex of
  0 : begin
       TheList := GetSystemModuleList
          (ws_NoDirectory, ws_Unique, ws_InstanceCount);
       FillModuleGrid;
       ModulesLabel.Caption :=
         'System-wide modules: ' + IntToStr(TheList.Count);
       TheList.Free;
      end;

  1 : begin
       TheList := GetSystemModuleList
         (ws_NoDirectory, ws_Unique, ws_InstanceCount);
       if TheList.Count > 0
        then begin
             ProcessListBox.ItemIndex := 0;
             DisplayProcessModules;
            end;
      end;
 end; { case }
end;

{
Special screen update procedure that retrieves
the module data for the currently selected
process.
}
procedure TForm1.DisplayProcessModules;
var
 Idx : Integer;
 s : String;
 p : Integer;
begin
 if ProcessListBox.Items.Count > 0
  then begin
       ClearModuleGrid;
       Idx := ProcessListBox.ItemIndex;
       TheList := GetProcessModules(ProcessListBox.Items[Idx],
         ws_NoDirectory, ws_InstanceCount);

       if TheList.Count > 0
        then for Idx := 1 to TheList.Count do
         begin
          s := TheList.Strings[Idx - 1];
          p := pos('<', s);
          ModuleGrid.Cells[0, Idx] := copy(s, 1, p - 1);
          delete(s, 1, p);
          s := Spaces(15) + s;
          ModuleGrid.Cells[1, Idx] := s;
          ModuleGrid.RowCount := ModuleGrid.RowCount + 1;
         end;
```

```
        ModulesLabel.Caption := 'Modules for this process: '
           + IntToStr(TheList.Count);
        TheList.Free;
       end;
end;

{
Clear all rows in the Module String Grid, and reset the
number of rows to one.
}
procedure TForm1.ClearModuleGrid;
var
 Idx : Integer;
begin
 for Idx := 1 to ModuleGrid.RowCount - 1 do
  begin
   ModuleGrid.Cells[0, Idx] := '';
   ModuleGrid.Cells[1, Idx] := '';
  end;
 ModuleGrid.RowCount := 2;
end;

{
Fill in the Process listbox, line by line, from the
global String List.
}
procedure TForm1.FillProcessList;
var
 Idx : Integer;
begin
 if TheList.Count > 0
  then for Idx := 0 to TheList.Count - 1 do
    ProcessListBox.Items.Add(TheList.Strings[Idx]);
end;

{
Fill in the Module String Grid, line by line, from the
global String List.
}
procedure TForm1.FillModuleGrid;
var
 s : String;
 p : Integer;
 Idx : Integer;
begin
 if TheList.Count > 0
  then begin
       for Idx := 1 to TheList.Count do
        begin
         s := TheList.Strings[Idx - 1];
         p := pos('<', s);
         ModuleGrid.Cells[0, Idx] := copy(s, 1, p - 1);
         delete(s, 1, p);
         s := Spaces(15) + s;
```

```
          ModuleGrid.Cells[1, Idx] := s;
          ModuleGrid.RowCount := ModuleGrid.RowCount + 1;
        end; { for }
      end;
end;

procedure TForm1.QuitBtnClick(Sender: TObject);
begin
 Close;
end;

procedure TForm1.RefreshBtnClick(Sender: TObject);
begin
 RefreshForm;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
 ModuleGrid.Colwidths[1] :=
   ModuleGrid.Width - ModuleGrid.ColWidths[0] - 22;
 ModuleGrid.Cells[0, 0] := 'Name';
 ModuleGrid.Cells[1, 0] := 'System instances';
 ModuleRBGroup.ItemIndex := 0;
end;

procedure TForm1.ModuleRBGroupClick(Sender: TObject);
begin
 RefreshForm;
end;

procedure TForm1.ProcessListBoxClick(Sender: TObject);
begin
 if ModuleRBGroup.ItemIndex > 0 then DisplayProcessModules;
end;

end.
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

There's nothing really special about the code in the demo. The upper window always shows all active processes in the system snapshot. If the radio button marked "System wide" is selected, the lower window lists all modules in the system, including the number of instances of each. If the "Selected process only" button is selected, the lower window lists only the modules for the process highlighted in the upper window. And of course, the Refresh button takes a new snapshot and updates the screen. The main thing to remember is that every call to a WalkDemo routine that returns a string list does so by creating a new object; that object must later be disposed exactly once.

This was all quite instructional, and a lot of fun as well. But my investigations only scratched the surface, and there are many more things to learn. One of the things I discovered is that the ToolHelp routines are compatible only with Win95 and *not* NT, at least for now.

I get the distinct impression that I'll be able to use some of this stuff in my applications. And I'll certainly have to come back and investigate more thoroughly when I have more time.

End of entry, March 28.

When Ace and Helen arrived at the office, there was still no word on the DNA test. Ace took out a bottle of Purely Poulsbo from the refrigerator, and found a couple of clean glasses in the cupboard. He poured two fingers' worth into each glass, then handed one to Helen. He had no more than touched the glass to his lips when there came a sharp rap at the door.

It was Marge Reynolds. Ace could hear excitement in her voice as she exchanged pleasantries with Helen. Marge quickly came to the point.

"I know you've been looking for clues to last night's robbery," she began. "I saw you out there digging through the mud in the parking lot this afternoon. But I've been keeping my eyes open, too—looking for anything suspicious."

She paused for an acknowledgment of interest.

“Go on,” he urged.

“This evening, as I was walking past the public phone booth on the other side of the courtyard you know, the one that faces your kitchen window—I found a piece of paper stuck in a bush, about a foot above the ground.”

Marge fumbled around in the pocket of her bulky knit sweater. “I’m sure it’s here somewhere. I put it—oh, here it is,” she said, pulling out a lavender-colored scrap of paper and examining it one more time. “It looks like it was written by a woman. It’s got your name and phone number on it, and two other words: ‘kidnapped heiress.’ Do you think it means anything?”

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

CHAPTER 14
The Oracle Returns

DON TAYLOR

- Sharing Event Handlers
- Windows 95 Memory Files
- Detecting Multiple Instances Of An Application
- Preventing Application Execution
- Floating Toolbars

While the Delphi Avenger discovers how to make Delphi apps detect both themselves and Delphi, (floating a toolbar as a bonus) Ace detects one too many X's in a very strange equation—but ultimately drags the case to a jaw-dropping conclusion.

Ace switched on the Halogen lamp on his desk and held the lavender piece of paper up to the light.

"Pretty expensive stuff," he said. "It's fine linen. I can just make out what appears to be the edge of a watermark." He turned the scrap so the light coming from the lamp reflected off the paper. "The writing's in a feminine hand, all right," he said. "By the size of the loops, I'd say the woman who wrote this has a strong personality, consistent with the woman who had the bravado to call me last night. And this was definitely written with a very expressive pen, probably a Habasher No. 4374 tip, using what appears to be Mandolay's 'Nightshade' ink. Other than that, there's not much I can deduce."

"May I see it?" asked Helen.

“Sure,” Ace replied. Handing over the note, he added, “One more thing. You’ll notice the paper has been scented with some kind of perfume.”

Helen’s eyes widened as she passed the paper scrap under her nose. “This is not just any ordinary perfume,” she said. “It’s a very expensive fragrance. *Chez Monieux*.”

“Okay, so it’s expensive perfume,” he said, a bit testily.

“I didn’t mean to belittle you in any way, Sweetheart. It’s just that women notice little things that men often don’t. In this case, it’s that only a sophisticated, well-to-do woman would wear this perfume—*Chez Monieux*.”

“Hey, lighten up. I already said she used expensive paper, and a very nice pen and ink, didn’t I? Just because I’m not a perfume expert—”

“I think what Helen is trying to say,” Marge interjected, “is the *name of the perfume* is *Chez Monieux*.”

Ace froze for a moment, as if wondering what to say. His eyes darted quickly between those of Marge and Helen. “I knew that,” he replied guardedly.

“Remember?” Helen said. “It’s part of that new exclusive line of fashion accessories produced for Muffy. You know, the *Pure Prophet* collection?”

“I knew that, too,” Ace said. “But where does all this get us?”

“Well, I have a theory,” Helen began. “I think Melvin Bohacker has for some time been looking to get even with you. He has probably recently gotten involved with a woman of means; a woman used to having her own way about things, and someone to whom pride is a precious commodity.”

She looked to her companions for a confirmation to continue. Ace rolled his eyes, but Marge was obviously hanging on Helen’s every word.

“When this woman—let’s call her ‘Madame X’—found out about how Melvin had lost face because of you, she coaxed him to get even. Together, they concocted the plan they executed last night: Driving her car, they pulled in here and parked in the space next to yours. He got out and hid in the shadows somewhere while she walked to the pay phone on the other side of the courtyard. From there, she was able to watch you through your kitchen window as she dialed the number she had earlier scrawled on a corner of her personal note paper.”

“So how did the note get where it was found?” Ace asked.

“I have a theory about that, too. The pay phone is one of those open types, mounted on a pole. Madame X probably laid the note down and it was blown away by a puff of wind. There’s a light by the pole. But if the note was blown even a few feet away, it would have been in relative darkness. She was probably in a hurry, and may not even have missed the note.”

“And the glove?”

“Bohacker undoubtedly wore gloves to break into your apartment, so he wouldn’t leave any fingerprints. He probably took them off as he was getting

back into the car, and dropped one on the ground. He may have run over it with the car, partially burying it in the mud."

"I'm sorry," Ace said, waving his hand in front of his face. "It all sounds plausible enough, except for one thing: Bohacker isn't capable of doing it, even with the goading of some rich babe—heiress or not."

Helen sighed. "I guess we'll just have to wait for the results from the DNA testing. That should prove who is right, once and for all."

"DNA testing?" Marge asked. "What about DNA testing? And what is this business with a glove?"

Ace recounted the story of the glove to Marge. It took nearly half an hour to describe all the details to her satisfaction.

"It's sure been an exciting day," she said. "I wish I could sit around here with you kids and wait for the test results. But there's a new tenant moving into #193 tonight—a bachelor—and I want to see what kind of appliances he owns."

Marge Reynolds walked to the front door, her polyester knit pants hugging every bulge, looking like a sack full of cats on their way to the Pound. She squeezed through the door and quietly closed it behind her.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Sharing Event Handlers

In an untidy little room across town, an ominous figure was bent over a book, reading intently.

Casebook No. 16, March 29: Among the bills in today's mail, I found a wedding invitation from Blade Porter and Cinder Holte-Mullingham-LaFong, a couple Helen and I had gotten to know while we were attending Art School. The announcement read "Come and share our joyous event with us." We had always assumed they would get married. Now it was official.

As I tossed the card on my desk, I got to thinking about ways it's possible to "share events" in an entirely different way—by using common handlers to deal with events that require similar actions.

I decided to create a simple application that would let me explore this concept. After a little pondering, I settled on a little "To Do List" program that would be a spin-off of the Drag/Drop Demo I had written a few days ago.

As in the earlier demo, the new program would feature an edit box in which the user could type a comment. Instead of dragging it to a calendar, this time I would provide three separate string grids—one for morning tasks, another for afternoon duties, and a final one for evening activities. The three grids would be placed on separate pages of a tabbed notebook. I created a design model of the form, which can be seen in Figure 14.1.

FIGURE 14.1 The shared events demo.

Taking a First Run

The scenario would be to select the proper page and then drag the task description string from the edit box to the grid on that page. The operation would be virtually identical for each of the three pages, the only difference being which grid would receive the string. There should therefore be a way for the three string grids to share their OnDragOver and OnDragDrop event handlers in common.

I suppose this is a personal preference, but I would like shared event handlers to have a more generic name than what is automatically generated by Delphi. I decided to call the two handlers **GridDragOver** and **GridDragDrop**.

Before proceeding any further, I want to document the 8-step method I used to create generic-sounding routine names for the handlers:

1. Double-click on the name of the event to be handled, as it appears in the Object Inspector.

2. Delphi automatically names the routine, combining the names of the component and the event to produce a unique identifier. More importantly, it automatically generates the call parameter list for the specific event. (Being lazy, I would much rather have Delphi do this for me than create it from scratch.)

3. Type anything (a semicolon does nicely) between the **begin** and **end** of the empty routine created by Delphi. This will prevent Delphi from automatically deleting the routine if you try to save it.

4. Edit the name of the routine, then double-click to highlight it, and press Ctrl-C to copy the new name to the clipboard.

5. Jump to the beginning of the file, and find the original handler declaration within the form's declaration. Highlight the original name there, and tap Ctrl-V to replace it with the new name.

6. At some point in the process, Delphi will complain that the name it originally assigned (still present in the Object Inspector) can't be found. Give Delphi the okay to remove it.

7. Move the cursor to the event slot in the Object Inspector, where this all began. Use the Ctrl-V combination to paste in the new name.

8. Save the file.

The routines I wrote appear in Listing 14.1.

Listing 14.1 Common handlers for OnDragOver and OnDragDrop events

```
{ Common event handler for all the grids' OnDragOver event. }
procedure TShareEventDemoForm.GridDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
 { We can accept anything, but only from the EditBox. }
 Accept := Source is TEdit;
end;

{ Common event handler for all the grids' OnDragDrop event. }
procedure TShareEventDemoForm.GridDragDrop(Sender, Source : TObject;
  X, Y : Integer);
begin
 { Drop the load on the currently selected grid. }
 DropEditString(CurrentGrid);
end;

{ Common event handler support routine that drops the string
  in the EditBox onto the specified grid. Also clears the EditBox. }
procedure TShareEventDemoForm.DropEditString(AGrid : TStringGrid);
begin
 if AGrid <> nil
  then with AGrid do
   begin
    Cells[0, RowCount - 1] := EditBox.Text;
    RowCount := RowCount + 1;
    EditBox.Text := '';
```

```
   end; { with }
end;

{ Return a pointer to the currenly selected tab sheet. }
function TShareEventDemoForm.CurrentGrid : TStringGrid;
begin
 Result := nil;
 if PageControl.ActivePage = MorningSheet
  then Result := MorningGrid else
 if PageControl.ActivePage = AfternoonSheet
  then Result := AfternoonGrid else
 if PageControl.ActivePage = EveningSheet
  then Result := EveningGrid;
end;
```

The OnDragOver routine is very straightforward. As long as the source object for the drag is the edit box, it can be accepted by any of the grids. I chose to split the OnDragDrop handling into two routines: a main handler and a support method. The handler merely calls the support routine, using a function which returns a pointer to the grid on the currently selected tabsheet. Delphi's ability to mask the use of pointers really comes into play here, making it very straightforward to pull some shenanigans with pointers, while keeping the program exceedingly readable. Given the pointer, the **DropEditString** routine adds the string from the edit box to the appropriate string grid, updating the grid's row counter and clearing out the edit box.

The handlers worked as expected for the grid on the first tabsheet (**MorningGrid**). I went back to the Object Inspector and selected the same handlers for the other two grids. Again, everything worked perfectly.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Down a Crooked Path

It must be a quirk in my personality (or maybe I'm just a nerd at heart), but this was just too easy. I wanted to take the process a step further, where I could drop a string from the edit box onto any of the string grids, simply by dropping it onto the appropriate tab. In a way, I wish I had never started down this road.

I first learned that although PageControls appear similar to the TabSet and Notebook components provided with Delphi 1.0, they have some significant differences, not the least of which concerns the tabs. TabSets have a method that will tell you the index of a tab, given an x-y position. The wrapper-relationship of PageControls gives most of the power to the TabSheets, leaving little to the PageControl, which is pretty well limited to knowing which is the currently selected sheet.

I would need to calculate where the tabs were, so I could know which tab the mouse was pointing to. From that information, I could easily determine the associated tabsheet. But there was another problem: PageControls automatically size each tab's width individually, based on the length of its caption string. What to do?

I decided to support drops onto tabs only if the tabs had been manually sized by setting the PageControl's TabHeight and TabWidth properties to some value other than zero. I should have stopped there. But I decided I also wanted to provide a feature that would automatically size the tabs at equal width, based on the length of the longest caption. That eventually led me to a much longer version of the code, the complete version of which is shown in Listing 14.2.

Listing 14.2 Complete code for the common event handler demo application

```
{————————————————————————————————————————}
{        Event Handler Sharing Demonstration        }
{            SHARMAIN.PAS : Main Unit               }
{            By Ace Breakpoint, N.T.P.              }
{             Assisted by Don Taylor                }
{                                                    }
{ Application that demonstrates the sharing of     }
{ event handlers within an application, as part    }
{ of a drag/drop operation.                        }
{                                                    }
{ Written for *Kick-Ass Delphi Programming*         }
{ Copyright (c) 1996 The Coriolis Group, Inc.       }
{              Last Updated 3/29/96                  }
{————————————————————————————————————————}
unit SharMain;
```

```
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, StdCtrls, ExtCtrls, ComCtrls;

type
  TShareEventDemoForm = class(TForm)
    EditBox: TEdit;
    Label1: TLabel;
    QuitBtn: TButton;
    Panel1: TPanel;
    PageControl: TPageControl;
    MorningSheet: TTabSheet;
    AfternoonSheet: TTabSheet;
    EveningSheet: TTabSheet;
    MorningGrid: TStringGrid;
    AfternoonGrid: TStringGrid;
    EveningGrid: TStringGrid;
    procedure FormCreate(Sender: TObject);
    procedure QuitBtnClick(Sender: TObject);
    procedure EditBoxMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure GridMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure GridDragOver(Sender, Source: TObject;
      X, Y: Integer; State: TDragState; var Accept: Boolean);
    procedure GridDragDrop(Sender, Source : TObject;
      X, Y : Integer);
    procedure PageControlDragOver(Sender, Source: TObject; X, Y: Integer;
      State: TDragState; var Accept: Boolean);
    procedure PageControlDragDrop(Sender, Source: TObject;
      X, Y: Integer);
  private
    CopyDrag : Boolean;
    function ManualTabsSet : Boolean;
    function CurrentGrid : TStringGrid;
    function TabGrid(X : Integer) : TStringGrid;
    procedure SetTabSizes;
    procedure DropEditString(AGrid : TStringGrid);
    procedure DropGridString(TargetGrid : TStringGrid);
  public
    { Public declarations }
  end;

var
  ShareEventDemoForm: TShareEventDemoForm;

implementation

{$R *.DFM}

{ Return the length in pixels of a string for display,
  given a handle to the window in which it will appear,
  along with a handle to the window's font. }
function StringWidth
  (WinHnd : HWND; FntHnd : HWND; Text : String) : Integer;
var
 DCHnd : HWND;
```

```
 StrSize : TSize;
 TextArr : array[0..127] of char;
begin
 Result := -1;
 DCHnd := GetDC(WinHnd);
 if GetMapMode(DCHnd) = MM_TEXT
  then begin
        SelectObject(DCHnd, FntHnd);
        StrPCopy(TextArr, Text);
        if GetTextExtentPoint32(DCHnd, @TextArr, Length(Text), StrSize)
         then Result := StrSize.Cx
       end;
 ReleaseDC(WinHnd, DCHnd);
end;

{ Return the height in pixels of a font for display,
  given a handle to the window in which it will appear,
  along with a handle to the window's font. The height
  must include the ascent, descent and inner leading. }
function FontHeight(WinHnd : HWND; FntHnd : HWND) : Integer;
var
 DCHnd : HWND;
 TextMex : TTextMetric;
begin
 Result := -1;
 DCHnd := GetDC(WinHnd);
 if GetMapMode(DCHnd) = MM_TEXT
  then begin
        SelectObject(DCHnd, FntHnd);
        GetTextMetrics(DCHnd, TextMex);
        Result := TextMex.tmHeight;
       end;
 ReleaseDC(WinHnd, DCHnd);
end;

procedure TShareEventDemoForm.FormCreate(Sender: TObject);
begin
 PageControl.ActivePage := MorningSheet;
 SetTabSizes;
 CopyDrag := False;
end;

procedure TShareEventDemoForm.QuitBtnClick(Sender: TObject);
begin
 Close;
end;

procedure TShareEventDemoForm.EditBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
{ Before initiating a drag, make sure the left button
   is pressed, the EditBox has text in it, and that we're
   not in a double-click situation. }
 if (Button = mbLeft)
   and (EditBox.Text <> '')
   and not (ssDouble in Shift)
  then TEdit(Sender).BeginDrag(False);
end;
```

```
{ Common event handler for all the grids' OnMouseDown event. }
procedure TShareEventDemoForm.GridMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
 TheGrid : TStringGrid;
begin
 { We're initiating a drag from whichever string grid
   is currently selected. Set the CopyDrag flag if the
   control key is down. Before starting the drag, make
   sure the left mouse button is pressed, there is text
   in the selected row of the grid, and we're not in
   a double-click situation. }
 TheGrid := CurrentGrid;
 CopyDrag := ssCtrl in Shift;
 if (Button = mbLeft)
   and (TheGrid.Cells[0, TheGrid.Row] <> '')
   and not (ssDouble in Shift)
  then TStringGrid(Sender).BeginDrag(False);
end;

{ Common event handler for all the grids' OnDragOver event. }
procedure TShareEventDemoForm.GridDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
 { We can accept anything, but only from the EditBox. }
 Accept := Source is TEdit;
end;

{ Common event handler for all the grids' OnDragDrop event. }
procedure TShareEventDemoForm.GridDragDrop(Sender, Source : TObject;
  X, Y : Integer);
begin
 { Drop the load on the currently selected grid. }
 DropEditString(CurrentGrid);
end;
procedure TShareEventDemoForm.PageControlDragOver(Sender,
          Source: TObject; X, Y: Integer;
          State: TDragState; var Accept: Boolean);
begin
 { A drop on a tab is acceptable if it comes fron the EditBox,
   or if it comes from a grid -- as long as the tab isn't the
   one for the grid originating the drag. In either case, tabs must
   have been manually set. }
 Accept := ManualTabsSet and
           (
           (Source is TEdit)
             or
           ((Source is TStringGrid) and (CurrentGrid <> TabGrid(X)))
           );
end;

procedure TShareEventDemoForm.PageControlDragDrop(Sender,
                              Source: TObject; X, Y: Integer);
begin

  { Get the string from the proper source and drop it on the grid
    associated with the tab at position X. }
  if (Source is TEdit) then DropEditString(TabGrid(X));
  if (Source is TStringGrid) then DropGridString(TabGrid(X));
```

```
end;

{ Return True only if both tab height and width have been set manually. }
function TShareEventDemoForm.ManualTabsSet : Boolean;
begin
 Result := (PageControl.TabHeight > 0) and (PageControl.TabWidth > 0);
end;

{ Return a pointer to the grid on the currenly selected tab sheet.}
function TShareEventDemoForm.CurrentGrid : TStringGrid;
begin
 Result := nil;
 if PageControl.ActivePage = MorningSheet
  then Result := MorningGrid else
 if PageControl.ActivePage = AfternoonSheet
  then Result := AfternoonGrid else
 if PageControl.ActivePage = EveningSheet
  then Result := EveningGrid;
end;

{ Return a pointer to the grid on the tab sheet associated with
  the tab located at position X. }
function TShareEventDemoForm.TabGrid(X : Integer) : TStringGrid;
var
 Idx : Integer;
begin
 Result := nil;
 with PageControl do
  begin
   Idx := X div TabWidth;
   case Idx of
    0 : Result := MorningGrid;
    1 : Result := AfternoonGrid;
    2 : Result := EveningGrid;
   end; { case }
  end; { with }
end;

{ Adjust tab height and width as required to give a pleasing
  appearance, and to ensure tabs each have the same height and
  width. }
procedure TShareEventDemoForm.SetTabSizes;
var
 i : Integer;
 Len : Integer;
 MaxWidth : Integer;
 s : String;
begin
 with PageControl do
  begin
   if TabWidth > 0
    then begin
          MaxWidth := -1;
          for i := 0 to PageCount - 1 do
           begin
            s := Pages[i].Caption;
            Len := StringWidth(Handle, Font.Handle, s);
            if Len > MaxWidth then MaxWidth := Len;
           end;
```

```
          if MaxWidth > 0 then TabWidth := MaxWidth + 10;
         end;

   if TabHeight > 0
    then PageControl.TabHeight := FontHeight(Handle, Font.Handle) + 5;

  end; { with }
end;

{ Common event handler support routine that drops the string
  in the EditBox onto the specified grid. Also clears the EditBox. }
procedure TShareEventDemoForm.DropEditString(AGrid : TStringGrid);
begin
 if AGrid <> nil
  then with AGrid do
   begin
    Cells[0, RowCount - 1] := EditBox.Text;
    RowCount := RowCount + 1;
    EditBox.Text := '';
   end; { with }
end;

{ Common event handler support routine that drops the string
  in the selected row of the currently selected grid onto
  the specified grid. If it is a "move" operation, the current
  grid is purged of the string and the grid is compressed. }
procedure TShareEventDemoForm.DropGridString(TargetGrid : TStringGrid);
var
 i : Integer;
begin
 if TargetGrid <> nil
  then begin
        with TargetGrid do
         begin
          Cells[0, RowCount - 1] :=
             CurrentGrid.Cells[0, CurrentGrid.Row];
          RowCount := RowCount + 1;
         end; { with }

        if not CopyDrag
         then with CurrentGrid do
          begin
           Cells[0, Row] := '';
           if Row < RowCount - 1
            then for i := Row to RowCount - 1 do
             Cells[0, i] := Cells[0, i + 1];
           RowCount := RowCount - 1;
          end; { with }
       end;
end;

end.
```

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

To properly compute the height and width of the string that would be written on a tab, I had to resort to routines within the Win95 API. I learned a couple of interesting things in the process. First, the **Height** sub-property of the **Font** property in a component includes the height of a character (ascender part plus descender part), but not the internal leading which is used for special purposes, like displaying umlauts (those two little dots) above some international characters.

I wanted to know the true height, which is provided by a call to **GetTextMetrics**. I created a general-purpose function called **FontHeight** that would return the height of the font on a component, given a handle to that component and a handle to its associated font. A check is made within **FontHeight** to make sure the map mode is **MM_TEXT**, meaning the value returned is for the screen and will be measured in pixels.

The companion routine, **StringWidth**, uses a similar technique to pass a string to **GetTextExtentPoint32**. The return value is the approximate length (in pixels) that will be required to display the string. (The value is only approximate because it doesn't account for any kerning done within a font.)

The **SetTabSizes** method is called by the form's **OnCreate** handler to see if the tabs need adjusting. If this routine determines that both **TabHeight** and **TabWidth** properties for the PageControl have been set to a non-zero value during design, **SetTabSize** takes over and adjusts the sizes according to the font metrics and the longest caption.

The **TabGrid** function uses the **TabWidth** and X-coordinate information provided during the drag to determine the tab being referenced, and from that, it returns a pointer to the associated string grid. **PageControlDragDrop** also uses **TabGrid** to determine which string grid will receive the string that will be dropped.

Just One More Thing…

Listing 14.2 reflects some additional capability I just couldn't resist adding to this example: the ability to drag an item from a string grid and copy or move it to one of the other grids by dropping it on the grid's tab. This required writing a common OnMouseDown handler for the grids, and extending the OnDragOver and OnDragDrop handlers for the PageControl. I also added **CopyDrag**, a boolean flag that is set whenever the Ctrl key is pressed and a drag is started in any of the grids.

DropGridString performs the lion's share of the work when items are dragged from a string grid to a tab. If the control key was not held down during the drag, **DropGridString** executes some extra steps that turn a straight copy into a move, clearing out the selected item from the source grid and then squeezing out the dead space.

An executing version of the demo is shown in Figure 14.2. This little application is really a lot of fun to play with. You can easily drag items back and forth between the pages, copying and moving, zigging and zagging. I think sharing event handlers is a lot more fun than attending a wedding. Especially my own.

FIGURE 14.2 The shared events demo app in action.

End of entry, March 29.

The fax machine in Ace's office began to whirr. Helen immediately jumped to her feet.

"Ace, there's a fax coming in," she said. "Hurry—it must be the test results!"

Breakpoint crossed the room and deftly ripped the page from the machine, just as the beep indicated it had finished printing. "Now we'll see which one of us is right about this Bohacker accusation," he said smugly.

He stood there, staring at the page and fidgeting for nearly a minute before Helen couldn't stand it any longer.

"Well, what does it say?" she demanded. "Was it Bohacker, or wasn't it?"

"Uh, it isn't completely clear. You see, these quick tests aren't always conclusive, and—"

"Let me see that," she said, grabbing the page from his hand. She quickly skimmed the report and then turned to her companion.

"It says right here—and I quote: 'The test results indicate a nearly point-for-point match between the two samples, with a congruence in the 95th percentile, perfectly consistent with DNA tests of this type.' That means the

hair and blood samples match, doesn't it?"

"Well, yes," Ace admitted. "But—"

"Then it *must* be Melvin Bohacker, just as I said. And whereever he is, you can bet Madame X is there with him. Now all we have to do is find out where he has gone."

"Norton City."

"What?"

"Norton City," Ace repeated. "Biff told me earlier this evening that Bohacker was going to Norton City sometime tonight. Trust me on this."

"But *where* in Norton City," she asked. "He could be in any one of a hundred places."

"I think I have a way to find that out," Ace said, as he strode across the room and flipped on his computer. "I'm going online—and I'm going to get some answers!"

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Using Memory Files

Casebook No. 16, April 1: I guess one of the most-often asked questions about Delphi is how to write an application that can limit itself to one instance in the system. During the past year, I've discovered several ways to accomplish that goal. But one of them was so much fun I thought I would document it here.

For an application to detect another instance of itself, it must be able to query the system in some way. Under Win 3.1, an application could check a handle called **hPrevInst** to see if there was already an instance running. But that all changed with Win95.

One way to accomplish the goal would be to use the WalkStuf unit I developed earlier. Using the **ModuleSysInstCount** function would return a value that would indicate how many copies of the program were already running. It would be simple for the application to abort execution if that count was something other than zero. But that wouldn't work with NT.

A common technique for communicating information between applications is the use of a unique global key that is accessible to multiple instances of a program. A classic example is the use of a unique file. When an application starts up, it checks for the presence of a file with a certain name (e.g., 'FOOBAR99.DAT'). If the file is present, there is another instance of the program already running. If the file is not found, the newly instantiated program creates it. When the program shuts down, it deletes the file.

One problem with this approach is caused by anomalies such as system crashes and power failures. Since the "flag" (in this case, a file) has been stored on a non-volatile medium, it's still there when the errant system is rebooted. Under those conditions, the first instance that comes along will see the file, assume another instance is running, and shut down. That means exactly zero instances of the program will run. Some manual housekeeping is required to remove the file and get everything back to normal.

Win95 offers a much nicer alternative through its shared memory files. This time, the files are in volatile memory (or at least it's treated that way, even if it's being swapped between RAM and the hard disk). And unlike many of the operations within Win95, memory files can be shared between processes.

I created a simple application to test the theory that memory files could be used to detect multiple instances of a program. The running application is shown in Figure 14.3, and its code appears in

Listing 14.3.

FIGURE 14.3 The single-instance program at runtime.

Listing 14.3 A simple single-instance program

```
{──────────────────────────────────────────────────────────────}
{               The One Instance Demo                          }
{             INSTMAIN.PAS : Main Form                         }
{             By Ace Breakpoint, N.T.P.                        }
{              Assisted by Don Taylor                          }
{                                                              }
{ Application that demonstrates how to prevent                 }
{ more than one instance of a program from running             }
{ in the Win95 environment.                                    }
{                                                              }
{ Written for *The Delphi Programming Explorer*                }
{ Copyright (c) 1995, 1996 The Coriolis Group, Inc.            }
{               Last Updated 4/1/96                            }
{──────────────────────────────────────────────────────────────}

unit InstMain;

interface

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

type
  TForm1 = class(TForm)
    ExitBtn: TButton;
    Label1: TLabel;
    procedure ExitBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.ExitBtnClick(Sender: TObject);
begin
 Close;
end;
```

```
end.
```

Of course, the form does nothing by itself. The idea is for a multiple instance to detect the presence of a previous instance and then shut itself down. And while this could be done in the startup code of the form, it is much more polite to accomplish this work in a way that doesn't display the new instance at all. That means taking care of it before the application even begins.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Before the Beginning

A lot of people don't seem to know that it's possible to execute code in the project file, even before the application is initialized. That's the case here. The project file for this experimental application is shown in Listing 14.4.

Listing 14.4 Project file for the One Instance Demo

```
{————————————————————————————————————————————}
{              The One Instance Demo                }
{           ONEINST.DPR : Project File              }
{            By Ace Breakpoint, N.T.P.              }
{             Assisted by Don Taylor                }
{                                                   }
{ Application that demonstrates how to prevent      }
{ more than one instance of a program from running  }
{ in the Win95 environment.                         }
{                                                   }
{ Written for *The Delphi Programming Explorer*     }
{ Copyright (c) 1995, 1996 The Coriolis Group, Inc. }
{              Last Updated 4/1/96                  }
{————————————————————————————————————————————}

program OneInst;

uses
  Windows,
  Forms,
  InstMain in 'InstMain.pas' {Form1};

const
```

```
 MemFileSize = 1024;
 MemFileName = 'one_inst_demo_memfile';

var
 MemHnd : HWND;

{$R *.RES}

begin
 { Attempt to create a file mapping object }
 MemHnd := CreateFileMapping(HWND($FFFFFFFF),
                             nil,
                             PAGE_READWRITE,
                             0,
                             MemFileSize,
                             MemFileName);

 { If it didn't already exist, then run the app… }
 if GetLastError <> ERROR_ALREADY_EXISTS
  then begin
        Application.Initialize;
        Application.CreateForm(TForm1, Form1);
        Application.Run;
       end;

 CloseHandle(MemHnd);

end.
```

This is a kick. First, an attempt is made to create a file mapping object with the **CreateFileMapping** API call. Whether the object already exists or is actually created, a handle to the object is returned and assigned to **MemHnd**. Calling **CreateFileMapping** with a handle value of $FFFFFFFF causes the operating system to use its own paging file instead of the conventional file system, so the file can be shared between processes; all the process needs to know is the name of the file. Although the file is set up for read/write, a call is not made here to **MapViewOfFile**, which would give the program access to the file's contents through a pointer. For the purpose of this example, it's sufficient just to check for the file's existence.

If the memory file already existed at the time of the call to **CreateFileMapping**, a handle to the file is returned to the caller, and an error value of **ERROR_ALREADY_EXISTS** is sent to the system. If that error isn't found by **GetLastError**, it means there isn't an instance already running, and it's safe to go ahead.

Because a handle was returned whether or not the file was created, the handle must be closed before the application finally shuts down. The first program to call **CreateFileMapping** creates the file mapping object; the last program to close a handle to it causes the system to destroy the object. This is the equivalent of deleting the file.

End of entry, April 1.

Ace clicked the Print button on a dialog and his laser printer came alive. He printed four pages of information, then pulled the sheets from the tray and examined them.

“Now there’s no doubt in my mind,” he said with a newly found excitement in his voice.

Helen was tempted to say something about her being right after all, but thought better of it. Besides, Ace was heading for the door.

“I’m coming with you,” she said, picking up her raincoat and purse.

“Sorry, Baby,” Ace replied. “This one might be dangerous, so you’re staying here. Wait by the phone in case something goes wrong.”

“I suppose you’re right,” she said reluctantly. “But be careful, Sweetheart.” Then she tenderly kissed him on the lips.

“I should be back in an hour or so,” he said. “If you haven’t heard from me by then, call the cops. Tell them I’m on a mission to bring in Bohacker!” One more kiss and he was gone.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Preventing Program Execution

Casebook No. 16, April 2: It was interesting to explore how to shut down a program when it finds a previous instance running. But something kept bugging me. What about the times you don't want an application to run unless another program *is* running?

This is the case with several demo versions of components, such as the Orpheus VCL library. If you create an application using what TurboPower Software calls the "trial" version of their components, they require Delphi to be running in order for the application to execute. How could something like this be accomplished?

The answer was so simple I almost couldn't believe it. The running version of this experiment is shown in Figure 14.4. The main form's code is shown in Listing 14.5, and the project source is detailed in Listing 14.6.

FIGURE 14.4 The Delphi Environment Detector running.

Listing 14.5 Code for the No Run Demo's main form

```
{──────────────────────────────────────────────────────}
{               The "No Run" Demo                      }
{            NRUNMAIN.PAS : Main Form                  }
{            By Ace Breakpoint, N.T.P.                 }
{             Assisted by Don Taylor                   }
{                                                      }
{ Main form for the application that demonstrates      }
{ how to prevent execution if Delphi 2.0 isn't         }
{ executing.                                           }
{                                                      }
{ Written for *Kick-Ass Delphi Programming*            }
{ Copyright (c) 1996 The Coriolis Group, Inc.          }
{              Last Updated 4/2/96                     }
{──────────────────────────────────────────────────────}

unit NRunMain;
```

```
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, WalkStuf;

type
  TForm1 = class(TForm)
    ExitBtn: TButton;
    Label1: TLabel;
    procedure ExitBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.ExitBtnClick(Sender: TObject);
begin
 Close;
end;

end.
```

Listing 14.6 No Run Demo project file

```
{————————————————————————————————————————————————————}
{                The "No Run" Demo                   }
{             NORUN.DPR : Project File               }
{            By Ace Breakpoint, N.T.P.               }
{             Assisted by Don Taylor                 }
{                                                    }
{ An application that demonstrates how to prevent    }
{ execution if Delphi 2.0 isn't running.             }
{                                                    }
{ Written for *Kick-Ass Delphi Programming*          }
{ Copyright (c) 1996 The Coriolis Group, Inc.        }
{              Last Updated 4/2/96                   }
{————————————————————————————————————————————————————}

program NoRun;

uses
  Forms, Dialogs,
  NRunMain in 'NRunMain.pas' {Form1},
  WalkStuf in 'WalkStuf.pas';

{$R *.RES}

begin
  Application.Initialize;
```

```
  { If there isn't a copy of Delphi 2.0 running, display an error
    message and kill the program. Otherwise, let this application
    execute. }
  if ModuleSysInstCount('DELPHI32.EXE') < 1
   then MessageDlg('Delphi 2.0 must be running to execute this program',
         mtError, [mbOK], 0)
   else begin
         Application.CreateForm(TForm1, Form1);
         Application.Run;
        end;
end.
```

The goal is to kill the application without the user even seeing the main form. So once again, code is added to the project file to accomplish the task. This time the **ModuleSysInstCount** function from the **WalkStuf** unit is used to make sure at least once instance of Delphi 2.x (DELPHI32.EXE) is running. If so, the program continues. If not, an error message is displayed.

One note: Because WalkStuf relies on ToolHelp32, this technique will only work with programs running under Win95.

End of entry, April 2.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Floating Toolbars

Casebook No. 16, April 3: I've always liked programs with toolbars that can be moved around on the screen. They're especially handy in artwork and page makeup programs, because you can have a tool palette right next to where you're working.

I sifted through the various components provided with Delphi, looking for something that could be used as the basis for a "floating" toolbar. I suppose I could have used an additional form, but I didn't need anything fancy. Something that would be limited to the main form's client area would be just fine.

It seemed that Panels would do the trick nicely, except for one thing: they aren't movable at runtime. But a little investigation revealed they have the provision for mouse event handling. After a few bumps and blind alleys, I came up with the demo program documented in Listing 14.7.

Listing 14.7 Code for the Floating Toolbar demo

```
{————————————————————————————————————————}
{            The "Floating Toolbar" Demo                }
{             TOOLMAIN.PAS : Main Form                  }
{             By Ace Breakpoint, N.T.P.                 }
{              Assisted by Don Taylor                   }
{                                                       }
{ An application that demonstrates the use of           }
{ moveable TPanel objects as floating toolbars.         }
{                                                       }
{ Written for *Kick-Ass Delphi Programming*             }
{ Copyright (c) 1996 The Coriolis Group, Inc.           }
{                Last Updated 4/3/96                    }
{————————————————————————————————————————}

unit ToolMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, FileCtrl, ExtCtrls, Buttons;

type
  TDirection = (otHorizontal, otVertical);
```

```
  TForm1 = class(TForm)
    Toolbar: TPanel;
    ExitSB: TSpeedButton;
    ZoomInSB: TSpeedButton;
    ZoomOutSB: TSpeedButton;
    ControlPanel: TPanel;
    GranRBGroup: TRadioGroup;
    MarginRBGroup: TRadioGroup;
    OrientRBGroup: TRadioGroup;
    ExitBtn: TButton;
    LEDSB: TSpeedButton;
    procedure ExitBtnClick(Sender: TObject);
    procedure ToolbarMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure ToolbarMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure ToolbarMouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure FormCreate(Sender: TObject);
    procedure GranRBGroupClick(Sender: TObject);
    procedure MarginRBGroupClick(Sender: TObject);
    procedure ExitSBClick(Sender: TObject);
    procedure OrientRBGroupClick(Sender: TObject);
  private
    DraggingPanel : Boolean;
    DragStartX : Integer;
    DragStartY : Integer;
    GridSize : Integer;
    MarginSize : Integer;
    procedure OrientToolBar(Direction : TDirection);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.ExitBtnClick(Sender: TObject);
begin
 Close;
end;

procedure TForm1.ToolbarMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 if Button = mbLeft
  then begin
        DraggingPanel := True;
        DragStartX := X;
        DragStartY := Y;
       end;
end;

procedure TForm1.ToolbarMouseUp(Sender: TObject; Button: TMouseButton;
```

```
  Shift: TShiftState; X, Y: Integer);
begin
 DraggingPanel := False;
end;

procedure TForm1.ToolbarMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
var
 DeltaX : Integer;
 DeltaY : Integer;
 SafetyMargin : Integer;
begin
 if DraggingPanel
  then with Toolbar do
   begin
    DeltaX := X - DragStartX;
    DeltaY := Y - DragStartY;
    if GridSize > MarginSize
     then SafetyMargin := GridSize
     else SafetyMargin := MarginSize;

    if (abs(DeltaX) > GridSize - 1)
     then if DeltaX > 0
           then begin
                 if (ControlPanel.Left - Left) > SafetyMargin
                  then Left := Left + DeltaX
                  else Left := ControlPanel.Left - SafetyMargin;
                end
           else begin
                 if (Left + Width) > SafetyMargin
                  then Left := Left + DeltaX
                  else Left := SafetyMargin - Width;
                end;

    if (abs(DeltaY) > GridSize - 1)
     then if DeltaY > 0
           then begin
                 if (Form1.ClientHeight - Top) > SafetyMargin
                  then Top := Top + DeltaY
                  else Top := Form1.ClientHeight - SafetyMargin;
                end
           else begin
                 if Top + Height > SafetyMargin
                  then Top := Top + DeltaY
                  else Top := SafetyMargin - Height;
                end;

   end; { with }
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 GranRBGroup.ItemIndex := 0;
 MarginRBGroup.ItemIndex :=0;
 OrientRBGroup.ItemIndex := 0;
end;

procedure TForm1.GranRBGroupClick(Sender: TObject);
begin
```

```
 case GranRBGroup.ItemIndex of
  0 : GridSize := 1;
  1 : GridSize := 10;
  2 : GridSize := 20;
 end; { case }
end;

procedure TForm1.MarginRBGroupClick(Sender: TObject);
begin
 case MarginRBGroup.ItemIndex of
  0 : MarginSize := 5;
  1 : MarginSize := 10;
  2 : MarginSize := 15;
 end; { case }
end;

procedure TForm1.ExitSBClick(Sender: TObject);
begin
 Close;
end;

procedure TForm1.OrientRBGroupClick(Sender: TObject);
begin
 case OrientRBGroup.ItemIndex of
  0 : OrientToolBar(otHorizontal);
  1 : OrientToolBar(otVertical);
 end; { case }
end;

procedure TForm1.OrientToolbar(Direction : TDirection);
begin
 with Toolbar do
  begin
   Left := 20;
   Top := 20;

   case Direction of
    otHorizontal :
     begin
      Width := (4 * ExitSB.Width) + 20;
      Height := ExitSB.Height + 10;
      ExitSB.Top := 6;
      ZoomInSB.Top := 6;
      ZoomOutSB.Top := 6;
      LEDSB.Top := 6;

      ExitSB.Left := 11;
      ZoomInSB.Left := ExitSB.Left + ExitSB.Width;
      ZoomOutSB.Left := ZoomInSB.Left + ZoomInSB.Width;
      LEDSB.Left := ZoomOutSB.Left + ZoomOutSB.Width;
     end;

    otVertical :
     begin
      Width := ExitSB.Width + 10;
      Height := (4 * ExitSB.Height) + 20;
      ExitSB.Left := 6;
      ZoomInSB.Left := 6;
      ZoomOutSB.Left := 6;
```

```
      LEDSB.Left := 6;

      ExitSB.Top := 11;
      ZoomInSB.Top := ExitSB.Top + ExitSB.Height;
      ZoomOutSB.Top := ZoomInSB.Top + ZoomInSB.Height;
      LEDSB.Top := ZoomOutSB.Top + ZoomOutSB.Height;
     end;
   end; { case }
  end; { with }
end;

end.
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

As can be seen, the panel (Toolbar) must handle three mouse events—OnMouseDown, OnMouseMove and OnMouseUp. The OnMouseDown handler simply makes sure it's the left mouse button that has been pressed. If so, it documents the initial position of the mouse and it sets a status variable that indicates a dragging process is underway.

The OnMouseMove handler is quite a bit more complex, mainly because it is making sure the panel doesn't move outside the client area of the form, where it might get lost. Basically, the **ToolbarMouseMove** handler computes the difference between the original mouse position and the current mouse position, and then it adds the differential to the original **Left** and **Top** properties of Toolbar to move it. A provision has been made to move Toolbar by jumps of 1, 10 or 20 pixels, giving it the same feeling as moving it in the "lock to grid" mode when designing with Delphi. I also added a margining capability, to make sure a piece of the toolbar can always be grabbed with the mouse, in those cases when the toolbar gets shoved off in a corner.

The OnMouseUp handler is obvious: All it has to do is turn off the status variable.

I added three radiobutton groups to the form, so grid and margin sizes can be set on the fly. I also included a capability for switching the toolbar between horizontal and vertical orientations.

An executing version of the program is shown in Figure 14.5. It's kind of fun to drag the toolbar around, and the first button on the toolbar is actually functional—it ends the program.

FIGURE 14.5 Floating a Delphi toolbar.

End of entry, April 3.

Ace Gets the Goods

As the Delphi Avenger glanced up from the Casebook, it was obvious that something had gone awry. The room had slowly filled with a thick carpet of fog that now hugged the floor like a Basset Hound on its day off. A brilliant white light emanated from the cloud, illuminating everything in the room with an eerie cast. The Avenger drew in a quick breath of air, a breath that stung the nostrils with the numbing cold of carbon dioxide. Something was very, very wrong. Then suddenly, from across the room there came a violent explosion, as the rickety door blew off its hinges and crashed inward to the floor. In the open doorway, illuminated from behind by the neon bug zapper in the hallway, stood a triumphant Ace Breakpoint!

* * *

The odor that permeated the seedy little apartment was unmistakable, a veritable snack food smorgasbord. I crossed the room powerfully, knowing I had not only solved this case, but that I had wrested complete control of the story from The Third Person. Three giant steps, and I had closed the gap between myself and the pathetic creature that cowered against the opposite wall—the now-defeated entity that once dared to call itself the Delphi Avenger.

"Breakpoint!" Bohacker hissed, dropping my Casebook on the floor. "How did you know?"

"Easy," I snarled. "From the very beginning of this case, I knew something was amiss. It just took me a while to realize the "miss" was *you*… *Miss Bohacker!*" I shouted, as I tore the false mustache from her upper lip.

"Cursors!" she spat through clenched teeth. She quickly scrutinized me from head to toe, then back again, like a wild animal trapped in a corner. The look of defeat was gone, and in its place was a twisted smile that quite unnerved me. "Yes, it's me," she grinned. "Melvin Bohacker's evil twin sister, *Mevlyn*."

A chill ran down my spine. Suddenly I felt like I was playing a scene in a very bad "B" movie. How did I get myself into these situations? I just shook my head.

"Sure," she said disdainfully. "Be *Mr. High and Mighty*. But imagine for just one minute what *your* life would be like, spending every day of your childhood following in Melvin Bohacker's footsteps."

And following his nutritional guidelines, as well, I mused, noticing the mass of Twinkie and HoHo wrappers strewn all over the apartment. But strangely, I did feel a twinge of empathy. I encouraged her to resume her confession.

"He was exactly eight minutes older," she said. "So *he* got all the good stuff. I got whatever was left. And when it came time for college, there was only enough money to send one of us. So of course it was Melvin who went off to

school, while I stayed home. I probably would have starved to death, if I hadn't started the designer clothes business."

"Wait a second," I said. "You mean you're the owner of Bohacker Industries? The people who make *Bohacker Blues* jeans?"

"You hadn't made the connection?" she asked wryly. "Then you're also not aware that my company produces the *Purely Prophet* designer line for your little friend, Muffy Katz."

"Why, you must be worth a bundle!" I exclaimed.

"Oh, yes—I have a net worth of several million. Which is fine, I suppose—if you're merely interested in money. But Melvin is the one who got all the *love*. He got the *accolades*. He's the one who got the college education. And he's the one who went on to become one of the most esteemed and revered members of the entire scientific community—a Windows 95 programmer."

Her eyes burned with hatred. "Since childhood, I've been a dysfunctional, codependent victim, and I've sworn I would get even with Melvin, no matter how long it took. Then a few weeks ago, it became obvious," she said, staring dreamily off into the distance. "I would steal your Casebook, and armed with the secrets it contains, I would become the best Windows programmer in the world—better even than Melvin. And certainly better than you."

She stood silently for several seconds, then turned back to continue the confession. "At the same time," she said thoughtfully, "I could frame Melvin for the robbery. I called him earlier today, to torment him and then lure him out of town tonight, so everyone would think he had taken it on the lam. It was a perfect plan. But something went wrong. Evidently I underestimated you."

"You forgot I used to be a detective," I replied. "And you left a trail of clues so transparent that even an MIS manager from Bayport could have followed them."

"Like the glove?" she asked. "Losing it was an accident. I thought of trying to get it back. But even so, it should have pointed to Melvin, not me."

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

"Sure it did. The muddy glove left at the crime scene contained some hair that was an exact DNA match with Melvin's. But at that point, I knew this had to be a frame, albeit a very sophisticated one. There were two reasons for that conclusion. You see, after the drubbing I gave him when he tried to double-cross me a year ago, I knew Melvin would never have the nerve to try anything underhanded with me again. I broke his spirit."

"And the second reason?" she queried.

"Whatever I might think of your brother, I didn't believe he was in the habit of wearing *ladies' gloves*," I chuckled. "So I queried the Department of Licensing computer and got a copy of his driver's license. That netted me his birthdate. I knew he was originally from California. So my next link was to a large database, where I searched through all the California county records."

She was one step ahead of me. "So you found our birth certificates," she said.

"Yeah. That's when I discovered there were actually *two* babies born to Chester and Martha Bohacker that night," I continued. "—a boy named Melvin, and a *girl* they called *Mevlyn*, born a few minutes later. Twins. That's why the DNA samples matched almost perfectly."

"All right. So you found out I was a twin. How did you trace me to this dump?" she asked.

"I guessed you had been in town for at least a couple of weeks to implement your plan. There were no new guests registered at the Gooey Duck Grill and Motel, the only place in town to stay. So I examined the local phone company's records, looking for new service connections made in the last couple of weeks. This is a small town. It wasn't long before my search led me here to the Sleeveless Arms. One final link to California had given me your car registration. When it matched your vehicle parked out front, I knew I had the goods. Typical gumshoe stuff."

“Gumshoe, perhaps. But typical? I don’t think so,” Mevlyn said, her voice suddenly becoming soft and low. The sudden change in her tenor caught my attention. Her eyes no longer filled with hatred but with icy fire, she removed her dark, wide-brimmed hat and lazily flung it across the room, carelessly tossing her head. Her silky, brunette tresses spilled over her shoulders, flowed down her back, turned at her heels, and rolled across the three-foot expanse of floor to the wall, where it lapped up onto the molding. It was obvious I would have to watch my step with this woman.

Someone next door began to play an alto saxophone. Mevlyn picked up my Casebook from the floor where she had dropped it. “Listen, Ace,” she said, lifting her eyes to mine and tossing me a coquettish smile. “Maybe we just got off to a bad start, you and I. Maybe you could just leave this itsy-bitsy book with me, and we could get to be *really close friends.*”

She pursed her lips, and I noticed she was caressing the buttery-smooth leather cover on my hardbound Casebook as if it were a prize-winning Pomeranian. This situation had all the earmarks of something that could turn ugly very fast. I had to keep my distance. Three bass saxophones had joined the alto, and now they were so loud it seemed they were right there in the room. *These cheap apartments really do have thin walls*, I thought.

“I’m afraid I can’t do that,” I said, stepping back. “I already have a really close friend. Her name is Helen.”

“I know,” she replied, steadily advancing toward me. “I’ve seen her. Nice girl, kind of… country. Sweet face. Cute little figure. But tell me,” she said, a wry little smile spreading across her mouth in anticipation, “how do you think she’d like to have *these?!*”…

With a single motion, and while still holding the Casebook, she grabbed both of her lapels and threw open her full-length raincoat. I gasped and reeled backward, my wide-eyed gaze fixed upon the two incredible objects displayed inside the open raincoat directly in front of me: Sticking out of the inner pocket were a pair of front-row tickets for the Seattle concert on the Stones’ 1999 “Old Rocker” Tour. She smoothly removed the passes and pressed them into my palm, closing my fist around them.

“I’m glad you like them,” she said.

“How—how did you manage to get these?” I stammered, trying to regain my composure.

“Money talks, Sweetheart,” she replied. “Go on. Take them. They’re yours.”

I stared at the tickets for what seemed like an hour. “I’m—I’m sorry, I, uh, just can’t accept a bribe,” I finally said, still visibly shaken. I reluctantly shoved the ill-gotten booty back into her pocket. “I have no choice. I’ll have to turn you in.”

“Can’t blame a girl for trying,” she said, rather matter-of-factly. “So what do you say, Ace…how about a little kiss, just to show there’s no hard feelings?”

She flashed me a devilish smile and pressed close to me. She slipped her arms

under mine and pulled me to her, still maintaining her tight grip on the Casebook. “A…*kiss*?” I asked stupidly, trying to stall for enough time to get my bearings. A tenor sax had now joined the others to form a quintet, and some idiot had started accompanying them on a kettle drum that resonated inside my chest and echoed in both of my ears.

Her liquid, steel-gray eyes were now riveted on mine. “Of course, silly,” she replied playfully. “You know what a kiss is, don’t you? It’s when two people each press their lips together, and then *together* they press their lips together… like… this….”

Until now, I hadn’t really noticed her mouth. But as her moist, ruby lips inched their way toward mine, her breath laden with the sweet scent of Twinkies, I could think of nothing else. I found myself in a trance, suspended in time with a small orchestra playing Barry Manilow’s greatest hits, all at once. What could I do? What about Helen? The voluptuous, trembling red lips came closer and closer….

Just as our lips were about to meet, I heard a muffled *thump* behind me, and a small explosion went off in my brain. Everything slowly melted from Technicolor into black and white and then into a palette of grays. Mevlyn’s lips became twisted in a maniacal laugh that seemed to echo from every wall. I caught a whiff of the distinctive perfume she wore, the same scent that had permeated the scrap of note paper. Now it completely filled my nostrils and burned indelibly into my brain.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

"*Chez Monieux*," I heard myself say to Mevlyn in a half-whisper. Then the whole eerie scene faded to black.

* * *

When I came to, my head was pounding and I had a mouth full of cotton candy. Struggling to my feet, I threw away the paper cone and I spit the sticky pink mass into the wastebasket. I glanced at my watch. It was 7:34 `p.m.` There was no sign of either Mevlyn or the Casebook, but downstairs I could hear the tell-tale sound of a racing engine as her slow-moving white Bronco lumbered away from the curb.

On the table was a lavender-colored envelope, on its face a message written in a familiar feminine hand. I picked it up and shook my head until I was finally able to focus on the writing in front of me:

> *Dearest Ace,*
>
> *Inside you'll find a little something just for you.*
>
> *I think we really could have hit it off together. But a girl's gotta do what a girl's gotta do. Perhaps we will meet again.*
>
> *I suppose it's like this: You can't always get what you want. But if you try sometimes, you get what you need.*
>
> *Always,*
>
> *Mevlyn*

I eagerly tore open the scented envelope and emptied its contents on the desk—a single strip of thin, white cardboard with a perforation near one end.

I picked up the ticket and stared at it, reviewing the whole crazy scene in my mind. First I had been struck with her beauty, which had left a permanent

impression on my psyche. Then I had been struck with my hard-bound Casebook, which had left a considerable goose egg on the back of my head. The lump would eventually heal. Right now I had to get my life back together.

I guessed that when my socially-conscious friends heard about this incident, they would all encourage me to try to better understand Mevlyn and to forget the pain and suffering she had caused me. And I supposed I would eventually be able to do that. But it would be a long time before I would again trust any woman with a twisted grin and a maniacal laugh.

Epilogue

Casebook No. 17, April 13: It was the best of times. It was the worst of times. For me, the past 24 hours had been a Dickens of a time. It had been an adventure I never wanted to repeat.

Helen was waiting for me when I got back from the Sleeveless Arms. At least she had been partially right—the perpetrator was a Bohacker. But I had been right, too—he would never have had the courage to try anything with me. Thanks to the information I had gotten from Biff, I managed to track Bohacker to the Gates Motel, where I reached him by phone….

“Hey, I had a ‘date’ tonight with someone you know,” I said. “And it looks like she got the better of me.”

“Yeah? Tell me about it,” he replied. “Been there. Done that. Got the T-shirt.” I guessed I had only an inkling of what it must have been like, growing up with Mevlyn. I decided to change the subject to something less painful for both of us.

“Tell me, Bo—are you still doing all your visual programming in C++?” I asked.

“Naw,” he replied. “I’ve found something much simpler that I’m using for a lot of my stuff. I can quickly produce visual programs for networked, client/server environments. Although it *is* somewhat like using C++.”

It was an opportune time to put the screws to him. “Is it fast?” I asked.

“Well, it’s actually about one-tenth the speed I’m used to, execution-wise,” he admitted. “But with just-in-time compilers—and now with—uh… Well, things are going to be a lot better.”

He was trying to cover up for some bad choices. Obviously, things weren’t going well for him. I decided to blow him off.

“I’ve gotta go. Had a big day, you know. Get some good sleep. And if you ever see your sister, say ‘Hi’ for me, okay?”

“Sure will,” he said.

I placed the receiver back on its hook. It was obvious he still hadn’t discovered Delphi. One thing was for sure: He wasn’t going to hear about it from me.

My watch told me it was 11:39 `p.m.` I needed to ponder a few things. I

poured a cup of Dark Roast and took a big gulp of the hot brew. I now knew what it was like to have my office ransacked and my property stolen. It's not an experience I want to have again. Like my Dad, Jack Breakpoint, always told me, "Take care of your stuff, son. There's always someone out there who wants it more than you do." Then Mom would add, "And always wear your best underwear, in case you get hit by a bus."

One thing was for sure: I wouldn't leave my Casebooks lying around any more. In all probability I would never see the stolen one again. Thankfully, it was only my most recent Casebook, one of several. Someone told me the best parts of it are on a CD-ROM included with a new book on Delphi programming. I can't remember who it was for sure who told me. I think it was The Editor.

Which reminds me—I had a long talk about The Author with Helen tonight, before she went home. She convinced me that if I really do appreciate what The Author has done for me, I need to talk to him more often and to tell others about him. All things considered, I guess that's the least I can do.

Against my better judgment, I picked up a copy of the latest CD from *Not Ourselves Today* at a convenience store on the way home. It's entitled "Transcend Dental Medication." I hit the stop button halfway through the first track, "Rinse and Spit." It was too much like sitting through a root canal to suit my taste.

I sat down at my kitchen table and took another sip of the bitter brown water. Outside I could hear the sounds of the relentless rain as it pounded against the window. I pulled out my wallet and extracted the lone ticket from the envelope left at The Sleeveless Arms. For some reason I just couldn't get it off my mind.

Perhaps I *would* go to that concert all by myself. It would be a good opportunity to get away from the hustle and bustle of life in a small town. It was my kind of music.

And after all, you never can tell who may show up in the seat beside you.

End of entry, April 13.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Previous | Table of Contents | Next

CHAPTER 15
An Age-Old Problem

DON TAYLOR

- Windows 95 Broadcast Messaging
- Registering Windows Message IDs
- Message Sender And Receiver Components
- Semaphores

Ace takes the stage to explain the machineries of Windows 95 broadcast messaging to an extremely mature audience, at least one member of which seems *remarkably* familiar.

I awoke with a start. The clock by my recliner told me it was 1:23 p.m., which meant I had only seven minutes to make it all the way across town. I tossed the pizza box off my lap, and in what seemed only a few seconds later, I found myself in an unfamiliar part of town. I somehow managed to find the address I had penciled in my day planner.

The sign at the parking lot entrance read “Hacker Have—An Upward-Compatible Digital Retirement Community.” I vaguely remember hearing rumors about the construction of an Old Programmers Home here in Poulsbo. But as I pulled into the parking lot, I wondered why I hadn’t noticed the place before. It was a huge two-story complex that must have covered the better part of an entire city block.

I held my attaché case above my head and vainly tried to dodge the raindrops as I raced for the main entrance. I hustled through the automatic doors and checked my watch. 1:30 p.m. exactly — I had just made it.

There was no one to be seen in the lobby, so I decided to poke around a bit to satisfy my curiosity. Who knows, I thought, someday maybe I'll be living here myself. I picked the second of three corridors leading from the lobby.

The first six doors down that hallway were all marked "Restroom." I quietly slipped inside the seventh door, under a sign that read "Recreation Room." I guess I expected the place to be full of ex-programmers working out on exercise equipment. To my surprise, I found a room containing 30 individual cubicles, each with its own client PC, all connected to a network server running the multiplayer version of Doom.

I walked completely around the room, but couldn't find one person. I supposed a bunch of old programmers might be insular enough to socialize only through games over a network. But why was the room empty? It wasn't meal time. Where was everybody? I was standing there scratching my head when a voice came from behind me.

"May I help you?"

I whirled around to face the questioner. She was a relatively young woman, probably early thirties, a tad on the hefty side.

"My name's Breakpoint," I said. "Ace Breakpoint."

"I thought so," she replied. "You're here to present the in-service lecture on programming for Windows 95. I'm Dinah Setz, the Volunteer Coordinator volunteer," she said, extending her hand. After a few pleasantries, she invited me to follow her to the multipurpose room where the lecture was to be held.

"I'll be your guide while you're here," Dinah said, as we walked down the long hallway. "If you have any needs, or if there are any questions I can answer, please feel free to ask."

"There is one thing," I said. "We just passed four more restrooms. Why so many?"

"Having them helps alleviate some of the little problems that burden many of our guests," she replied, flashing me a knowing smile.

"Oh, I get it," I said. "The old geezers can't pack the mail any more."

"We prefer to call them our 'Legacy Application Developers' or more simply, 'guests,'" she said. "We have both men *and* women here, by the way. And yes, many of our guests experience some common challenges due to the number of their years. Since they are all programmers, we euphemistically refer to the results of these problems as 'errors,' and we have assigned standard error numbers to most of them."

"I don't think I understand," I said.

"Well, the most common occurrence is *Error 214: Collection overflow error*. It frequently cascades from an *Error 11: Line too long*. That's why we have included a large number of restrooms in the facility."

"I see. Uh, listen," I said, noticing we had come to the dining room. "If you

don’t mind, I’d like to get a little something to bring in with me. Just in case I need some extra energy during the lecture.”

“Certainly,” she said. “Help yourself.”

I expected to get a snack from the kitchen. On entering the dining room, I was surprised to find there *was* no kitchen—only a series of vending machines that encircled the room full of tables and chairs. Dinah said the place saved a lot of money by just ordering takeout food and billing it to the guests’ accounts. I walked up to one of the machines and dropped in four quarters.

“Ready to go,” I said, slipping a candy bar into my pocket. We walked past an expanse of windows, where Dinah pointed out Hacker Haven Cemetery, located at the rear of the property.

“Nice place,” I commented, noticing the immaculately manicured lawns. “Beautiful place. I’ll bet you charge an arm and a leg to get in there.”

“It’s largely a matter of convenience,” she said. “With the cemetery right here on the property, our guests never have to leave us. We like to think of it as our way of enabling our programmers to terminate and stay resident. Well, here is the multipurpose room.”

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Facing the Situation

Dinah opened the door and motioned for me to go in. “I’ll stay with you,” she said, “in case you have any problems. Just walk up on the stage.”

The whole place was buzzing with excitement. *So this is where everyone had gone*, I thought. This was truly going to be an event. I cautiously made my way up the aisle, past rows full of gray-haired programmers seated in ergonomic easy chairs. On the stage was a complete 5GHz Pentagram server with 64 gig of RAM, full T-1 net access, and a 64 million color projection system. I fished around in my pocket until I located the Delphi system CD and the floppy that contained the entry-level code examples I had prepared for today’s lecture. I sat down at the server and installed Delphi. Then I slipped the floppy into the powerful machine and waited in anticipation. I didn’t have to wait long.

Dinah picked up the microphone, and a hush fell over the room. “We are very fortunate today,” she began, “to have with us Mr. Ace Breakpoint, an expert who is going to share with us some of the finer points of programming under Windows 95 with Delphi 2.0.” She glanced my way and smiled, and I shot back a look of near panic. This was not what I had intended. Not at all.

“I’m sure we all remember,” she continued, “the lecture Mr. Bohacker gave us last week on entry-level programming. So now, let’s welcome Mr. Breakpoint, as he presents more advanced topics.”

I’m sure she must have motioned to me to take the floor. All I remember was the sick feeling in the pit of my stomach, and some guy in the third row shouting, “Yeah—let’s see if he can top Bohacker!” I suddenly felt I was having a bad dream—like the one where you walk into class on the day of the final exam, and it’s the first time you’ve ever attended the class. But this was even worse. Why had I volunteered to do this gig? I honestly couldn’t remember even talking to anyone about it.

I stepped up to the microphone and tried to get a grip on myself. “What we’ll be covering today won’t quite be ‘finer points’ of Win95 programming. Instead you’ll hear some really basic information I’m sure you’ll all want to know—”

I felt a dull pain between my ribs, and looked down to see an old coot who had gotten up from his seat in the front row. “Listen, Sonny,” he said, poking me in the chest with the tip of his telescoping presentation pointer. “All I want to know is how you would have two different Win95 applications communicate between themselves, using standard Windows messaging.”

Specifying the Problem

“Yeah!” someone else shouted. Then, one by one, residents began to voice their requirements for a problem specification.

“What if you’ve got multiple instances of sender and receiver apps—how are you going to satisfy them all?” yelled a shriveled little man in the fourth row. “And how are you going to make sure any message you’re broadcasting throughout the system won’t trigger an unwanted side effect on some other application that happens to use the same message ID?”

“How about the ability to selectively send a message to a single form, all forms, or certain classes of forms with reception capability?” added the man in the first row with a bag over his head. He looked somehow familiar....

“Componentize it, componentize it!” shouted another man. One by one, every person in the audience began to pick up the chant. Pandemonium reigned for several minutes. I tried to get my bearings and come up with some sort of approach.

As luck would have it, several staff members came in, rolling carts loaded with takeout pepperoni pizza. Within two or three minutes, the entire room had quieted down once again, as the residents munched on the tasty fare. I was able to use those precious moments to create the diagram shown in Figure 15.1.

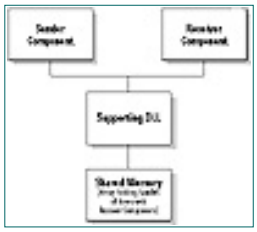

FIGURE 15.1 Relationship of entities comprising Sender and Receiver components.

“Ladies and gentlemen,” I began, as I projected my bitmap on the screen. “Here is one possible approach to the problem you’ve described. We could create two separate components, a sender component and a receiver component. Each of these components would share some common routines contained in a Dynamic Link Library (DLL). On creation, each receiver component would register itself with the DLL, which would maintain in shared memory an array of records. Each record would contain the handle values and class names of all active receivers, along with a boolean that indicates whether

or not that record was occupied. A receiver would unregister itself with the DLL as part of the receiver's destructor routine."

I swept my eyes across the crowd, looking for major objections. Seeing none, I continued. "Sender components can have access to the handles of the receivers by querying the DLL. By calling routines included in the DLL, a sender can send a message selectively to any or all of the handles it gets from the query."

"Wait a second," came a voice from the back of the room. "You skipped over a very important point. You're talking about sending messages to a form in another process. How are you going to make sure the message IDs you're going to broadcast aren't used by some other process?"

"By being different," I replied. "Windows gives us an API call named **RegisterWindowMessage**. This function accepts a string and returns a message ID that is guaranteed to be unique throughout the system. If a second application tries to register the same string, it is given the same unique message ID. This is how the sender and receiver components can share a unique message—by simply sharing a string that is made a property of each component."

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

"There's a serious flaw in your logic, Kid," said a totally bald-headed codger sitting in front of me. "You may have a unique message, all right. But you won't be able to create a handler for it, because you won't know the message ID until runtime."

I thought for a moment. *How was I going to handle this one?* Then it came to me.

"A radio wave," I said.

"What? Are you crazy?" shouted some curmudgeon in the third row.

"No, really. We can send out a message the way a transmitter sends a modulated signal to a receiver. The transmitter impresses a signal—information—onto a carrier wave of a designated frequency. Because the receiver is tuned to that same designated frequency and is constantly monitoring the incoming signal, it is able to pick off the information from the carrier wave, process it, and extract the information."

Dead silence. I looked out over the crowd. But all I could see was 249 eyes staring back at me.

"Thanks to a call to **RegisterWindowMessage**," I continued, "both the sender and receiver of our intended message will wind up with the same unique message ID—call it **FMessageID**. That's like the frequency of the carrier wave. The real message will be sent as a package inside the message with the unique ID—sort of a private message. The sender will transmit a message to the receiver with the standard **SendMessage** function, using the receiver form's handle and the unique "carrier" message ID. But the private message ID and any additional message information will be packed into **wParam** and **lParam**. It will look something like this." I typed in the function shown in Listing 15.1.

Listing 15.1 Prototype of a message transmission routine

```
function TMsgSender.DoBroadcast(Handle : Integer;
                                wParam : Word;
                                lParam : Longint) : Longint;
begin
 Result := BroadcastIt(FMessageID, Handle, wParam, lParam);
end;
```

They looked skeptical, but they were still hanging in there. "On the receiver side, we declare as a property a generic event that can be called whenever the form containing the receiver detects the unique message ID. If we call the event **FOnIDMessage**, the routine might look something like this," I said, punching in the procedure shown in Listing 15.2.

Listing 15.2 Prototype of a message reception routine

```
procedure TMsgReceiver.WndProc(var Msg : TMessage);
```

```
begin
 if Msg.Msg = FMessageID
  then begin
        if Assigned(FOnIDMessage)
         then Msg.Result := FOnIDMessage(Msg.wParam, Msg.lParam);
       end
  else Dispatch(Msg);
end;
```

“Hold on, Wizard Boy,” came a voice from somewhere off to the left. “Now you’ve overstepped your bounds. You’re presuming the receiver component has somehow taken over the message processing for the form’s window. But you haven’t subclassed the window.”

“It will be part of the component’s design for it to automatically subclass the form on which it resides,” I explained. “But enough of this talk. Let’s write some code so we can explore this whole concept.”

I looked around the room. For the first time, they had all sat back in their chairs. I had their undivided attention. Now I just had to do the impossible: Make them happy.

Designing the DLL

By far, the DLL would be the biggest challenge. It was also the centerpiece of the whole process, so it seemed like a logical place to start. With the residents shouting out their own ideas for what needed to be included in the DLL, we hammered out a rough version. I couldn’t believe how quickly we came up with the final version of the code, shown in Listing 15.3.

Listing 15.3 DLL for sender/receiver component support

```
{————————————————————————————————————————}
{         Message Broadcasting in Delphi 2           }
{           BCASTDLL.DLL : Library File              }
{            By Ace Breakpoint, N.T.P.               }
{             Assisted by Don Taylor                 }
{                                                    }
{ DLL that provides the underlying function of       }
{ managing a list of registered components in        }
{ shared memory.                                     }
{                                                    }
{ Written for *Kick-Ass Delphi Programming*          }
{ Copyright (c) 1996 The Coriolis Group, Inc.        }
{               Last Updated 5/2/96                  }
{————————————————————————————————————————}

library BcastDll;

uses
  SysUtils,
  Classes,
  Windows;

type
 PMsgReceiverRec = ^MsgReceiverRec;
 MsgReceiverRec =
  record
   Assigned : Boolean;
   WndHandle : HWND;
   WndClassName : ShortString;
  end; { record }
```

```
const
 MaxReceivers = 100;
 ArraySize = MaxReceivers * SizeOf(MsgReceiverRec);
 MemFileName = 'msg_reg_array';
 SemaphoreName = 'msg_bcast_semaphore';

var
 ArrayHnd     : HWND;
 ArrayBasePtr : PMsgReceiverRec;
 SemaphoreHnd : HWND;
 DLLValid     : Boolean;
 SaveExit     : Pointer;
 LastErr      : Longint;
{ BcastDLLValid returns a Boolean that lets the caller know
  that the shared memory and semaphores are operational, so
  all of the other exported routines are valid. }
function BcastDLLValid : Boolean;
begin
 Result := DLLValid;
end;

{ RegisterReceiver adds the specified handle and class name
  to the array of receivers in shared memory. It returns the
  index (plus 1) of the new entry, or -1 if it couldn't be found. }
function RegisterReceiver(Hnd : THandle; CName : ShortString) : Integer;
var
 Idx : Integer;
 Found : Boolean;
 RecPtr : PMsgReceiverRec;
begin
 Idx := 0;
 Result := -1;
 RecPtr := ArrayBasePtr;
 Found := False;
 WaitForSingleObject(SemaphoreHnd, INFINITE);

 { Find an unused slot }
 repeat
  Found := RecPtr^.Assigned = False;
  if not Found
   then begin
         Inc(Idx);
         Inc(RecPtr);
        end;
 until Found or (Idx >= MaxReceivers);

 if Found
  then begin { Put the record in the slot }
         FillChar(RecPtr^, 0, SizeOf(MsgReceiverRec));
         with RecPtr^ do
          begin
           Assigned := True;
           WndHandle := Hnd;
           WndClassName := CName;
          end; { with }
         Result := Idx + 1;
       end;
 ReleaseSemaphore(SemaphoreHnd, 1, nil);
```

```
end;

{ UnregisterReceiver removes the record with the specified
  handle from the array. It returns the former index (plus 1)
  of the specified receiver, or -1 if it couldn't be found. }
function UnregisterReceiver(Hnd : THandle) : Integer;
var
 Idx : Integer;
 Found : Boolean;
 RecPtr : PMsgReceiverRec;
begin
 Idx := 0;
 Result := -1;
 RecPtr := ArrayBasePtr;
 Found := False;
 WaitForSingleObject(SemaphoreHnd, INFINITE);

 { Locate the record }
 repeat
  Found := RecPtr^.Assigned and (RecPtr^.WndHandle = Hnd);
  if not Found
   then begin
         Inc(Idx);
         Inc(RecPtr);
        end;
 until Found or (Idx >= MaxReceivers);

 if Found
  then begin
        RecPtr^.Assigned := False;
        Result := Idx + 1;
       end;

 ReleaseSemaphore(SemaphoreHnd, 1, nil);
end;

{ BroadcastToClass uses SendMessage to broadcast the specified
  message to all registered components with the specified class.
  Returns as a result the value from the first message that
  returns a non-zero result. }
function BroadcastToClass(CName : ShortString;
                          MessageID : Word;
                          wParam : Word;
                          lParam : Longint) : Longint;
var
 Idx : Integer;
 RecPtr : PMsgReceiverRec;
begin
 Idx := 0;
 Result := 0;
 RecPtr := ArrayBasePtr;
 WaitForSingleObject(SemaphoreHnd, INFINITE);

 { Send the message to all matching entries }
 repeat
  if RecPtr^.Assigned and (RecPtr^.WndClassName = CName)
   then begin
         if Result = 0
```

```
          then Result :=
            SendMessage(RecPtr^.WndHandle, MessageID, wParam, lParam)
          else SendMessage(RecPtr^.WndHandle, MessageID, wParam, lParam);
        end;

  Inc(Idx);
  Inc(RecPtr);
 until Idx >= MaxReceivers;
 ReleaseSemaphore(SemaphoreHnd, 1, nil);
end;

{ BroadcastToAll uses SendMessage to broadcast the specified
  message to all registered components. Returns as a result
  the value from the first message that returns a non-zero result. }
function BroadcastToAll(MessageID : Word;
                        wParam : Word;
                        lParam : Longint) : Longint;

var
 Idx : Integer;
 RecPtr : PMsgReceiverRec;
begin
 Idx := 0;
 Result := 0;
 RecPtr := ArrayBasePtr;
 WaitForSingleObject(SemaphoreHnd, INFINITE);

 repeat
  if RecPtr^.Assigned
   then begin
         if Result = 0
          then Result :=
            SendMessage(RecPtr^.WndHandle, MessageID, wParam, lParam)
          else SendMessage(RecPtr^.WndHandle, MessageID, wParam, lParam);
        end;

   Inc(Idx);
   Inc(RecPtr);
  until Idx >= MaxReceivers;
 ReleaseSemaphore(SemaphoreHnd, 1, nil);
end;

{ BroadcastToOne uses SendMessage to broadcast the specified
  message to only one specified registered component. Returns as
  a result the value received from the component. }
function BroadcastToOne(MessageID : Word;
                        RcvNum : Integer;
                         wParam : Word;
                         lParam : Longint) : Longint;

var
 RecPtr : PMsgReceiverRec;
begin
 Result := 0;
 RecPtr := ArrayBasePtr;
 if (RcvNum > 0) and (RcvNum <= MaxReceivers)
  then begin
        WaitForSingleObject(SemaphoreHnd, INFINITE);
```

```
        Inc(RecPtr, RcvNum - 1);
        if RecPtr^.Assigned
         then Result :=
          SendMessage(RecPtr^.WndHandle, MessageID, wParam, lParam);

        ReleaseSemaphore(SemaphoreHnd, 1, nil);
       end;
end;

{ NumClassMsgReceivers returns the quantity of registered
  receivers of the specified class. }
function NumClassMsgReceivers(CName : ShortString) : Integer;
var
 Idx : Integer;
 RecPtr : PMsgReceiverRec;
begin
 Idx := 0;
 Result := 0;
 RecPtr := ArrayBasePtr;
 WaitForSingleObject(SemaphoreHnd, INFINITE);
repeat
 if RecPtr^.Assigned and (RecPtr^.WndClassName = CName) then Inc
(Result);
  Inc(Idx);
  Inc(RecPtr);
 until Idx >= MaxReceivers;

 ReleaseSemaphore(SemaphoreHnd, 1, nil);
end;

{ NumMsgReceivers returns the total quantity of registered
  receivers, regardless of class. }
function NumMsgReceivers : Integer;
var
 Idx : Integer;
 RecPtr : PMsgReceiverRec;
begin
 Idx := 0;
 Result := 0;
 RecPtr := ArrayBasePtr;
 WaitForSingleObject(SemaphoreHnd, INFINITE);
 repeat
  if RecPtr^.Assigned then Inc(Result);
  Inc(Idx);
  Inc(RecPtr);
 until Idx >= MaxReceivers;
 ReleaseSemaphore(SemaphoreHnd, 1, nil);
end;

{ First ReceiverAssigned returns the index + 1
  of the first registered receiver in the list.
  If the list is empty, -1 is returned. }
function FirstReceiverAssigned : Integer;
var
 Idx : Integer;
 Found : Boolean;
 RecPtr : PMsgReceiverRec;
```

```
begin
 Idx := 0;
 RecPtr := ArrayBasePtr;
 WaitForSingleObject(SemaphoreHnd, INFINITE);

 repeat
  Found := RecPtr^.Assigned;
  if not Found
   then begin
         Inc(Idx);
         Inc(RecPtr);
        end;
 until Found or (Idx >= MaxReceivers);

 if Found
  then Result := Idx + 1
  else Result := -1;

 ReleaseSemaphore(SemaphoreHnd, 1, nil);
end;

{ NextReceiverAssigned returns the index number + 1
  of the next registered receiver in the list, following
  a specified "receiver number" (i.e., the starting
  index + 1). If there are no more receivers in the
  list, -1 is returned. }
function NextReceiverAssigned(RcvrNum : Integer) : Integer;
var
 Idx : Integer;
 Found : Boolean;
 RecPtr : PMsgReceiverRec;
begin
 if (RcvrNum >= 1) and (RcvrNum < MaxReceivers)
  then begin
        Idx := RcvrNum;
        RecPtr := ArrayBasePtr;
        Inc(RecPtr, Idx);
        WaitForSingleObject(SemaphoreHnd, INFINITE);

        repeat
         Found := RecPtr^.Assigned;
         if not Found
          then begin
                Inc(Idx);
                Inc(RecPtr);
               end;
        until Found or (Idx >= MaxReceivers);

        if Found
         then Result := Idx + 1
         else Result := -1;
        ReleaseSemaphore(SemaphoreHnd, 1, nil);
       end
  else Result := -1;
end;

{ Finalize unmaps and releases the shared memory, then
  releases the semaphore. If there is another instance of
```

```
  this DLL active at the moment, the memory and semaphore
  will not go away. When all other instances of the DLL
  have been closed, the OS will destroy the reserved
  memory and semaphore. }
procedure FinalizeLibrary;
begin
 UnmapViewOfFile(ArrayBasePtr);
 CloseHandle(ArrayHnd);
 CloseHandle(SemaphoreHnd);

 { Restore the exit chain }
 ExitProc := SaveExit;
end;

exports
 BcastDLLValid index 1,
 RegisterReceiver index 2,
 UnregisterReceiver index 3,
 BroadcastToClass index 4,
 BroadcastToAll index 5,
 BroadcastToOne index 6,
 NumClassMsgReceivers index 7,
 NumMsgReceivers index 8,
 FirstReceiverAssigned index 9,
 NextReceiverAssigned index 10;

begin
 { Library initialization code }

 DLLValid := False;

 {Attempt to create a memory-mapped file. If it already exists,
  that's fine. In either case we will get a valid handle. }
 ArrayHnd := CreateFileMapping(HWND($FFFFFFFF),
                               nil,
                               PAGE_READWRITE,
                               0,
                               ArraySize,
                               MemFileName);

 { If we failed, we're done. }
 if (ArrayHnd = 0) then Exit;

 { Map the file by getting a pointer to it }
 LastErr := GetLastError;
 ArrayBasePtr := MapViewOfFile(ArrayHnd, FILE_MAP_WRITE, 0, 0, 0);

 { If we didn't get a valid pointer, we're done. }
 if ArrayBasePtr = nil
  then begin
        CloseHandle(ArrayHnd);
        Exit;
       end;

 { If we actually created the mapped memory, clear it. }
 if LastErr <> ERROR_ALREADY_EXISTS
  then FillChar(ArrayBasePtr^, 0, ArraySize);
```

```
  { Create a semaphore to control access to the mapped memory. If
    the handle comes back zero, we failed for some reason. }
  SemaphoreHnd := CreateSemaphore(nil, 0, 1, SemaphoreName);
  LastErr := GetLastError;
  if (SemaphoreHnd = 0)
   then begin
         CloseHandle(ArrayHnd);
         UnmapViewOfFile(ArrayBasePtr);
         Exit;
        end;

  { Save then exit chain and insert the finalization routine. }
  SaveExit := ExitProc;
  ExitProc := @FinalizeLibrary;

  { If we created the semaphore, then force it to its "signaled" state. }
  if LastErr <> ERROR_ALREADY_EXISTS
   then ReleaseSemaphore(SemaphoreHnd, 1, nil);

   DLLValid := True;
end.
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Startup Code

Perhaps the most interesting part of the DLL is what happens when it starts and ends. It was pointed out by several of the residents that each process which employs a sender or receiver component would undoubtedly load its own copy of the DLL, meaning there could be several DLL instances loaded and running simultaneously. To allow that, several things must be true:

- There must be only one (global) memory area containing the references to which form handles have been registered;
- All instances of the DLL must be capable of cooperatively managing the global memory area;
- There must be a mechanism for a DLL instance to create the shared memory area if there is none, and to destroy the memory area when there is no longer a need for it, and
- Only one DLL at a time can be allowed to change the contents of the memory.

As it turns out, this gnarly little problem was not as bad as it sounded. Win95 provides some good tools not only for creating shared memory, but for managing access to it as well.

The **CreateFileMapping** function returns a handle to a memory file object. If the object doesn't exist, it gets created, and a chunk of the system's paging file—in this case, a chunk of memory big enough to hold the data for 100 receivers—is reserved. The "file" is given a name to ensure it is exclusive to this DLL.

If the memory file already exists, the handle to it is returned, but an **ERROR_ALREADY_EXISTS** error value is generated. By testing for this value, an instance of the DLL can determine if it has created the memory file. If so, it is free to use **FillChar** to clear the entire area—once we know its

location. (Since the memory file can be considered merely as a chunk of memory, that's how it will be treated from here on.)

The location of the memory is mapped to a pointer within the application through a call to **MapViewOfFile**, using the handle returned by **CreateFileMapping**. In this example, the value is assigned to **ArrayBasePtr**. The combination of calling **CreateFileMapping** and **MapViewOfFile** is a lot like making a call to **GetMem**.

Signals from a Semaphore

"Okay, Wunderkind," said the old guy wearing the shopping bag. "How are you gonna keep multiple instances of the DLL from clobbering the shared memory? What if one instance adds something, and then another instance comes in right behind it and adds it again?"

"That's a fair question," I replied. "I think this situation calls for a semaphore."

Win95 provides several synchronization objects, ranging from the trivial to the complex—semaphores, events, and mutexes. But for restricting access to the memory in this situation, the clear choice would be a semaphore. This puppy would enable different processes to cooperatively use the memory through a signal-and-counter system. Like the call to **CreateFileMapping**, the call to **CreateSemaphore** either returns the handle to a newly created object (in this case a semaphore), or the handle to an existing object, while generating an **ERROR_ALREADY_EXISTS** error code. If a new instance of the DLL creates the semaphore, it is given the sole responsibility for setting the semaphore to its signaled state.

By "signaled," it means the semaphore's counter has a non-zero value. Each time a process executes a call to **WaitForSingleObject** using the semaphore's handle, the semaphore's counter is examined. If it is non-zero, the counter is decremented by one and the next program statement in the process is executed. If the counter is already zero, the process goes into an efficient time-wasting loop, waiting for the counter to change to a non-zero value (or a timeout value to be exceeded).

The code in this example permits only two values (0 and 1) for the semaphore counter, effectively turning it into an off/on switch. One of our design goals was that, as long as one instance of a DLL was accessing the shared memory—for any purpose—no other instance could gain access at the same time. Listing 15.4 is an example routine that shows how we could use a semaphore to achieve that goal.

Listing 15.4 Using a semaphore to control memory access

```
procedure DoSomething;
begin
 WaitForSingleObject(SemaphoreHnd, INFINITE);
 DoSomeStuffWithTheMemory;
 ReleaseSemaphore(SemaphoreHnd, 1, nil);
end;
```

In this simple example, we first test the value of the semaphore. If another process is accessing the memory at the moment, the semaphore will be in its non-signaled state (0), and we will wait until forever (as specified by the **INFINITE** timeout constant), if necessary, for the semaphore to change to its signaled state (1).

Once the semaphore is signaled, its counter is automatically decremented by one (to zero), and our process goes ahead and **DoSomeStuffWithTheMemory** executes. While this is going on, any other process that examines the semaphore will find it unsignaled and (if the process has properly used a call to **WaitForSingleObject**) it will bide its time until we're through.

When we're all done with the memory, we place a call to **ReleaseSemaphore**, which will increment the semaphore's counter by the amount we specify. We use a value of one, in effect turning "on" the semaphore "switch."

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Shutdown Code

If we make it all the way through the DLL startup code without error, we set a variable (**DLLValid**) to attest to our success. But if we have succeeded, several handles have been created that must be released when Win95 unloads the library.

This is taken care of by creating **FinalizeLibrary**, an exit routine that will be executed when the DLL is shut down. The procedure is inserted in the program's chain of events by saving the pointer to the normal exit routine (**ExitProc**) in the variable **SaveExit**, and then pointing **ExitProc** to **FinalizeLibrary**.

When the library is unloaded, **FinalizeLibrary** will execute. It will release the map view assigned to **ArrayBasePtr**, and then release the handles to the memory file object and the semaphore. Finally, it will restore the link to the standard exit procedure, so the program will terminate normally.

And what about the memory file object itself? Win95 automatically takes care of that for us. Like many of the system calls that create objects, Win95 associates a counter with the memory file object. Each time a **CreateFile Mapping** call is executed, the system increments a counter for the associated object. Whenever a handle to that object is released, the counter is decremented. When the count reaches zero, Win95 knows the object is no longer being used, so it destroys it. Last one out, turn out the lights.

Examining the DLL Routines

Most of the routines exported by the DLL busy themselves with stepping through the records in the memory array. Since we know the base address of the array, the size of a record, and the maximum number of records, we can easily tiptoe our way through the array using the pointer arithmetic provided

by Object Pascal.

The **NumMsgReceivers** function provides a good example of this technique. A pointer of type **PMsgReceiverRec** is assigned to the base address. Each time a record is examined, the **Inc** procedure is used to advance the pointer by exactly the size of a **TMsgReceiverRec**. Tracking the process with an index variable prevents us from accessing memory outside the bounds of the memory block we have been allocated. The **NumMsgReceivers** function simply counts all the records that have their **Assigned** field set **True**.

The **Assigned** field is managed by two routines, **RegisterReceiver** and **UnregisterReceiver**. **RegisterReceiver** looks for the first unused record and, after clearing it, stores the handle value and class name passed into **RegisterReceiver**. It also sets the record's **Assigned** field **True**. **UnregisterReceiver**, to no great surprise, does pretty much the opposite. It searches the fields with **Assigned** set **True**, looking for a handle value that matches the one specified. If the record is found, it sets the **Assigned** field **False**.

BroadcastToOne, **BroadcastToClass**, and **BroadcastToAll** use the **SendMessage** API function to transmit a message to the handle of a specific form, handles of all active forms of a given class name, or the handles of all active forms in the list. **BroadcastToOne** returns the value returned to it from the call to **SendMessage**; the other two return the first non-zero value returned by a call to **SendMessage**.

FirstReceiverAssigned and **NextReceiverAssigned** together provide a way for a user of the DLL to construct a list of all currently active record numbers in the array. It works a bit like the DOS **FindFirst** and **FindNext** routines. In this case, **FirstReceiverAssigned** returns an index to the first active form in the array. If that index value is then passed to **NextReceiverAssigned**, it will return an index to the next form, starting its search from the specified index. If no active form is found, these routines return a negative one.

Finally, **BcastDLLValid** returns a boolean that indicates whether or not the DLL was able to set up (or connect to) the required shared memory and the semaphore.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Creating the Sender Component

The residents were quietly buzzing among themselves. In a back corner, the murmuring broke out into shouting, as two nearly bald-headed programmers began arguing the various merits of their favorite programming languages at the tops of their lungs. The altercation soon turned to fisticuffs, and it took three staff members to pull the septuagenarians apart.

“Is it always like this around here?” I asked Dinah.

“No,” she replied. “But yesterday one of the guests asked for a cup of Java with creamer, and someone else thought he said the Java language was for dreamers, and clobbered the poor man along side of the head with his standard-issue wrist brace. That one took five stitches.”

I examined the crowd more carefully. Sure enough, nearly everyone was wearing an identical brace.

She had apparently noticed my look of dismay. “Carpal Tunnel Syndrome,” she sighed, with a look full of sorrow. “It gets even the best of them, sooner or later. Researchers are still looking for a cure.”

I nodded in sympathy. The dispute now under control, I decided it would be an opportune time to segue back to developing the program.

With the DLL now under my belt, I was able to move to the next piece of the puzzle. I chose what I would call the MsgSender component. This component would, on command, take a snapshot of the list of registered form handles maintained by the DLL. The component would also be able to call selected DLL routines to broadcast messages and to determine the number of receiver components registered at any given time.

The MsgSender component, I reasoned, would need only one property: the unique message string that would be set to match a corresponding receiver component. The unique message ID, returned by using the message string in a call to **RegisterWindowMessage,** would not be made available outside the component.

Listing 15.5 illustrates the final version of the code for the MsgSender component.

Listing 15.5 Code for the MsgSender component

```
{————————————————————————————————————————}
{         Message Broadcasting in Delphi 2          }
{        MSGSENDR.PAS : MsgSender Component         }
{            By Ace Breakpoint, N.T.P.              }
{             Assisted by Don Taylor                }
{                                                   }
{ This component registers a special system-wide    }
{ message with Win95, and then is able to send      }
{ system-wide messages to other forms in the        }
{ system via MsgReceiver components.                }
```

```
{                                                       }
{ Written for *Kick-Ass Delphi Programming*             }
{ Copyright (c) 1996 The Coriolis Group, Inc.           }
{               Last Updated 5/2/96                     }
{———————————————————————————————————————————————————}

unit MsgSendr;

interface

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

type
  TBcastDLLValid = function : Boolean;
  TRegisterReceiver = procedure(Hnd : THandle; CName : ShortString);
  TUnregisterReceiver = procedure(Hnd : THandle);
  TBroadcastToClass = function(CName : ShortString;
                               MessageID : Word;
                               wParam : Word;
                               lParam : Longint) : Longint;

  TBroadcastToAll = function(MessageID : Word;
                             wParam : Word;
                             lParam : Longint) : Longint;

  TBroadcastToOne = function(MessageID : Word;
                             RcvNum : Integer;
                             wParam : Word;
                             lParam : Longint) : Longint;

  TNumClassMsgReceivers = function(CName : ShortString) : Integer;
  TNumMsgReceivers = function : Integer;
  TFirstReceiverAssigned = function : Integer;
  TNextReceiverAssigned = function(RcvrNum : Integer) : Integer;

  TMsgSender = class(TComponent)
  private
   FIDString : ShortString;
   FMessageID : Word;
   LibraryHandle : THandle;
   DLLValid : Boolean;
   BcastDLLValid : TBcastDLLValid;
   BroadcastToClass : TBroadcastToClass;
   BroadcastToAll : TBroadcastToAll;
   BroadcastToOne : TBroadcastToOne;
   NumClassMsgReceivers : TNumClassMsgReceivers;
   NumMsgReceivers : TNumMsgReceivers;
   FirstReceiverAssigned : TFirstReceiverAssigned;
   NextReceiverAssigned : TNextReceiverAssigned;
   procedure SetIDStr(NewID : ShortString);
   procedure RegisterIDStr;

  public
   ReceiverList : TStringList; { Read only -- do not modify! }
   constructor Create(AOwner : TComponent); override;
   destructor Destroy; override;
   function ClassBroadcast(CName : ShortString;
                           wParam : Word;
```

```
                              lParam : Longint) : Longint;

    function AllBroadcast(wParam : Word; lParam : Longint) : Longint;
    function OneBroadcast
      (RcvNum : Integer; wParam : Word; lParam : LongInt) : Longint;
    function NumClassReceivers(CName : ShortString) : Integer;
    function NumReceivers : Integer;
    procedure UpdateReceiverList;

  published
    property IDString : ShortString read FIDString write SetIDStr;
  end;

procedure Register;

implementation

constructor TMsgSender.Create(AOwner : TComponent);
begin
 DLLValid := False;
 inherited Create(AOwner);
 FMessageID := 0;

 if not (csDesigning in ComponentState)
  then begin
        LibraryHandle := LoadLibrary('BCASTDLL.DLL');
        if LibraryHandle > HINSTANCE_ERROR then
         then begin

              @BroadcastToClass :=
                    GetProcAddress(LibraryHandle, 'BroadcastToClass');
              @BroadcastToAll :=
                    GetProcAddress(LibraryHandle, 'BroadcastToAll');
              @BroadcastToOne :=
                    GetProcAddress(LibraryHandle, 'BroadcastToOne');
              @NumClassMsgReceivers :=
                    GetProcAddress(LibraryHandle, 'NumClassMsgReceivers');
              @NumMsgReceivers :=
                    GetProcAddress(LibraryHandle, 'NumMsgReceivers');
              @FirstReceiverAssigned :=
                    GetProcAddress(LibraryHandle, 'FirstReceiverAssigned');
              @NextReceiverAssigned :=
                    GetProcAddress(LibraryHandle, 'NextReceiverAssigned');
              @BcastDLLValid :=
                    GetProcAddress(LibraryHandle, 'BcastDLLValid');
              DLLValid := BcastDLLValid;
              RegisterIDStr;
             end
        else MessageDlg('Could not load DLL "BCASTDLL.DLL"', mtError,
          [mbOK], 0);
       end;
 ReceiverList := TStringList.Create;
end;

destructor TMsgSender.Destroy;
begin
 ReceiverList.Free;

 if not (csDesigning in ComponentState)
  then if LibraryHandle > HINSTANCE_ERROR then
```

```
    FreeLibrary(LibraryHandle);
 inherited Destroy;
end;

procedure TMsgSender.SetIDStr(NewID : ShortString);
begin
 if NewID <> FIDString
  then begin
       FIDString := NewID;
       RegisterIDStr;
      end;
end;

procedure TMsgSender.RegisterIDStr;
var
 IDStr : Array [0..255] of Char;
begin
 if not (csDesigning in ComponentState) and (Length(FIDString) > 0)
  then begin
       StrPCopy(IDStr, FIDString);
       FMessageID := RegisterWindowMessage(IDStr);
      end
  else FMessageID := 0;
end;

function TMsgSender.ClassBroadcast(CName : ShortString;
                                   wParam : Word;
                                   lParam : Longint) : Longint;
begin
 if DLLValid
  then Result := BroadcastToClass(CName, FMessageID, wParam, lParam)
  else Result := -1;
end;

function TMsgSender.AllBroadcast(wParam : Word;
  lParam : Longint) : Longint;
begin
 if DLLValid
  then Result := BroadcastToAll(FMessageID, wParam, lParam)
  else Result := -1;
end;

function TMsgSender.OneBroadcast(RcvNum : Integer;
                                 wParam : Word;
                                 lParam : Longint) : Longint;
begin
 if DLLValid
  then Result := BroadcastToOne(FMessageID, RcvNum, wParam, lParam)
  else Result := -1;
end;

function TMsgSender.NumClassReceivers(CName : ShortString) : Integer;
begin
 if DLLValid
  then Result := NumClassMsgReceivers(CName)
  else Result := -1;
end;

function TMsgSender.NumReceivers : Integer;
begin
```

```
 if DLLValid
  then Result := NumMsgReceivers
  else Result := -1;
end;

procedure TMsgSender.UpdateReceiverList;
var
 RcvrNum : Integer;
begin
 ReceiverList.Clear;
 if DLLValid
  then begin
        RcvrNum := FirstReceiverAssigned;
        if RcvrNum > 0
         then begin
               ReceiverList.Add(IntToStr(RcvrNum));
               repeat
                RcvrNum := NextReceiverAssigned(RcvrNum);
                if RcvrNum > 0 then ReceiverList.Add(IntToStr(RcvrNum));
               until RcvrNum < 0;
              end;
       end;
end;

procedure Register;
begin
  RegisterComponents('Ace''s Stuff', [TMsgSender]);
end;

end.
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

For the most part, this component was a simple interface between the form that would own the component and routines in the DLL. **DLLValid** is a boolean, set to indicate the result of attempting to load the DLL. **DLLValid** is checked by methods that call DLL routines. The DLL itself is not loaded if Delphi is in the design state.

Broadcast functions were implemented using the corresponding DLL routines, returning a value of minus one if they failed. **UpdateReceiverList** uses the DLL's **FirstReceiverAssigned** and **NextReceiverAssigned** functions to prepare a fresh snapshot of the state of the registered receiver list. I decided to give the MsgSender's owner the sole responsibility for calling **UpdateReceiverList**.

For a finishing touch, I used Delphi's Image Editor to create a resource file containing a bitmap for the face of the component. Although the process was easy, it had to be done exactly right or it wouldn't work. So I'll document the process here.

After opening the Image Editor, I selected the New option from the File menu. From the choices, I selected "Component Resource File (.dcr)". I then selected the New option from the Resource menu, and selected "Bitmap". I specified a bitmap size of 24 by 24 pixels, and VGA (16 color) resolution.

Before creating the artwork itself, I saved the resource file with the still-blank bitmap. This is the most critical part of the process. First, I clicked on the default name given to the bitmap resource ("Bitmap1") and changed the name to "TMSGSENDER". Although the tradition is to use all caps, it isn't necessary. But the name must *exactly* match the type identifier given to the component. Next, I saved the file, with the name "MsgSendr.dcr". Once again, this is critical: The file name must exactly match the .dcu file (which is the same as the .pas file) that contains the code for the component. Figure 15.2 shows the state of the Image Editor once I had saved the file. From that point, I was able to create my beautiful piece of artwork and save it.

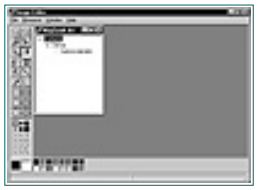

FIGURE 15.2 The Delphi Image Editor and the blank bitmap.

Creating the Receiver Component

The pizza now entirely consumed, the natives once again became restless. I was about to suggest an intermission when I looked at my watch in horror: It still read 1:30! Absolutely no time had passed!

I made a mental note to get the watch to the local jeweler. When I glanced up, I saw that one of the residents had his hand up. *Finally*, I thought. *Decorum is about to break out.*

“Excuse me,” the man said.

I nodded for him to proceed with his question.

“Who are you? And what are you doing here?” he asked. He looked around at the other programmers and continued. “And where is Arnold? Isn’t Arnold supposed to be here?”

A titter drifted through the audience, and some jerk in the third row holding a bucket of extra greasy chicken began to guffaw. Totally confused, I looked to Dinah for a clue.

“Who’s Arnold?” I asked her.

“One of the guests,” she replied. “An inveterate troublemaker. He’s been put in Detention for violating the Internet curfew,” she said.

I pointed with my thumb to the man who had raised his hand. “And what about this guy?” I asked, trying hard not to move my mouth.

She didn’t make a sound, but I could read her lips. “That’s Bernie,” she said. “*Error 1: Out of memory.*”

Whoever said that truth was stranger than fiction certainly hadn’t experienced this place. I sighed and began the task of creating the receiver component.

TMsgReceiver turned out to be more of a challenge than **TMsgSender**. I should have expected that, since I knew going in I would have to subclass the window of the component’s owner.

The MsgReceiver component would have to perform only four basic tasks:

1. Register itself with the DLL when it is created, if the application is executing;
2. Register the unique message string with Windows to obtain the unique message ID;
3. Unregister itself with the DLL when the component is destroyed; and

4. Call a specified handler procedure when the component receives the unique message ID.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

The final version of **TMsgReceiver**'s code is shown in Listing 15.6.

Listing 15.6 Code for the MsgReceiver component

```
{──────────────────────────────────────────────────────}
{         Message Broadcasting in Delphi 2             }
{        MSGRCVR.PAS : MsgReceiver Component           }
{            By Ace Breakpoint, N.T.P.                 }
{             Assisted by Don Taylor                   }
{                                                      }
{ This component effectively subclasses the form       }
{ it is placed on, replacing the form's WndProc        }
{ and registering a special system-wide message        }
{ with Win95. Only one of these components is          }
{ allowed per form.                                    }
{                                                      }
{ Written for *Kick-Ass Delphi Programming*            }
{ Copyright (c) 1996 The Coriolis Group, Inc.          }
{               Last Updated 5/2/96                    }
{──────────────────────────────────────────────────────}

unit MsgRcvr;

interface

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

type
 EOwnerNotWinControl = class(Exception);
 EMultipleMsgReceivers = class(Exception);

 TBcastDLLValid = function : Boolean;
 TRegisterReceiver = function(Hnd : THandle;
   CName : ShortString) : Integer;
 TUnregisterReceiver = function(Hnd : THandle) : Integer;

 TMsgReceiverEvent = function(wParam : Word;
   lParam : Longint) : Longint of object;
```

```
 TMsgReceiver = class(TComponent)
  private
   FIDString : ShortString;
   FMessageID : Word;
   FReceiverNum : Integer;
   LibraryHandle : THandle;
   DLLValid : Boolean;
   OriginalWndProc : TFarProc;
   NewWndProc : TFarProc;
   WndProcHooked : Boolean;
   OwnerHandle : THandle;
   BcastDLLValid : TBcastDLLValid;
   RegisterReceiver : TRegisterReceiver;
   UnregisterReceiver : TUnregisterReceiver;
   procedure SetIDStr(NewID : ShortString);
   procedure RegisterIDStr;

  protected
   FOnIDMessage : TMsgReceiverEvent;
   procedure WndProc(var Msg : TMessage); virtual;
   procedure DefaultHandler(var Msg); override;
   procedure HandleWMDestroy(var Msg : TWMDestroy); message WM_Destroy;

  public
   constructor Create(AOwner : TComponent); override;
   destructor Destroy; override;

  published
   property IDString : ShortString read FIDString write SetIDStr;
   property ReceiverNum : Integer read FReceiverNum;
   property OnIDMessage : TMsgReceiverEvent read FOnIDMessage
     write FOnIDMessage;
  end;

procedure Register;

implementation

constructor TMsgReceiver.Create(AOwner : TComponent);
var
 i : Integer;
begin
 DLLValid := False;
 FReceiverNum := -1;

 if not (AOwner is TWinControl)
  then raise
    EOwnerNotWinControl.Create('Owner must be TWinControl or descendant');

 for i := 0 to AOwner.ComponentCount - 1 do
  if AOwner.Components[i] is TMsgReceiver
   then raise
     EMultipleMsgReceivers.Create('Only one TMsgReceiver allowed per
     form');

 inherited Create(AOwner);
 OwnerHandle := (AOwner as TWinControl).Handle;
 NewWndProc := MakeObjectInstance(WndProc);
```

```
 FMessageID := 0;

 if not (csDesigning in ComponentState)
  then begin
        OriginalWndProc :=
         TFarProc(GetWindowLong((AOwner as TWinControl).Handle,
             gwl_WndProc));
        SetWindowLong((AOwner as TWinControl).Handle, gwl_WndProc,
             Longint(NewWndProc));
        WndProcHooked := True;

        LibraryHandle := LoadLibrary('BCASTDLL.DLL');
        if LibraryHandle > HINSTANCE_ERROR
         then begin
               @RegisterReceiver :=
                 GetProcAddress(LibraryHandle, 'RegisterReceiver');
               @UnregisterReceiver :=
                 GetProcAddress(LibraryHandle, 'UnregisterReceiver');
               @BcastDLLValid :=
                 GetProcAddress(LibraryHandle, 'BcastDLLValid');
               DLLValid := BcastDLLValid;
               RegisterIDStr;
              end
         else MessageDlg('Could not load DLL "BCASTDLL.DLL"', mtError,
           [mbOK], 0);
       end;
 if DLLValid then FReceiverNum := RegisterReceiver(OwnerHandle,
   AOwner.ClassName);
end;

destructor TMsgReceiver.Destroy;
begin
 if WndProcHooked
  then SetWindowLong((Owner as TWinControl).Handle, gwl_WndProc,
            Longint(OriginalWndProc));

 FreeObjectInstance(NewWndProc);
 if DLLValid then UnregisterReceiver(OwnerHandle);

 if not (csDesigning in ComponentState)
  then if LibraryHandle > HINSTANCE_ERROR then
    FreeLibrary(LibraryHandle);
 Inherited Destroy;
end;

procedure TMsgReceiver.HandleWMDestroy(var Msg : TWMDestroy);
begin
 SetWindowLong((Owner as TWinControl).Handle, gwl_WndProc,
   Longint(OriginalWndProc));
 WndProcHooked := False;
end;

procedure TMsgReceiver.WndProc(var Msg : TMessage);
begin
 if Msg.Msg = FMessageID
  then begin
        if Assigned(FOnIDMessage)
         then Msg.Result := FOnIDMessage(Msg.wParam, Msg.lParam);
       end
```

```
  else Dispatch(Msg);
end;

procedure TMsgReceiver.DefaultHandler(var Msg);
begin
 with TMessage(Msg) do
  Result := CallWindowProc(OriginalWndProc,
    (Owner as TWinControl).Handle,
              Msg, wParam, lParam);
end;

procedure TMsgReceiver.SetIDStr(NewID : ShortString);
begin
 if NewID <> FIDString
  then begin
        FIDString := NewID;
        RegisterIDStr;
       end;
end;

procedure TMsgReceiver.RegisterIDStr;
var
 IDStr : Array [0..255] of Char;
begin
 if not (csDesigning in ComponentState) and (Length(FIDString) > 0)
  then begin
        StrPCopy(IDStr, FIDString);
        FMessageID := RegisterWindowMessage(IDStr);
       end
  else FMessageID := 0;
end;

procedure Register;
begin
 RegisterComponents('Ace''s Stuff', [TMsgReceiver]);
end;

end.
```

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Subclassing the Owner Window

All Delphi objects (that is, those based on **TObject** and hence part of the VCL) have inherent messaging capability. Figure 15.3(a) shows the event processing chain for a **TObject**, with its built-in **Dispatch** and **DefaultHandler** routines.

The job of the **Dispatch** routine is to examine any event reaching it, first checking to see if the event's message ID matches those of any special event handlers. These special handlers are the ones added to the **TObject** descendant by the programmer, and are of the form:

```
procedure HandleMyEvent (var Msg : TMessage);
  message MyEven tMessageID;
```

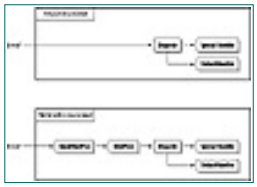

FIGURE 15.3 The event handling process for TObject and TWinControl types.

If a match is found, the special handler corresponding to the event's message ID is called. Otherwise, the built-in **DefaultHandler** routine gets called. Unless modified, the standard **DefaultHandler** does nothing.

Standard event processing for a **TWinControl** descendant is slightly more sophisticated, as can be seen in Figure 15.3(b). This time there are two more methods involved, **MainWndProc** and **WndProc**. **MainWndProc** is the main window procedure for each type of component in an application. When a message is received by any **TWinControl** descendant, it is first checked by

MainWndProc to see if the message relates to an exception. If not, the message is passed to the virtual method **WndProc**.

Classically, a window's **WndProc** routine is sort of "Message Central." It is within this routine that certain messages can be trapped or handled in a special way. The **WndProc** method for a **TWinControl** descendant, for example, passes keyboard events to **Dispatch** only if the component isn't being dragged.

Under normal circumstances, Delphi provides a great deal of flexibility for handling special events. The ability to create special handlers for any **TObject** descendant as described above is a simple and powerful technique, as long as you know the message IDs of the events you want to handle at compile time.

But what we want here cannot be done that simply, for two reasons. First, we won't know the message ID we're looking for until the program is executing and has called **RegisterWindowMessage**. And second, the component must receive the event first, before its owner form sees it. In other words, we must break into the message chain of the owner, handle anything corresponding to the special message ID, and then pass everything else back to the owner to process in the normal manner.

Figure 15.4 illustrates the process. Here we have broken the chain in the form, so **MainWndProc** passes events to a new **WndProc** method written for the MsgReceiver component. The MsgReceiver's **WndProc** is aware of the special message ID and the procedure assigned to handle that ID.

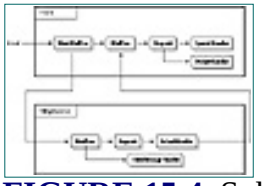

FIGURE 15.4 Subclassing the TForm's window with TMsgReceiver.

If the ID of the event matches the special message ID, the assigned **OnIDMessage** handler gets called. Otherwise, MsgReceiver's **Dispatch** method is called. Since there are no special handlers defined for MsgReceiver, the event is passed on to **DefaultHandler**. **DefaultHandler** has been modified to simply send the event back to the form's original **WndProc**, where processing resumes normally. That means any special event handlers written for the form will execute as if nothing had been changed.

This process of breaking in to a window's message chain and inserting an additional **WndProc** is called *subclassing*. It's a standard technique used all the time in Windows programming. There is just less need for it when programming with Delphi.

In this example, the subclassing is performed dynamically in the MsgReceiver's constructor. The call to **MakeObjectInstance** creates **NewWndProc**, a pointer to a procedure of the form

```
procedure AWndProc(var Msg : TMessage),
```

which is a prototype for a method used to handle messages sent to window procedures. If the application is executing (i.e., not in design mode), three

statements are then performed to complete the subclassing process. First, a pointer to the owner's original **WndProc** is stored in **OriginalWndProc**. Next, **SetWindowLong** is used to tell Windows that the MsgReceiver's **WndProc** (pointed to by **NewWndProc**) is the new place to send messages to the owner. Finally, a boolean (**WndProcHooked**) is set to indicate the subclassing process has succeeded. If the DLL loads correctly, the MsgReceiver's **RegisterIDStr** method will be called, which will in turn call **RegisterWindowMessage** and set the **FMessageID** field.

The MsgReceiver's **WndProc** checks the incoming event to see if it matches **FMessageID**. If so, it attempts to call the handler specified in the **OnIDMessage** property. If the event doesn't match **FMessageID**, **WndProc** calls the MsgReceiver's **Dispatch** method. Since there are no special message handlers defined for the MsgReceiver, the event is automatically passed to the MsgReceiver's **DefaultHandler** method, where we simply use **CallWindowProc** to pass the message to the owner's original **WndProc**. We have accomplished exactly what is illustrated in Figure 15.4.

What has all this gained us? We've set up a pre-processing system for the form containing the MsgReceiver. Instead of having to know exactly what we're looking for (the message ID) and what we're going to do when we get it (the handler) at compile time, we have shifted the whole decision-making process to execution time. (In fact, we could even dynamically switch both the message ID string and the procedure chosen to handle it at any time during the program's execution. A frightening concept.)

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Other Interesting Stuff

Just as there is a way to subclass a window, there must be a way to restore the system when the program terminates. This feat is accomplished in a couple of places. If the component receives a **WM_Destroy** message, it must immediately restore the original messaging, before anything else takes place; otherwise, there will be no valid window by the time execution gets to the MsgReceiver's **Destroy** destructor—an ugly situation. The **Handle WMDestroy** method takes care of this, setting the **WndProcHooked** variable **False** to indicate that subclassing has been eliminated.

The **Destroy** destructor accomplishes the same task, assuming subclassing is still in place when it is called. Once subclassing has been eliminated, the destructor can safely free the object instance for **NewWndProc**. Before calling the inherited destructor, though, the component is unregistered and the DLL is unloaded.

Two other interesting things happen within MsgReceiver's **OnCreate** handler. First, before creating the component we must make sure the potential owner is a qualified, card-carrying member of the **TWinControl** family. If it isn't, we won't have a **MainWndProc** or **WndProc** to work with, and the results won't be pretty. If the owner is not a **TWinControl** or descendant, an exception is raised before the inherited **Create** constructor can be called.

Second, we only want to allow one MsgReceiver component per form. Can you imagine keeping track of *several levels* of subclassing? Only if you're a masochist. After the **TWinControl** test, a check is made for prior instances of a **TMsgReceiver** type in the owner's list of components. If a prior is found, an exception is raised, and we're outta here!

One last thing: MsgReceiver contains a read-only property called **Receiver Num**, the value returned when the component is registered with the DLL.

Using the same overall procedure as I had used for the MsgSender, I created a bitmap for the **TMsgReceiver** object. Done at last—well, almost.

Creating a Receiver Demo

"You're really doing well," Dinah said. "I haven't seen our guests this interested in a long time. Even Mr. Bohacker wasn't able to command such attention."

I suppose I needed the encouragement. But I realized that this time I *really was* doing well. I couldn't believe how well the code was coming together. And to be able to pull this off under such extreme pressure made it all the sweeter.

And I really was commanding respect. Most of the programmers were intently watching my every move, hanging on every syllable leaving my mouth. Of course, there were a few exceptions. Probably 20 or 25 old fogies were napping at any given time. The three women sitting in the last row who I thought were knitting were actually surfing the Web through a wireless link to their laptop PCs. But there was one guy that *really* bugged me.

"Psssst. Psssssssst." I leaned away from the microphone, attempting to attract Dinah's attention. She finally looked up from the copy of *Weekly World News* she was perusing.

"What's with the guy in the leftmost seat in the front row?" I whispered. "He's been sitting there this whole time, just staring off into space. I haven't even seen him blink."

"That's Herbie," she whispered back. "He's a little, uh, you know… We just say he's in Permanent Screen Saver mode."

I pulled the candy bar out of my pocket and tore off the wrapper. I still had two applications to write, and I was in need of some extra energy. I bit off the end of the bar and nearly gagged. The Foo Bar was the candy confection universally chosen by programmers worldwide. Its delicate balance of fat calories, sugar, and caffeine made it the perfect productivity snack food. Today I had grabbed a Foo Bar++ from the machine, thinking it would have even more of the ingredients my body needed to make it through this endless afternoon. It wasn't until I had snarfed a big bite that I realized the ingredient that had been added for extra boost was ground dried prunes. I took a gulp of water from the glass perched on the lectern and then shoved the remainder of the bar back in my pocket.

With the component design complete, I needed to create a demo application for each component—two programs that would highlight most of the features of the components.

From the several suggestions pitched at me by the audience, I chose a scenario where a sender application would send predetermined codes to a receiver, causing it to set a series of indicators. We decided to emulate the little lights on some automobile odometers that indicate when you should change your oil or filter, or rotate your tires. The receiver demo app would illuminate red or green indicators to indicate the condition, as transmitted by a sender app. In fact, we

could have multiple instances of both sender and receiver apps if we wished; the methods we had chosen for communication should support that concept.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

I decided to write the receiver demo first, because it seemed the easier of the two. In fact, it was pretty simple. The first thing was to come up with a set of command constants to use for private message IDs. I developed the Constants unit shown in Listing 15.7.

Listing 15.7 Constants for the Sender and Receiver demo applications

```
{──────────────────────────────────────────────────────────}
{         Message Broadcasting in Delphi 2                 }
{           CONSTANT.PAS : Constants Unit                  }
{            By Ace Breakpoint, N.T.P.                     }
{             Assisted by Don Taylor                       }
{                                                          }
{ Constants for use by the Sender and Receiver             }
{ applications.                                            }
{                                                          }
{ Written for *Kick-Ass Delphi Programming*                }
{ Copyright (c) 1996 The Coriolis Group, Inc.              }
{               Last Updated 5/2/96                        }
{──────────────────────────────────────────────────────────}

unit Constant;

interface

const
 { Command codes }
 mdSetAll = 100;
 mdSetChangeOil = 101;
 mdSetChangeFilter = 102;
 mdSetRotateTires = 103;

 mdResetAll = 200;
 mdResetChangeOil = 201;
 mdResetChangeFilter = 202;
 mdResetRotateTires = 203;
```

```
 { Status codes }
 mdNoReply = 0;
 mdReady = 1;
 mdSuccess = 2;
 mdRcvChange = 3;

implementation

end.
```

Creating the Receiver demo was a breeze. A picture of the design version, next to the Object Inspector, is shown in Figure 15.5. The code is shown in Listing 15.8.

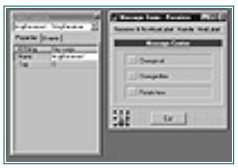

FIGURE 15.5 The Receiver demo at design time.

Listing 15.8 Code for the Receiver demo

```
{──────────────────────────────────────────────────────────}
{         Message Broadcasting in Delphi 2                 }
{        RECVMAIN.PAS : Receiver Main Form                 }
{            By Ace Breakpoint, N.T.P.                     }
{             Assisted by Don Taylor                       }
{                                                          }
{ This demo application receives commands from             }
{ the Sender application, through the use of the           }
{ MsgSender and MsgReceiver components.                    }
{                                                          }
{ Written for *Kick-Ass Delphi Programming*                }
{ Copyright (c) 1996 The Coriolis Group, Inc.              }
{               Last Updated 5/2/96                        }
{──────────────────────────────────────────────────────────}

unit RecvMain;

interface

uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
      Dialogs,
   StdCtrls, ExtCtrls, MsgRcvr, Constant;

type
  TReceiverForm = class(TForm)
    ExitBtn: TButton;
    MsgReceiver1: TMsgReceiver;
    Panel2: TPanel;
    HndLabel: TLabel;
    Label2: TLabel;
    RcvNumLabel: TLabel;
    Label1: TLabel;
    Panel1: TPanel;
```

```
    Panel3: TPanel;
    Bevel1: TBevel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Bevel2: TBevel;
    ChangeOilInd: TPanel;
    Bevel3: TBevel;
    ChangeFiltInd: TPanel;
    Bevel4: TBevel;
    RotateInd: TPanel;
    procedure ExitBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    function MsgReceiver1IDMessage(wParam: Word;
      lParam: Longint): Longint;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  ReceiverForm: TReceiverForm;

implementation

{$R *.DFM}

procedure TReceiverForm.ExitBtnClick(Sender: TObject);
begin
 Close;
end;

procedure TReceiverForm.FormCreate(Sender: TObject);
begin
 HndLabel.Caption := IntToStr(Self.Handle);
 RcvNumLabel.Caption := IntToStr(MsgReceiver1.ReceiverNum);
 ChangeOilInd.Color := clYellow;
 ChangeFiltInd.Color := clYellow;
 RotateInd.Color := clYellow;
end;

function TReceiverForm.MsgReceiver1IDMessage(wParam: Word;
  lParam: Longint): Longint;
begin
 case wParam of
  mdSetChangeOil : ChangeOilInd.Color := clRed;
  mdSetChangeFilter : ChangeFiltInd.Color := clRed;
  mdSetRotateTires : RotateInd.Color := clRed;
  mdResetChangeOil : ChangeOilInd.Color := clLime;
  mdResetChangeFilter : ChangeFiltInd.Color := clLime;
  mdResetRotateTires : RotateInd.Color := clLime;
 end; { case }

 Result := mdSuccess;
```

```
end;

end.
```

The centerpiece of the Receiver demo is the **MsgReceiverIDMessage** method. This is the handler specified in the **OnIDMessage** property of the MsgReceiver component. **MsgReceiverIDMessage** tests the **wParam** portion of the message to set the colors of the three indicators. It then returns a message result of **mdSuccess**, which will be returned to the MsgSender component issuing the command.

I could have simply defined a string constant for the **IDString** property of the MsgSender and MsgReceiver components in my Constants unit, and then assigned the string to the properties of each of the components at runtime. I guess after all this mysterious mumbo-jumbo, I was overcome by a fit of conformity to the standard way of doing things.

To demonstrate the ability of a MsgReceiver to get the index number assigned by the DLL, I displayed **ReceiverNum** in the top panel of the Receiver form. I also displayed the form's handle value, the same value that would be registered by the DLL.

Previous | Table of Contents | Next

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Creating a Sender Demo

“That receiver demo-thingy isn’t such a big deal,” said the old man with the bucket of chicken, as he took a big bite out of a thigh. “I could do just as well—even on one of my ‘bad’ days.”

“Me, too,” said the bald and toothless old geezer sitting next to him, as he wiped off the grease running down his friend’s chin with a paper towel.

“Any of us could, in the Old Days,” someone agreed.

“Yeah, we didn’t have it so easy back then,” came another voice. “Didn’t have all these new-fangled development tools. Had to walk ten miles to school in the snow. And the only debugger we had was a can of Raid.”

“You think that’s bad?” someone else chimed in. “We had to walk 50 feet across ice-cold linoleum, just to get from our cubicles to the vending machines. And we had to write everything in assembly code.”

I looked around the room. It was getting more tense by the moment.

“You’re all a bunch of dweebs,” said a heavy-set man in the fifth row, as he stood up and faced the crowd. “You all go around humming the background tunes from Doom episodes. Where I was working, we didn’t even have assemblers. We had to write everything with ones and zeros. There were times we ran out of ones, and we had to make do.”

The scene probably would have turned into a full-scale riot, except for a fortuitous event. Suddenly, the door at the back of the room flew open and hit the wall so hard it sounded like a small explosion. Everyone instantly stopped arguing and turned to the door to see who had made the noise, and as they did I could hear several people suck in their breath. Framed in the doorway was a tall, slender man with a grizzled face and a toothpick protruding from between his lips. His full-length coveralls were sky-blue, an insignia adorning the left breast pocket, and he wore a leather belt with a spray bottle hanging low from each side. In his right hand he held a rubber squeegee.

“Anybody seen Eddie today?” he growled, the toothpick following every movement of his lower lip. He stood there for several seconds, pounding the business end of the squeegee against the palm of his left hand. Receiving no answer to his question, he eventually grunted something indiscernible and then ambled on down the hallway.

I shot Dinah a questioning glance, and she responded without my having to ask.

“That was Brad, our Chief of Janitors,” she explained. “He’s looking for Eddie Rivers, one of our, well, problem guests.”

“A man with a problem? Doesn’t sound at all unusual. Everybody here seems to have some problem or another.”

“Not Eddie. His is a *recurring* problem. *Error 212: Stream registration error*. It happened three times, yesterday alone.”

I turned to face the audience. The crowd had once again quieted down, and it was now time for the final piece of this whole puzzle—a demo that would transmit commands to the Receiver demo. But I wanted the Sender demo to be much more. After all, I was up against Bohacker again, and my reputation was on the line. I had to perform Big Time.

I decided to pop a timer on the form, to force a periodic query of what the DLL had registered. By monitoring that information, I could pull off some pretty cute stuff. A snapshot of the design version of the form is shown in Figure 4.6. The final version of the code is detailed in Listing 15.9.

FIGURE 15.6 The Sender demo at design time.

Listing 15.9 Code for the Sender demo

```
{————————————————————————————————————————}
{         Message Broadcasting in Delphi 2           }
{         SENDMAIN.PAS : Sender Main Form            }
{            By Ace Breakpoint, N.T.P.               }
{             Assisted by Don Taylor                 }
{                                                    }
{ This demo application sends commands to the        }
{ Receiver application, through the use of the       }
{ MsgSender and MsgReceiver components.              }
{                                                    }
{ Written for *Kick-Ass Delphi Programming*          }
{ Copyright (c) 1996 The Coriolis Group, Inc.        }
{              Last Updated 3/31/96                  }
{————————————————————————————————————————}

unit SendMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, MsgSendr, ExtCtrls, Constant;

type
  TSenderMainForm = class(TForm)
    SendBtn: TButton;
    ExitBtn: TButton;
    MsgSender1: TMsgSender;
    RefreshTimer: TTimer;
    RecipientGroup: TGroupBox;
    AllRB: TRadioButton;
    OneRB: TRadioButton;
```

```
    OneCombo: TComboBox;
    Panel1: TPanel;
    Label1: TLabel;
    NumRcvrsLabel: TLabel;
    Bevel1: TBevel;
    Bevel2: TBevel;
    Label2: TLabel;
    StatusLabel: TLabel;
    StatusTimer: TTimer;
    MsgGroup: TGroupBox;
    ChangeOilCB: TCheckBox;
    ChangeFiltCB: TCheckBox;
    RotateCB: TCheckBox;
    procedure ExitBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure RefreshTimerTimer(Sender: TObject);
    procedure StatusTimerTimer(Sender: TObject);
    procedure SendBtnClick(Sender: TObject);
    procedure AllRBClick(Sender: TObject);
    procedure OneRBClick(Sender: TObject);
  private
    NumReceivers : Integer;
    procedure RefreshRecipientGroup;
    procedure ShowStatus(MsgNum : Integer);
  public
    { Public declarations }
  end;

var
  SenderMainForm: TSenderMainForm;

implementation

{$R *.DFM}

procedure TSenderMainForm.RefreshRecipientGroup;
var
 CurrReceivers : Integer;
 i : Integer;

begin
 CurrReceivers := MsgSender1.NumReceivers;
 if NumReceivers <> CurrReceivers
  then begin
        NumReceivers := CurrReceivers;
        NumRcvrsLabel.Caption := IntToStr(NumReceivers);
        if NumReceivers > 0
         then begin
               OneRB.Enabled := True;
               OneCombo.Items.Clear;
               MsgSender1.UpdateReceiverList;
               for i := 0 to MsgSender1.ReceiverList.Count - 1 do
                OneCombo.Items.Add(MsgSender1.ReceiverList[i]);
               OneCombo.ItemIndex := 0;
               OneCombo.Enabled := OneRB.Checked;
              end
```

```
          else begin
                AllRB.Checked := True;
                OneRB.Enabled := False;
                OneCombo.Text := '';
                OneCombo.Enabled := False;
               end;
         ShowStatus(mdRcvChange);
        end;

end;

procedure TSenderMainForm.ShowStatus(MsgNum : Integer);
var
 Msg : ShortString;
begin
 case MsgNum of
  mdReady : Msg := 'Ready';
  mdSuccess : Msg := 'Complete';
  mdNoReply : Msg := 'No reply';
  mdRcvChange : Msg := 'Number of receivers changed';
 else
   Msg := 'Unknown status response';
 end; { case }
 StatusLabel.Caption := Msg;
 StatusTimer.Enabled := True;
end;

procedure TSenderMainForm.ExitBtnClick(Sender: TObject);
begin
 Close;
end;

procedure TSenderMainForm.FormCreate(Sender: TObject);
begin
 NumRcvrsLabel.Caption := IntToStr(MsgSender1.NumReceivers);
 AllRB.Checked := True;
 OneRB.Enabled := False;
 OneCombo.Enabled := False;
 ShowStatus(mdReady);
end;

procedure TSenderMainForm.RefreshTimerTimer(Sender: TObject);
begin
 RefreshRecipientGroup;
end;

procedure TSenderMainForm.StatusTimerTimer(Sender: TObject);
begin
 StatusTimer.Enabled := False;
 StatusLabel.Caption := '';
end;

procedure TSenderMainForm.SendBtnClick(Sender: TObject);
var
 Status : Integer;
 RcvrNum : Integer;
```

```
begin
 with MsgSender1 do
 if AllRB.Checked
  then begin
        if ChangeOilCB.Checked
         then AllBroadcast(mdSetChangeOil, 0)
         else AllBroadcast(mdResetChangeOil, 0);

        if ChangeFiltCB.Checked
         then AllBroadcast(mdSetChangeFilter, 0)
         else AllBroadcast(mdResetChangeFilter, 0);

        if RotateCB.Checked
         then Status := AllBroadcast(mdSetRotateTires, 0)
         else Status := AllBroadcast(mdResetRotateTires, 0);

         ShowStatus(Status);
       end
  else begin
        RcvrNum := StrToInt(OneCombo.Items[OneCombo.ItemIndex]);
        if ChangeOilCB.Checked
         then OneBroadcast(RcvrNum, mdSetChangeOil, 0)
         else OneBroadcast(RcvrNum, mdResetChangeOil, 0);

        if ChangeFiltCB.Checked
         then OneBroadcast(RcvrNum, mdSetChangeFilter, 0)
         else OneBroadcast(RcvrNum, mdResetChangeFilter, 0);

        if RotateCB.Checked
         then Status := OneBroadcast(RcvrNum, mdSetRotateTires, 0)
         else Status := OneBroadcast(RcvrNum, mdResetRotateTires, 0);

         ShowStatus(Status);
       end;

 ChangeOilCB.Checked := False;
 ChangeFiltCB.Checked := False;
 RotateCB.Checked := False;
end;

procedure TSenderMainForm.AllRBClick(Sender: TObject);
begin
 OneCombo.Enabled := False;
end;

procedure TSenderMainForm.OneRBClick(Sender: TObject);
begin
 OneCombo.Enabled := True;
end;

end.
```

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Each time **RefreshTimer** fires, it calls the **RefreshRecipientGroup** method. **RefreshRecipientGroup** determines whether the number of registered receivers has changed since the last inquiry. If so, a status indicator is updated. If there is at least one receiver, the list of items in the combo box is updated with the indexes of the receivers. If there are no receivers registered, the combo box is disabled.

The Send button's OnClick handler is the busiest routine on this form. If commands are to be sent to all registered receivers, **SendBtnClick** simply composes three messages—one for each check box—and uses the MsgSender's **AllBroadcast** method to send the messages on their way. If the commands are to be sent to a single receiver, **SendBtnClick** composes the messages, but sends them via the **OneBroadcast** method, computing the receiver's index number from the combo box's item index.

A **ShowStatus** method is provided to display the results of some of the operations, including a change in the number of receivers, and the response to a transmitted command. **StatusTimer** automatically blanks the status display after three seconds.

I fired off the Sender app, then brought up the Receiver app. They worked famously, as shown in Figure 15.7. At the urging of the crowd, I brought up more instances. At one point, I had two Senders and four Receivers on the screen, all working perfectly. Using the "One receiver" radio button and combo box, I could send individual commands to any of the receivers.

FIGURE 15.7 Sender and Receiver at runtime.

A Rude Awakening

I had met the challenge of the day, and although it had been the longest afternoon of my life, I had performed well. I felt good.

“And in conclusion,” I said to the assembled masses, “there isn’t any problem that a combination of persistence and determination can’t overcome. Thank you, ladies and gentlemen, for being here. You’ve been a wonderful audience.”

I stood back from the microphone, ready to accept the inevitable applause. I had no idea what was about to happen.

“Sounds like ‘Spam’ to me,” shouted the greaseball with the chicken in his lap.

“Yea—let’s flame him!” yelled his toothless companion with the billiard ball head.

It was a matter of unfortunate timing that brought four staff members into the room at that moment, prepared to distribute the golden, finger-like sponge cakes bursting with creme filling intended for the residents’ mid-afternoon snack. An instant later those palatable pastries had turned into weapons of wrath, hundreds of them pelting my person.

Out of the corner of my eye, I saw the toothless old geezer heading in my direction, climbing up onto the stage, making his way right up to me, and finally, shoving his wrinkled face so close to mine that our noses touched. Whoever said that bad breath is better than no breath at all was overly optimistic. This guy was, for me, the ultimate argument against growing old.

“Whaddya want with me?” I asked him.

“I just want you to know what it’s gonna be like for you, if you don’t take better care of yourself,” he said, looking me straight in the eye. “Count on it.”

I stared back at his weak brown eyes through lenses as thick as soda bottles. There was something familiar about those eyes. Then it struck me.

“You’re…me!” I shouted.

“In yer dreams, Bud,” he replied. Then he began to laugh....

My eyes snapped open and I sat bolt upright. Beads of sweat poured down my face as I fought to regain my bearings. I had never left. I was sitting in the recliner in my office, the half-full pizza box still sitting on my lap. I got up, wiping the sweat from my forehead on my sleeve. I walked to the sink and doused my face with cold water.

I hate dream sequences, I thought. But it was no use. I was just a two-bit character in an expensive technical book, and I knew it. Perhaps tomorrow would be a better day.

I grabbed a bottle of Purely Poulsbo from the fridge and started to contemplate what had happened. Maybe Helen had been right after all. Perhaps I should take better care of myself. Balanced meals. Vitamin supplements. More

exercise.

I picked up the pizza box and tossed it in the trash bin. I could do better.

This time out, The Author had given me a glimpse at one possible scenario. If that scenario was intended to be someone's future, I didn't want it to be mine.

I guess time and cotton briefs are a lot alike—they can slowly creep up on you, and if you don't do anything about it, you can eventually find yourself in a really tight situation.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Kick Ass Delphi Programming
(Publisher: The Coriolis Group)
Author(s): Don Taylor, Jim Mischel, John Penman, Terence Goggin
ISBN: 1576100448
Publication Date: 09/01/96

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Table of Contents