Mastering™

Marco Cantù

2025 Annotated Edition

Delphi™ 5

The Ultimate Delphi Resource—
for New and Experienced
Developers Alike

Written by the Winner of Borland's
1999 *Spirit of Delphi Award*

Marco Cantù

Mastering Delphi 5
2025 Annotated Edition

Original Edition: Sybex, 1995
2025 Annotated Edition: Marco Cantu, 2025
Release 0.2 – February 17th, 2025

Author: Marco Cantù
Publisher: Sybex (original edition), Marco Cantù (2025 edition)

Mastering Delphi 5 2025 Annotated Edition
This PDF version is a draft dated February 17th, 205

More information at http://www.marcocantu.com/md52025

To my wife, Lella, the love of my life[1]

1 I've kept the dedication of the book, as it was originally. In fact, that is still true today!

Preface To The 2025 Commented Edition

As you know I wrote several Mastering Delphi books over the course of the years. I thoughts a few times about writing a new one... but the task is fairly daunting, given Delphi (as an IDE and considering the libraries and target platforms it now supports) has dramatically grown in size and complexity, and you'd now need several thousand pages to cover the product adequately, and not even in depth. While I have several draft of my older books, it turns out Mastering Delphi 5 is the oldest one I have in an electronic version with images and proper formatting. A few years back, I acquired the rights of this edition from my original publisher (Sybex, now part of Wiley) and considering a new edition I asked a person to reformat the text, import the images, and turn this into a complete volume.

That was a few years back. More recently, I found this edited and formatted manuscript, and decided to make it public rather than keeping it on my hard drive. The

text of the book is, with minor and limited changes, the original text covering version 5 of Delphi, released in 1999.

This is not a book on recent versions of Delphi: A few of the sections are clearly dated, but most of the core content covering the key features of the product is still actual today. However, publishing it as is would have been of very limited use and possibly confusing. Therefore I've made two primary changes to the book. First I've captured some updated images of the IDE and of the running applications. I've kept some of the original images alongside, though, mixing the old and the new. The different is so striking I don't even need to call them out. Second I've added a large number of footnotes to underline new features, significant changes, code I'd write differently, assertions that are no longer true. I haven't rewritten the text, as this would have been way more time consuming, but I've pointed out many facts, giving ideas and suggestions for further study, or just providing some tidbits and facts, along with many links to additional information available online. I've used footnotes to reduce the impact on the existing text, compared to adding notes in the text flow or doing direct edits.

But you might still wonder, who is this book for? Although it might appeal them, *this is not only for the nostalgic*, although some of the old timers might find it interesting to read it. *It is for anyone who wants to understand Delphi.* Even covering the product how it was many years ago, **this book helps understanding all of Delphi's core concepts**.

You might be wondering if this is possible because Delphi is an old product. This is certainly *not* the case. It underlines the fact the product has a great history, but also that it has kept and keeps evolving in a fantastic way while maintaining its core tenets and offering an *unparalleled degree of compatibility in the development tools space*. The fact that most of the code in this old book can be compiled and run today, producing modern looking Windows 11 applications is a testament of the power of Delphi.

This preface is the only new section of the book. From now on, this is the old book with my comments and annotation. Have a good reading!

Acknowledgments

This incarnation of *Mastering Delphi* marks the fifth year of the Delphi era[2]. As it has for many other programmers, Delphi has been my primary interest throughout these years; and writing, consulting, teaching, and speaking at conferences about Delphi have absorbed more and more of my time, leaving other languages and programming tools in the dust of my office[3]. Because my work and my life are quite intertwined, many people have been involved in both, and I wish I had enough space and time to thank them all as they deserve. Instead, I'll just mention a few particular people and say a warm "Thank You" to the entire Delphi community (also for the Spirit of Delphi 1999 Award I've been happy to share with Bob Swart).

The first official thanks are for the Borland programmers and managers[4] who made Delphi possible and continue to improve it: Chuck Jazdzewski, Danny Thorpe[5], Eddie Churchill, Allen Bauer, Steve Todd, Mark Edington, Jim Tierney, Ravi

---

2 I made further editions of Mastering Delphi for Delphi 6, Delphi 7, and Delphi 2005 with the same publisher. Later I moved to self publishing and started the "Delphi 20xx Handbook" series, focused on specific new features of the given version of Delphi, rather than providing the broad overview of the Mastering Delphi volumes. That's one of the reasons for this new "reprint" of Mastering Delphi 5. You can find more on my web site www.marcocantu.com.

3 A few years ago I ended up accepting a Product Manager position at Embarcadero (now part of Idera Inc.), the company who owns Delphi. So my focus on Delphi continues to be a full time focus even today, although with a different perspective. I have used Delphi for 30 years and continue to do so. My knowledge of the technologies behind the product has grown during the years I've been working for Embarcadero.

Kumar, Jörg Weingarten, Anders Ohlsson, and all the others I have not had a chance to meet. I'd also like to give particular mention to my friends Ben Riga (the current Delphi product manager), Charlie Calvert, John Kaster, and David I (all three at Borland Developer's Relations). I cannot forget the help I received from Zack Urlocker and Nan Borreson.

The next thanks are for the Sybex editorial and production crew, many of whom I don't even know. Special thanks go to Denise Santoro, Jim Compton, and Diane Lowery for their editorial acumen; I'd also like to thank Richard Mills, Kristine O'Callaghan, Maureen Forys, Teresa Trego, Jennifer Campbell, Carol Iverson, and Tony Jonick.

This edition of *Mastering Delphi* has had an incredibly picky and detailed review from Delphi team member Danny Thorpe. His highlights and comments have improved the book in all areas: technical content, accuracy, examples, and even readability. Thanks a lot. Past editions of the book also had special contributions: Tim Gooch worked on Part V for Mastering Delphi 4 and Giuseppe Madaffari contributed a lot of database material for this and the last edition. Many improvements to the text and sample programs were suggested by technical reviewers of the past editions (Juancarlo Añez, Ralph Friedman, Tim Gooch, and Alain Tadros) and from other reviews done over the years by Bob Swart, Giuseppe Madaffari, and Steve Tendon.

Special thanks go to my friends Bruce Eckel, Andrea Provaglio, Norm McIntosh, Johanna and Phil of the BUG-UK, Ray Konopka, Mark Miller, Cary Jensen, Chris Frizelle of *The Delphi Magazine,* Foo Say How, John Howe, Mike Orriss, Chad "Kudzu" Hower, Dan Miser, and Marco Miotti. Also, a very big "Thank You" to all the attendees of my Delphi programming courses, seminars, and conferences in Italy, the United States, France, the United Kingdom, Singapore, the Netherlands, Germany, and Sweden.

Aside from the people involved with Delphi, my biggest thanks go to my patient wife, Lella, who (while carrying a child[6]) had to spend another summer with little vacation, as the book always took more time than I expected. Many of our friends provided healthy breaks in the work: Sandro and Monica with Luca, Stefano and

---

4 Needless to say none of this developers and managers work at Embarcadero any more, after 25 years. I've kept this Acknowledgments section as it was, despite the changes to my like and that of all of the people mentioned here.

5 Unfortunately we lost Danny a few years back. Danny was the technical reviewer of the original edition of this book, Mastering Delphi 5, and I've long been in touch with him after he left the company.

6 That child is now a young adult. We also have a second one, who's also grown up.

Elena, Marco and Laura with Matteo, Bianca, Chiara, Luca and Elena, Chiara and Daniele with Leonardo, Laura, Vito and Marika with Sofia. Our parents, brothers, sisters, and their families were very supportive, too. It was nice to spend some of our free time with them and our six nephews, Matteo, Andrea, Giacomo, Stefano, Andrea, and Pietro.

Finally, I would like to thank all of the people, many of them unknown, who enjoy life and help to build a better world. If I never stop believing in the future and in peace, it is also because of them.

Introduction

The first time Zack Urlocker[7] showed me a yet-to-be-released product code-named Delphi, I realized that it would change my work—and the work of many other software developers. I used to struggle with C++ libraries for Windows, and Delphi was and still is the best combination of object-oriented programming and visual programming for Windows.

Delphi 5 simply builds on this tradition and on the solid foundations of the VCL to deliver another astonishing and all-encompassing software development tool. Looking for database, client/server, multi tier, intranet, or Internet solutions? Looking for control and power? Looking for fast productivity? With Delphi 5 and the plethora of techniques and tips presented in this book, you'll be able to accomplish all this[8].

---

7 Zack was the first Delphi Product Manager, and made a career in many other management positions including at MySQL and, more recently, several startups.

8 Most of the features discussed here are still valid in the latest versions of Delphi, even if they represent a subset of the available features. A few have been discontinued or are not recommended any more, and this will all be covered in footnotes.

Five Versions and Counting

Some of the original Delphi features that attracted me were its form-based and object-oriented approach, its extremely fast compiler, its great database support, its close integration with Windows programming, and its component technology. But the most important element was the Object Pascal language, which is the foundation of everything else.

Delphi 2 was even better! Among its most important additions were these: the Multi-Record Object and the improved database grid, OLE Automation support and the variant data type, full Windows 95 support and integration, the long string data type, and Visual Form Inheritance. Delphi 3 added to this the Code Insight technology, DLL debugging support, component templates, the TeeChart, the Decision Cube[9], the Web Broker technology, component packages, ActiveForms, and an astonishing integration with COM, thanks to interfaces.

Delphi 4 gave us the AppBrowser editor, new Windows 98 features, improved OLE and COM support, extended database components, and many additions to the core VCL classes, including support for docking, constraining, and anchoring controls. There are a great many new features in Delphi 4, as you can still discover by reading this book if you missed the last edition.

Delphi 5 adds to the picture many more improvements of the IDE (too many to list here), extended database support (with specific ADO and InterBase datasets), an improved version of MIDAS[10] with Internet support, the TeamSource version-control tool[11], translation capabilities, the concept of frames, many new components, and much more, as you'll see in the following pages.

Delphi is a great tool, but it is also a complex programming environment that involves many elements. This book will help you master Delphi programming, including the Object Pascal language, Delphi components (both using the existing ones and developing your own), database and client/server support, the key elements of Windows and COM programming, and Internet and Web development.

---

9 The Decision Cube is a feature that was later dropped form the product.

10 MIDAS was later turned into DataSnap and some of the related technologies are still around, even if the world of multi-tier development and web services has changed a bit since the early days. Today's multi-tier solutions tend to use the REST architecture, which is true for RAD Server, the current multi-tier technology in Delphi.

11 More recent versions of Delphi have added support for modern version control systems, including Subversion and Git.

You do not need an in-depth knowledge of any of these topics to read this book, but you do need to know the basics of Pascal programming. Having some familiarity with Delphi will help you considerably, particularly after the introductory chapters. The book starts covering its topics in depth immediately; much of the introductory material from previous editions has been removed. Some of this left-out material and an introduction to Pascal[12] is available on the author's Web site and can be a starting point if you are not confident with Delphi basics. Each new Delphi 5 feature is covered in the relevant chapters throughout the book.

The Structure of the Book

The book is divided into five parts:

- Part I, "Delphi and Object Pascal," introduces new features of the Delphi 5 Integrated Development Environment (IDE) in Chapter 1 and then moves to the Object Pascal language and the Visual Component Library (VCL), providing both foundations and advanced tips[13].
- Part II, "Using Components," covers standard components, Windows common controls, graphics, menus, dialog, scrolling, docking, multiple-page controls, Multiple Document Interface, and many other topics[14].
- Part III, "Writing Database Applications," covers plain database access, advanced Paradox topics, in-depth coverage of the data-aware controls, client/server programming, InterBase Express, and ADO[15].

---

12 This material on the Pascal language later turned into the Essential Pascal e-book, but it is also included in my "Object Pascal Handbook", a book I'm maintaining up to date over time. It is available as a PDF via Embarcadero and as a printed book on Amazon, see www.marcocantu.com/objectpascalhandbook/ for more information.

13 While the IDE changed considerably a fair number of the techniques and tips still applied today. Also the core of the language remains the same, even if additions have been relevant. So most of the content of Part I is quite relevant.

14 The core techniques related with using components have not changed at all. So these foundation chapters provide a very good introduction to Delphi, even after many years. I'll mention changes in notes, like on all other chapters, of course.

15 The database part of the product has seen many significant changes, with the demise of Paradox and the introduction of the dbExpress library and later the migration to FireDAC. ADO components are still available and the core classes in the DB.pas unit did not change that much.

- Part IV, “Components and Libraries,” covers Delphi component and Dynamic Link Library (DLL) development; it then looks at COM and OLE, covering Windows shell extensions, OLE Automation, and ActiveX development[16].
- Part V, “Real-World Delphi Programming,” discusses many common programming techniques, such as multithreading, memory handling, debugging, using resources, printing support, file handling, programming TCP/IP sockets, Internet development, Web server-side extensions, three-tier architectures, and distributed database applications build upon the MIDAS technology[17].

As this brief summary suggests, the book covers topics of interest to Delphi users at nearly all levels of programming expertise, from “advanced beginners” to component developers.

In this book, I’ve tried to skip reference material almost completely and focus instead on techniques for using Delphi effectively. Because Delphi provides extensive online documentation, to include lists of methods and properties of components in the book would not only be superfluous, it would also make it obsolete as soon as the software changes slightly. I suggest that you read this book with the Delphi help files at hand, to have reference material readily available. You can find some more Delphi reference material on my Web site, as described later.

However, I’ve done my best to allow you to read the book away from a computer if you prefer. Screen images and the key portions of the listings should help in this direction. The book uses just a few conventions to make it more readable. All the source code elements, such as the keywords, the names of properties, classes, and functions appear in `this font`, and listings are formatted as they appear in the Delphi editor, with boldfaced keywords and italic comments and strings.

---

16 The section on components and libraries is still surprisingly up-to-date, as the COM layer in Windows is still there and Delphi's support saw limited improvements (as it was already very good and Microsoft didn't touch COM for many years, focusing on the newer .NET framework).

17 While some core techniques like multi-threading are still based on the same foundations, most of what relates to Web development saw significant improvements. Still WebBroker is still an architecture heavily used today.

Free Source Code on the Web

This book focuses on examples. After the presentation of each concept or Delphi component, you'll find a working program example (sometimes more than one) that demonstrates how the feature can be used. All told, there are more than 200 examples presented in the book[18]. Most of the examples are quite simple and focus on a single feature. More complex examples are often built step-by-step, with intermediate steps including partial solutions and incremental improvements.

note Some of the database examples also require you to have the Delphi sample database DBDEMOS installed; it is part of the default Delphi installation.

Besides the archive with the minimal source code files required to build the programs, a second archive includes an HTML version of the source code, with full syntax highlighting, along with a complete cross-reference of keywords and identifiers (class, function, method, and property names, among others). The cross-reference is an HTML file, so you'll be able to use your browser to easily find all the programs that use a Delphi keyword or identifier you're looking for.

The directory structure of the downloaded files is quite simple. Basically, each part of the book has its own folder, with a subfolder for each chapter, and a further subfolder for each example (e.g., `Part2\06\Borders`). In the text, the examples are simply referenced by name (e.g., Borders).

note Be sure to read the source code archive's Readme file, which contains important information about using the software legally and effectively.

How to Reach the Author

If you find any problems in the text or examples in this book, I would be happy to hear from you. Besides reporting errors and problems, please give us your unbiased

18 I have not updated or modified the source code demos, although I might do it in the future. I'll occasionally point out to code that needs an update. The source code demos are available on more modern repositories, like github.com/MarcoDelphiBooks/MasteringDelphi5.

opinion of the book and tell us which examples you found most useful and which you liked least. There are several ways you can provide this feedback[19]:

- My own Web page (www.marcocantu.com) hosts news and tips, technical articles, the free online book "Essential Pascal," Delphi 5 reference information we could not fit in this book, Delphi links, and my collection of Delphi components and tools.
- Finally, you can reach me via e-mail at marco@marcocantu.com. My mailbox is usually quite full and, regretfully, I cannot reply promptly to every request. Please write to me in English or Italian.

19 Here I've made an exception to the "no edits" rule and removed referenced to the original publisher, Sybex. My contact information is still valid, but some is missing like my blog (blog.marcocantu.com).

Chapter 1: Delphi And Object Pascal

In a visual programming tool such as Delphi, the role of the environment is at times even more important than the programming language. Delphi 5 provides many new features in its visual development environment, and this chapter covers them in detail. This chapter isn't a complete tutorial but mainly a collection of tips and suggestions aimed at the average Delphi user. In other words, it's not for newcomers. I'll be covering the new features of the Delphi 5 Integrated Development Environment (IDE) and some of the advanced and/or little-known features of previous versions as well, but in this chapter I won't provide a step-by-step introduction. Throughout this book I'll assume you already know how to carry out the basic hands-on operations of the IDE, and all the chapters after this one focus on programming issues and techniques.

If you *are* a beginning programmer, don't be afraid. The Delphi Integrated Development Environment is quite intuitive to use. Delphi itself includes a manual (available in Acrobat format on the Delphi CD[20]) with a tutorial that introduces the

20 There is no Delphi CD any more, but you can find tutorials and documentation at docwiki.embarcadero.com/RADStudio/.

development of Delphi applications. You can also find a step-by-step introduction to the Delphi 5 IDE on my Web site, www.marcocantu.com. The short online book *Essential Pascal*[21] is based on material from the first chapters of earlier editions of *Mastering Delphi*.

Editions of Delphi 5

Before delving into the details of the Delphi programming environment, let's take a side step to underline two key ideas. First, there isn't a single edition of Delphi 5; there are many of them. Second, any Delphi environment can be customized. For these reasons, Delphi screens you see illustrated in this chapter may differ from those on your own computer. Here are the current editions of Delphi:

- The basic version (the "Standard" edition) is aimed at Delphi newcomers and casual programmers.[22]
- The second level (the "Professional" edition) is aimed at professional developers. It includes all the basic features, plus database programming support, extensive Web server support (WebBroker), and some of the external tools. This book generally assumes you are working with at least the Professional edition.
- The full-blown Delphi (the "Enterprise" edition, previously called the "Client/Server Suite") is aimed at developers building enterprise applications. It includes SQL Links for native Client/Server BDE connections, ADO and InterBase Express components, support for multiuser applications, internationalization, and three-tier architecture, and many other tools, including the SQL Monitor. Some chapters cover features included only in Delphi Enterprise; these sections are specifically identified.

---

21 See www.marcocantu.com/epascal/ for the latest information about this e-book.

22 The "Standard" edition of Delphi has been long discontinued. It was temporarily replaced by a Turbo edition (now discontinued as well). Later the company introduced a new low cost version called Starter edition. Today you can use the free Delphi Community Edition (if you quality in terms of use case and earnings) or buy the "Professional" and "Enterprise" versions, which continue to be the core offerings, with differences not radically changed since Delphi 5. Check the latest product description and Feature Matrix at www.embarcadero.com/products/delphi for information on differences between the versions, so you can download or buy the one that better serves your needs.

Some of the features of Delphi Enterprise are available as an "upsell" to owners of Delphi Professional. Although this is a marketing decision that may change in the future, you should be able to buy ADO components and TeamSource (for cooperation among programmers). If you can't justify the cost of the full Enterprise edition for your work, you may be able to buy Delphi Professional plus the specific subsystems you want separately from the Borland Online Store[23].

Besides the different editions available, there are a number of ways to customize the Delphi environment. In the screen illustrations throughout the book, I've tried to use a standard user interface (as it comes out of the box); however, I have my preferences, of course, and I generally install many add-ons, which might be reflected in some of the screen shots.

The Delphi 5 IDE

The Delphi 5 IDE includes some of the broadest changes Borland has introduced since it upgraded Delphi 1 to Delphi 2. Among the new features are a redesigned Object Inspector, a new Project Manager, the ability to save the position of the desktop windows, the to-do list, and much more. Most of the features are quite easy to grasp, but it's worth examining them with some care so that you can start using Delphi 5 at its full potential.

Command-Line Options

The first thing to notice is that there are changes even before you start Delphi. In fact, the delphi32.exe program[24], which starts the IDE, has many new command-line options. Most of these options (listed in the Help topic "IDE command-line options"[25]) are aimed at advanced users and allow you to track the status of the Delphi IDE itself.

---

23 Extra add-ins are currently not sold separately any more.

24 The executable file that starts the IDE is now called "*bds.exe*" (which originally was a short version of Borland Developer Studio), some of the command lines parameters mentioned here still work and little known by developers.

25 See docwiki.embarcadero.com/RADStudio/en/IDE_Command_Line_Switches_and_Options

For example, you can load a program into the debugger or attach the system to a process that's already running (topics I'll discuss along with other debugging features in Chapter 18).

Other features might be useful even to the casual programmer. The `-ns` ("no splash") flag skips the splash screen, and the `-np` ("no project") flag tells Delphi not to open an empty project on startup. (This allows for a fast boot because it prevents the loading of any package of components, which are attached to projects.)

Probably the most commonly used feature isn't strictly a command-line option, or even a startup option. You can easily specify a project, project group, or Pascal source code file to open. When Delphi is already running, double-clicking a filename or icon in Windows Explorer doesn't open a new copy of the IDE, it simply opens a PAS or DFM file in the current copy of Delphi. When you select a project file (.DPR), Delphi first closes the current project after asking you to save any changes[26].

From the command line you can load a project and let Delphi automatically build or make it (with the `-b` and `-m`) options, immediately closing the IDE after the operation is completed. This doesn't seem terribly useful; for compiling a series of large projects with a script or batch file, you should instead use the faster command-line compiler[27] (which doesn't need the IDE).

Saving the Desktop Settings

Building on past versions of Delphi and on the support for docking that was added in Win32, Delphi has since version 4 allowed programmers to customize the IDE in a number of ways, typically opening many windows and arranging them and docking them to each other. However, programmers often need to open one set of windows at design time and a different set at debug time. Similarly, programmers might need one layout when working with forms and a completely different layout when writing components or low-level code using only the editor. Rearranging the IDE for each of these needs is a tedious task.

With Delphi 5, every time you come up with an arrangement of IDE windows you like for a specific purpose, you can save it with a name and restore it easily. Also, you can make one of these groupings your default debugging setting, so that it will

---

26 This behavior has changed: As you activate a project in Explorer, the IDE by default adds the project to the current project group, rather than replacing the currently open project.

27 Starting from Delphi 2005 the command line compilation can also be invoked using a MSBuild script, which is what the IDE does anyway. Compiling outside of the IDE is also directly available as a compiler project option.

be restored automatically when you start the debugger. All these features are available in the new Desktops toolbar, shown in Figure 1.1. (It's the only toolbar with a combo box.) You can also work with desktop settings using the View ➤ Desktops menu. This has the features of the toolbar, and also allows you to delete one of the saved settings[28].

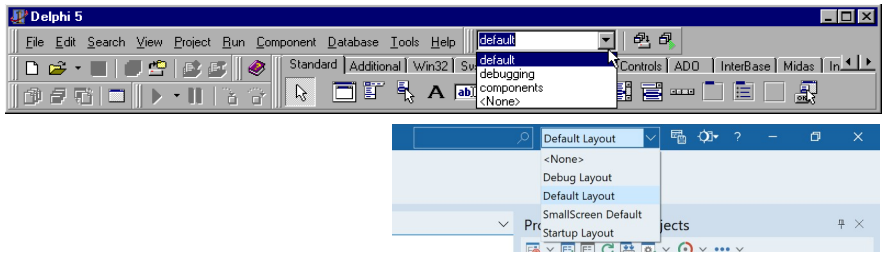


Figure 1.1: The main window of Delphi 5 includes the Desktops toolbar, which you can use to reload a configuration of the IDE windows. Images captured in Delphi 5 and Delphi 12.

Desktop setting information is saved in DST files[29], which are INI files in disguise. The saved settings include the position of the main window, the Project Manager, the Alignment Palette, the Object Inspector (including its new property category settings), the editor windows (with the status of the Code Explorer and the Message View), and many others, plus the docking status of the various windows.

Here is a small excerpt from a DST file, which should be easily readable:

```
[Main Window]
Create=1
Visible=1
State=0
Left=0
Top=0
Width=1024
Height=105
ClientWidth=1016
ClientHeight=78

[ProjectManager]
Create=1
Visible=0
State=0
...
Dockable=1
```

28 While the UI has changed the same idea remains today, with the addition of a new default desktop settings called "Startup Layout", used when no project is open.

29 In recent versions, the DST files are saved in the folder with the version number under C:\Users\<username>\AppData\Roaming\Embarcadero\BDS\xxx. The file content remains largely the same.

```
[AlignmentPalette]
Create=1
Visible=0
...

[PropertyInspector]
Create=1
Visible=1
...
Dockable=1
SplitPos=85
ArrangeBy=Name
HiddenCategories=Legacy
ShowStatusBar=1
```

note Desktop settings override project settings. This helps eliminate the problem of moving a project between machines (or between developers) and having to rearrange the windows to your liking. Delphi 5 separates per-user and per-machine preferences from the project settings, to better support team development.

The To-Do List

Another brand-new feature of Delphi 5's IDE is the to-do list[30]. This is a list of tasks you still have to do to complete a project, a collection of notes for the programmer (or programmers, as this tool can be very handy in a team). While the idea is not new, the key concept of the to-do list in Delphi 5 is that it works as a two-way tool.

In fact, you can add or modify to-do items by adding special comments to the source code of any file of a project; you'll then see the corresponding entries in the list. But you can also visually edit the items in the list to modify the corresponding source code comment. For example, here is how a to-do list item might look like in the source code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // TODO -oMarco: Add creation code
end;
```

The same item can be visually edited in the window shown in Figure 1.2.

30 While superseded by modern developer collaboration tools for tracking changes and work, the To-Do list has still a nice role and I think it has been a bit neglected, while I find it handy to leave notes for myself and occasionally for others using this format rather than using a general comment, as the IDE makes it easier to find them.

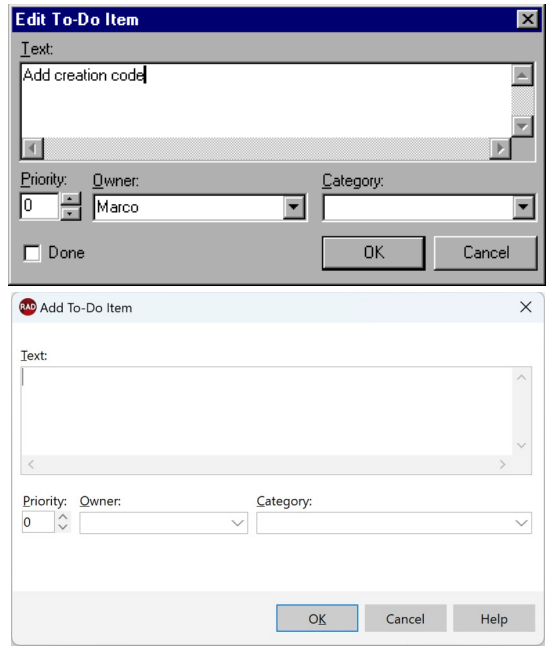


Figure 1.2: The Edit To-Do Item window can be used to modify a to-do item, an operation you can also do directly in the source code. Images captured in Delphi 5 and Delphi 12.

The exception to this two-way rule is the definition of project-wide to-do items. You must add these items directly to the list. To do that, you can either use the Ctrl+A key combination in the To-Do List window or right-click in the window and select Add from the shortcut menu. These items are saved in a special file with the .TODO extension.

There are multiple options you can use with a `TODO` comment. You can use `-o` (as in the code excerpt above) to indicate the owner, the programmer who entered the comment; the `-c` option to indicate a category; or simply a number from 1 to 5 to indicate the priority (0, or no number, indicates that no priority level is set). For example, using the Add To-Do Item command on the editor's shortcut menu (or the Ctrl+Shift+T shortcut[31]) generated this comment:

```
{ TODO 2 -oMarco : Button pressed }
```

Delphi treats everything after the colon, up to the end of line or the closing brace, depending on the type of comment, as the text of the to-do item.

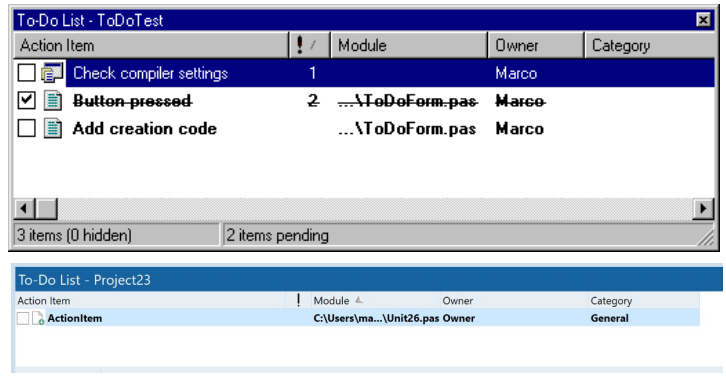
31 The shortcut still opens the Edit To-Do item dialog above, but you can also type "*todo*" in the editor and press space to trigger the generation of this line:

```
{TODO -oOwner -cGeneral : ActionItem}
```

Finally, in the To-Do List window you can check off an item to indicate that it has been done. The source code comment will change from TODO to DONE. You can also change the comment in the source code manually to see the check mark appear in the To-Do List window.

One of the most powerful elements of this architecture is the main To-Do List window, shown in Figure 1.3, which can automatically collect to-do information from the source code files as you type them. The items of this list are part of this chapter's ToDoTest example (which does nothing but has lots of things to do). The list items in this window show the various attributes I've just described, along with the source code files where they are defined. The initial check box is marked for Done items, which also have their text crossed out.

Figure 1.3: The To-Do List window for the ToDoTest example. Images captured in Delphi 5 and Delphi 12.



note To try out ToDoTest and all the program examples in this book, you need to download the source code[32]. Every reader should download the source code in order to get the full value of this book. Each time the text mentions a new program example by name, you should look for a folder of that name among the downloaded files and read the complete source code. For most examples you'll also want to compile the program and run it.

The To-Do List window has a shortcut menu that allows you to add, edit, or delete items, filter and sort them, and export them to the Clipboard. The command used to perform this last operation, Copy As, lets you export the items either as text or as an HTML table, which can be customized using the Table Properties command. The HTML table settings include a nice preview, as you can see in Figure 1.4. The information is not saved in an HTML file; it's just copied to the Clipboard. You have to

32 In this case, I've deleted portions of the text as the old locations don't exist any more. Again, the correct link is github.com/MarcoDelphiBooks/MasteringDelphi5.

open your favorite HTML editor (or Notepad or a text window in the Delphi editor) to save it to a file.[33]

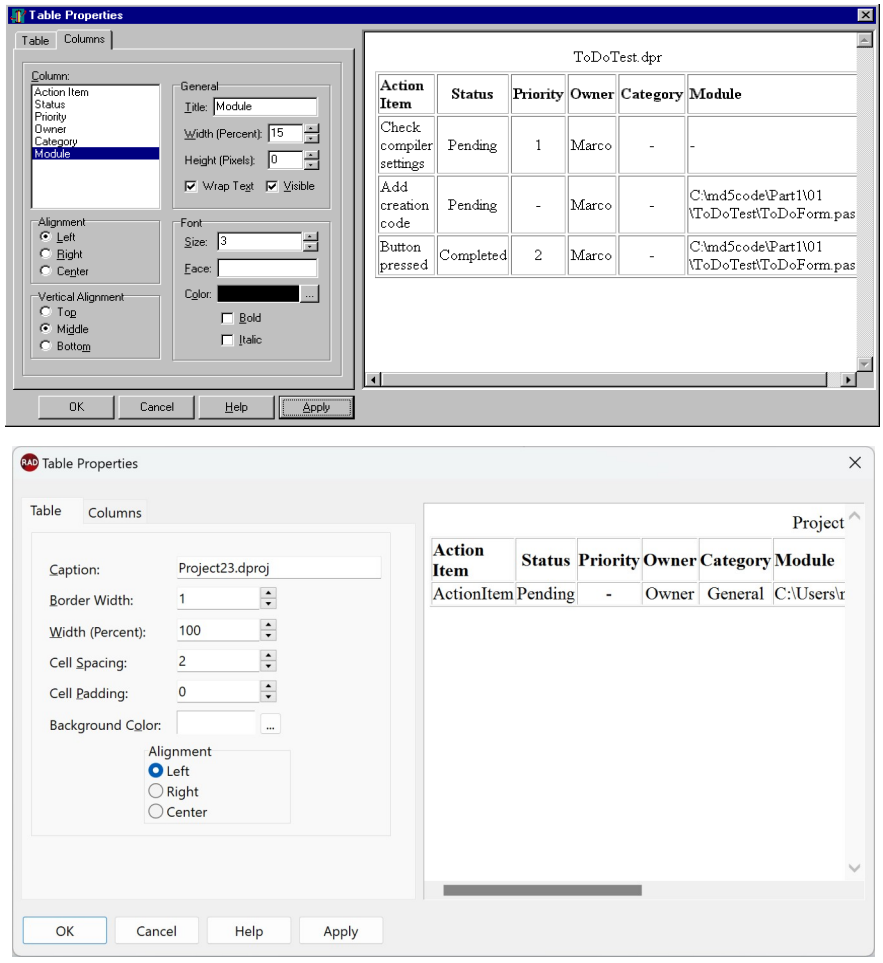


Figure 1.4: The HTML table preview of the to-do list. Images captured in Delphi 5 and Delphi 12.

33 I realized I had not seen that HTML preview in so many years, I thought the feature had been dropped, but – as Figure 1.4 shows – it's still in the most recent versions of Delphi.

The AppBrowser Editor

The editor included with Delphi 5 hasn't changed much from Delphi 4. However, Delphi 4 had many new features, so it's worth briefly examining this tool. Delphi 4 introduced three fundamental innovations: a Code Explorer window (which lists all the definitions of a unit), support for navigation (similar to that of a Web browser), and Class Completion (a code-generation technology).

Delphi 5 adds to the editor a new keyboard mapping for Visual Studio emulation and the ability to extend the editor with custom key mapping modules. These last settings are defined in the new Key Bindings tab of the Editor Properties dialog box, which you can activate with the Tools ➢ Editor Options[34] command. This new dialog box shows the environment settings related to the editor.

note The custom key mapping modules can be written using new Tools API features added to Delphi 5. You can write a completely new key mapping module or simply add a few extra shortcut keys to the existing one. This advanced topic is not covered in the book, but you can find examples in the Editor Keybinding folder of Delphi's Demos directory. One of these additional key bindings, called Buffer List, is installed by default and available by pressing the Ctrl+B key combination.

The Delphi editor allows you to work on several files at once, using a "notebook with tabs" metaphor, and you can also open multiple editor windows[35]. You can jump from one page of the editor to the next by pressing Ctrl+Tab (or Shift+Ctrl+Tab to move in the opposite direction). There are a number of options that affect the editor, located in the new Editor Properties dialog box. You have to go to the Preferences page of the Environment Options[36] dialog box, however, to set the editor's AutoSave feature, which saves the source code files each time you run the program (preventing data loss in case the program crashes badly).

I won't discuss the various settings of the editor, as they are quite intuitive and are described in the online Help. What is not officially documented is that you can use two entries of the Windows Registry to set the initial width and height of the editor[37]

34 This is now found in the Editor section of the Tools | Options dialog box.

35 Starting with very recent versions of Delphi, you can also use "split views" which is the ability to slit an editor horizontally or vertically to see more than one file, but also see two different locations of the same file side by side. *I like this "split views" new feature a lot!*

36 Now under Tools | Options. I won't keep adding footnotes for each occurrence, it's a general changes how options are now surfaced I a single all-encompassing dialog box.

37 This entire concept doesn't exist any more, given the editor is now docked to the main IDE window.

(to make it as large as your screen, for example). Go to the Delphi section in the Registry[38], `HKEY_CURRENT_USER/Software/Borland/Delphi/5.0`, and add under the `Editor` key two new `DWORD` items, called `DefaultHeight` and `DefaultWidth`, indicating the height and width of the editor in pixels. To modify the Windows Registry you can use the RegEdit.EXE application under Windows 95 and 98 or RegEdt32.EXE under NT[39].

Another tip to remember is that beginning with Delphi 4, using Cut and Paste commands is not the only way to move source code. You can also select and drag words, expressions, or entire lines of code. You can also copy text instead of moving it, by pressing the Ctrl key while dragging.

The Code Explorer

The *Code Explorer* window[40], which is generally most useful when it's docked on the side of the editor, simply lists all of the types, variables, and routines, defined in a unit, plus other units appearing in `uses` statements. For complex types, such as classes, the Code Explorer can list detailed information including a list of fields, properties, and methods. All the information is updated as soon as you start typing in the editor. You can use the Code Explorer to navigate in the editor. If you double-click one of the entries in the Code Explorer, the editor jumps to the corresponding declaration.

While all that is quite obvious after you've used Delphi for a few minutes, there are some features of the Code Explorer that are not so intuitive. One important point is that you have full control of the layout of the information, and you can reduce the depth of the tree usually displayed in this window by customizing the Code Explorer. Collapsing the tree can help you make your selections more quickly. You can configure the Code Explorer by using the corresponding page of the Environment Options[41], as shown in Figure 1.5.

---

38 In recent releases, that's a key under Computer\HKEY_CURRENT_USER\Software\Embarcadero\BDS\23.0 or similar (depending on the internal product version number).

39 Today, it's just called *regedit.exe*.

40 The content of what was the Code Explorer windows now displayed in the Structure view in case a source code file is open in the editor (while the same pane doubles as a form layout view when a designer is selected). The Code Explorer pane has now some more information, including Error Insight, the list of errors in the given unit.

41 In recent versions these settings are available under Tools | Options, User Interface, Structure.

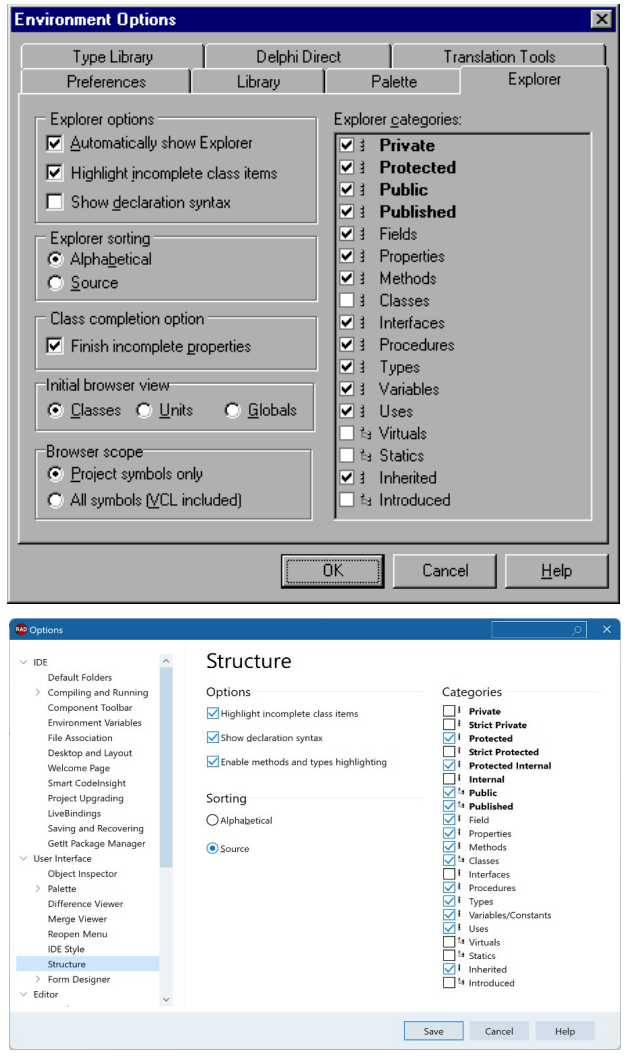


Figure 1.5:
You can configure the Code Explorer in the Environment Options dialog box. Images captured in Delphi 5 and Delphi 12: The content it surprisingly similar.

Notice that when you deselect one of the Explorer Categories items on the right side of this page of the dialog box, the Explorer doesn't remove the corresponding elements from view, it simply adds the node in the tree. For example, if you deselect the Uses check box, Delphi doesn't hide the list of the used units from the Code Explorer. On the contrary, the used units are listed as main nodes instead of being kept in the Uses folder. As another example, by disabling the Types, Classes, and Variables selections, you obtain the output shown in Figure 1.6.

The most important settings are probably those related to classes. The definitions related to a class can be arranged in three ways:

- According to the private, protected, public, and published categories
- According to the methods and fields categories
- All together in a single group

As each item of the Code Explorer tree has an icon marking its type, arranging by field and method seems less important than arranging by access specifier. My preference is to show all items in a single group, as this requires the fewest mouse clicks to reach each item. Selecting items in the Code Explorer, in fact, provides a very handy way of navigating the source code of a large unit. When you double-click on a method in the Code Explorer, the focus moves to the definition in the class declaration (in the interface portion of the unit). You can use the Ctrl+Shift combination with the up or down arrow keys to jump from the definition of a method or procedure in the interface portion of a unit to its complete definition in the implementation portion (or back again).

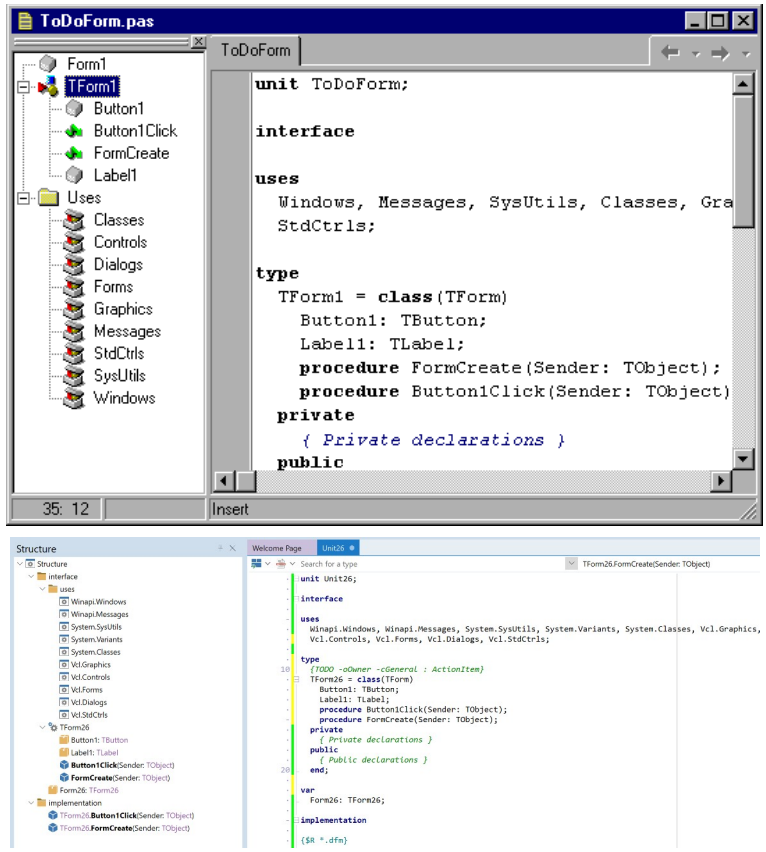


Figure 1.6: Some of the folders of the Code Explorer can be removed by removing the items from the corresponding settings. Images captured in Delphi 5 and Delphi 12.

note Some of the Explorer Categories shown in Figure 1.5 are used by the new Project Explorer (or Browser) introduced in Delphi 5, rather than by the Code Explorer. These include, among others, the Virtuals, Statics, Inherited, and Introduced grouping options.

The Code Explorer is not only an output and browsing tool. In fact, you can use it for entering new items in each category. Actually, the type of the new item generally depends on what you type. A name that starts with the `procedure` or `function` keywords is automatically considered a method, while a name followed by a semicolon and a data type is considered a field. The editing capabilities of the Code Explorer are too limited to provide any real advantage compared to editing in the source-code window. It would be nice to have dragging capabilities, for example, to move a field or method to a different visibility section or copy it to another class.

note *Field*, *methods*, *public*, *private*...? If you're not familiar with the terminology of the Object Pascal language, you'll find good coverage of these terms in Chapter 2. I've used them here without explaining them simply because most readers of this book probably have at least some exposure to Delphi and its programming language.

Browsing in the Editor

Another feature of the AppBrowser editor is the *Tooltip Symbol Insight*. Move the mouse over a symbol in the editor, and a Tooltip will show you where the identifier is declared. This feature can be particularly important for tracking identifiers, classes, and functions within an application you are writing, and also for referring to the source code of the Visual Component Library (VCL).

note Although it may seem a good idea at first, you cannot use Tooltip Symbol Insight to find out which unit declares an identifier you want to use. If the corresponding unit is not already included, in fact, the Tooltip won't appear.

The real bonus of this feature, however, is that you can turn it into a navigational aid. When you hold down the Ctrl key and move the mouse over the identifier Delphi creates an active link to the definition instead of showing the Tooltip. These links are displayed with the blue color and underline style that are typical of Web browsers, and the pointer changes to a hand whenever it's positioned on the link, as shown in Figure 1.7.

For example, you can Ctrl-click on the `TLabel` identifier to open its definition in the VCL source code. As you select references, the editor keeps track of the various positions you've jumped to, and you can move backward and forward among them—again as in a Web browser. You can also click on the drop-down arrows near the Back and Forward buttons to view a detailed list of the lines of the source code files you've already jumped to, for more control over the backward and forward movement.

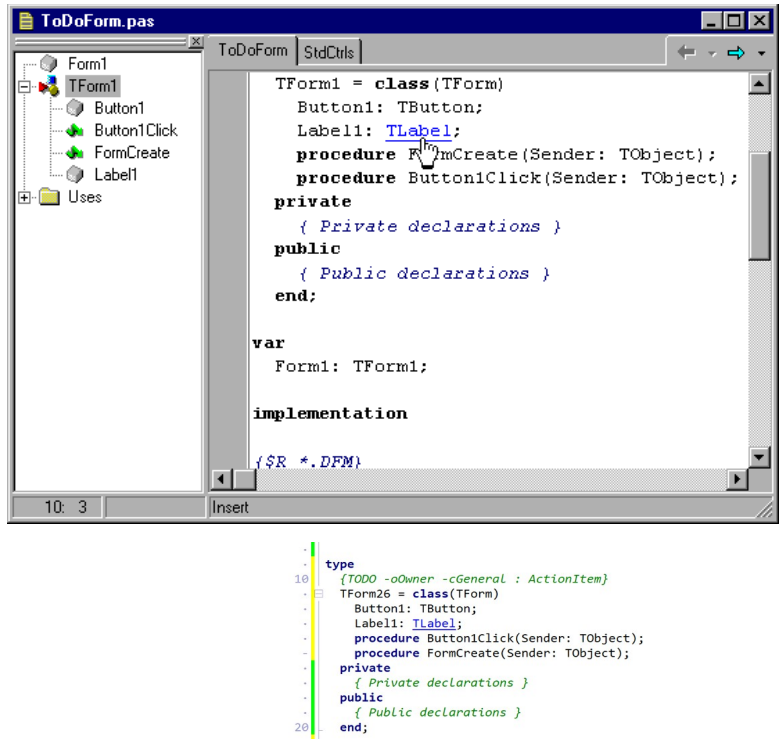


Figure 1.7: Delphi's browsing capability is activated by keeping the Ctrl key pressed and moving the mouse over an identifier. Images captured in Delphi 5 and Delphi 12.

How can you jump directly to the VCL source code if it is not part of your project? The AppBrowser editor can find not only the units in the Search path (which are compiled as part of the project), but also those in Delphi's Debug Source, Browsing, and Library paths. These directories are searched in the order I've just listed, and you can set them in the Directories/Conditionals page[42] of the Project Options dialog box and in the Library page of the Environment Options dialog box. By default, Delphi adds the VCL source code directories in the Browsing path of the environment, which has the following declaration:

```
$(DELPHI)\source\vcl;$(DELPHI)\source\rtl\Corba;
$(DELPHI)\source\rtl\Sys;$(DELPHI)\source\rtl\Win;
$(DELPHI)\source\Internet
```

42 The Browsing Path is now configured the Language | Delphi | Library section of the Tools | Options dialog box. The default path is very long, and not worth listing here.

In this series of paths, the declaration `$(DELPHI)` stands for the directory where Delphi is installed[43].

Class Completion

The third important feature of Delphi's AppBrowser editor is *Class Completion*, activated by pressing the Ctrl+Shift+C key combination. Adding an event handler to an application is a fast operation, as Delphi automatically adds the declaration of a new method to handle the event in the class and provides you with the skeleton of the method in the implementation portion of the unit. This is part of Delphi's support for visual programming.

Newer versions of Delphi also simplify life in a similar way for programmers who write a little extra code behind event handlers. The new code-generation feature, in fact, applies to general methods, message-handling methods, and properties. For example, if you type the following code in the class declaration:

```
public
  procedure Hello (MessageText: string);
```

and then press Ctrl+Shift+C, Delphi will provide you with the definition of the method in the implementation section of the unit, generating the following lines of code:

```
{ TForm1 }
procedure TForm1.Hello(MessageText: string);
begin
end;
```

This is really handy, compared with the traditional approach of many Delphi programmers, which is to copy and paste one or more declarations, add the class names, and finally duplicate the `begin .. end` code for every method copied.

Class Completion can also work the other way around. You can write the implementation of the method with its code directly, and then press Ctrl+Shift+C to generate the required entry in the class declaration.

Glancing back at the Explorer settings shown in Figure 1.5, you'll see one option for Class Completion—you can use it to complete the definition of a property. If you simply type in a brand-new form class,

```
property X: Integer;
```

---

43 This is now replaced by the $(BDS) symbolic reference.

and activate Class Completion, Delphi generates a `SetX` method for the property and adds the `FX` field to the class. The resulting code looks like this:

```
type
  TForm1 = class(TForm)
  private
    FX: Integer;
    procedure SetX(const Value: Integer);
  public
    property X: Integer read FX write SetX;
  end;

implementation

procedure TForm1.SetX(const Value: Integer);
begin
  FX := Value;
end;
```

This really saves a lot of typing. In fact, you can even partially control how Class Completion generates Set and Get methods for the property, as discussed in Chapter 3 in the section devoted to properties.

Code Insight

Besides the Code Explorer, Code Completion, and the navigational features, the Delphi editor still supports the *Code Insight*[44] technology originally introduced in Delphi 3. Collectively, the Code Insight techniques are based on a constant background parsing, both of the source code you write and of the source code of the system units your source code refers to. Code Insight comprises five capabilities:

- **Code Completion** allows you to choose the property or method of an object simply by looking it up on a list, or by typing its initial letters. To activate it you can simply type the name of an object, such as `Button1`, then add the dot, and wait. To force the display of the list, press Ctrl+Spacebar; to remove it when you don't want it, press Esc. Code Completion also lets you look for a proper value in an assignment statement. As you type **:=** after a variable or property, Delphi will list all the other variables or objects of the same type, plus the objects having properties of that type. While the list is visible, you can right-click on it to change the order of the items, sorting either by scope or by name.

44 Most of the Code Insight features are now based on a DelphiLSP engine, a Delphi implementation of the Language Server Ptotocol defined by Microsoft. The behavior in the IDE remains almost unchanged.

- **Code Templates** allow you to insert one of the predefined code templates, such as a complex statement with an inner `begin..end` block. Code Templates must be activated manually, by typing Ctrl+J to show a list of all of the templates[45]. If you type a few letters (such as a keyword) before pressing Ctrl+J, Delphi will list only the templates starting with those letters.
- **Code Parameters** display, in a hint or Tooltip window, the data type of a function's or method's parameters while you are typing it. Simply type the function or method name and the open (left) parenthesis, and the parameter names and types appear immediately in a popup hint window. To force the display of Code Parameters, you can press Ctrl+Shift+spacebar. As a further help, the current parameter appears in boldface type.
- **Tooltip Expression Evaluation** is a debug-time feature. It shows you the value of the identifier, property, or expression that is under the mouse cursor.
- **Tooltip Symbol Insight** lets you see the definition of an identifier in a Tooltip, as discussed earlier, in the section "Browsing in the Editor."

You can enable and disable or configure each of these features in the Code Insight page of the Editor Options dialog box[46], shown in Figure 1.8.

45 Since Delphi 2006, Code Templates have been replaced and superseded by the more powerful Live Templates, which are invoked either by the Tab key or the plain Space key, but are still listed if you press the original Ctrl+J shortcut key. Live Templates are covered in my "Delphi 2007 Handbook".

46 The configuration is now under the Editor | Language page of the Tools | Options dialog box. The page has multiple tabs including a "Code Insigth" one, as shown in Figure 1.8.

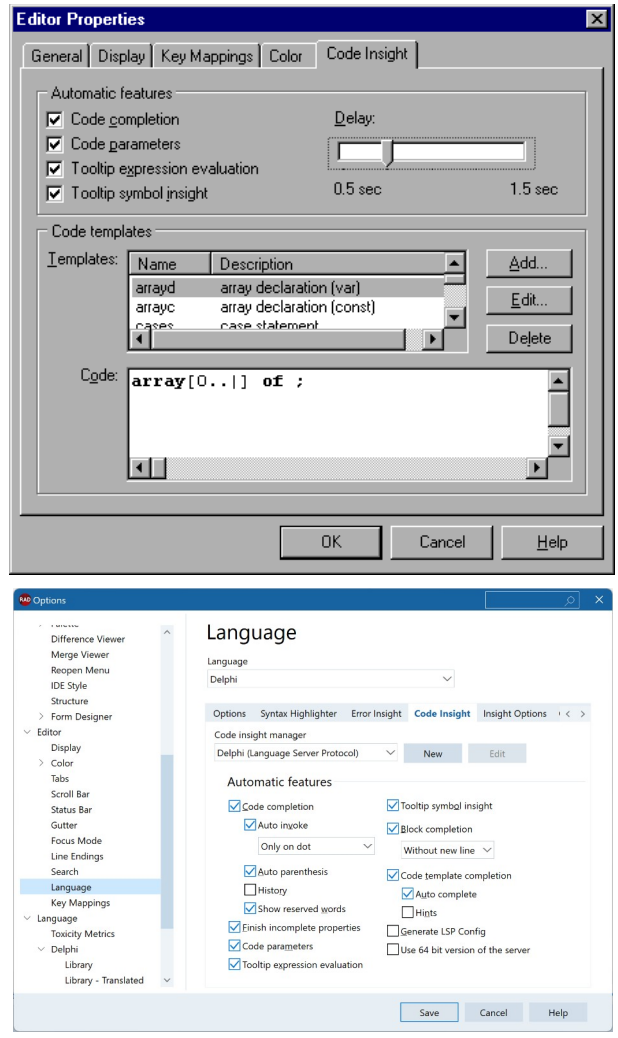


Figure 1.8: The Code Insight page of the Editor Options dialog box allows you to activate or disable each of these technologies and to set the delay time. Images captured in Delphi 5 and Delphi 12.

note When the code you've written is not correct, Code Insight won't work, and you may see just a generic error message indicating the situation. It is possible to display specific Code Insight errors in the Message pane (which must already be open; it doesn't open automatically to display compilation errors). To activate this feature you need to set another undocumented registry entry, setting the string key `Delphi\5.0\Compiling\ShowCodeInsiteErrors` to the value "1".[47]

47 This feature is now active by default and it can be configured in the same page of the Tools Options dialog box, in the "Error Insight" tab.

More Editor Shortcut Keys

The editor has many more shortcut keys, which depend on the editor style you've selected. Here are a few of the less-known shortcuts, most of which are useful:

- Ctrl+Shift plus a number key from 0 to 9 activates a bookmark, indicated in a "gutter" margin on the side of the editor. To jump back to the bookmark you can press the Ctrl key plus the number key. The usefulness of bookmarks in the editor is limited by the fact that a new bookmark can override an existing one and that bookmarks are not persistent[48]; they are lost when you close the file.
- Ctrl+E activates the incremental search. You can press Ctrl+E and then directly type the word you want to search for, without the need to go through a special dialog box and click the Enter key to do the actual search.
- Ctrl+Shift+I indents multiple lines of code at once. The number of spaces used is the one that is set by the Block Indent option in the Editor page of the Environment Options dialog box. Ctrl+Shift+U is the corresponding key for unindenting the code.
- Ctrl+O+U toggles the case of the selected code; you can also use Ctrl+K+E to switch to lowercase and Ctrl+K+F to switch to uppercase.
- Ctrl+Shift+R starts recording a macro, which you can later play by using the Ctrl+Shift+P shortcut. The macro records all the typing, moving, and deleting operations done in the source code file. Playing the macro simply repeats the sequence—an operation that has little meaning once you've moved on to a different source code file. I have yet to find a use for this technique, although I guess Borland uses it for testing purposes[49].
- Holding down the Alt key, you can drag the mouse to select rectangular areas of the editor, not just consecutive lines and words.

---

48 This is not true any more: Editor bookmarks are saved along with other local project settings.

49 I started using this feature (which is now surfaced with specific buttons at the bottom of the editor pane) when I need to perform repeated editing, like deleting or adding the same text to multiple lines. It can be very effective.

The Form Designer

Another Delphi window you'll interact with very often is the Form Designer, a visual tool for placing components on forms. In the Form Designer you can select a component directly with the mouse or through the Object Inspector, a handy feature when a control is behind another one or is very small. If a control covers another one completely, you can use the Esc key to select the parent control of the current one. You can press Esc one or more times to select the form, or press and hold Shift while you click on the selected component. This will deselect the current component and select the form by default.

note What if you need to move a control at design time by dragging it, but its area is covered by a child control? Just drag the child control and then press the Esc key (while holding down the mouse button) to switch the dragging operation to the parent control.

There are two alternatives to using the mouse to set the position of a component. You can either set values for the `Left` and `Top` properties, or you can use the arrow keys while holding down Ctrl. Using arrow keys is particularly useful for fine-tuning an element's position. (The Snap to Grid option works only for mouse operations.[50]) Similarly, by pressing the arrow keys while you hold down Shift, you can fine-tune the size of a component. (If you press Shift+Ctrl along with an arrow key, the component will be moved only at grid intervals.) Unfortunately, during these fine-tuning operations the component hints with the position and size are not displayed.

To align multiple components or make them the same size, you can select several components and set the `Top`, `Left`, `Width`, or `Height` property for all of them at the same time. To select several components, you can click on them with the mouse while holding down the Shift key, or, if all the components fall into a rectangular area, you can drag the mouse to "draw" a rectangle surrounding them. When you've selected multiple components, you can also set their relative position using the Alignment dialog box (with the Align command of the form's shortcut menu) or the Alignment palette (accessible through the View ➢ Alignment Palette[51] menu command).

50 The design time guidelines now available in Delphi offer you a lot of power for aligning components to the sides or the text baseline and effectively replace some of the techniques described here and later. Notice also that you now get some of the hints that were missing when I wrote the text.

51 Now available with the menu View | Toolbars | Align.

When you've finished designing a form, you can use the Lock Controls command of the Edit menu to avoid accidentally changing the position of a component in a form. This is particularly helpful, as there is no real Undo operation on forms (only an Undelete one), but the setting is not persistent.

Among its other features, the Form Designer offers a number of Tooltip hints:

- As you move the pointer over a component, the hint shows you the name and type of the component. This is an alternative to the Show Component Captions environment setting, which I tend to keep always active.
- As you resize a control, the hint shows the current size (the `Width` and `Height` properties). Of course, this feature is available only for controls, not for non-visual components (which are indicated in the Form Designer by icons).
- As you move a component, the hint indicates the current position (the `Left` and `Top` properties).

Finally, what may be the most important new Delphi 5 feature of the Form Designer is that you can save DFM (Delphi Form Module) files in plain text instead of the traditional binary resource format[52]. You can toggle this option for an individual form with the Form Designer's shortcut menu, or you can set a default value for newly created forms in the Preferences page of the Environment Options dialog box (see Figure 1.9).

52 Using textual DFM files has now long been the default in Delphi.

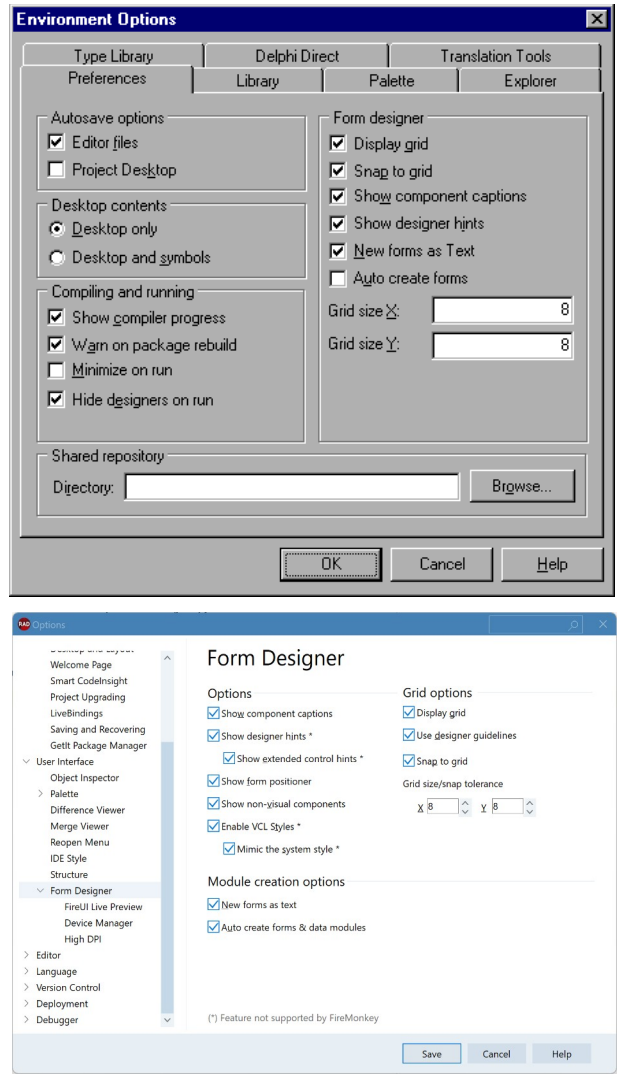


Figure 1.9:
The Preferences page of the Environment Options dialog box in Delphi 5 allows you to determine whether forms will be created by default and whether the DFM files will hold plain text. Images captured in Delphi 5 and Delphi 12.

In the same page you can also specify whether the secondary forms of a program will be automatically created at startup, a decision you can always reverse for each individual form (using the Forms page of the Project Options dialog box). But the most obvious difference between Delphi 5 and past versions, when working with forms, is the Object Inspector.

Having DFM files stored as text is a welcome addition; it lets you better operate with version-control systems. Programmers won't get a real advantage from this

feature, as you could already open the binary DFM files in the Delphi editor with a specific common of the shortcut menu of the designer. Version control systems, on the other hand, need to store the textual version of the DFM files to be able to compare them and capture the differences between two versions of the same file. This was probably introduced in Delphi 5 in conjunction with the new TeamSource version control system interface, discussed in Chapter 19[53].

In any case, note that if you use DFM files as text, Delphi will still convert them into a binary resource format before including them in the executable file of your programs. DFM are linked into your executable in binary format to reduce the executable size (although they are not really compressed) and to improve run-time performance (they can be loaded faster).

note Earlier versions of the Delphi IDE won't recognize text DFM files. When you open a textual DFM in Delphi 4 (or past versions), you'll get an error. To fix it, you should manually first use Delphi 5 to convert the DFM file to the binary format, using the shortcut menu of the Form Designer. (On a computer that doesn't have Delphi 5, you can use the Delphi 4 command-line tool `CONVERT`.[54]) When you open an existing DFM in the Delphi 5 IDE, the original DFM format will be preserved (unless you explicitly change it using the Text DFM shortcut menu item), thus allowing you to reopen the same form in past version of Delphi.

The Object Inspector in Delphi 5

If you have used Delphi in the past, you will immediately see that there is something new in the Object Inspector. The most important changes involve the graphical drop-down lists and the property categories.

The first element is the simplest to use. The drop-down list for a property in the Object Inspector can include graphical elements. Many of the relevant properties use this feature by default: `Color`, `Cursor` and its variations, generally the `ImageIndex` property of components connected with an ImageList (such as an action, a menu item, or a toolbar button), the Pen and Brush styles, and a few others. For example, Figure 1.10 shows the list of cursors (`Cursor` properties)[55]. Of course, developers of Delphi components and add-ins will be able to customize this feature, and you'll see more graphical drop-down elements in the future. See the fol-

53 Given this entire feature is no longer available (and it has been removed from the product for a long time), I'm going to remove that section of the book.

54 The *convert.exe* tools continues to exist and be available in the *bin* folder today

55 The list of cursors is still displayed today, even if I haven't included an updated image.

lowing section “Drop-Down Fonts in the Object Inspector” for a simple customization of this window.

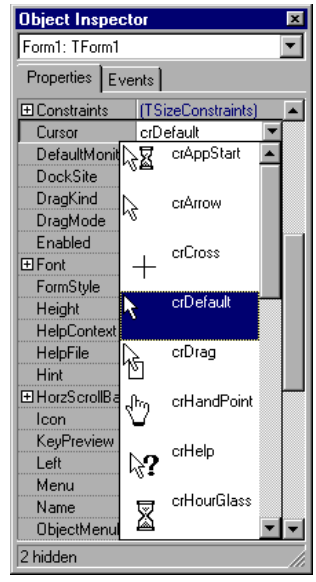


Figure 1.10:
A graphical drop-down list in the Delphi 5 Object Inspector, showing available cursors.

It takes a little more time to get used to the property categories[56]. To understand this feature, you first need to make it visible. To display properties by category instead of by name, right-click in the Object Inspector and choose the proper Arrange option from the shortcut menu. You can see the effect of this choice in Figure 1.11. Looking carefully at this figure, you may notice something strange—the `Align` property is available in two different categories. This is a general rule; categories are not exclusive, and a property can register itself for multiple categories.

56 While the ability to group Object Inspector properties in categories still exists today (see Figure 1.11), this feature is not frequently used and generally not recommended. Because of this, I’ve skipped capturing new versions of some of the other figures.

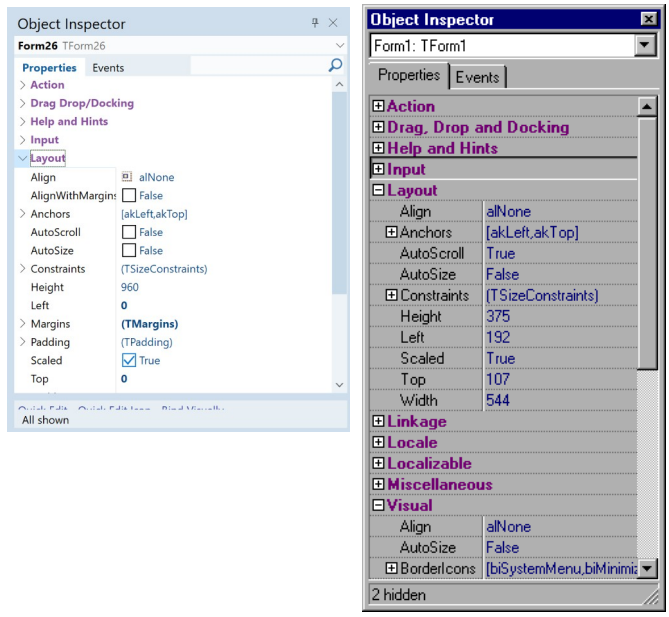


Figure 1.11:
The effect of arranging properties by category. Images captured in Delphi 5 and Delphi 12.

Categories have the benefit of reducing the complexity of the Object Inspector. You can use the View submenu from the shortcut menu to hide properties of given categories, regardless of the way they are displayed (that is, even if you prefer the traditional arrangement by name, you can still hide the properties of some categories). For example, in Figure 1.12 you can see the properties of a form arranged by name, but only the properties within the Visual and Input categories. In fact, as you can see in the status bar of the Object Inspector, 44 properties are hidden. The arrangement and the visibility you select will affect events, as well.

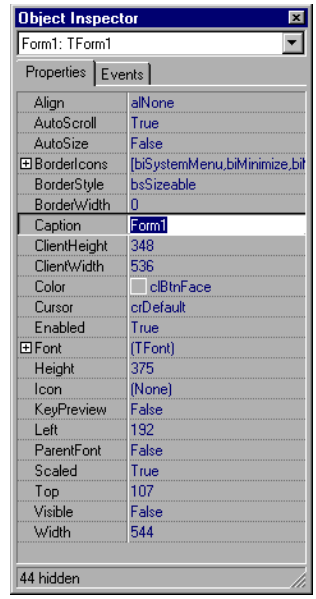


Figure 1.12: You can hide properties of some categories, even when they are arranged by name.

note Another new feature of the Delphi 5 Object Inspector is the ability to select the component referenced by a property. To do this, double-click with the left mouse button on the property value while keeping the Ctrl key pressed. For example, if you have a MainMenu component in a form and you are looking at the properties of the form in the Object Inspector, you can select the MainMenu component by moving to the MainMenu property of the form and Ctrl+double-clicking on the value of this property. This selects the main menu indicated as the value of the property in the Object Inspector. This feature can be very useful when you have many connected components; for example, when using multiple data-source and dataset components.[57]

57 The ability to jump to the connected component has later been extended with the ability to expand the properties of the connected component in place, as if it was a local property with sub-properties.

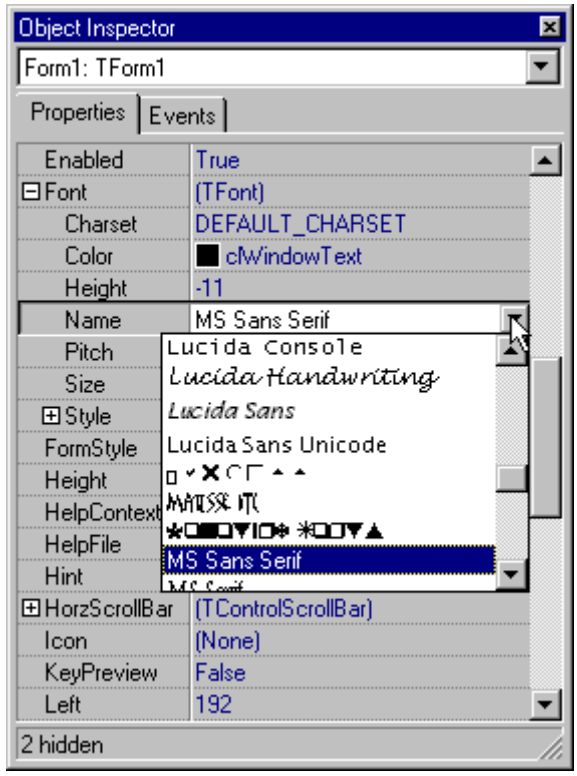
Drop-Down Fonts in the Object Inspector[58]

The Delphi 5 Object Inspector has graphical drop-down lists for several properties. You might want to add one showing the actual image of the font you are selecting, corresponding to the `Name` subproperty of the `Font` property. This capability is actually built into Delphi 5, but it has been disabled because most computers have a large number of fonts installed and rendering them can really slow down the computer. If you want to enable this feature, you have to install in Delphi a package that enables the `FontNamePropertyDisplayFontNames` global variable of the DsgnIntf unit. I've done this in the OiFontPk package, which you can find among the program examples for this chapter[59].

Once this package is installed, you can move to the `Font` property of any component, and use the graphical Name drop-down menu, as displayed below:

58 This Object Inspector customization still works today, but it is rarely used as painting the drop down list of fonts with the actual fonts can be very slow, compared to showing the font names only.

59 Again, this is not recommended as this makes the display terribly slow. The feature can be enabled without the special add-in package.



There is a second, more complex customization of the Object Inspector I like and use frequently, a custom font for the entire Object Inspector, to make its text more visible. This feature is particularly useful for public presentations[60]. You can find the package to install custom fonts in the Object Inspector on my Web site, `www.marcocantu.com`.

60 I won't recommend using this old add-in package either, I doubt it's going to work smoothly.

Secrets of the Component Palette

The Component Palette[61] is very simple to use, but there are a few things you might not know. There are four simple ways to place a component on a form:

- After selecting a control in the palette, click within the form to set the position for the control, and press-and-drag the mouse to size it.
- After selecting any component, simply click within the form to place it with the default height and width.
- Double-click the icon in the palette to add a component of that type in the center of the form.
- Shift-click on the component icon to place several components of the same kind in the form. To stop this operation, simply click on the standard selector (the arrow icon) on the left side of the Component Palette.

You can select the Properties command on the shortcut menu of the palette to completely rearrange the components in the various pages, possibly adding new elements or just moving them from page to page. In the resulting Properties page, you can simply drag a component from the Components list box to the Pages list box to move that component to a different page.

note When you have too many pages in the Component Palette, you'll need to scroll them to reach a component. There is a simple trick you can use in this case: Rename the pages with shorter names, so that all the pages will fit on the screen. Obvious—once you've thought about it.

The real undocumented feature of the Component Palette is the "hot-track" activation. By setting special keys of the Registry, you can simply select a page of the palette by moving over the tab, without any mouse click. The same feature can be applied to the component scrollers on both sides of the palette, which show up when a page has too many components.

To activate this hidden feature you have to add an `Extras` key under `HKEY_CURRENT_USER\Software\Borland\Delphi\5.0`. Under this key you have to

61 The Component Palette has been replaced by the Tools Palette, but some of the description in this section (like the ways to select components) still applies. Delphi still has also a Components Toolbar that acts, behaves, and can be customized much like the original Component palette, although it's not a stable and reliable feature and Embarcadero has hinted at deprecating and removing it.

enter two string values, `AutoPaletteSelect` and `AutoPaletteScroll`, and set each value to the string `'1'`.

Defining Event Handlers

There are several techniques you can use to define a handler for an event of a component:

- Select the component, move to the Events page, and either double-click in the white area on the right side of the event or type a name in that area and press the Enter key.
- For many controls, you can double-click on them to perform the default action, which is to add a handler for the `OnClick`, `OnChange`, or `OnCreate` events.

When you want to remove an event handler you have written from the source code of a Delphi application, you could delete all of the references to it. However, a better way is to delete all of the code from the corresponding procedure, leaving only the declaration and the `begin` and `end` keywords. The text should be the same as what Delphi automatically generated when you first decided to handle the event. When you save or compile a project, Delphi removes any empty methods from the source code and from the form description (including the reference to them in the Events page of the Object Inspector). Conversely, to keep an event handler that is still empty, consider adding a comment to it (even simply the `//` characters), so that it will not be removed.

Copying and Pasting Components

An interesting feature of the Form Designer is the ability to copy and paste components from one form to another or to duplicate the component in the form. During this operation Delphi duplicates all the properties and keeps the connected event handlers, and, if necessary, changes the name of the control (which must be unique in each form).

It is also possible to copy components from the Form Designer to the editor and vice versa. When you copy a component to the Clipboard, Delphi also places the textual description there. You can even edit the text version of a component, copy the text to the Clipboard, and then paste it back into the form as a new component. For example, if you place a button on a form, copy it, and then paste it into an editor

(which can be Delphi's own source code editor or any word processor), you'll get the following description:

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'Button1'
  TabOrder = 0
end
```

Now, if you change the name of the object, its caption, or its position, for example, or add a new property, these changes can be copied and pasted back to a form. Here are some sample changes:

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'My Button'
  TabOrder = 0
  Font.Name = 'Arial'
end
```

Copying this description and pasting it into the form will create a button in the specified position with the caption *My Button* in an Arial font.

To make use of this technique, you need to know how to edit the textual representation of a component, what properties are valid for that particular component, and how to write the values for string properties, set properties, and other special properties. When Delphi interprets the textual description of a component or form, it might also change the values of other properties related to those you've changed, and it might change the position of the component so that it doesn't overlap a previous copy. Of course, if you write something completely wrong and try to paste it into a form, Delphi will display an error message indicating what has gone wrong.

You can also select several components and copy them all at once, either to another form or to a text editor. This might be useful when you need to work on a series of similar components. You can copy one to the editor, replicate it a number of times, make the proper changes, and then paste the whole group into the form again.

From Component Templates to Frames

When you copy one or more components from one form to another, you simply copy all of their properties. A more powerful approach is to create a *component template*, which makes a copy of both the properties and the source code of the event handlers. As you paste the template into a new form, by selecting the pseudo-component from the palette, Delphi will replicate the source code of the event handlers in the new form.

To create a component template, select one or more components and issue the Component ➢ Create Component Template menu command. This opens the Component Template Information dialog box (see Figure 1.13) where you enter the name of the template, the page of the Component palette where it should appear, and an icon.

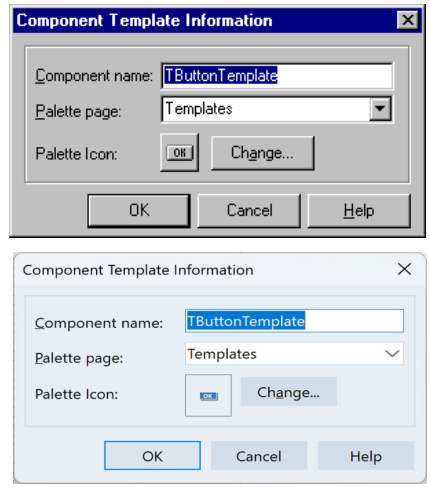


Figure 1.13: The Component Template Information dialog box. Images captured in Delphi 5 and Delphi 12.

By default, the template name is the name of the first component you've selected followed by the word *Template*. The default template icon is the icon of the first component you've selected, but you can replace it with an icon file. The name you give to the component template will be used to describe it in the Component Palette (when Delphi displays the pop-up hint).

All the information about component templates is stored in a single file, DELPHI32.DCT[62], but there is apparently no way to retrieve this information and edit a

62 The file is now *bds.dct,* stored in C:\Users\xxx\AppData\Roaming\Embarcadero\BDS\xxx.

template. What you can do, however, is place the component template in a brand-new form, edit it, and install it again as a component template *using the same name*. This way you can overwrite the previous definition.

note A group of Delphi programmers can share component templates by storing them in a common directory, adding to the Registry the entry `CCLibDir` under the key `Software\Borland\Delphi\5.0\Component Templates`.[63]

Component templates are handy when different forms need the same group of components and associated event handlers. The problem is that once you place an instance of the template in a form, Delphi makes a copy of the components and their code, which is no longer related to the template. There is no way to modify the template definition itself, and it is certainly not possible to make the same change effective in all the forms that use the template. Am I asking too much? Not at all. This is what the new *frames* technology in Delphi 5 does.

A frame is a sort of panel you can work with at design time in a way similar to a form. You simply create a new frame, place some controls in it, and add code to the event handlers. After the frame is ready you can open a form, select the Frame pseudo-component from the Standard page of the Component Palette, and choose one of the available frames (of the current project). After placing the frame in a form, you'll see it as if the components were copied to it. If you modify the original frame (in its own designer), the changes will be reflected in each of the instances of the frame.

You can see a simple example, called Frames1, in Figure 1.14[64]. A screen snapshot doesn't really mean much; you should open the program or rebuild a similar one if you want to start playing with frames.

Like forms, frames define classes, so they fit within the VCL object-oriented model much more easily than Component Templates. Chapter 4 provides an in-depth look at the VCL and includes a more detailed description of frames. As you might imagine from this short introduction, frames are a powerful new technique.

63 The registry key is still exists, but I'm not sure if this undocumented configuration works today.

64 Frames work today, for both VCL and FireMonkey, even if I haven't captured a new image for Figure 1.14.

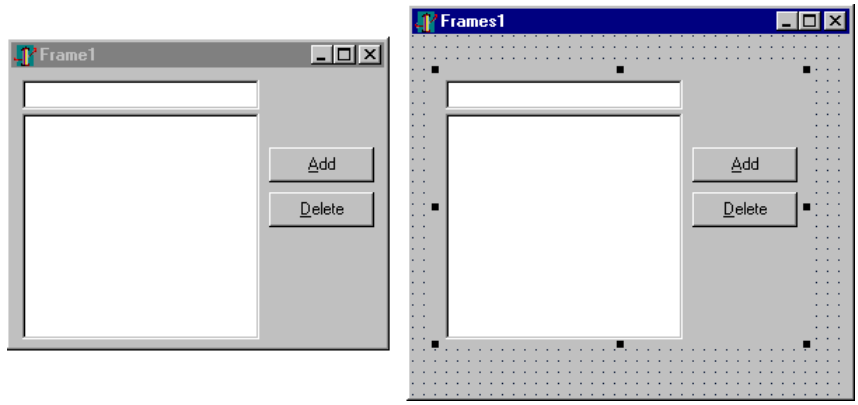


Figure 1.14: The Frames1 example demonstrates the use of frames. The frame (on the left) and its instance inside a form (on the right) are kept in synch.

Managing Projects

One of the new features of the Delphi 4 IDE was the multi-target Project Manager (View ➢ Project Manager). The Project Manager works on a project *group*, which can have one or more projects under it. For example, a project group can include a DLL and an executable file, or multiple executable files.

In Figure 1.15 you can see the Project Manager with the project group for the examples of the current chapter. As you can see, the Project Manager is based on a tree view, which shows the hierarchical structure of the project group, the projects, and all of the forms and units that make up each project. You can use the simple toolbar and the more complex shortcut menus of the Project Manager to operate on it. The shortcut menu is context-sensitive; its options depend on the selected item. There are menu items to add a new or existing project to a project group, to compile or build a specific project, or to open a unit.

Of all the projects in the group only one is active, and this is the project you operate upon when you select a command such as Project ➢ Compile. The Project pull-down of the main menu has two commands you can use to compile or build all the projects of the group. (Strangely enough, these commands are not available in the shortcut menu of the Project Manager for the project group.[65]) When you have mul-

65 They were not, now they've been added. The Project Manager has seen many extensions over the years, but its core behavior is still what I described here (see also Figure 1.15).

tiple projects to build, you can set a relative order by using the Build Sooner and Build Later commands. These two commands basically rearrange the projects in the list.

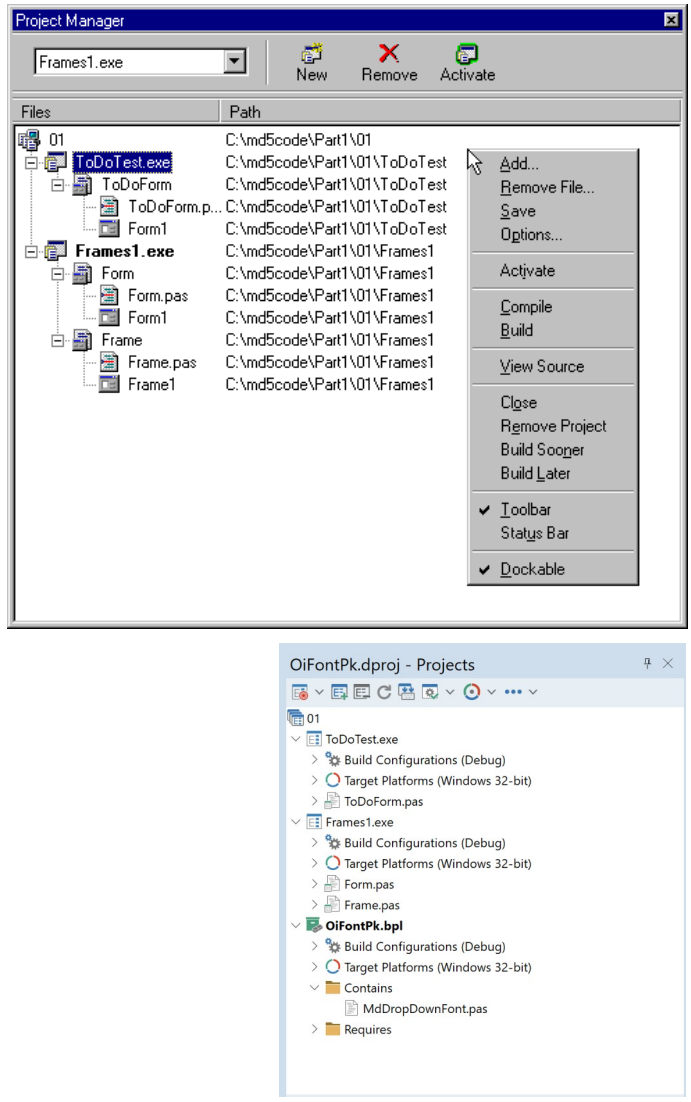


Figure 1.15: Delphi's multi-target Project Manager. Images captured in Delphi 5 and Delphi 12.

Delphi 5 adds some features to the Project Manager. You can now drag source code files from Windows folders or Windows Explorer onto a project in the Project Manager window to add them to that project. Unfortunately, you cannot drag an existing

project or package file to add it to the entire project group. You can also drag from one project to another of the same project group.

Another big advantage is that the Project Manager automatically selects as current project the one you are working with, for example, opening a file. You can easily see which project is selected and change it by using the combo box on the top of the form[66].

note Besides adding Pascal files and projects, you can add Windows resource files to the Project Manager; they are compiled along with the project. Simply move to a project, select the Add shortcut menu, and choose *Resource file (*.rc)* as the file type. This resource file will be automatically bound to the project, even without a corresponding `$R` directive.

Delphi saves the project groups with the new `.BPG` extension, which stands for Borland Project Group[67]. This feature comes from C++Builder and from past Borland C++ compilers, a history that is clearly visible as you open the source code of a project group, which is basically that of a makefile in a C/C++ development environment[68].

Project Options

The Project Manager doesn't provide a way to set the options of two different projects at one time. What you can do instead is invoke the Project Options dialog from the Project Manager for each project[69]. The first page of Project Options (Forms) lists the forms that should be created automatically at program startup and the forms that are created manually by the program. The next page (Application) is used to set the name of the application and the name of its Help file, and to choose its icon. Other Project Options choices relate to the Delphi compiler and linker, version information, and the use of run-time packages.

66 That combo box is still available in the form of a drop down split button, the first button of the Project Manager toolbar, with the symbol of a target superimposed.

67 This is not the case any more. Project and project groups are now XML files in the MSBuild format, as this is the tool for building applications since Delphi 2007, as detailed in my "Delphi 2007 Handbook". I took the freedom of removing the project group files listed in the original book, as they are totally useless in today's Delphi.

68 The format was later changed to the MSBUILD XML format. You can still open a Delphi 5 project group file today, although the IDE will ask you to save it in the current format.

69 The Project Options dialog still exists, but the sequence of pages has changed, with many more features available.

There are two ways to set compiler options. One is to use the Compiler page of the Project Options dialog. The other is to set or remove individual options in the source code with the `{$X+}` or `{$X-}` commands, where you'd replace `X` with the option you want to set. This second approach is more flexible, since it allows you to change an option only for a specific source-code file, or even for just a few lines of code. The source-level options override the compile-level options.

All of the Project Options are saved automatically with the project, but in a separate file with a .DOF extension[70]. This is a text file you can easily edit. You should not delete this file if you have changed any of the default options. Delphi also saves the compiler options in another format in a CFG file, for command line compilation.

Another alternative for saving compiler options is to press Ctrl+O+O (press the O key twice while keeping Ctrl pressed). This inserts, at the top of the current unit, compiler directives that correspond to the current project options, as in the following listing[71]:

```
{$A+,B-,C+,D+,E-,F-,G+,H+,I+,J+,K-,L+,M-,N+,O+,P+,Q-,
R-,S-,T-,U-,V+,W-,X+,Y+,Z1}

{$MINSTACKSIZE $00004000}

{$MAXSTACKSIZE $00100000}

{$IMAGEBASE $00400000}

{$APPTYPE GUI}
```

Compiling and Building Projects

There are several ways to compile a project. If you run it (by pressing F9 or clicking the Run toolbar icon), Delphi will compile it first. When Delphi compiles a project, it compiles only the files that have changed.

---

70 Project options files are gone as well, and so are their command line counterparts (as compilation now follows the same steps both from the command line and from the IDE. In current versions of Delphi, the project settings are saved in project files or in build configurations, can be shared among projects, have release and build variations, and much more. Still the core application settings haven't changed much. Coverage of project settings management is my Delphi Handbooks, as they were extended from version to version.

71 There are now many more lines inserted, but the keyboard shortcut and the overall concept remain the same.

If you select Compile ➢ Build All instead[72], every file is compiled, even if it has not changed. You should only need this second command infrequently, since Delphi can usually determine which files have changed and compile them as required. The only exception is when you change some project options. In this case you have to use the Build All command to put the new options into effect.

To build a project, Delphi first compiles each source code file, generating a Delphi compiled unit (DCU). (This step is performed only if the DCU file is not already up to date.) The second step, performed by the linker, is to merge all the DCU files into the executable file, optionally with compiled code from the VCL library (if you haven't decided to use packages at run time). The third step is binding into the executable file any optional resource files, such as the RES file of the project, which hosts its main icon, and the DFM files of the forms. You can better understand the compilation steps and follow what happens during this operation if you enable the Show Compiler Progress option (in the Preferences page of the Environment Options dialog box).

note Delphi doesn't always properly keep track of when to rebuild units based on other units you've modified. This is particularly true for the cases (and there are many) in which user intervention confuses the compiler logic. For example, renaming files, modifying source files outside the IDE, copying older source files or DCU files to disk, or having multiple copies of a unit source file in your search path can break the compilation. Every time the compiler shows some strange error message, the first thing you should try is the Build All command to resynchronize the make feature with the current files on disk.

The Compile command can be used only when you have loaded a project in the editor. If no project is active and you load a Pascal source file, you cannot compile it. However, if you load the source file *as if it were a project*, that will do the trick and you'll be able to compile the file. To do this, simply select the Open Project toolbar button and load a PAS file. Now you can check its syntax or compile it, building a DCU.[73]

I've mentioned before that Delphi allows you to use run-time packages, which affect the distribution of the program more than the compilation process. Delphi packages are dynamic link libraries (DLLs) containing Delphi components. By using packages, you can make an executable file much smaller. However, the program won't

72 The menu command is now Project | Build.

73 In recent versions of Delphi (probably since MSBuild was introduced) this trick doesn't work any more. There is apparently no way to compile an individual source code file outside of a project in the IDE. However, you can easily compile a single Pascal source code files with the command line compiler.

run unless the proper dynamic link libraries (such as vcl50.bpl, which is quite large) are available on the computer where you want to run the program.

If you add the size of this dynamic library to that of the small executable file, the total amount of disk space required by the apparently smaller program built with run-time packages is much larger than the space required by the apparently bigger stand-alone executable file. Of course if you have multiple applications on a single system, you'll end up saving a lot, both in disk space and memory consumption at run time. The use of packages is often but not always recommended. I'll discuss all the implications of packages in detail in Chapter 13, where we'll build some packages, and in Chapter 14, which is devoted to DLLs and packages.

note You don't have to use the stock vcl50.bpl package if you only need a small set of VCL units. You can create your own mini-VCL package, as long as you don't call it vcl50.bpl.

In both cases, Delphi executable files are extremely fast to compile, and the speed of the resulting application is comparable to that of a C or C++ program. Delphi compiled code runs at least five times faster than the equivalent code in interpreted or "semicompiled" tools[74].

Conditional Compilation for Different versions of Delphi

You can test the `VER130` define to check whether you are compiling with Delphi 5 or an earlier version. This can be useful if you want to compile the same program with different versions of Delphi and make minor changes to the source code in each of the versions. If you want to add some specific Delphi 5 code, you can write that code as follows:

```
{$IFDEF VER130}
  // Delphi 5 specific code
{$ENDIF}
```

Each of the past versions of the Delphi included a specific define, so you can write a complex statement to provide alternative coding solutions for different Delphi versions. The numbering scheme starts from the last version of Pascal compiler from

74 I'm not sure if this specific number ("five times faster", which I assume was in reference to Visual Basic) makes sense today, with many alternatives between compiled and interpreted code. Delphi programs are still native and remain fast. Some of the options used today for desktop development, like JavaScript, are clearly in a different league, both in terms of slower performance and in terms of the complex deployment dependencies.

Borland before Delphi, Borland Pascal with Object version 7, and also includes the versions of the Pascal compiler included in Borland C++Builder[75]:

- `VER80` for Delphi 1
- `VER90` for Delphi 2
- `VER93` for C++Builder 1
- `VER100` for Delphi 3
- `VER110` for C++Builder 3
- `VER120` for Delphi 4
- `VER125` for C++Builder 4

Exploring a Project[76]

Past versions of Delphi included an Object Browser, which you could use when a project was compiled to see a hierarchical structure of its classes and to look for its symbols and the source code lines where they are referenced. Delphi 5 includes a similar but enhanced tool, with a new name—Project Explorer. Like the Code Explorer, it is updated automatically as you type, without recompiling the project.

The Project Explorer retains from the Object Browser the main structure of Classes, Units, and Globals, but it lets you choose whether to look only for symbols defined within your project or for those from both your project and the VCL. You can see an example with project symbols only in Figure 1.16.

75 The list has been added to the Delphi docwiki, and it can be found at docwiki.embarcadero.-com/RADStudio/en/Compiler_Versions. The Delphi 12 compiler defines VER360.

76 This entire feature isn't part of recent versions of Delphi, so you can skip this section.

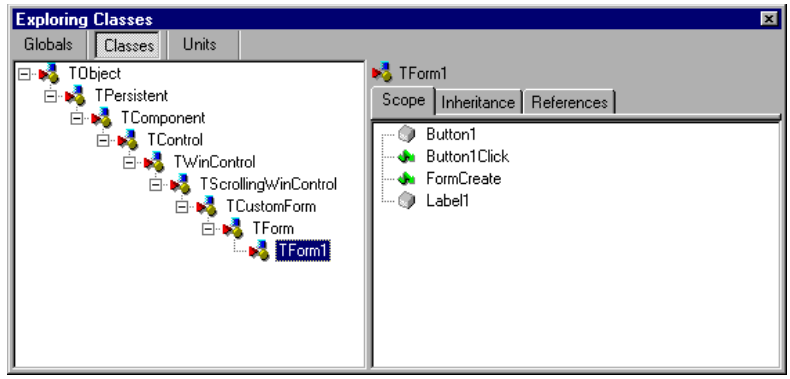


Figure 1.16: The Project Explorer, a completely updated Object Browser[77]

You can change the settings of this Explorer and those of the Code Explorer in the Explorer page of the Environment Options (see Figure 1.5) or by selecting the Properties command in the shortcut menu of the Project Explorer. Some of the Explorer categories you see in this window are specific to the Project Explorer, others relate to both tools.

Additional and External Delphi Tools

Besides the IDE, when you install Delphi you get other, external tools. Some of them, such as the Database Desktop, the Package Collection Editor (PCE.EXE), and the Image Editor (ImagEdit.EXE), are available from Tools menu of the IDE. In addition, the Client/Server edition has a link to the SQL Monitor (SqlMon.EXE)[78].

Other tools that are not directly accessible from the IDE include many command-line tools you can find in the `Bin` directory of Delphi. For example, there is a command-line Delphi compiler (DCC.EXE), a Borland resource compiler (BRC32.EXE and BRCC32.EXE), and an executable viewer (TDump.EXE).[79]

---

77 As mentioned earlier, this feature is no longer available in recent versions of Delphi.

78 Most of these tools are now gone. The Tools menu in Delphi 12 includes by default the *Bitmap Style Designer*, the *FireDAC Explorer*, the *FireDAC Monitor*, the *REST Debugger*, the *XML Mapper*, and – in some editions – the *RAD Server Console*.

79 These low level tools, instead, are still available today, even if with some differences. There are multiple Delphi compilers, for example.

Finally, some of the sample programs that ship with Delphi are actually useful tools that you can compile and keep at hand. I'll discuss some of these tools in the book, as needed. Here are a few of the useful and higher-level tools[80]:

- **WinSight** (WS.EXE) is a Windows "message spy" program available in the `Bin` directory.[81]
- **Database Explorer** can be activated from the Delphi IDE or as a stand-alone tool, using the DBExplor.EXE program of the `Bin` directory.[82]
- **Convert** (Convert.EXE) is a command-line tool you can use to convert `DFM` files into the equivalent textual description and vice versa.
- **Turbo Grep** (Grep.EXE) is a command-line search utility, much faster than the embedded Find in Files mechanism but not so easy to use.
- **Turbo Register Server** (TRegSvr.EXE) is a tool you can use to register ActiveX libraries and COM servers. The source code of this tool is available under Demos/ActiveX/TRegSvr.[83]
- **Resource Explorer** is a powerful resource viewer (but not a full-blown resource editor) you can find under Demos/ResXplor[84].
- The Delphi 5 CD also includes a separate installation for **Resource Workshop[85]**. This is an old 16-bit resource editor that can also manage Win32 resource files. It was formerly included in Borland C++ and Pascal compilers for Windows, and it was much better than the standard Microsoft resource editors then available. Although its user interface hasn't been updated and it doesn't handle long file names, this tool can still be very useful for building custom or special resources. It also lets you explore the resources of existing executable files. You'll find more information about Windows resources and the use of Resource Workshop in Chapter 19.

---

80 *Convert*, *Grep*, and *TRegSvr* still exist today. For the other tools, see the respective footnotes.

81 The *WinSight* tool is not available any more. There are similar free utilities for Windows.

82 *DbExplorer* has been replaced by equivalent FireDAC utilities, some of which are listed in the Tools menu, as covered in a previous footnote.

83 The tool is still available, but not its source code.

84 This demo is no longer part of the core product demos.

85 Not only there is no CD, but also no version of the *Resource Explorer* available with the product. There are similar free utilities for Windows.

The Files Produced by the System

Delphi produces a number of files for each project, and you should know what they are and how they are named. There are basically two elements that have an impact on how files are named: the names you give to a project and its units, and the predefined file extensions used by Delphi. Table 1.1 lists the extensions of the files you'll find in the directory where a Delphi project resides[86]. The table also shows when or under what circumstances these files are created and their importance for future compilations. Extensions that are new to Delphi 5 are marked in bold.

Table 1.1: Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.BMP, .ICO, .CUR	Bitmap, icon, and cursor files: standard Windows files used to store bitmapped images.	Development: Image Editor	Usually not, but they might be needed at run time and for further editing.
.BPG	Borland Project Group[87]: the files used by the new multiple-target Project Manager. It is a sort of makefile.	Development	Required to recompile all the projects of the group at once.
.BPL	Borland Package Library: a DLL including VCL components to be used by the Delphi environment at design time or by applications at run time. (These files used a .DPL extension in Delphi 3.)	Compilation: Linking	You'll distribute packages to other Delphi developers and, optionally, to end-users.
.CAB	The Microsoft Cabinet compressed-file format used for Web deployment by Delphi. A CAB file can store multiple compressed files.	Compilation	Distributed to users.
.CFG	Configuration file with project options. Similar to the DOF files.	Development	Required only if special compiler options have been set.
.DCP	Delphi Component Package: a file with symbol information for the code that was compiled	Compilation	Required when you use packages. You'll distribute it only to other developers along with DPL files.

86 Most of these file types are still used, but not all of them. I haven't added here new files available, including the new project files in MSBUILD format, but only added a few comments to the original list.

87 This files is not used any more, replaced by Project Group files, with the .groupproj extension.

	into the package. It doesn't include compiled code, which is stored in DCU files.		
.DCU	Delphi Compiled Unit: the result of the compilation of a Pascal file.	Compilation	Only if the source code is not available. DCU files for the units you write are an intermediate step, so they make compilation faster.
.DFM	Delphi Form File: a binary file with the description of the properties of a form (or a data module) and of the components it contains.	Development	Yes. Every form is stored in both a PAS and a DFM file.
.~DF[88]	Backup of Delphi Form File (DFM).	Development	No. This file is produced when you save a new version of the unit related to the form and the form file along with it.
.DFN[89]	**Support file for the Integrated Translation Environment (there is one DFN file for each form and each target language).**	**Development (ITE)**	**Yes (for ITE). These files contain the translated strings that you edit in the Translation Manager.**
.DLL	Dynamic Link Library: another version of an executable file.	Compilation: Linking	See .EXE.
.DOF[90]	Delphi Option File: a text file with the current settings for the project options.	Development	Required only if special compiler options have been set.
.DPK	Delphi Package: the project source code file of a package.	Development	Yes.
.DPR	Delphi Project file. (This file actually contains Pascal source code.)	Development	Yes.
.~DP	Backup of the Delphi Project file (. DPR).	Development	No. This file is generated automatically when you save a new version of a project file.
.DSK	Desktop file: contains information about the position of the Delphi windows, the files open in the editor, and other Desktop settings.	Development	No. You should actually delete it if you copy the project to a new directory.

88 Backup files are now saved in sequence under the __history sub-folder of the project source code folder and they use a different logic. The same is true for all of the backup files listed in this table.

89 This format still exists but the translation support isn't installed any more as part of Delphi. The same is true for other file formats associated with the old translation system. The feature can currently be installed using the GetIt package manager.

90 Project options are now part of the .dproj project file.

.DSM[91]	Delphi Symbol Module: stores all the browser symbol information.	Compilation (but only if the Save Symbols option is set)	No. Object Browser uses this file, instead of the data in memory, when you cannot recompile a project.
.DTI[92]	**Design Time Information, used by the new Data Module Designer**	**Development**	**No. This file stores "design-time only" information, not required by the resulting program but very important for the programmer.**
.EXE	Executable file: the Windows application you've produced	Compilation: Linking	No. This is the file you'll distribute. It includes all of the compiled units, forms, and resources.
.HTM	Or `.HTML`, for HyperText Markup Language: the file format used for Internet Web pages	Web deployment of an ActiveForm	No. This is not involved in the project compilation.
.LIC	The license files related to an OCX file.	ActiveX Wizard and other tools	No. It is required to use the control in another development environment.
.OBJ	Object (compiled) file, typical of the C/C++ world.	Intermediate compilation step, generally not used in Delphi	It might be required to merge Delphi with C++ compiled code in a single project.
.OCX	OLE Control eXtension: a special version of a DLL, containing ActiveX controls or forms.	Compilation: Linking	See .EXE.
. PAS	Pascal file: the source code of a Pascal unit, either a unit related to a form or a stand-alone unit.	Development	Yes.
.~PA	Backup of the Pascal file (.PAS).	Development	No. This file is generated automatically by Delphi when you save a new version of the source code.
.RES, .RC	Resource file: the binary file associated with the project and usually containing its icon. You can add other files of this type to a project. When you create custom resource files you might use also the textual format, .RC.	Development Options dialog box. The ITE (Integrated Translation Environment) generates resource files with special comments.	Yes. The main RES file of an application is rebuilt by Delphi according to the information in the Application page of the Project Options dialog box.
.RPS	**Translation Repository (part of the Integrated Translation**	**Development (ITE)**	**No. Required to manage the translations.**

91 This file format and the associated feature don't exist any more.

92 This feature is also long gone, with the matching file format.

	Environment).		
.TLB	Type Library: a file built automatically or by the Type Library Editor for OLE server applications.	Development	This is a file other OLE programs might need.
.TODO	**To-do list file, holding the items related to the entire project.**	**Development**	**No. This file hosts notes for the programmers.**
.UDL	**Microsoft Data Link**	**Development**	**Used by ADO to refer to a data provider. Similar to an alias in the BDE world (see Chapter 12).**

Besides the files generated during the development of a project in Delphi, there are many others generated and used by the IDE itself. In Table 1.2 I've provided a short list of extensions worth knowing about. Most of these files are in proprietary and undocumented formats, so there is little you can do with them.

Table 1.2: Selected Delphi IDE Customization File Extensions[93]

Extension	File Type
.DCI	Delphi Code Templates
.DRO	Delphi's Object Repository (The repository should be modified with the Tools ➢ Repository command.)
.DMT	Delphi Menu Templates
.DBI	Database Explorer Information
.DEM	Delphi Edit Mask (Files with country-specific formats for edit masks)
.DCT	Delphi Component Templates
.DST	Desktop settings file (one for each desktop setting you've defined)

Looking at Source Code Files

I've just listed some files related to the development of a Delphi application, but I want to spend a little time to cover their actual format. The fundamental Delphi files are Pascal source code files, which are plain ASCII text files. The bold, italic, and colored text you see in the editor depend on syntax highlighting, but they are not saved with the file. It is worth noting that there is one single file for the whole code of the form, not just small code fragments.

93 Many of these files don't exist any more. Desktops settings and component template files are still used.

note In the listings in the book I've tried to match the bold syntax highlighting of the editor for keywords and the italic for strings and comments.

For a form, the Pascal file contains the form class declaration and the source code of the event handlers. The values of the properties you set in the Object Inspector are stored in a separate form description file (with a `.DFM` extension). The only exception is the `Name` property, which is used in the form declaration to refer to the components of the form.

The DFM file is a binary and in Delphi 5 can be saved either as a plain text file or in the traditional Windows Resource format. You can set the default format you want to use for new projects in the Preferences page of the Environment Options dialog box, and you can toggle the format of individual forms with the Text DFM command of a form's shortcut menu. A plain-text editor can read only the text version. However, you can load DFM files of both types in the Delphi editor, which will, if necessary, first convert them into a textual description. The simplest way to open the textual description of a form (whatever the format) is to select the View As Text command on the shortcut menu in the Form Designer. This closes the form, saving it if necessary, and opens the DFM file in the editor. You can later go back to the form using the View As Form command on the shortcut menu in the editor window.

You can actually edit the textual description of a form, although this should be done with extreme care. As soon as you save the file, it will be turned back into a binary file. If you've made incorrect changes, compilation will stop with an error message and you'll need to correct the contents of your DFM file before you can reopen the form. For this reason, you shouldn't try to change the textual description of a form manually until you have a good knowledge of Delphi programming.

note In the book I'll often show you excerpts of DFM files. With most of these excerpts, I'll only be showing the most relevant components or properties; generally, I will have removed the positional properties, the binary values, and other lines providing little useful information.

In addition to the two files describing the form (PAS and DFM), a third file is vital for rebuilding the application. This is the Delphi project file (DPR), which is another Pascal source code file. This file is built automatically, and you seldom need to change it manually. You can see this file with the View ➢ Project Source menu command.

Some of the other, less relevant, files produced by the IDE use the structure of Windows INI files, in which each section is indicated by a name enclosed in square brackets. For example, this is a fragment of an option file (DOF)[94]:

```
[Compiler]
A=1
B=0
ShowHints=1
ShowWarnings=1

[Linker]
MinStackSize=16384
MaxStackSize=1048576
ImageBase=4194304

[Parameters]
RunParams=
HostApplication=
```

The same structure is used by the Desktop files (DSK), which store the status of the Delphi IDE for the specific project, listing the position of each window. Here is a small excerpt:

```
[MainWindow]
Create=1
Visible=1
State=0
Left=2
Top=0
Width=800
Height=97
```

note A lot of information related to the status of the Delphi environment is saved in the Windows Registry, as well as in DSK and other files. I've already indicated a few special undocumented entries of the Registry you can use to activate specific features. You should explore the HKEY_CURRENT_USER/Software/Borland/Delphi/5.0[95] section of the Registry to examine all the setting of the Delphi IDE (including all those you can modify with the Project Options and the Environment Options dialog boxes, as well as many others).

94 As mentioned earlier, option files content is now part of the .drpoj project file.

95 This is still true, although the location in the registry is now *HKEY_CURRENT_USER\Software\Embarcadero\BDS\xx.0*.

The Object Repository

Delphi has several menu commands you can use to create a new form, a new application, a new data module, a new component, and so on. These commands are located in the File menu and in other pull-down menus. What happens if you simply select File ➢ New[96]? Delphi opens the Object Repository, which is used to create new elements of any kind: forms, applications, data modules, thread objects, libraries, components, automation objects, and more.

The New dialog box (shown in Figure 1.17) has a number of pages, hosting all the new elements you can create, existing forms and projects stored in the Repository, Delphi wizards, and the forms of the current project (for visual form inheritance). The pages and the entries in this tabbed dialog box depend on the specific version of Delphi, so I won't list them here.

note The Object Repository has a shortcut menu that allows you to sort its items in different ways (by name, by author, by date, or by description) and to show different views (large icons, small icons, lists, and details). The Details view gives you the description, the author, and the date of the tool, information that is particularly important when looking at wizards, projects, or forms that you've added to the Repository.

The simplest way to customize the Object Repository is to add new projects, forms, and data modules as templates. You can also add new pages and arrange the items on some of them (not including the New and "current project" pages). Adding a new template to Delphi's Object Repository is as simple as using an existing template to build an application. When you have a working application you want to use as a starting point for further development of similar programs, you can save the current status to a template, ready to use later on. Simply use the Project ➢ Add to Repository command, and fill in its dialog box.

Just as you can add new project templates to the Object Repository, you can also add new form templates. Simply move to the form that you want to add and select the Add To Repository command of its shortcut menu. Then indicate the title, description, author, page, and icon in its dialog box.

You might want to keep in mind that as you copy a project or form template to the repository and then copy it back to another directory, you are simply doing a copy and paste operation. This isn't much different than copying the files manually.

96 Beside the fact that the menu command is now File | New | Other and the totally different UI, the role, content, and behavior of the Object Repository remains very similar to what's described here.

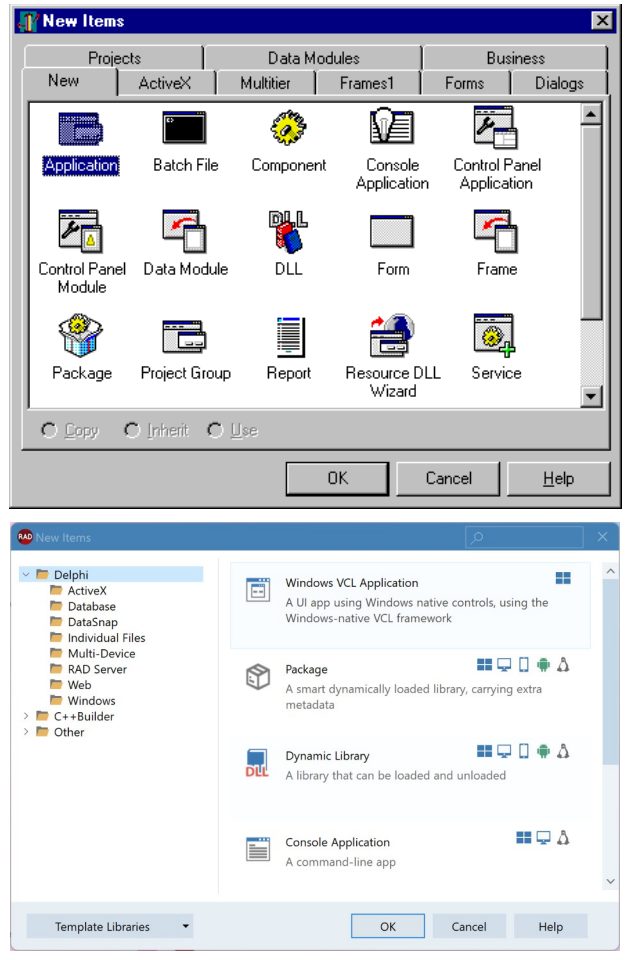


Figure 1.17: The first page of the New dialog box, generally known as the "Object Repository". Images captured in Delphi 5 and Delphi 12.

The Empty Project Template

When you start a new project, it automatically opens a blank form, too. If you want to base a new project on one of the form objects or Wizards, this is not what you want, however. To solve this problem, you can add an Empty Project template to the Gallery.

The steps required to accomplish this are simple[97]:

1. Create a new project as usual.
2. Remove its only form from the project.
3. Add this project to the templates, naming it *Empty Project*.

When you select this project from the Object Repository, you gain two advantages: You have your project without a form, and you can pick a directory where the project template's files will be copied. There is also a disadvantage—you have to remember to use the File ➢ Save Project As command to give a new name to the project, because saving the project any other way automatically uses the default name in the template.

To further customize the Repository, you can use the Tools ➢ Repository command. This opens the Object Repository dialog box, which you can use to move items to different pages, to add new elements, or to delete existing ones. You can even add new pages, rename or delete them, and change their order. An important element of the Object Repository setup is the use of defaults:

- Use the New Form check box below the list of objects to designate a form as the one to be used when a new form is created (File ➢ New Form).
- The Main Form check box indicates which type of form to use when creating the main form of a new application (File ➢ New Application) when no special New Project is selected.
- The New Project check box, available when you select a project, marks the default project that Delphi will use when you issue the File ➢ New Application command.

Only one form and only one project in the Object Repository can have each of these three settings marked with a special symbol placed over its icon. If no project is selected as New Project, Delphi creates a default project based on the form marked as Main Form. If no form is marked as the main form, Delphi creates a default project with an empty form.

When you work on the Object Repository, you work with forms and modules saved in the `OBJREPOS` sub-directory of the Delphi main directory[98]. At the same time, if you use a form or any other object directly without copying it, then you end up having some files of your project in this directory. It is important to realize how the

97 I haven't actually tried these steps, but I assume they still work.

98 Current folder is still under the application folder, at C:\Program Files (x86)\Embarcadero\Studio\xx.0\ObjRepos

Repository works, because if you want to modify a project or an object saved in the Repository, the best approach is to operate on the original files, without copying data back and forth to the Repository.

Installing New DLL Wizards

Technically, new wizards come in two different forms: They may be part of components or packages, or they may be distributed as stand-alone DLLs. In the first case, they would be installed the same way you install a component or a package. When you've received a stand-alone DLL, you should add the name of the DLL in the Windows Registry under the key Software\Borland\Delphi\5.0\Experts[99]. Simply add a new string key under this key, choose a name you like (it doesn't really matter what it is), and use as text the path and filename of the wizard DLL. You can look at the entries already present under the `Experts` key to see how the path should be entered.

What's Next?

This chapter has presented an overview of the new and more advanced features of Delphi 5 programming environment, including a number of tips and suggestions about some lesser-known features that were already available in previous Delphi versions. I didn't provide a step-by-step description of the IDE, partly because it is generally simpler to start *using* Delphi than it is to read about how to use it. Moreover, there is a detailed Help file describing the environment and the development of a new simple project; and you might already have some exposure to one of the past versions of Delphi or a similar development environment.

We haven't finished covering new features of Delphi 5 IDE, though. I'll discuss the new Data Module Designer in Chapter 10, new debugging features in Chapter 18, and TeamSource and the Integrated Translation Environment in Chapter 19[100]. But now we are ready to spend the next three chapters looking into the Object Pascal language and the VCL library. Then, in Part II, we'll start focusing on the user interface of applications and using the components available in Delphi.

---

99 The registry key is now HKEY_CURRENT_USER\Software\Embarcadero\BDS\xx.0\Experts

100 As already mentioned, these two features don't exist any more (or are not officially supported any more).

Chapter 2: Object-Oriented Programming In Delphi

Most modern programming languages support *object-oriented programming* (OOP). OOP languages are based on three fundamental concepts: encapsulation (usually implemented with classes), inheritance, and polymorphism (or late binding).

You can write Delphi applications even without knowing the details of Object Pascal. As you create a new form, add new components, and handle events, Delphi prepares most of the related code for you automatically. But knowing the details of

the language and its implementation will help you to understand precisely what Delphi is doing and to master the language completely.

A single chapter doesn't allow space for a full introduction to the principles of object-oriented programming and the Object Pascal language[101]. Instead, I will outline the key OOP features of the language and show how they relate to everyday Delphi programming. Even if you don't have a precise knowledge of OOP, the chapter will introduce each of the key concepts so that you won't need to refer to other sources.

note If you don't know the basics of the Pascal language (which is not covered in this book), you can refer to the online electronic version of the text *Essential Pascal* at www.marcocantu.com. The language has not changed significantly from Delphi 4 to Delphi 5.[102]

Introducing Classes and Objects

Class and *object* are two terms commonly used in Object Pascal and other OOP languages. However, because they are often misused, let's be sure we agree on their definitions. A *class* is a user-defined data type, which has a state (its representation) and some operations (its behavior). A class has some internal data and some methods, in the form of procedures or functions, and usually describes the generic characteristics and behavior of a number of similar objects.

An *object* is an instance of a class, or a variable of the data type defined by the class. Objects are *actual* entities. When the program runs, objects take up some memory for their internal representation. The relationship between object and class is the same as the one between variable and type.

To declare a new class data type in Object Pascal, with some local data fields and some methods, use the following syntax:

```
type
  TDate = class
    Month, Day, Year: Integer;
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean;
```

101 I've published a book covering the Delphi Object Pascal language in detail. It's called "Object Pascal Handbook" and it's available in print on Amazon. See www.marcocantu.com/objectpascalhandbook/ for more information.

102 My ebook Essential Pascal remains available for free.

```
  end;
```

The function and the procedure declared above should be fully defined in the implementation portion of the same unit, including the class declaration. You can let Delphi generate a skeleton of the definition of the methods by using the Class Completion feature of the editor (simply press Ctrl+C while the cursor is within the class definition). You can tell the methods are part of the `TDate` class by class-name prefixing (using a dot in between), as in the following code:

```
procedure TDate.SetValue(m, d, y: Integer);
begin
  Month := m;
  Day := d;
  Year := y;
end;

function TDate.LeapYear: Boolean;
begin
  // call IsLeapYear in SysUtils.pas[103]
  Result := IsLeapYear (Year);
end;
```

note The convention in Delphi is to use the letter *T* as a prefix for the name of every class you write and every other type (*T* stands for *Type*). This is just a convention—to the compiler, *T* is just a letter like any other—but it is so common that following it will make your code easier to understand. In the book I'll try to stick with this convention.

Once the class has been defined, we can create an object and use it as follows:

```
var
  ADay: TDate;
begin
  // create
  ADay := TDate.Create;
  // use
  ADay.SetValue (1, 1, 2000);
  if ADay.LeapYear then
    ShowMessage ('Leap year: ' + IntToStr (ADay.Year));
  // destroy
  ADay.Free;
end;
```

The notation used is nothing unusual, but it is powerful. We can write a complex function (such as `LeapYear`) and then access its value for every `TDate` object as if it were a primitive data type. Notice that `ADay.LeapYear` is an expression similar to

103 Using today's notation, the unit name is now *System.SysUtils.pas*.

`ADay.Year`, although the first is a function call and the second a direct data access. As we'll see in the next chapter, the notation used by Object Pascal to access properties is again the same.

Delphi's Object Reference Model

In some OOP languages, declaring a variable of a class type creates an instance of that class. Object Pascal, instead, is based on an *object reference model*. The idea is that each variable of a class type, such as `ADay` in the code fragment above, does not hold the value of the object. Rather, it contains a reference, or a *pointer,* to indicate the memory location where the object has been stored.

note The object reference model is powerful yet easier to use than other models. Other OOP languages use similar models, notably Eiffel and Java[104]. In my opinion, adopting this model was one of the best design decisions made by the Delphi development team at Borland.

The only problem with this approach is that when you declare a variable, you don't create an object in memory, you only reserve the memory location for a reference to an object. Object instances must be created manually, at least for the objects of the classes you define. Instances of a component you place on a form are built automatically by Delphi.

To create an instance of an object, we can call its `Create` method, which is a constructor. As you can see in the last code fragment, the constructor is applied to the class, not to the object. Where does the `Create` method come from? It is a constructor of the class `TObject`, from which all the other classes inherit. Once you have created an object and you've finished using it, you need to dispose of it (to avoid filling up memory you don't need any more, which causes what is known as a *memory leak*). This can be accomplished by calling the `Free` method (yet another method of the `TObject` class), as demonstrated in the previous listing. As long as you create objects when you need them and free them when you're finished with them, the object reference model works without a glitch.

Private, Protected, and Public

A class can have any amount of data and any number of methods. However, for a good object-oriented approach, data should be hidden, or *encapsulated,* inside the

104 Also C# uses the same machanism.

class using it. When you access a date, for example, it makes no sense to change the value of the day by itself. In fact, changing the value of the day might result in an invalid date, such as February 30. Using methods to access the internal representation of an object limits the risk of generating erroneous situations, as the methods can check whether the date is valid and refuse to modify the new value if it is not. Encapsulation is important because it allows the class writer to modify the internal representation in a future version.

The concept of encapsulation is quite simple: just think of a class as a "black box" with a small, visible portion. The visible portion, called the *class interface,* allows other parts of a program to access and use the objects of that class. However, when you use the objects, most of their code is hidden. You seldom know what internal data the object has, and you usually have no way to access the data directly. Of course, you are supposed to use methods to access the data, which is shielded from unauthorized access. This is the object-oriented approach to a classical programming concept known as *information hiding*.

Object Pascal has three access specifiers: `private`, `protected`, and `public`[105]. A fourth one, `published`, will be discussed in the next chapter. Here are the three basic ones:

- The `private` directive denotes fields and methods of a class that are not accessible outside the unit (the source code file) that declares the class.
- The `public` directive denotes fields and methods that are freely accessible from any other portion of a program as well as in the unit in which they are defined.
- The `protected` directive is used to indicate methods and fields with limited visibility. Only the current class and its subclasses can access `protected` elements. We'll discuss this keyword again in the "Protected Fields and Encapsulation" section.

Generally, the fields of a class should be `private`; the methods are usually `public`. However, this is not always the case. Methods can be `private` or `protected` if they are needed only internally to perform some partial computation. Fields can be `protected` or `public` when you want an easy and direct access and you are fairly sure that their type definition is not going to change.

---

105 Two further access specifiers, *strict private* and *strict protected* were added to match the behavior of other OOP languages having no special rules for classes declared within the same unit or source code file. The strict versions of the access specifiers provide an even more robust encapsulation, but remain rarely used by Delphi developers.

note Instead of having `public` fields, you should generally use properties, as we'll see in detail in the next chapter. Properties are an extension to the encapsulation mechanism of other OOP languages and are very important in Object Pascal.

Access specifiers only restrict code outside your unit from accessing certain members of classes declared in the interface section of your unit. This means that if two classes are in the same unit, there is no protection for their private fields. Only by placing a class in the interface portion of a unit will you limit the visibility from classes and functions in other units to the public method and fields of the class.

As an example, consider this new version of the `TDate` class:

```
type
  TDate = class
  private
    Month, Day, Year: Integer;
  public
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

In this version, the fields are now declared to be `private`[106], and there are some new methods. The first, `GetText`, is a function that returns a string with the date. You might think of adding other functions, such as `GetDay`, `GetMonth`, and `GetYear`, which simply return the corresponding `private` data, but similar direct data-access functions are not always needed. Providing access functions for each and every field might reduce the encapsulation and make it harder to modify the internal implementation of a class. Access functions should be provided only if they are part of the logical interface of the class you are implementing.

The second new method is the `Increase` procedure, which increases the date by one day. This is far from simple, because you need to consider the different lengths of the various months as well as leap and nonleap years. What I'll do to make it easier to write the code is to change the internal implementation of the class to use Delphi's `TDateTime` type for the internal implementation. The class will change to

```
type
  TDate = class
  private
    fDate: TDateTime;
  public
    procedure SetValue (m, d, y: Integer);
```

106 You could consider using *strict private*, instead.

```
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

Notice that because the only change is in the `private` portion of the class, you won't have to modify any of your existing programs that use it. This is the advantage of encapsulation!

---

note The `TDateTime` type is actually a floating-point number. The integral portion of the number indicates the date since 12/30/1899, the same base date used by OLE Automation and Microsoft applications. (Use negative values to express previous years.) The decimal portion indicates the time as a fraction. For example, a value of `3.75` stands for the second of January 1900, at 6:00 p.m. (three-quarters of a day). To add or subtract dates, you can simply add or subtract the number of days, which is much simpler than adding days with a day/month/year representation.

Encapsulation and Forms

One of the key ideas of encapsulation is to reduce the number of global variables used by a program. A global variable can be accessed from every portion of a program. For this reason, a change in a global variable affects the whole program. On the other hand, when you change the representation of a field of a class, you only need to change the code of some methods of that class and nothing else. Therefore, we can say that information hiding refers to *encapsulating changes*.

Let me clarify this idea with an example. When you have a program with multiple forms, you can make some data available to every form by declaring it as a global variable in the interface portion of the unit of one of the forms:

```
var
  Form1: TForm1;
  nClicks: Integer;
```

This works but has two problems. First, the data is not connected to a specific instance of the form, but to the entire program. If you create two forms of the same type, they'll share the data. If you want every form of the same type to have its own copy of the data, the only solution is to add it to the form class:

```
type
  TForm1 = class(TForm)
  public
    nClicks: Integer;
  end;
```

The second problem is that if you define the data as a global variable or as a public field of a form, you won't be able to modify its implementation in the future without affecting the code that uses the data. For example, if you only have to read the current value from other forms, you can declare the data as private and provide a method to read the value:

```
type
  TForm1 = class(TForm)
  public
    function GetClicks: Integer;
  private
    nClicks: Integer;
  end;

function TForm1.GetClicks: Integer;
begin
  Result := nClicks;
end;
```

An even better solution is to add a property to the form, as we'll see in the next chapter.

The Self Keyword

We've seen that methods are very similar to procedures and functions. The real difference is that methods have an implicit parameter, which is a reference to the current object. Within a method you can refer to this parameter—the current object—using the `Self` keyword. This extra hidden parameter is needed when you create several objects of the same class, so that each time you apply a method to one of the objects, the method will operate only on its own data and not affect the other sibling objects.

For example, in the `SetValue` method of the `TDate` class, listed earlier, we simply use `Month`, `Year`, and `Day` to refer to the fields of the current object, something you might express as

```
Self.Month := m;
Self.Day := d;
```

This is actually how the Delphi compiler translates the code, *not* how you are supposed to write it. The `Self` keyword is a fundamental language construct used by the compiler, but at times it is used by programmers to resolve name conflicts and to

make tricky code more readable. (The C++ and Java languages[107] have a similar feature based on the keyword `this`.)

All you really need to know about `Self` is that the technical implementation of a call to a method differs from that of a call to a generic subroutine. Methods have an extra hidden parameter, `Self`. Because all this happens behind the scenes, you do not need to know how `Self` works at this time.

---

note If you look at the definition of the `TMethod` data type in the VCL, you'll see that it is a record with a `Code` field and a `Data` field. The first is a pointer to the function's address in memory, the second the value of the `Self` parameter to use when calling that function address. We'll discuss method pointers in the next chapter.

Creating Components Dynamically

In Delphi, the `Self` keyword is often used when you need to refer to the current form explicitly in one of its methods. The typical example is the creation of a component at run time, where you must pass the owner of the component to its `Create` constructor and assign the same value to its `Parent` property. (The difference between `Owner` and `Parent` properties is discussed in the next chapter.) In both cases, you have to supply the current form as parameter or value, and the best way to do this is to use the `Self` keyword.

To demonstrate this kind of code, I've written the CreateC[108] example (the name stands for *Create Component*). This program has a simple form with no components and a handler for its `OnMouseDown` event. I've used `OnMouseDown` because it receives as its parameter the position of the mouse click (differently from the `OnClick` event). I need this information to create a button component in that position. Here is the code of the method:

```
procedure TForm1.FormMouseDown (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  Btn: TButton;
begin
  Btn := TButton.Create (Self);
  Btn.Parent := Self;
  Btn.Left := X;
```

---

107 And C#, as well.

108 A lot of the example of the book are short, to comply with the 8 char file name limitations of the DOS word. I know it can sound odd, but it was common at the time. The original Delphi library unit names were all 8 char maximum for the same reason.

```
  Btn.Top := Y;
  Btn.Width := Btn.Width + 50;
  Btn.Caption := Format ('Button at %d, %d', [X, Y]);
end;
```

The effect of this code is to create buttons at mouse-click positions, with a caption indicating the exact location, as you can see in Figure 2.1. In the code above, notice in particular the use of the `Self` keyword, as the parameter of the `Create` method and as the value of the `Parent` property.

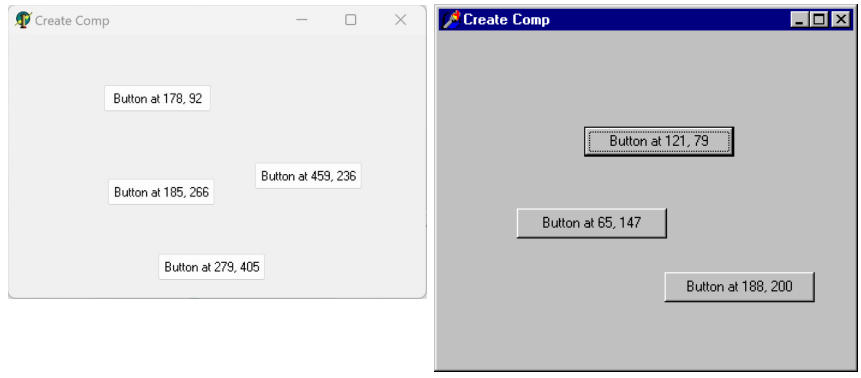


Figure 2.1: The output of the CreateC example, which creates Button components at run time. Images from the original book and captured today after rebuilding it in Delphi 12, running on Windows 11.

It is very common to write code like the above method using a `with`[109] statement, as in the following listing:

```
procedure TForm1.FormMouseDown (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  with TButton.Create (Self) do
  begin
    Parent := Self;
    Left := X;
    Top := Y;
    Width := Width + 50;
    Caption := Format ('Button in %d, %d', [X, Y]);
  end;
end;
```

109 I've later changed my mind regarding the *with* statement: I don't recommend using it as it can lead to hard-to-spot bugs, given the scope of the symbols used is not always clear.

note When writing a procedure like the code you've just seen, you might be tempted to use the `Form1` variable instead of `Self`. In this specific example, that change wouldn't make any practical difference, but if there are multiple instances of a form, using `Form1` would really be an error. In fact, if the `Form1` variable refers to the first form of that type being created, by clicking in another form of the same type, the new button will always be displayed in the first form. Its owner and Parent will be `Form1` and not the form the user has clicked onto. In general, referring to a particular instance of a class when the current object is required is a bad OOP practice.[110]

Constructors

To allocate the memory for the object, we call the `Create` method. This is a *constructor,* a special method that you can apply to a class to allocate memory for an instance of that class. The instance is returned by the constructor and can be assigned to a variable for storing the object and using it later on. The default `TObject.Create` constructor initializes all the data of the new instance to zero.

If you want your instance data to start out with a nonzero value, then you need to write a custom constructor to do that. The new constructor can be called `Create`, or it can have any other name: simply use the `constructor` keyword in front of it. Notice that in this case, you don't need to call `TObject.Create`: every constructor can automatically allocate the memory for an object instance simply by applying this special method to the related class.

The main reason to add a custom constructor to a class is to initialize its data. If you create objects without initializing them, calling methods later on may result in odd behavior or even a run-time error. Instead of waiting for these errors to appear, you should use preventive techniques to avoid them in the first place. One such technique is the consistent use of constructors to initialize objects' data. For example, we must call the `SetValue` procedure of the `TDate` class after we've created the object. As an alternative, we can provide a customized constructor, which creates the object and gives it an initial value.

Although in general you can use any name for a constructor, keep in mind that if you use a name other than `Create`, the `Create` constructor of the base `TObject` class will still be available. If you are developing and distributing code for others to use, a programmer calling this default constructor might bypass the initialization code you've provided. By defining a `Create` constructor with some parameters, you

110 I've defined a rule "Never use Form1" (or a reference to a specific form) in your code. While not an absolute rules, it's a very good idea in almost all cases.

replace the default definition with a new one and make its use compulsory. This is possible for generic classes, but it should be avoided for custom components. As we'll see in Chapter 13, when you inherit from `TComponent`, you should override the default `Create` constructor with one parameter and avoid disabling it.

In the same way that a class can have a custom constructor, it can have a custom destructor, a method declared with the `destructor` keyword and called `Destroy`, which can perform some resource cleanup before an object is destroyed. Just as a constructor call allocates memory for the object, a destructor call frees the memory. Destructors are needed only for objects that acquire resources in their constructors or during their lifetime.

Instead of calling `Destroy` directly, a program should call `Free`, which calls `Destroy` only if the object exists—that is, if it is not `nil`[111]. Keep in mind, however, that calling `Free` doesn't set the object to `nil` automatically; this is something you should do yourself! The reason is that the object doesn't know which variables may be referring to it, so it has no way to set them all to `nil`.

note Delphi 5 has finally introduced a simple `FreeAndNil` procedure you can use to free an object and set its reference to `nil` at the same time. Simply call `FreeAndNil (Obj1)` instead of calling `Obj1.Free` and then setting `Obj1` to `nil`.[112]

Overloaded Methods and Constructors

Starting with Delphi 4, Object Pascal supports overloaded functions and methods: you can have multiple methods with the same name, provided that the parameters are different. By checking the parameters, the compiler can determine which of the versions of the routine you want to call.

There are two basic rules:

- Each version of the method must be followed by the `overload` keyword.
- The differences must be in the number or type of the parameters or both. The return type, instead, cannot be used to distinguish among two methods.

111 Delphi's *nil* is the equivalent of *null* in other programming languages.

112 There have been discussions in the Delphi community whether *FreeAndNil* should be used or if its use implies a bad code architecture. In general, I tend to agree that the use of *FreeAndNil* should be a rare occurrence. Then a lcoal variable is about to get out of scope, setting it to *nil* has no benefit.

Overloading can be applied to global functions and procedures and to methods of a class. This feature is particularly relevant for constructors, because we can have multiple constructors and call them all `Create`, which makes them easy to remember.

note Historically, overloading was added to C++ to allow the use of multiple constructors that each have the same name (the name of the class). In Object Pascal, this feature was considered unnecessary simply because multiple constructors can have different specific names. The increased integration of Delphi with C++Builder has motivated Borland to make this feature available in both languages. Technically, when C++Builder constructs an instance of a Delphi VCL class, it looks for a Delphi constructor named `Create` and nothing but `Create`. If the Delphi class has constructors by other names, they cannot be used from C++Builder code. Therefore, when creating classes and components you intend to share with C++Builder programmers, you should be careful to name all your constructors `Create` and distinguish between them by their parameter lists (using `overload`). This is not required by Delphi, but it is required for C++Builder to use your Delphi classes.

As an example of overloading, I've added to the `TDate` class two different versions of the `SetValue` method:

```
type
  TDate = class
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;

procedure TDate.SetValue (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
end;

procedure TDate.SetValue(NewDate: TDateTime);
begin
  fDate := NewDate;
end;
```

After this simple step, I've added to the class two separate `Create` constructors, one with no parameters, which hides the default constructor, and one with the initialization values. The constructor with no parameters uses as the default value today's date:

```
type
  TDate = class
  public
    constructor Create; overload;
    constructor Create (y, m, d: Integer); overload;

constructor TDate.Create (y, m, d: Integer);
```

```
begin
  fDate := EncodeDate (y, m, d);
end;

constructor TDate.Create;
begin
  fDate := Date;
end;
```

Having these two constructors makes it possible to define a new `TDate` object in two different ways:

```
var
  Day1, Day2: TDate;
begin
  Day1 := TDate.Create (1999, 12, 25);
  Day2 := TDate.Create; // today
```

The Complete TDate Class

Throughout this chapter, I've shown you bits and pieces of the source code for different versions of a `TDate` class. The first version was based on three integers to store the year, the month, and the day; a second version used a field of the `TDateTime` type provided by Delphi. Here is the complete interface portion of the unit that defines the `TDate` class:

```
unit Dates;

interface

type
  TDate = class
  private
    fDate: TDateTime;
    function GetYear: Integer;
  public
    constructor Create; overload;
    constructor Create (y, m, d: Integer); overload;
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    procedure Increase (NumberOfDays: Integer = 1);
    procedure Decrease (NumberOfDays: Integer = 1);
    function GetText: string;
  end;

implementation
```

```
...
```

The aim of the new methods, `Increase` and `Decrease` (which have a default value for their parameter), is quite easy to understand. If called with no parameter, they change the value of the date to the next or previous day. If a `NumberOfDays` parameter is part of the call, they add or subtract that number:

```
procedure TDate.Increase (NumberOfDays: Integer = 1);
begin
  fDate := fDate + NumberOfDays;
end;
```

`GetText` returns a string with the formatted date, using the `DateToStr` function:

```
function TDate.GetText: string;
begin
  GetText := DateToStr (fDate);
end;
```

We've already seen most of the methods in the previous sections, so I won't provide the complete listing; you can find it in the code of the ViewDate example I've written to test the class. The form has a caption to display a date and six buttons, which can be used to modify the date. You can see the main form of the ViewDate example at run time in Figure 2.2. To make the label component look nice, I've given it a big font, made it as wide as the form, set its `Alignment` property to `taCenter`, and set its `AutoSize` property to `False`.

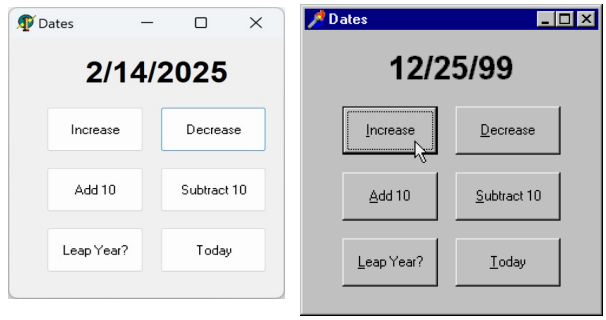


Figure 2.2: The output of the ViewDate example at start-up. Images captured now and in the original book.

The start-up code of this program is in the `OnCreate` event handler. In the corresponding method, we create an instance of the `TDate` class, initialize this object, and then show its textual description in the `Caption` of the label, as shown in Figure 2.2.

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
```

```
  TheDay := TDate.Create (1999, 12, 25);
  LabelDate.Caption := TheDay.GetText;
end;
```

`TheDay` is a private field of the class of the form, `TDateForm`. By the way, the name for the class is automatically chosen by Delphi when we change the `Name` property of the form to `DateForm`. The object is then destroyed along with the form:

```
procedure TDateForm.FormDestroy(Sender: TObject);
begin
  TheDay.Free;
end;
```

When the user clicks one of the six buttons, we need to apply the corresponding method to the `TheDay` object and then display the new value of the date in the label:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
  TheDay.SetValue (Date);
  LabelDate.Caption := TheDay.GetText;
end;
```

An alternative way to write the last method is to destroy the current object and create a new one:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
var
  NewDay: TDate;
begin
  TheDay.Free;
  NewDay := TDate.Create;
  TheDay := NewDay;
  LabelDate.Caption := TheDay.GetText;
end;
```

In this particular circumstance, this is not a very good approach (because creating a new object and destroying an existing one entails a lot of time overhead, when all we need is to change the object's value), but it allows me to show you a couple of Object Pascal techniques. The first thing to notice is that we destroy the previous object before assigning a new one. The assignment operation, in fact, replaces the reference, leaving the object in memory (even if no pointer is referring to it). When you assign an object to another object, Delphi simply copies the reference to the object in memory to the new object/reference.

If you really want to change the data inside an existing object, copy each field, or provide a specific method to copy the internal data. Some classes of the VCL have an `Assign` method, which does this *deep-copying*. To be more precise, all the VCL classes inheriting from `TPersistent` have the `Assign` method, but most of those

inheriting from `TComponent` don't implement it, raising an exception when it is called.

Inheriting from Existing Types

We often need to use a slightly different version of an existing class that we have written or that someone has given to us. For example, you might need to add a new method or slightly change an existing one. You can do this easily by modifying the original code, unless you want to be able to use the two different versions of the class in different circumstances. Also, if the class was originally written by someone else (including Borland), you might want to keep your changes separate.

A typical alternative is to make a copy of the original type definition, change its code to support the new features, and give a new name to the resulting class. This might work, but it also might create problems: in duplicating the code you also duplicate the bugs; and if you want to add a new feature, you'll need to add it two or more times, depending on the number of copies of the original code you've made. This approach results in two completely different data types, so the compiler cannot help you take advantage of the similarities between the two types.

To solve these kinds of problems in expressing similarities between classes, Object Pascal allows you to define a new class directly from an existing one. This technique is known as *inheritance* (or *subclassing,* or *derivation*) and is one of the fundamental elements of object-oriented programming languages. To inherit from an existing class, you only need to indicate that class at the beginning of the declaration of the subclass. For example, Delphi does this automatically each time you create a new form:

```
type
  TForm1 = class(TForm)
  end;
```

This simple definition indicates that the `TForm1` class inherits all the methods, fields, properties, and events of the `TForm` class. You can apply any public method of the `TForm` class to an object of the `TForm1` type. `TForm`, in turn, inherits some of its methods from another class, and so on, up to the `TObject` class.

As a simple example of inheritance, we can change the ViewDate program slightly, deriving a new class from `TDate` and modifying one of its functions, `GetText`. You can find this code in the `DATES.PAS` file of the ViewD2 example.

```
type
  TNewDate = class (TDate)
  public
     function GetText: string;
  end;
```

In this example, `TNewDate` is derived from `TDate`. It is common to say that `TDate` is an *ancestor* class or *parent* class of `TNewDate` and that `TNewDate` is a *subclass, descendant* class, or *child* class of `TDate`.

To implement the new version of the `GetText` function, I used the `FormatDateTime` function, which uses (among other features) the predefined month names available in Windows; these names depend on the user's regional and language settings. Many of these regional settings are actually copied by Delphi into constants defined in the library, such as `LongMonthNames`, `ShortMonthNames`, and many others you can find under the *Currency and date/time formatting variables* topic in the Delphi Help file. Here is the `GetText` method, where *'dddddd'* stands for the long data format:

```
function TNewDate.GetText: string;
begin
  GetText := FormatDateTime ('dddddd', fDate);
end;
```

note Using regional information, the ViewD2 program automatically adapts itself to different Windows user settings. If you run this same program on a computer with regional settings referring to a language other than English, it will automatically show month names in that language. To test this behavior, you just need to change the regional settings; you don't need a new version of Windows. Notice that regional-setting changes immediately affect the running programs.

Once we have defined the new class, we need to use this new data type in the code of the form of the ViewD2 example. Simply define the `TheDay` object of type `TNewDate`, and call its constructor in the `FormCreate` method:

```
type
  TDateForm = class(TForm)
    ...
  private
    TheDay: TNewDate; // updated declaration
  end;

procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TNewDate.Create (1998, 12, 25); // updated
  DateLabel.Caption := TheDay.GetText;
end;
```

Without any other changes, the new ViewD2 example will work properly. The `TNewDate` class inherits the methods to increase the date, add a number of days, and so on. In addition, the older code calling these methods still works. Actually, to call the new version of the `GetText` method, we don't need to change the source code! The Delphi compiler will automatically bind that call to a new method. The source code of all the other event handlers remains exactly the same, although its meaning changes considerably, as the new output demonstrates (see Figure 2.3).

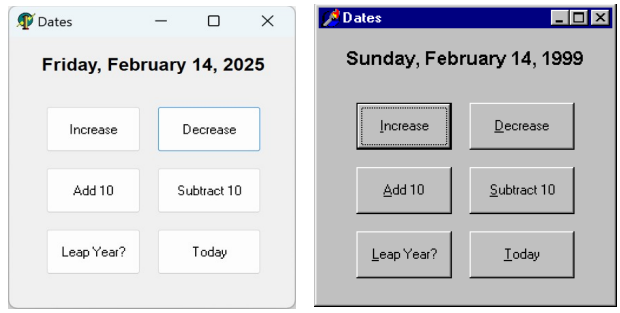


Figure 2.3: The output of the ViewD2 program, with the name of the month and of the day depending on Windows regional settings. Images captured now and in the original book.

Protected Fields and Encapsulation

The code of the `GetText` method of the `TNewDate` class compiles only if it is written in the same unit as the `TDate` class. In fact, it accesses the `fDate` private field of the ancestor class. If we want to place the descendant class in a new unit, we must either declare the `fDate` field as protected or add a simple, possibly protected, method in the ancestor class to read the value of the private field.

Many developers believe that the first solution is always the best, because declaring most of the fields as protected will make a class more extensible and will make it easier to write subclasses. However, this violates the idea of encapsulation. In a large hierarchy of classes, changing the definition of some protected fields of the base classes becomes as difficult as changing some global data structures. If ten derived classes are accessing this data, changing its definition means potentially modifying the code in each of the ten classes.

In other words, flexibility, extension, and encapsulation often become conflicting objectives. When this happens, you should try to favor encapsulation. If you can do so without sacrificing flexibility, that will be even better. Often this intermediate

solution can be obtained by using a virtual method, a topic I'll discuss in detail below in the section "Late Binding and Polymorphism." If you choose not to use encapsulation in order to obtain faster coding of the subclasses, then your design might not follow the object-oriented principles.

Accessing Protected Data of Other Classes

> *If you are new to Delphi and to OOP, this is a rather advanced section you might want to skip the first time you are reading this book, as it might be confusing.*

We've seen that in Delphi, the `private` and `protected` data of a class is accessible to any functions or methods that appear *in the same unit as the class*. For example, consider this simple class (part of the Protection example):

```
type
  TTest = class
  protected
    ProtectedData: Integer;
  public
    PublicData: Integer;
    function GetValue: string;
  end;
```

The `GetValue` method simply returns a string with the two integer values:

```
function TTest.GetValue: string;
begin
  Result := Format ('Public: %d, Protected: %d',
    [PublicData, ProtectedData]);
end;
```

Once you place this class in its own unit, you won't be able to access its protected portion from other units directly. Accordingly, if you write the following code,

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  Obj.ProtectedData := 20;  // won't compile
  ShowMessage (Obj.GetValue);
  Obj.Free;
end;
```

the compiler will issue an error message, *Undeclared identifier: "ProtectedData."* At this point, you might think there is no way to access the protected data of a class defined in a different unit. (This is what Delphi manuals and most Delphi books say.) However, there is a way around it. Consider what happens if you create an apparently useless derived class, such as

```
type
  TFake = class (TTest);
```

Now, if you make a direct cast of the object to the new class and access the protected data through it, this is how the code will look:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  TFake (Obj).ProtectedData := 20; // compiles!
  ShowMessage (Obj.GetValue);
  Obj.Free;
end;
```

This code compiles and works properly, as you can see by running the Protection program. How is it possible for this approach to work? Well, if you think about it, the `TFake` class automatically inherits the protected fields of the `TTest` base class, and because the `TFake` class is in the same unit as the code that tries to access the data in the inherited fields, the protected data is accessible. As you would expect, if you move the declaration of the `TFake` class to a secondary unit, the program won't compile any more.

Now that I've shown you how to do this, I must warn you that violating the class-protection mechanism this way is likely to cause errors in your program (from accessing data that you really shouldn't), and it runs counter to good OOP technique. However, there are times when using this technique is the best solution, as you'll see by looking at the VCL source code and the code of many Delphi components. Two simple examples that come to mind are accessing the `Text` property of the `Tcontrol` class and the `Row` and `Col` positions of the `DBGrid` control. These two ideas are demonstrated by the TextProp and DBGridCol examples respectively. (These examples are quite advanced, so I suggest that only programmers with a good background of Delphi programming read them at this point in the text—other readers might come back later.)

Although the first example shows a reasonable example of using the typecast *cracker,* the DBGrid example of `Row` and `Col` is actually a counter example, one that illustrates the risks of accessing bits that the class writer chose not to expose. The

row and column of a `DBGrid` do not mean the same thing as they do in a `DrawGrid` or `StringGrid` (the base classes). First, `DBGrid` does not count the fixed cells as actual cells (it distinguishes data cells from decoration), so your row and column indexes will have to be adjusted by whatever decorations are currently in effect on the grid (and those can change on the fly). Second, the `DBGrid` is a virtual view of the data. When you scroll up in a `DBGrid`, the data may move underneath it, but the currently selected row might not change.

This technique is often described as a *hack,* and it should be avoided whenever possible. The problem is not accessing protected data of a class in the same unit but declaring a class for the sole purpose of accessing protected data of an existing object of a different class! The danger of this technique is in the hard-coded typecast of an object from a class to a different one.

Inheritance and Type Compatibility

Pascal is a strictly typed language. This means that you cannot, for example, assign an integer value to a `Boolean` variable, at least not without an explicit typecast. The rule is that two values are type-compatible only if they are of the same data type, or (to be more precise) if their data type has the same name and their definition comes from the same unit.

There is an important exception to this rule in the case of class types. If you declare a class, such as `TAnimal`, and derive from it a new class, say `TDog`, you can then assign an object of type `TDog` to a variable of type `TAnimal`. That is because a dog is an animal! So, although this might surprise you, the following constructor calls are both legal:

```
var
  MyAnimal1, MyAnimal2: TAnimal;
begin
  MyAnimal1 := TAnimal.Create;
  MyAnimal2 := TDog.Create;
```

As a general rule, you can use an object of a descendant class any time an object of an ancestor class is expected. However, the reverse is not legal; you cannot use an object of an ancestor class when an object of a descendant class is expected. To simplify the explanation, here it is again in code terms:

```
MyAnimal := MyDog;  // This is OK
MyDog := MyAnimal;  // This is an error!!!
```

Before we look at the implications of this important feature of the language, you can try out the Animals1 example, which defines the two simple `TAnimal` and `TDog` classes:

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
  private
    Kind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
  end;
```

The two `Create` methods simply set the value of `Kind`, which is returned by the `GetKind` function. The form displayed by this example, shown in Figure 2.4, has a private field of type `TAnimal`. An instance of this class is created and initialized when the form is created and each time one of the radio buttons is selected.

```
procedure TFormAnimals.FormCreate(Sender: TObject);
begin
  MyAnimal := TAnimal.Create;
end;

procedure TFormAnimals.RbtnDogClick(Sender: TObject);
begin
  MyAnimal.Free;
  MyAnimal := TDog.Create;
end;
```

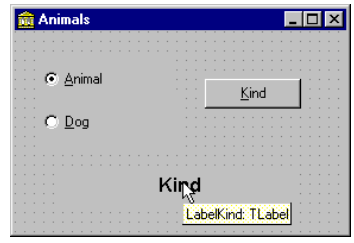


Figure 2.4: The form of the Animals1 example (in Delphi 5)

Finally, the Kind button calls the `GetKind` method for the current animal and displays the result in the label:

```
procedure TFormAnimals.BtnKindClick(Sender: TObject);
```

```
begin
  KindLabel.Caption := MyAnimal.GetKind;
end;
```

Late Binding and Polymorphism

Pascal functions and procedures are usually based on *static binding,* which is also called *early binding*. This means that a method call is resolved by the compiler or the linker, which replaces the request with a call to the specific memory location where the function or procedure resides. (This is known as the *address* of the function.) Object-oriented programming languages allow the use of another form of binding, known as *dynamic binding,* or *late binding*. In this case, the actual address of the method to be called is determined at run time based on the type of the instance used to make the call.

The advantage of this technique is known as *polymorphism*. Polymorphism means you can write a call to a method, applying it to a variable, but which method Delphi actually calls depends on the type of the object the variable relates to. Delphi cannot determine until run time the actual class of the object the variable refers to, simply because of the type-compatibility rule discussed in the previous section.

note The term *polymorphism* is quite a mouthful. A glance at the dictionary tells us that in a general sense it refers to something having more than one form. In the OOP sense, then, it refers to the fact that there may be several versions of a given method and that a single method call can refer to any of these versions.

For example, suppose that a class and its subclass (let's say `TAnimal` and `TDog`) both define a method, and this method has late binding. Now you can apply this method to a generic variable, such as `MyAnimal`, which at run time can refer either to an object of class `TAnimal` or to an object of class `TDog`. The actual method to call is determined at run time, depending on the class of the current object.

The Animals2 example extends the Animals1 program to demonstrate this technique. In the new version, the `TAnimal` and the `TDog` classes have a new method: `Voice`, which means to output the sound made by the selected animal, both as text and as sound. This method is defined as `virtual` in the `TAnimal` class and is later overridden when we define the `TDog` class, by the use of the `virtual` and `override` keywords:

```
type
  TAnimal = class
  public
    function Voice: string; virtual;

  TDog = class (TAnimal)
  public
    function Voice: string; override;
```

Of course, the two methods also need to be implemented. Here is a simple approach:

```
uses
  MMSystem;

function TAnimal.Voice: string;
begin
  Voice := 'Voice of the animal';
  PlaySound ('Anim.wav', 0, snd_Async);
end;

function TDog.Voice: string;
begin
  Voice := 'Arf Arf';
  PlaySound ('dog.wav', 0, snd_Async);
end;
```

note This example uses a call to the `PlaySound` API function, defined in the MMSystem unit. The first parameter of this function is the name of the WAV sound file or the system sound you want to execute. The second parameter indicates an optional resource file containing the sound. The third parameter indicates (among other options) whether the call should be synchronous or asynchronous; that is, whether the program should wait for the sound to finish before continuing with the following statements.

Now what is the effect of the call `MyAnimal.Voice`? It depends. If the `MyAnimal` variable currently refers to an object of the `TAnimal` class, it will call the method `TAnimal.Voice`. If it refers to an object of the `TDog` class, it will call the method `TDog.Voice` instead. This happens only because the function is `virtual`.

The call to `MyAnimal.Voice` will work for an object that is an instance of any descendant of the `TAnimal` class, even classes that are defined after this method call or outside its scope. The compiler doesn't need to know about all the descendants in order to make the call compatible with them; only the ancestor class is needed. In other words, this call to `MyAnimal.Voice` is compatible with all future `TAnimal` subclasses.

This is the key technical reason why object-oriented programming languages favor reusability. You can write code that uses classes within a hierarchy without any

knowledge of the specific classes that are part of that hierarchy. In other words, the hierarchy—and the program—is still extensible, even when you've written thousands of lines of code using it. Of course, there is one condition—the ancestor classes of the hierarchy need to be designed very carefully.

The Animals2 program demonstrates the use of these new classes and has a form similar to that of the previous example. This code is executed by clicking on the button:

```
procedure TFormAnimals.BtnVerseClick(Sender: TObject);
begin
  LabelVoice.Caption := MyAnimal.Voice;
end;
```

In Figure 2.5, you can see an example of the output of this program. By running it, you'll also hear the corresponding sounds produced by the `PlaySound` API call.

Figure 2.5: The output of the Animals2 example. Image from the original book.

Overriding, Redefining, and Reintroducing Methods

As we have just seen, to override a late-bound method in a descendant class, you need to use the `override` keyword. Note that this can take place only if the method was defined as `virtual` in the ancestor class. Otherwise, if it was a static method, there is no way to activate late binding, other than by changing the code of the ancestor class.

The rules are simple: A method defined as static remains static in every subclass, unless you hide it with a new virtual method having the same name. A method defined as `virtual` remains late-bound in every subclass. There is no way to change this, because of the way the compiler generates different code for late-bound methods.

To redefine a static method, you simply add a method to a subclass having the same parameters or different parameters than the original one, without any further specifications. To override a `virtual` method, you must specify the same parameters and use the `override` keyword:

```
type
  MyClass = class
    procedure One; virtual;
    procedure Two; {static method}
  end;

  MySubClass = class (MyClass)
    procedure One; override;
    procedure Two;
  end;
```

There are typically two ways to override a method. One is to replace the method of the ancestor class with a new version. The other is to add some more code to the existing method. This can be accomplished by using the `inherited` keyword to call the same method of the ancestor class. For example, you can write

```
procedure MySubClass.One;
begin
  // new code
  ...
  // call inherited procedure MyClass.One
  inherited One;
end;
```

You might wonder why you need to use the `override` keyword. In other languages, when you redefine a method in a subclass, you automatically override the original one. However, having a specific keyword allows the compiler to check the correspondence between the names of the methods of the ancestor class and the subclass (misspelling a redefined function is a common error in other OOP languages), check that the method was virtual in the ancestor class, and so on.

Furthermore, if you define a static method in any class inherited by a class of the library, there will be no problem, even if the library is updated with a new virtual method having the same name as a method you've defined. Because your method is not marked by the `override` keyword, it will be considered a separate method and not a new version of the one added to the library (something that would probably break your code).

The support for overloading introduced in Delphi 4 added some further complexity to this picture. A subclass can provide a new version of a method using the `overload` keyword. If the method has different parameters than the version in the base class,

it becomes effectively an overloaded method; otherwise it replaces the base class method. Here is an example:

```
type
  TMyClass = class
    procedure One;
  end;

  TMySubClass = class (TMyClass)
    procedure One (S: string); overload;
  end;
```

Notice that the method doesn't need to be marked as `overload` in the base class. However, if the method in the base class is virtual, the compiler issues the warning *Method 'One' hides virtual method of base type 'TMyClass'.* To avoid this message from the compiler and to instruct the compiler more precisely on your intentions, you can use the new `reintroduce` directive:

```
type
  TMyClass = class
    procedure One; virtual;
  end;

  TMySubClass = class (TMyClass)
    procedure One (S: string); reintroduce; overload;
  end;
```

You can find this code in the Reintr example and experiment with it further.

Virtual versus Dynamic Methods

In Delphi, there are two different ways to activate late binding. You can declare the method as `virtual`, as we have seen before, or declare it as `dynamic`. The syntax of these two keywords is exactly the same, and the result of their use is also the same. What is different is the internal mechanism used by the compiler to implement late binding.

Virtual methods are based on a *virtual method table* (VMT, also known as a *vtable*). A virtual method table is an array of method addresses. For a call to a virtual method, the compiler generates code to jump to an address stored in the *n*th slot in the object's virtual method table.

Virtual method tables allow fast execution of the method calls. Their main drawback is that they require an entry for each virtual method for each descendant class, even if the method is not overridden in the subclass. At times, this has the effect of propagating virtual method table entries throughout a class hierarchy (even for methods

that aren't redefined). This might require a lot of memory just to store the same method address a number of times.

Dynamic method calls, on the other hand, are dispatched using a unique number indicating the method. The search for the corresponding function is generally slower than the simple one-step table lookup for virtual methods. The advantage is that dynamic method entries only propagate in descendants when the descendants override the method. For large or deep object hierarchies, using dynamic methods instead of virtual methods can result in significant memory savings with only a minimal speed penalty.

From a programmer's perspective, the difference between these two approaches lies only in a different internal representation and slightly different speed or memory usage. Apart from this, virtual and dynamic methods are the same.

Message Handlers

A late-bound method can be used to handle a Windows message, too, although the technique is somewhat different. For this purpose Delphi provides yet another directive, `message`, to define message-handling methods, which must be procedures with a single `var` parameter. The `message` directive is followed by the number of the Windows message the method wants to handle. For example, the following code allows you to handle a user-defined message, with the numeric value indicated by the `wm_User` Windows constant:

```
type
  TForm1 = class(TForm)
    ...
    procedure WmUser (var Msg: TMessage);
      message wm_User;
  end;
```

The name of the procedure and the actual type of the parameters are up to you, although there are a number of predefined record types for the various Windows messages. This technique can be extremely useful for veteran Windows programmers, who know all about Windows messages and API functions.

---

note The ability to handle Windows messages and call API functions as you do when you are programming Windows with the C language may horrify some programmers and delight others. But in Delphi, when writing Windows applications, you will seldom need to use `message` methods. Only when you are writing complex components in Delphi will you have to deal with low-level messages and API calls.

Abstract Methods

The `abstract` keyword is used to declare methods that will be defined only in subclasses of the current class[113]. The `abstract` directive fully defines the method; it is not a forward declaration. If you try to provide a definition for the method, the compiler will complain. In Object Pascal, you can create instances of classes that have `abstract` methods. However, when you try to do so, Delphi's 32-bit compiler issues the warning message: *Constructing instance of <class name> containing abstract methods*. If you happen to call an `abstract` method at run time, Delphi will raise an exception, as demonstrated by the following Animals3 example.

note C++ and Java use a more strict approach: in these languages, you cannot create instances of abstract classes.

You might wonder why you would want to use `abstract` methods. The reason lies in the use of polymorphism. If class `TAnimal` has the `abstract` method `Voice`, every subclass can redefine it. The advantage is that you can now use the generic `MyAnimal` object to refer to each animal defined by a subclass and invoke this method. If this method was not present in the interface of the `TAnimal` class, the call would not have been allowed by the compiler, which performs static type checking. Using a generic `MyAnimal` object, you can call only the method defined by its own class, `TAnimal`.

You cannot call methods provided by subclasses, unless the parent class has at least the declaration of this method—in the form of an `abstract` method. The next example, Animals3, demonstrates the use of `abstract` methods and the abstract call error. Here are the interfaces of the classes of this new example:

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
    function Voice: string; virtual; abstract;
  private
    Kind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
    function Voice: string; override;
```

113 In recent versions of Delphi you can also use the abstract keyword to decorate a class as a whole, a syntax originally introduced in the .NET version of the compiler.

```
    function Eat: string; virtual;
  end;

  TCat = class (TAnimal)
  public
    constructor Create;
    function Voice: string; override;
    function Eat: string; virtual;
  end;
```

The most interesting portion is the definition of the class `TAnimal`, which includes a virtual `abstract` method: `Voice`. It is also important to notice that each derived class overrides this definition and adds a new virtual method, `Eat`. What are the implications of these two different approaches? To call the `Voice` function, we can simply write the same code as in the previous version of the program:

```
LabelVoice.Caption := MyAnimal.Voice;
```

How can we call the `Eat` method? We cannot apply it to an object of the `TAnimal` class. The statement

```
LabelVoice.Caption := MyAnimal.Eat;
```

generates the compiler error *Field identifier expected*.

To solve this problem, you can use run-time type information (RTTI) to cast the `TAnimal` object to a `TCat` or `TDog` object; but without the proper cast, the program will raise an exception. You will see an example of this approach in the next section. Adding the method definition to the `TAnimal` class is a typical solution to the problem, and the presence of the `abstract` keyword favors this choice.

Run-Time Type Information[114]

The Object Pascal type–compatibility rule for descendant classes allows you to use a descendant class where an ancestor class is expected. As I mentioned earlier, the reverse is not possible.

Now suppose that the `TDog` class has an `Eat` method, which is not present in the `TAnimal` class. If the variable `MyAnimal` refers to a dog, it should be possible to call

---

114 The core form of RTTI, described in this section, is still available today. On top of it, there is now in Delphi an extended RTTI and a specific unit with classes you can use to access a large amount of type information at runtime.

the function. But if you try, and the variable is referring to another class, the result is an error. By making an explicit typecast, we could cause a nasty run-time error (or worse, a subtle memory overwrite problem), because the compiler cannot determine whether the type of the object is correct and the methods we are calling actually exist.

To solve the problem, we can use techniques based on run-time type information. Essentially, because each object "knows" its type and its parent class, we can ask for this information with the `is` operator or using some of the methods of the `TObject` class (discussed in the next chapter). The parameters of the `is` operator are an object and a class type, and the return value is a Boolean:

```
if MyAnimal is TDog then
  ...
```

The `is` expression evaluates as `True` only if the `MyAnimal` object is currently referring to an object of class `TDog` or a type descendant from `TDog`. This means that if you test whether a `TDog` object is of type `TAnimal`, the test will succeed. In other words, this expression evaluates as `True` if you can safely assign the object (`MyAnimal`) to a variable of the data type (`TDog`).

Now that you know for sure that the animal is a dog, you can make a safe typecast (or type conversion). You can accomplish this direct cast by writing the following code:

```
if MyAnimal is TDog then
begin
  MyDog := TDog (MyAnimal);
  Text := MyDog.Eat;
end;
```

This same operation can be accomplished directly by the second RTTI operator, `as`, which converts the object only if the requested class is compatible with the actual one. The parameters of the `as` operator are an object and a class type, and the result is an object converted to the new class type. We can write the following snippet:

```
MyDog := MyAnimal as TDog;
Text := MyDog.Eat;
```

If we only want to call the `Eat` function, we might also use an even shorter notation:

```
(MyAnimal as TDog).Eat;
```

The result of this expression is an object of the `TDog` class data type, so you can apply to it any method of that class. The difference between the traditional cast and the use of the `as` cast is that the second one raises an exception if the type of the object

is not compatible with the type you are trying to cast it to. The exception raised is `EInvalidCast` (exceptions are described at the end of this chapter).

To avoid this exception, use the `is` operator and, if it succeeds, make a plain typecast (in fact there is no reason to use `is` and `as` in sequence, doing the type check twice):

```
if MyAnimal is TDog then
  TDog(MyAnimal).Eat;
```

Both RTTI operators are very useful in Delphi because you often want to write generic code that can be used with a number of components of the same type or even of different types. When a component is passed as a parameter to an event-response method, a generic data type is used (`TObject`), so you often need to cast it back to the original component type:

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

This is a common technique in Delphi, and I'll use it in a number of examples throughout the book. In Chapter 4, we'll discuss the `is` and `as` operators again, while focusing on some alternative RTTI techniques based on methods of the `TObject` class. The two RTTI operators, `is` and `as`, are extremely powerful, and you might be tempted to consider them as standard programming constructs. Although they are indeed powerful, you should probably limit their use to special cases. When you need to solve a complex problem involving several classes, try using polymorphism first. Only in special cases, where polymorphism alone cannot be applied, should you try using the RTTI operators to complement it. *Do not use RTTI instead of polymorphism*. This is bad programming practice, and it results in slower programs. RTTI, in fact, has a negative impact on performance, because it must walk the hierarchy of classes to see whether the typecast is correct. As we have seen, virtual method calls require just a memory lookup, which is much faster.

Visual Form Inheritance

To better understand derivation among classes, you can use visual form inheritance. In short, you can simply inherit a form from an existing one, adding new compo-

nents or altering the properties of the existing ones. But what is the real advantage of visual form inheritance?

Well, this mostly depends on the kind of application you are building. If it has a number of forms, some of which are very similar to each other or simply include common elements, then you can place the common components and the common event handlers in the base form and add the specific behavior and components to the sub-classes. For example, if you prepare a standard parent form with a toolbar, a logo, default sizing and closing code, and the handlers of some Windows messages, you can then use it as the parent class for each of the forms of an application.

You can also use visual form inheritance to customize an application for different clients, without duplicating any source code or form definition code; just inherit the specific versions for a client from the standard forms. Remember that the main advantage of visual inheritance is that you can later change the original form and automatically update all the derived forms. This is a well-known advantage of inheritance in object-oriented programming languages. But there is a beneficial side effect: polymorphism. You can add a virtual method in a base form and override it in a subclassed form. Then you can refer to both forms and call this method for each of them.

note Delphi 5 includes a new feature, called *Frames,* which resembles visual form inheritance. In both cases you can work at design time on two versions of a form. However, in visual form inheritance, you are defining two different classes (parent and derived), whereas with frames, you work on a class and an instance. Frames will be discussed in detail in Chapter 4.

Inheriting from a Base Form

The rules governing visual form inheritance are quite simple, once you have a clear idea of what inheritance is. Basically, a subclass form has the same components as the parent form as well as some new components. You cannot remove a component of the base class, although (if it is a visual control) you can make it invisible. What's important is that you can easily change properties of the components you inherit.

Notice that if you change a property of a component in the inherited form, any modification of the same property in the parent form will have no effect. Changing other properties of the component will affect the inherited versions, as well. You can resynchronize the two property values by using the Revert to Inherited local menu command of the Object Inspector. The same thing is accomplished by setting the two properties to the same value and recompiling the code. After modifying multi-

ple properties, you can re-synchronize them all to the base version by applying the Revert to Inherited command of the component's local menu.

An alternative technique is to open the textual description of the inherited form and remove the line that changes the value of the property. (We will look at the structure of this file in a second.) Besides inheriting components, the new form inherits all the methods of the base form, including the event handlers. You can add new handlers in the inherited form and also override existing handlers.

To describe how visual form inheritance works, I've built a very simple example, called VFI. I'll describe step-by-step how to build it. First, start a new project, and add four buttons to its main form. Then select File ➢ New[115], and choose the page with the name of the project in the New Items dialog box (see Figure 2.6). Here you can choose the form from which you want to inherit. The new form has the same four buttons. Here is the initial textual description of the new form:

```
inherited Form2: TForm2
  Caption = 'Form2'
end
```

And here is its initial class declaration, where you can see that the base class is not the usual `TForm` but the actual base class form:

```
type
  TForm2 = class(TForm1)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Notice the presence of the `inherited` keyword in the textual description; also notice that the form indeed has some components, although they are defined in the base class form. If you move the form and add the caption of one of the buttons, the textual description will change accordingly:

115 File | New | Other in recent versions of Delphi. The category is "Inheritable items", rather than the name of the project.

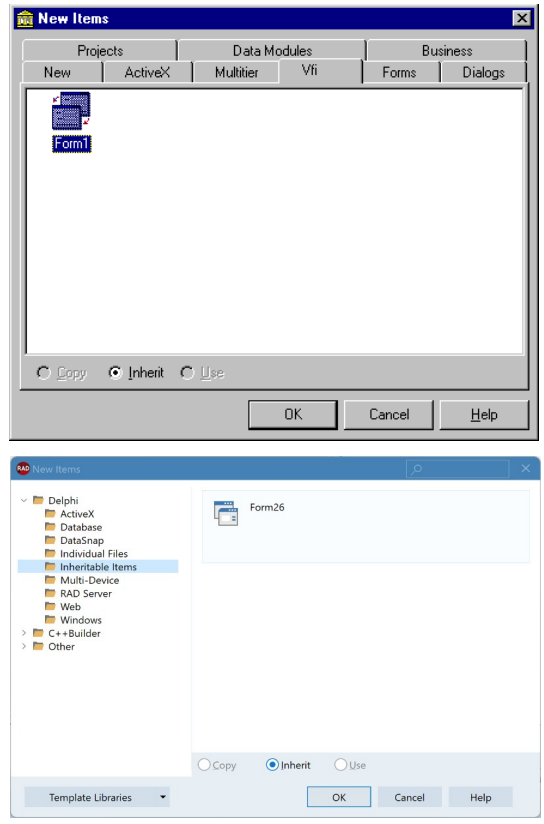


Figure 2.6:
The New Items dialog box allows you to create an inherited form. Images captured in Delphi 5 and Delphi 12.

```
inherited Form2: TForm2
  Left = 313
  Top = 202
  Caption = 'Form2'
  inherited Button2: TButton
    Caption = 'Beep...'
  end
end
```

Only the properties with a different value are listed (and by removing these properties from the textual description of the inherited form, you can reset them to the value of the base form, as I mentioned before). I've actually changed the captions of most buttons, as you can see in Figure 2.7.

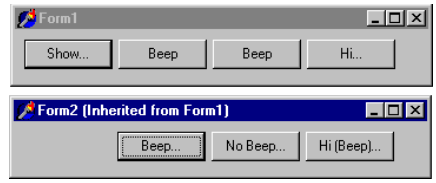


Figure 2.7: The two forms of the VFI example at run time. Image from the original book.

Each of the buttons of the first form has an `OnClick` handler, with simple code. The first button shows the inherited form calling its `Show` method; the second and the third buttons call the `Beep` procedure; and the last button displays a simple message calling `ShowMessage ('Hi')`.

What happens in the inherited form? First we should remove the Show button because the secondary form is already visible. However, we cannot delete a component from an inherited form. An alternative solution is to leave the component there but set its `Visible` property to `False`. The button will still be there but not visible (as you can guess from Figure 2.7). The other three buttons will be visible but with different handlers. This is simple to accomplish. If you select the `OnClick` event of a button in the inherited form (by double-clicking it), you'll get an empty method slightly different from the default one:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
end;
```

The `inherited` keyword stands for a call to the corresponding event handler of the base form. This keyword is always added by Delphi, even if the handler is not defined in the parent class (and this is reasonable, because it might be defined later) or if the component is not present in the parent class (which doesn't seem like a great idea to me). It is very simple to execute the code of the base form and perform some other operations:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
  ShowMessage ('Hi');
end;
```

This is not the only choice. An alternative approach is to write a brand-new event handler and not execute the code of the base class, as I've done for the third button of the VFI example:

```
procedure TForm2.Button3Click(Sender: TObject);
```

```
begin
  ShowMessage ('Hi');
end;
```

Still another choice includes calling a base-class method after some custom code has been executed, calling it when a condition is met, or calling the handler of a different event of the base class, as I've done for the fourth button:

```
procedure TForm2.Button4Click(Sender: TObject);
begin
  inherited Button3Click (Sender);
  inherited;
end;
```

You probably won't do this very often, but you must be aware that you can. Of course, you can consider each method of the base form as a method of your form, and call it freely. This example allows you to explore some features of visual form inheritance, but to see its true power you'll need to look at real-world examples more complex than this book has room to explore. There is something else I want to show you here: *visual form polymorphism*.

Polymorphic Forms

The problem is simple. If you add an event handler to a form and then change it in an inherited form, there is no way to refer to the two methods using a common variable of the base class, because the event handlers use static binding by default.

Confusing? Here is an example, which is intended for experienced Delphi programmers. Suppose you want to build a bitmap viewer form and a text viewer form in the same program. The two forms have similar elements, a similar toolbar, a similar menu, an OpenDialog component, and different components for viewing the data. So you decide to build a base-class form containing the common elements and inherit the two forms from it. You can see the three forms at design time in Figure 2.8.

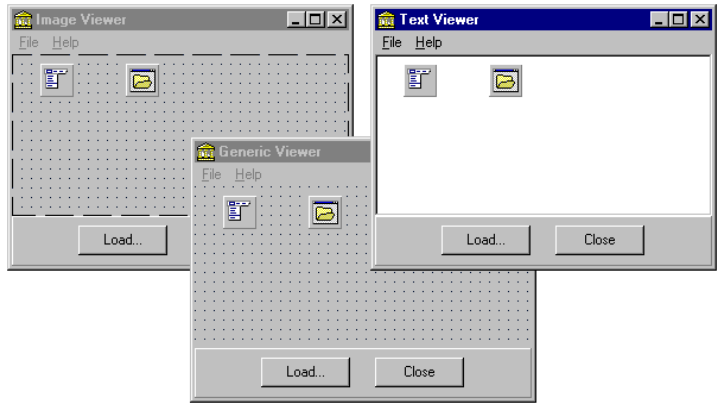


Figure 2.8: The base-class form and the two inherited forms of the PoliForm example at design time (in Delphi 5)

Here is the textual description of the main form:

```
object ViewerForm: TViewerForm
  Caption = 'Generic Viewer'
  Menu = MainMenu1
  object Panel1: TPanel
    Align = alBottom
    object ButtonLoad: TButton…
    object CloseButton: TButton…
  end
  object MainMenu1: TMainMenu
    object File1: TMenuItem...
      object Load1: TMenuItem...
      object N1: TMenuItem...
      object Close1: TMenuItem...
    object Help1: TMenuItem...
      object AboutPoliform1: TMenuItem...
  end
  object OpenDialog1: TOpenDialog...
end
```

The two inherited forms have only minor differences, but they feature a new component, either an image viewer (`TImage`) or a text viewer (`TMemo`):

```
inherited ImageViewerForm: TImageViewerForm
  Caption = 'Image Viewer'
  object Image1: TImage [0]
    Align = alClient
  end
  inherited OpenDialog1: TOpenDialog
    Filter = 'Bitmap file|*.bmp|Any file|*.*'
  end
end
inherited TextViewerForm: TTextViewerForm
```

```
  Caption = 'Text Viewer'
  object Memo1: TMemo [1]
    Align = alClient
  end
  inherited OpenDialog1: TOpenDialog
    Filter = 'Text files|*.txt|Any file|*.*'
  end
end
```

The main form includes some common code. The Close button and the File ➢ Close command call the `Close` method of the form. The Help ➢ About command shows a simple message box. The Load button of the base form has the follow-ing code:

```
procedure TViewerForm.ButtonLoadClick(Sender: TObject);
begin
  ShowMessage ('Error: File loading code missing');
end;
```

The File ➢ Load command, instead, calls another method:

```
procedure TViewerForm.Load1Click(Sender: TObject);
begin
  LoadFile;
end;
```

This method is defined in the `TViewerForm` class as

```
public
  procedure LoadFile; virtual; abstract;
```

Because this is an abstract method, we will need to redefine it (and override it) in the inherited forms:

```
type
  TImageViewerForm = class(TViewerForm)
    Image1: TImage;
    procedure ButtonLoadClick(Sender: TObject);
  public
    procedure LoadFile; override;
  end;
```

The code of this `LoadFile` method simply uses the `OpenDialog1` component to ask the user to select an input file and loads it into the image component:

```
procedure TImageViewerForm.LoadFile;
begin
  if OpenDialog1.Execute then
    Image1.Picture.LoadFromFile (
      OpenDialog1.Filename);
end;
```

The other inherited class has similar code, loading the text into the memo component. The project has one more form, a main form with two buttons, used to reload the files in each of the viewer forms. The main form is the only form created by the project when it starts. The generic viewer form is never created: it is only a generic base class, containing common code and components of the two sub-classes. The forms of the two subclasses are created in the `OnCreate` event handler of the main form:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  FormList [1] := TTextViewerForm.Create (Application);
  FormList [2] := TImageViewerForm.Create (Application);
  for I := 1 to 2 do
    FormList[I].Show;
end;
```

See Figure 2.9 for the resulting forms (with text and image already loaded in the viewers). `FormList` is a polymorphic array of forms, declared in the `TMainForm` class as:

```
private
  FormList: array [1..2] of TviewerForm;
```

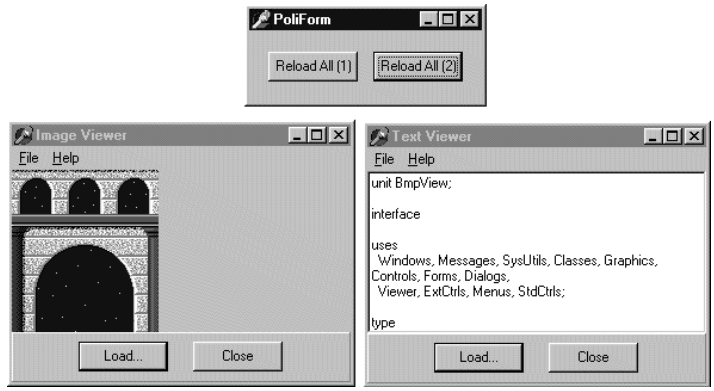


Figure 2.9: The PoliForm example at run time. Image from the original book.

Note that to make this declaration in the class, you need to add the Viewer unit (but not the specific forms) in the `uses` clause of the interface portion of the main form. The array of forms is used to load a new file in each viewer form when one of the two buttons is pressed. The handlers of the two buttons' `OnClick` events use different approaches:

```
procedure TMainForm.ReloadButton1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 2 do
    FormList [I].ButtonLoadClick (self);
end;

procedure TMainForm.ReloadButton2Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 2 do
    FormList [I].LoadFile;
end;
```

The second button simply calls a virtual method, and it will work without any problem. The first button calls an event handler and will always reach the generic `TFormView` class (displaying the error message of its `ButtonLoadClick` method). This happens because the method is static, not virtual.

Is there a way to make this approach work? Sure. Declare the `ButtonLoadClick` method of the `TFormView` class as virtual, and declare it as overridden in each of the inherited form classes, as we do for any other virtual method:

```
type
  TViewerForm = class(TForm)
    // components and plain methods...
    procedure ButtonLoadClick(Sender: TObject); virtual;
  public
    procedure LoadFile; virtual; abstract;
  end;
...
type
  TImageViewerForm = class(TViewerForm)
    Image1: TImage;
    procedure ButtonLoadClick(Sender: TObject); override;
  public
    procedure LoadFile; override;
  end;
```

Simple, isn't it? This trick really works, although it is never mentioned in the Delphi documentation. This ability to use virtual event handlers is what I actually mean by visual form polymorphism.

What's Next?

In this chapter, we have discussed the foundations of object-oriented programming in Object Pascal. We have considered the definition of classes, the use of methods, encapsulation, inheritance, polymorphism, and run-time type information. This is certainly a lot of information if you are a newcomer, but if you are fluent in another OOP language or if you've already used past versions of Delphi, you should be able to apply the topics covered in this chapter to your programming.

The next chapter continues our discussion of how Delphi implements OOP. It covers other language features, such as method pointers, class references, properties, events, and exceptions, which are particularly important to support Delphi's visual development style. Chapter 3 also shows how to define your own components. Chapter 4 then focuses on the structure of the VCL (Visual Component Library) and discusses a few important classes.

Understanding the secrets of Object Pascal and the structure of the VCL is vital for becoming an expert Delphi programmer. These topics form the foundation of working with the VCL; after exploring them in the next two chapters, we'll *finally* go on in Part II of the book to explore the development of real applications using all the various components provided by Delphi.

Chapter 3: Advanced Object Pascal

In the last chapter you've seen the foundations of the Object Pascal language used by Delphi: classes, objects, methods, constructors, inheritance, late binding, and run-time type information. Now we need to move one step further, by looking at some more advanced features of the language[116]. Some of the extensions discussed in this chapter, particularly the `published` keyword, properties, and events, are

116 Since the time this book was published, the Delphi language has been largely extended with significant improvements to classes (with strict access specifiers, nested types, class data, class properties, class constructors, and more), to the core language (inline routines, for-in loops, records with methods and operators overloading), and later with features opening up for different programming models, like generics, anonymous methods, and extended RTTI. All of these features build on top of the core capabilities of Delphi discussed in this book, which remain relevant.

strictly related to Delphi's visual programming model. In fact, while discussing these topics, I'll show you how to build a simple custom component.

Some other elements of Object Pascal, such as exceptions and interfaces, are not so closely related with the visual elements of Delphi. Still, it's important to know them, as well as a few other elements discussed in this chapter, to write correct code in your Delphi applications.

Class Methods and Class Data

When you define a field in a class, you actually specify that the field should be added to each object instance of that class. Each instance has its own independent representation (referred to by the `Self` pointer). In some cases, however, it might be useful to have a field that is shared by all the objects of a class.

Other object-oriented programming languages have formal constructs to express this, such as `static` in C++. But in Object Pascal, we can simulate this feature using the encapsulation provided at the unit level[117]. You can simply add a variable in the `implementation` portion of a unit, to obtain a class variable—a single memory location shared by all of the objects of a class.

If you need to access this value from outside the unit, you might use a method of the class. However, this forces you to apply this method to one of the instances of the class. An alternative solution is to declare a *class method*. A class method cannot access the data of any single object but can be applied to a class as a whole rather than to a particular instance. A class method is related to the class, not to its objects or instances (like a `static` member function in C++ or Java[118]).

To declare a class method in Object Pascal, you simply add the `class` keyword in front of it:

```
type
  MyClass = class
```

117 The lack of a formal declaration for class data has been filled with the "*class var*" construct, which let's you define true class data and works properly in case on inheritance and for generic classes, two areas in which the technique proposed in Mastering Delphi 5 falls short.

118 Another addition to the language, since the days this book was written, is the availability of "static" class methods, which are very similar to their C++, Java, or C# counterparts. The difference with the standard class methods in Delphi is that these have a hidden *self* parameter referring to the class, unlike static class methods which are for all purposes identical to global functions (to the point that they can be used as Windows callback functions).

```
    class function ClassMeanValue: Integer;
```

The use of class methods is not very common in Object Pascal, because you can obtain the same effect by adding a procedure or function to a unit declaring a class. Object-oriented purists, however, will definitely prefer the use of a class method over a routine unrelated to a class. And actually the VCL uses class methods quite often, although there are also many global subroutines. Notice that in Delphi, class methods can also be virtual[119], so they can be overridden and used to obtain polymorphism.

A Class with an Object Counter

When unit data is used to maintain general information related to the class (such as the number of objects created or a list of these objects), you can use class methods to access that data. That is exactly what the next example does.

The CountObj program is an extension of the CreateC example from the last chapter. The form is still quite bare, but I've added some new code. In particular I've added a brand-new class, which inherits from the `TButton` class of the VCL and adds a new feature, namely object counting. Here is the declaration of the new class:

```
type
  TCountButton = class (TButton)
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    class function GetTotal: Integer;
  end;
```

note What you see here is a perfectly working custom component. In this case, we won't register it and won't add it to Delphi's Components palette, even if this is not a particularly difficult operation. Customizing existing components can be really that simple! We'll cover this topic a little further in this chapter and in much more detail in Chapter 13.

Every time an object is created, the program increments the counter before calling the constructor of the base class. Every time an object is destroyed, the counter is decreased:

```
constructor TCountButton.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
```

119 This is a unique feature across programming languages, which combines nicely with another uncommon Delphi feature, class references.

```
  Inc (TotBtns);
end;

destructor TCountButton.Destroy;
begin
  Dec (TotBtns);
  inherited Destroy;
end;
```

The counter is a variable declared in the `implementation` portion of the unit and so is not accessible outside the unit. Only the class method allows us to read its current value. You can directly initialize this variable when it is defined:

```
implementation

var
  TotBtns: Integer = 0;

class function TCountButton.GetTotal: Integer;
begin
  Result := TotBtns;
end;
```

Now we can create objects of this new type by changing the code of the `FormMouseDown` method slightly:

```
begin
  with TCountButton.Create (Self) do
  begin
    Parent := Self;
    // same code as before...
```

Every time a `TCountButton` object is created, the current number of objects is displayed at the beginning of its caption. We can call the `GetTotal` class method for the newly created object (notice that we are inside a `with` statement), just as we call any plain method. However, we can call the same method without a valid object instance. This is what we do when the interval of a timer I've added to the form elapses:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Caption := Format ('CountObj: %d custom buttons',
    [TCountButton.GetTotal]);
end;
```

The `Caption` property in this code refers to the caption of the form. You can see the effect of this call in Figure 3.1. The drawback of this example is that we can only create objects and never destroy them, so we see the total number of live objects always increasing and never reducing its value.

To see that the number of objects in existence goes down to zero, we can try to check the number of objects after the form has been destroyed, along with the `TCountButton` objects it owns. This is the code I've added at the end of the unit:

```
finalization
  MessageBox (0, PChar (Format (
     'There are %d CountButton objects',
     [TCountButton.GetTotal])),
     'Finalization', mb_OK);
end.
```

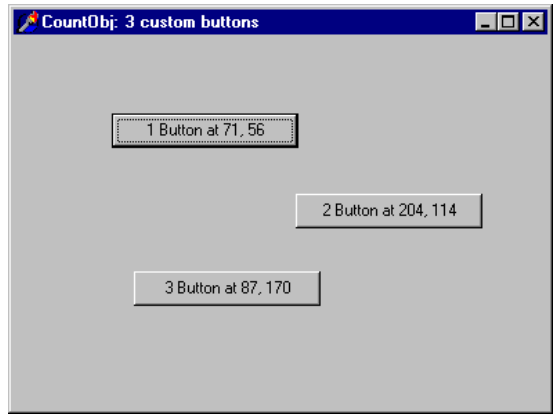


Figure 3.1: The output the CountObj example after a couple of TCountButton objects have been created. Image from the original book.

note In the `finalization` code above I had to use a Windows API function (`MessageBox`) instead of a Delphi procedure (such as `ShowMessage`). The reason is that the `finalization` code of the unit is executed after some of the Delphi global objects have been destroyed, so it is better not to rely on them.

The program simply displays a Windows message box indicating the number of objects in existence, a value obtained by calling the `GetTotal` class method. If you run the program, the number in the output is zero, although I have to say that this is not guaranteed but is due to the order in which objects are destroyed. The compiler uses a specific order for units initialization and finalization: starting with the project source code, the units referred to are initialized before and finalized after the units that refer to them. Typically, the project will initialize the Forms unit, which in turn initializes other VCL units, and then it initializes your form unit, which will first initialize the units describing the components you use (that is, those in the `uses` statement).

However, at first sight, it is not simply to determine when in this sequence the main form of the application and the components it owns are going to be destroyed. To test that everything actually works, I've added the same `MessageBox` call code in the handler of the `OnDestroy` event of the form, triggered before the form is destroyed.

If you run the program, you'll see that when the `FormDestroy` method is executed, all of the objects you've created still exist; but right after that, the objects are destroyed and the count decreases to zero. We'll see a more complete example, in which we'll destroy the buttons at run time, after we discuss method pointers in the following section "The Updated Counter Example."

Method Pointers

Another Delphi addition to the Object Pascal language is the concept of *method pointers*. A method pointer type is like a procedural type, but one that refers to a method[120]. Technically, a method pointer type is a procedural type that has an implicit `Self` parameter. In other words, a method pointer stores two addresses: the address of the method code and the address of an object instance (data). The address of the object instance will show up as `Self` inside the method body when the method code is called using this method pointer. This explains the definition of Delphi's generic `TMethod` type, a record with a `Code` field and a `Data` field.

The declaration of a method pointer type is similar to that of a procedural type, except that it has the keywords `of object` at the end of the declaration:

```
type
  IntProceduralType = procedure (Num: Integer);
  IntMethodPointerType = procedure (Num: Integer) of object;
```

When you have declared a method pointer, such as the one above, you can declare a variable of this type and assign to it a compatible method of another object. What's a compatible method? One that has the same parameters as those requested by the method pointer type, such as a single Integer parameter in the example above.

At first glance, the goal of this technique may not be clear, but this is one of the cornerstones of Delphi component technology. The secret is in the word *delegation*. If someone has built an object that has some method pointers, you are free to change the object's behavior simply by assigning new methods to the pointers. Does this sound familiar? It should.

---

120 The language now offers a different, related feature: anonymous methods. Methods pointers remain the foundation for event handlers.

When you add an `OnClick` event handler for a button, Delphi does exactly that. The button has a method pointer, named `OnClick`, and you can directly or indirectly assign a method of the form to it. When a user clicks the button, this method is executed, even if you have defined it inside another class (typically, in the form).

What follows is a listing that sketches the code actually used by Delphi to define the event handler of a button component and the related method of a form:

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

  MyButton = class
    OnClick: TNotifyEvent;
  end;

  TForm1 = class (TForm)
    procedure Button1Click (Sender: TObject);
    Button1: MyButton;
  end;

var
  Form1: TForm1;
```

Now inside a procedure, you can write

```
MyButton.OnClick := Form1.Button1Click;
```

The only real difference between this code fragment and the code of the VCL is that `OnClick` is a property name, and the actual data it refers to is called `FOnClick`. An event that shows up in the Events page of the Object Inspector, in fact, is nothing more than a property of a method pointer type.

This means, for example, that you can dynamically modify the event handler attached to a component at design time or even build a new component at run time and assign an event handler to it. For example, we could add to the form of the Counter example the following method:

```
procedure TForm1.ButtonClick (Sender: TObject);
begin
  ShowMessage ('Button pressed');
end;
```

and then write for each newly created button the following code:

```
with TCountButton.Create (Self) do
begin
  OnClick := ButtonClick;
```

With this code each of the buttons will react to a click of the mouse by showing a common message, because all components share the same handler. However, we

can use the `Sender` parameter of the event to customize it for each button. This is what I'll do in the example discussed in the sidebar "The Updated Counter Example," which is an even more complete extension of the Counter program.

The Updated Counter Example

Now that we know how to use method pointers, we can update the CountObj example by using them. The name of the new example is CountOb2. Its purpose is to add a handler for the `OnKeyPress` event of the new objects a user creates dynamically. Add the following code in the form class declaration:

```
procedure ButtonKeyPress(Sender: TObject; var Key: Char);
```

The parameters are those required for an event of this kind. If you select the `OnKeyPress` event for a component of a form and press the F1 key to invoke the Help file, you'll find the following declaration:

```
TKeyPressEvent = procedure (Sender: TObject; var Key: Char) of object;
property OnKeyPress: TKeyPressEvent;
```

As you can see in this last line, the event is based on the `TKeyPressEvent` method pointer type, listed in the line before. Therefore, we need to write a method that complies with this method pointer type, like the one presented in the previous section.

To connect this method with the `OnKeyPress` event of the buttons we create dynamically, we need just one line of code in the `FormMouseDown` method:

```
with TCountButton.Create (Self) do
begin
  ...
  // set the event handler
  OnKeyPress := ButtonKeyPress;
  // grab the input focus
  SetFocus;
end;
```

The second line of code moves the input focus to the newly created button, so that subsequent keyboard input will be directed to it.

Now we can write the code of the `ButtonKeyPress` method. Press the Ctrl+Shift+C key combination to activate Delphi's Code Completion, and then fill the method declaration with some actual code. In this example, we should destroy the button when the user presses the Backspace key. Because keyboard input is sent to the control that has the input focus, you can simply click a button or use the Tab key to select the button you want to destroy; then press the Backspace key.

The first approach I tried in developing this application was simply to destroy the object passed as `Sender` parameter, which is the object that received the event:

```
procedure TForm1.ButtonKeyPress(Sender: TObject;
  var Key: Char);
begin
  if Key = #8 then
    Sender.Free
end;
```

This code generates an exception. We cannot destroy an object while we are processing one of its events. Instead, we must delay the object destruction. There are basically two approaches. We can save the object we want to destroy in a private field of the form class and later destroy it, inside some code periodically activated by a timer. You can find this code in the CountOld example. Notice that the use of the timer causes a little flaw in the program: if two backspace keys are processed before the timer fires, only one button is going to be destroyed.

The second approach, implemented in the CountOb2 example, is to send a custom Windows message (such as `wm_User`) to the form using the `PostMessage` API function. This introduces a delay, because the message has to reach the window and will be retrieved and elaborated after the current event handler has completed its execution. To follow this second approach, we can write the following handler for the `OnKeyPress` event of each new button:

```
procedure TForm1.ButtonKeyPress(Sender: TObject;
  var Key: Char);
begin
  // if user pressed backspace
  if Key = #8 then
  begin
    // set this as the object to destroy
    ToDestroy := Sender as TButton;
    // post message to perform destruction
    PostMessage (Handle, wm_User, 0, 0);
  end;
end;
```

In this code `ToDestroy` is a private field of the form of the `TButton` data type. This field is automatically set to `nil` (no object to destroy) when the form is first created (this is the default initialization for class fields). When the user presses the Backspace key, the current button object (the `Sender` of the `ButtonKeyPress` method) is stored in the `ToDestroy` field. At this point, the `PostMessage` Windows API call sends a message to the current window (identified by the value of its `Handle` property). The handler of this message is defined in the form class as follows:

```
type
```

```
  TForm1 = class(TForm)
    ...
  private
    ToDestroy: TButton;
  public
    procedure WmUser (var Msg: TMessage); message wm_User;
```

Now we can look at the code of this method, in which the program can double-check whether there is a button to destroy before destroying it and setting it to `nil`:

```
procedure TForm1.WmUser (var Msg: TMessage);
begin
  // if there is an object to destroy
  if Assigned (ToDestroy) then
  begin
    // moves the input focus to the next control
    SelectNext (ToDestroy, True, True);
    // destroy the object and set the reference to nil
    FreeAndNil (ToDestroy);
  end;
  // update the form caption
  Caption := Format ('CountObj: %d custom buttons',
    [TCountButton.GetTotal]);
end;
```

To make the program behave a little better before destroying an object, I moved the input focus to the next control by calling the `SelectNext` method. Then the program calls the `FreeAndNil` procedure, which calls the `Free` method of the object, which in turn invokes the destructor `Destroy`. Because the destructor is virtual, the program invokes the overridden destructor of the `TCountButton` class, which decrements the object counter. For this reason I've placed the code that destroys the object before the code that updates the form caption. Before calling `Free`, `FreeAndNil` sets the `ToDestroy` reference to `nil`.

More about freeing objects and memory management is in the section "Objects and Memory," later in this chapter.

Class References

Having looked at several topics related to methods, we can now move on to the topic of *class references* and extend our example of dynamically creating components even further. The first point to keep in mind is that a class reference isn't a class, it isn't an object, and it isn't a reference to an object; it is simply a reference to a class type.

A class reference type determines the type of a class reference variable. Sounds confusing? A few lines of code might make this a little clearer. Suppose you have defined the class `TMyClass`. You can now define a new class reference type, related to that class:

```
type
  TMyClassRef = class of TMyClass;
```

Now you can declare variables of both types. The first variable refers to an object, the second to a class:

```
var
  AClassRef: TMyClassRef;
  AnObject: TMyClass;
begin
  AClassRef := TMyClass;
  AnObject := TMyClass.Create;
```

You may wonder what class references are used for. In general, class references allow you to manipulate a class data type at run time. You can use a class reference in any expression where the use of a data type is legal. Actually, there are not many such expressions, but the few cases are interesting. The simplest case is the creation of an object. We can rewrite the two lines above as follows:

```
AClassRef := TMyClass;
AnObject := AClassRef.Create;
```

This time I've applied the `Create` constructor to the class reference instead of to an actual class; I've used a class reference to create an object of that class.

---

note Class references remind us of the concept of *metaclass* available in other OOP languages. In Object Pascal, however, a class reference is not itself a class but only a type pointer. Therefore, the analogy with metaclasses (classes describing other classes) is a little misleading. Actually, `TMetaclass` is also the term used in C++Builder.

Class reference types wouldn't be as useful if they didn't support the same type-compatibility rule that applies to class types. When you declare a class reference variable, such as `MyClassRef` above, you can then assign to it that specific class and any subclass. So if `MyNewClass` is a subclass of my class, you can also write

```
AClassRef := MyNewClass;
```

Delphi declares a lot of class references in the run-time library and the VCL, including the following:

```
TClass = class of TObject;
```

```
ExceptClass = class of Exception;
TComponentClass = class of TComponent;
TControlClass = class of TControl;
TFormClass = class of TForm;
```

In particular, the `TClass` class reference type can be used to store a reference to any class you write in Delphi, because every class is ultimately derived from `TObject`. The `TFormClass` reference, instead, is used in the source code of most Delphi projects. The `CreateForm` method of the `Application` object, in fact, requires as parameter the class of the form to create:

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

The first parameter is a class reference, the second is a variable that stores a reference to the created object instance.

Finally, when you have a class reference you can apply to it the class methods of the related class. Considering that each class inherits from `TObject`, you can apply to each class reference some of the methods of `TObject`, including `InstanceSize`, `ClassName`, `ParentClass`, and `InheritsFrom`. I'll discuss these class methods and other methods of `TObject` class in the next chapter.

Creating Components Using Class References

What is the *practical* use of class references in Delphi? Being able to manipulate a data type at run time is a fundamental element of the Delphi environment. When you add a new component to a form by selecting it from the Component Palette, you select a data type and create an object of that data type. (Actually, that is what Delphi does for you behind the scenes.)

To give you a better idea of how class references work, I've built an example named ClassRef. The form displayed by this example is quite simple. It has three radio buttons, placed inside a panel in the upper portion of the form. When you select one of these radio buttons and click the form, you'll be able to create new components of the three types indicated by the button labels: radio buttons, push buttons, and edit boxes.

To make this program run properly, you need to change the names of the three components. The form must also have a class reference field:

```
private
  ClassRef: TControlClass;
  Counter: Integer;
```

The first field stores a new data type every time the user clicks one of the three radio buttons. Here is one of the three methods:

```
procedure TForm1.RadioButtonRadioClick(Sender: TObject);
begin
  ClassRef := TRadioButton;
end;
```

The other two radio buttons have `OnClick` event handlers similar to this one, assigning the value `TEdit` or `TButton` to the `ClassRef` field. A similar assignment is also present in the handler of the `OnCreate` event of the form, used as an initialization method.

The interesting part of the code is executed when the user clicks the form. Again, I've chosen the `OnMouseDown` event of the form to hold the position of the mouse click:

```
procedure TForm1.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  NewCtrl: TControl;
  MyName: String;
begin
  // create the control
  NewCtrl := ClassRef.Create (Self);
  // hide it temporarily, to avoid flickering
  NewCtrl.Visible := False;
  // set parent and position
  NewCtrl.Parent := Self;
  NewCtrl.Left := X;
  NewCtrl.Top := Y;
  // compute the unique name (and caption)
  Inc (Counter);
  MyName := ClassRef.ClassName + IntToStr (Counter);
  Delete (MyName, 1, 1);
  NewCtrl.Name := MyName;
  // now show it
  NewCtrl.Visible := True;
end;
```

The first line of the code for this method is the key. It creates a new object of the class data type stored in the `ClassRef` field. We accomplish this simply by applying the `Create` constructor to the class reference. Now you can set the value of the `Parent` property, set the position of the new component, give it a name (which is automatically used also as `Caption` or `Text`), and make it visible.

Notice in particular the code used to build the name; to mimic Delphi's default naming convention, I've taken the name of the class with the expression `ClassRef.ClassName`, using a class method of the `TObject` class. Then I've added a

number at the end of the name and removed the initial letter of the string. For the first radio button, the basic string is `TRadioButton`, plus the *1* at the end, and minus the *T* at the beginning of the class name—*RadioButton1*. Sounds familiar?

You can see two examples of the output of this program in Figure 3.2. Notice that the naming is not exactly the same as used by Delphi. Delphi uses a separate counter for each type of control; I've used a single counter for all of the components. If you place a radio button, a push button, and an edit box in a form of the ClassRef example, their names will be *RadioButton1, Button2,* and *Edit3*.

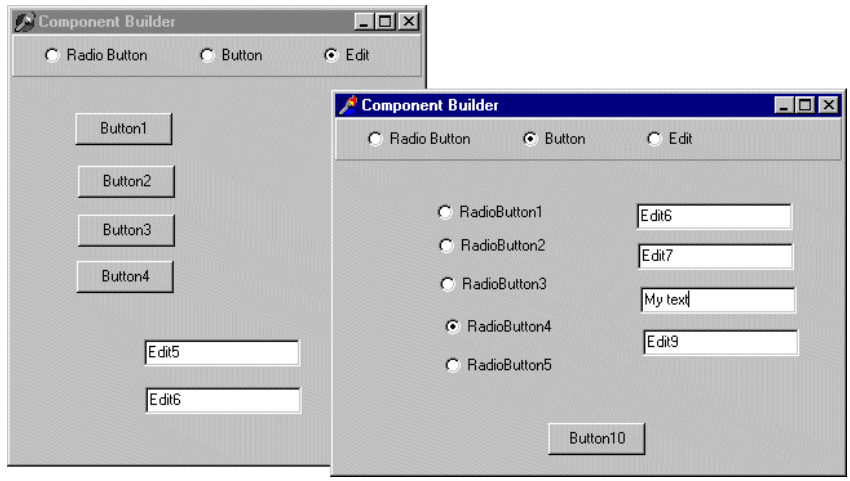


Figure 3.2: Two examples of the output of the ClassRef example, in two different windows

Objects and Memory

Memory management in Delphi is subject to two simple rules: You must destroy every object you create, and you must destroy each object only once. Delphi supports three types of memory management for dynamic elements (that is, elements not in the stack and the global memory area):

- Every time you create an object, you should also free it. If you fail to do so, the memory used by that object won't be released for other objects, until the program terminates.
- When you create a component, you can specify an owner component, passing the owner to the component constructor. The owner component (often a form)

becomes responsible for destroying all the objects it owns. In other words, when you free the form, it frees all the components it owns. So, if you create a component and give it an owner, you don't have to remember to destroy it.

- When you allocate memory for strings, dynamic arrays, and objects referenced by interface variables (discussed at the end of this chapter), Delphi automatically frees the memory when the reference goes out of scope. You don't need to free a string: when it becomes unreachable, its memory is released.

We'll see how the issue of memory management affects actual examples when discussing applications with multiple forms in Part II of the book.

Destroying Objects Only Once

Another problem is that if you call the destructor of an object twice, you get an error. A *destructor* is a method that de-allocates an object's memory. We can write code for a destructor, generally overriding the default `Destroy` destructor, to let the object execute some code before it is destroyed. In your code, of course, you don't have to handle memory de-allocation—this is something Delphi does for you.

`Destroy` is simply a virtual destructor of the `TObject` class. Most of the classes that require custom clean-up code when the objects are destroyed override this virtual method. The reason you should never define a new destructor is that objects are usually destroyed by calling the `Free` method, and this method calls the `Destroy` virtual destructor (possibly the overridden version) for you.

As I've just mentioned, `Free` is simply a method of the `TObject` class, inherited by all other classes. The `Free` method basically checks whether the current object (`Self`) is not `nil` before calling the `Destroy` virtual destructor.

note You might wonder why you can safely call `Free` if the object reference is `nil`, but you can't call `Destroy`. The reason is that `Free` is a known method at a given memory location, whereas the virtual function `Destroy` is determined at run time by looking at the type of the object, a very dangerous operation if the object doesn't exist any more.

Here is its pseudo-code (the actual code in the RTL source code files is written in assembler):

```
procedure TObject.Free;
begin
  if Self <> nil then
    Destroy;
```

```
end;
```

Next, we can turn our attention to the `Assigned` function. When we pass a pointer to this function, it simply tests whether the pointer is `nil`. So the following two statements (from the CountOb2 example) are equivalent, at least in most cases:

```
if Assigned (ToDestroy) then ...
if ToDestroy <> nil then ...
```

Notice that these statements test only whether the pointer is not `nil`; they do not check whether it is a valid pointer. If you write the following code

```
ToDestroy.Free;
if ToDestroy <> nil then
  ToDestroy.DoSomething;
```

the test will be satisfied, and you'll get an error on the line with the call to the method of the object. It is important to realize that calling `Free` doesn't set the object to `nil`.

Automatically setting an object to `nil` is not possible. You might have several references to the same object, and Delphi doesn't track them. At the same time, within a method (such as `Free`) we can operate on the object, but we know nothing about the object reference—the memory address of the variable we've used to call the method. In other words, inside the `Free` method or any other method of a class, we know the memory address of the object (`Self`), but we don't know the memory location of the variable referring to the object, such as `ToDestroy`. Therefore, the `Free` method cannot affect the `ToDestroy` variable.

However, when we call an external procedure, such as `FreeAndNil` in Delphi 5, the procedure knows about the object reference, passed as a parameter, and can act on it. Here is Delphi code for `FreeAndNil`[121]:

```
procedure FreeAndNil(var Obj);
var
  P: TObject;
begin
  P := TObject(Obj);
  // clear the reference before destroying the object
  TObject(Obj) := nil;
  P.Free;
end;
```

To sum things up, here are a couple of guidelines:

---

121 The code today is slightly different, as it assigns the nil value before freeing the object, to be safe in multi-threaded applications.

- Always call `Free` to destroy objects, instead of calling the `Destroy` destructor.
- Use `FreeAndNil`, or set object references to `nil` after calling `Free`, unless the reference is going out of scope immediately afterward.

Passing and Copying Objects

Another important element to discuss is passing objects as parameters or assigning an object to another one. If you write

```
var
  Button2: TButton;
begin
  Button2 := Button1;
```

you don't create a new object but rather a new reference to the same object in memory. There is only one object in memory, and both the `Button1` and `Button2` variables refer to it. The same happens if you pass an object as parameter to a function: you don't create a new object, but you refer to the same one in two different places of the code.

For example, by writing this procedure and calling it as follows, you'll modify the caption of `Button1`, or `Button` if you prefer:

```
procedure ChangeCaption (Button: TButton; Text: string);
begin
  Button.Caption := Text;
end;

// call...
ChangeCaption (Button1, 'Hello')
```

What if you need to create a new object, instead? You'll basically have to create it and then copy each of the relevant properties. Some classes, notably some VCL classes derived from `TPersistent`, define an `Assign` method to copy the data of an object. For example, you can write

```
ListBox1.Items.Assign (Memo1.Lines);
```

Even if you assign those properties directly, Delphi will execute a similar code for you. In fact, the `SetItems` method connected with the items property of the list box calls the `Assign` method of the `TStringList` class representing the actual items of the listbox.

You can use complex techniques based on streaming to clone a component in memory, but most of the time, creating a new object of the same type as the current one

and assigning a few properties to it might do the trick. To do this, you can ask the component its class and then use the class reference to create a new object of that type. Here is the code (extracted from the ObjClone example), which clones the `Sender` object:

```
procedure TForm1.ClickComp(Sender: TObject);
var
  ControlText: string;
begin
  with TControlClass (Sender.ClassType).Create (Self) do
  begin
    Parent := (Sender as TControl).Parent;
    Left := (Sender as TControl).Left + 10;
    Top := (Sender as TControl).Top + 10;
    SetLength (ControlText, 50);
    (Sender as TControl).GetTextBuf(
      PChar(ControlText), 50);
    ControlText := PChar(ControlText) + ' *';
    SetTextBuf (PChar (ControlText));
  end;
end;
```

This method takes the class of the `Sender` object, the component clicked by the user, and calls the `Create` constructor. To call the `Create` constructor of the `TControl` class instead of calling that of the `TObject` class, the program has to cast the class reference to the proper type. When we cast to `TControlClass` and then call `Create`, the result is an object of class `TControl`. This object is used inside the `with` statement, and the program sets its `Parent`, `Left`, and `Top` properties using information extracted from the `Sender` control.

At the end of the `with` statement, the program extracts the text of the `Sender` object, using the `GetTextBuf` method, which is available for every control. In fact, the `Text` and `Caption` properties aren't defined inside the `TControl` class. After adding an asterisk to the string, the program uses the string as the new text of the control, again calling the `SetTextBuf` method of the `TControl` class.

You can see the effect of cloning on some of the controls in Figure 3.3. The ObjClone program is also capable of cloning an entire form, using a similar technique.

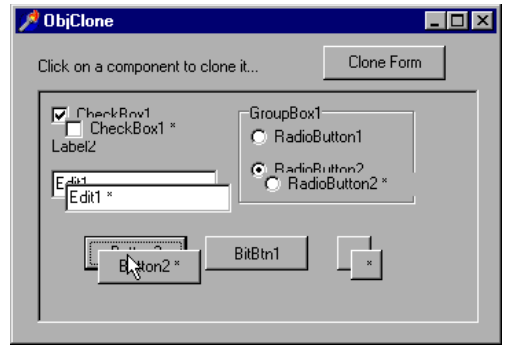


Figure 3.3:
The ObjClone example. Image from the original book.

Handling Exceptions

The last interesting feature of Object Pascal we will cover in this chapter is *exception handling*. The idea of exceptions is to make programs more robust by adding the capability of handling software or hardware errors in a simple and uniform way. A program can survive such errors or terminate gracefully, allowing the user to save data before exiting. Exceptions allow you to separate the error handling code from your normal code, instead of intertwining the two. You end up writing code that is more compact and less cluttered by maintenance chores unrelated to the actual programming objective.

Another benefit is that exceptions define a uniform and universal error-reporting mechanism, which is also used by Delphi components. At run time, Delphi raises exceptions when something goes wrong. If your code has been written properly, it can acknowledge the problem and try to solve it; otherwise, the exception is passed to its calling code, and so on. Ultimately, if no part of your code handles the exception, Delphi handles it, by displaying a standard error message and trying to continue the program.

The whole mechanism is based on four keywords:

- `try` delimits the beginning of a protected block of code.
- `except` delimits the end of a protected block of code and introduces the exception-handling statements, with this syntax form:

```
on exception-type do statement
```

- `finally` is used to specify blocks of code that must always be executed, even when exceptions occur.
- `raise` is the statement used to generate an exception. Most exceptions you'll encounter in your Delphi programming will be generated by the system, but you can also `raise` exceptions in your own code when it discovers invalid or inconsistent data at run time. The `raise` keyword can also be used inside a handler to *re-raise* an exception; that is, to propagate it to the next handler.

Here is an example of a simple protected block:

```
function DivideTwicePlusOne (A, B: Integer): Integer;
begin
  try
    // error if B equals 0
    Result := A div B;
    // do something else... skip if exception is raised
    Result := Result div B;
    Result := Result + 1;
  except
    on EDivByZero do
      Result := 0;
  end;
end;
```

In the exception-handling statement, we catch the `EDivByZero` exception, which is defined by Delphi. There are a number of these exceptions referring to run-time problems (such as a division by zero or a wrong dynamic cast), to Windows resource problems (such as out-of-memory errors), or to component errors (such as a wrong index). Programmers can also define their own exceptions; simply create a new sub-class of the default exception class or one of its sub-classes:

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

When you add a new element to an array that is already full (probably because of an error in the logic of the program), you can raise the corresponding exception by creating an object of this class:

```
if MyArray.Full then
  raise EArrayFull.Create ('Array full');
```

This `Create` method (inherited from the `Exception` class) has a string parameter to describe the exception to the user. You don't need to worry about destroying the object you have created for the exception, because it will be deleted automatically by the exception-handler mechanism.

The code presented in the previous excerpts is part of a sample program, called Except. Some of the routines have actually been slightly modified, as in the following `DivideTwicePlusOne` function:

```
function DivideTwicePlusOne (A, B: Integer): Integer;
begin
  try
    // error if B equals 0
    Result := A div B;
    // do something else... skip if exception is raised
    Result := Result div B;
    Result := Result + 1;
  except
    on EDivByZero do
    begin
      Result := 0;
      MessageDlg ('Divide by zero corrected',
        mtError, [mbOK], 0);
    end;
    on E: Exception do
    begin
      Result := 0;
      MessageDlg (E.Message,
        mtError, [mbOK], 0);
    end;
  end; // end except
end;
```

note When you run a program in the debugger, the debugger will stop the program by default when an exception is encountered. This is normally what you want, of course, because you'll know where the exception took place and can see the call of the handler step-by-step. In the case of the Except test program, however, this behavior will confuse the program's execution. In fact, even if the code is prepared to properly handle the exception, the debugger will stop the program execution at the source code line closest to where the exception was raised. Then, moving step-by-step through the code, you can see how it is handled. If you just want to let the program run when the exception is properly handled, run the program with the "Run without debugging" menu command.

In this code there are two different exception handlers after the same `try` block. You can have any number of these handlers, which are evaluated in sequence. For this reason, you need to place the broader handlers (the handlers of the ancestor `Exception` classes) at the end.

In fact, using a hierarchy of exceptions, a handler is also called for the subclasses of the type it refers to, as any procedure will do. This is polymorphism in action again. But keep in mind that using a handler for every exception, such as the one above, is not usually a good choice. It is better to leave unknown exceptions to Delphi. The default exception handler in the VCL displays the error message of the exception

class in a message box, and then resumes normal operation of the program. You can actually modify the normal exception handler with the `Application.OnException` event, as demonstrated in the ErrorLog example later in this chapter.

Another important element of the code above is the use of the exception object in the handler (see `on E: Exception do`). The object `E` of class `Exception` receives the value of the exception object passed by the `raise` statement. When you work with exceptions, remember this rule: You raise an exception by creating an object and handle it by indicating its type. This has an important benefit, because as we have seen, when you handle a type of exception, you are really handling exceptions of the type you specify as well as each descendant type.

Delphi defines a hierarchy of exceptions, and you can choose to handle each specific type of exception in a different way or handle groups of them together. You can find a list of the Delphi exception classes on www.marcocantu.com/d5ref[122].

Exceptions and the Stack

When the program raises an exception and the current routine doesn't handle it, what happens to your function call stack? The program starts searching for a handler among the functions already on the stack. This means that the program exits from existing functions and does not execute the remaining statements. To understand how this works, you can either use the debugger or add a number of simple message boxes to the code, to be informed when a certain source code statement is executed. In the next example, Except2, I've followed this second approach.

For example, when you press the Raise2 button in the form of the Except2 example, an exception is raised and not handled, so that the final part of the code will never be executed:

```
procedure TForm1.ButtonRaise2Click(Sender: TObject);
begin
  // unguarded call
  AddToArray (24);
  ShowMessage ('Program never gets here');
end;
```

Notice that this method calls the `AddToArray` procedure, which invariably raises the exception. When the exception is handled, the flow starts again after the handler and not after the code that raises the exception. Consider this modified method:

---

122 That page and that list don't exists any more.

```
procedure TForm1.ButtonRaise1Click(Sender: TObject);
begin
  try
    // this procedure raises an exception
    AddToArray (24);
    ShowMessage ('Program never gets here');
  except
  on EArrayFull do
      ShowMessage ('Handle the exception');
  end;
  ShowMessage ('ButtonRaise1Click call completed');
end;
```

The last `ShowMessage` call will be executed right after the second one, while the first is always ignored. I suggest that you run the program, change its code, and experiment with it to fully understand the program flow when an exception is raised.

The Finally Block

There is a fourth keyword for exception handling that I've mentioned but haven't used so far, `finally`. A `finally` block is used to perform some actions (usually cleanup operations) that should always be executed. In fact, the statements in the `finally` block are processed whether or not an exception takes place. The plain code following a `try` block, instead, is executed only if an exception was not raised or if it was raised and handled. In other words, the code in the `finally` block is always executed after the code of the `try` block, even if an exception has been raised.

Consider this method (part of the Except3 example), which performs some time-consuming operations and uses the hourglass cursor to show the user that it's doing something:

```
procedure TForm1.BtnWrongClick(Sender: TObject);
var
  I, J: Integer;
begin
  Screen.Cursor := crHourglass;
  J := 0;
  // long (and wrong) computation...
  for I := 1000 downto 0 do
    J := J + J div I;
  MessageDlg ('Total: ' + IntToStr (J),
    mtInformation, [mbOK], 0);
  Screen.Cursor := crDefault;
end;
```

Because there is an error in the algorithm (as the variable `I` can reach a value of 0 and is also used in a division), the program will break, but it won't reset the default cursor. This is what a `try-finally` block is for:

```
procedure TForm1.BtnTryFinallyClick(Sender: TObject);
var
  I, J: Integer;
begin
  Screen.Cursor := crHourglass;
  J := 0;
  try
    // long (and wrong) computation...
    for I := 1000 downto 0 do
      J := J + J div I;
    MessageDlg ('Total: ' + IntToStr (J),
      mtInformation, [mbOK], 0);
  finally
    Screen.Cursor := crDefault;
  end;
end;
```

When the program executes this function, it always resets the cursor, whether an exception (of any sort) occurs or not. The drawback to this version of the function is that it doesn't handle the exception. Strangely enough, this is not possible. A `try` block can be followed by either an `except` or a `finally` statement but not both of them at the same time. The typical solution is to use two nested `try` blocks, associating the internal one with a `finally` statement and the external one with an `except` statement or vice versa, as the situation requires. Here is the code of this third button of the Except3 example:

```
procedure TForm1.BtnTryTryClick(Sender: TObject);
var
  I, J: Integer;
begin
  Screen.Cursor := crHourglass;
  J := 0;
  try try
    // long (and wrong) computation...
    for I := 1000 downto 0 do
      J := J + J div I;
    MessageDlg ('Total: ' + IntToStr (J),
      mtInformation, [mbOK], 0);
  finally
    Screen.Cursor := crDefault;
  end;
  except
    on E: EDivByZero do
    begin
      // re-raise the exception with a new message
      raise Exception.Create ('Error in Algorithm');
```

```
    end;
  end;
end;
```

You should always protect blocks with the `finally` statement, to avoid resource or memory leaks in case an exception is raised. Handling the exception is probably less important, since Delphi can survive most of them.

Logging Errors

Most of the time, you don't know which operation is going to `raise` an exception, and you cannot (and should not) wrap each and every piece of code in a `try-except` block. An alternative approach is to let Delphi handle all the exceptions and pass them all to you, by handling the `OnException` event of the global `Application` object. In early versions of Delphi you could handle this event by writing a proper method and connecting in the code. Now Delphi provides the ApplicationEvents component we can use to build this example. (More on the global `Application` object and the ApplicationEvents component in Chapter 6).

In the ErrorLog example, I've added to the main form a copy of the Application-Events component, and added a handler for its `OnException` event:

```
procedure TFormLog.LogException(Sender: TObject; E: Exception);
var
  Filename: string;
  LogFile: TextFile;
begin
  // prepares log file
  Filename := ChangeFileExt (Application.Exename, '.log');
  AssignFile (LogFile, Filename);
  if FileExists (FileName) then
    Append (LogFile) // open existing file
  else
    Rewrite (LogFile); // create a new one

  // write to the file and show error
  Writeln (LogFile, DateTimeToStr (Now) + ':' + E.Message);
  if not CheckBoxSilent.Checked then
    Application.ShowException (E);

  // close the file
  CloseFile (LogFile);
end;
```

note The ErrorLog example uses the simple text file support provided by the traditional Turbo Pascal TextFile data type. You can assign a text file variable to an actual file and then simply read or write it. You can find more on TextFile operations in the book *Essential Pascal* (available on `www.marcocantu.com/epascal`).[123]

In the global exceptions handler, you can write to the log, for example, the date and time of the event, and also decide whether to show the exception as Delphi usually does (executing the `ShowException` method of the `TApplication` class). In fact, Delphi by default executes `ShowException` only if there is no `OnException` handler installed.

Finally, remember to close the file, flushing the buffers, every time the exception is handled or when the program terminates. I've chosen the first approach to avoid keeping the log file open for the lifetime of the application, potentially making it difficult to work on it. You can accomplish this in the `OnDestroy` event handler of the form:

```
procedure TFormLog.FormDestroy(Sender: TObject);
begin
  CloseFile (LogFile);
end;
```

The form of the program includes a check box to determine its behavior and two buttons generating simple exceptions. In Figure 3.4, you can see the ErrorLog program running and a sample exceptions log open in Notepad.

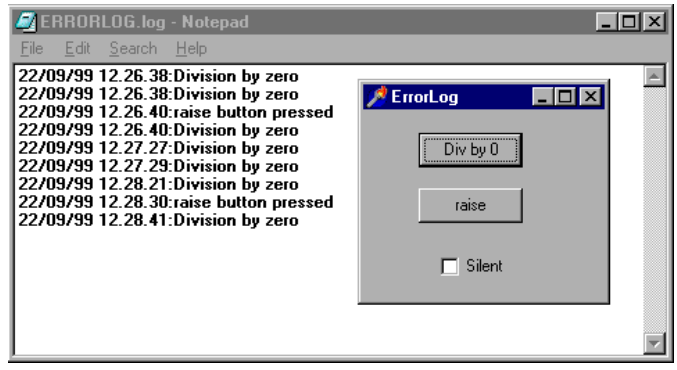


Figure 3.4: The ErrorLog example and the log it produces. Image from the original book.

123 Using the very old TextFile type is not recommended at all, however the code does work today in Delphi 12.

The published Access Specifier

Along with the `public`, `protected`, and `private` access directives, you can use a fourth one, called `published`. A `published` field or method is available not only at run time but also at design time. In fact, every component in the Delphi Components Palette has a published interface that is used by some Delphi tools, in particular the Object Inspector. A regular use of `published` fields is important when you write components. Usually, the `published` part of a component contains no fields or methods but has a new element of the language: properties.

When Delphi generates a form, it places the definitions of its components and methods in the first portion of its definition, before the `public` and `private` keywords. These fields and methods of the initial portion of the class are `published`. The default is `published` when no special keyword is added before an element of a component class.

note To be more precise, `published` is the default keyword only if the class was compiled with the `$M+` compiler directive or is descended from a class compiled with `$M+`. As this directive is used in the `TPersistent` class, most classes of the VCL and all of the component classes default to `published`. However, non-component classes in Delphi (such as `TStream` and `TList`) are compiled with `$M-` and default to public visibility.

The methods assigned to any event should be `published` methods, and the fields corresponding to your components in the form should be `published` to be automatically connected with the objects described in the DFM file and created along with the form. Only the components and methods in the initial `published` part of your form declaration can show up in the Object Inspector (in the list of components of the form or in the list of the available methods displayed when you select the drop-down list for an event).

Defining Properties

Now that we have looked at the `published` keyword, we can start focusing on other extensions of the Object Pascal language specifically tailored for visual, component-based programming. This section covers properties; later on, we'll look at events and build a first simple component.

Properties are attributes that determine the status and behavior of an object. A property is basically a name that is mapped to some `read` and `write` methods or that accesses some data directly. In other words, every time you read the value of a property or change it, you might be accessing a field (even a private one) or calling a method. For example, here is the definition of a property for a date object:

```
property Month: Integer
  read FMonth write SetMonth;
```

To access the value of the `Month` property, this code has to read the value of the private field `FMonth`, while to change the value it calls the method `SetMonth`. Different combinations are possible (for example, we could also use a method to read the value or directly change a field in the `write` directive), but the use of a method to change the value of a property is very common. Here are some alternatives:

```
property Month: Integer read GetMonth write SetMonth;
property Month: Integer read FMonth write Fmonth;
```

note When you write code that accesses a property, it is important to realize that a method might be called. The issue is that some of these methods take some time to execute; they can also produce a number of side effects, often including a (slow) repainting of the component on the screen. Although side effects of properties are seldom documented, you should be aware that they exist, particularly when you are trying to optimize your code.

The `write` directive of a property can also be omitted, making it a *read-only* property. Technically you can also omit the `read` directive and define a *write-only* property, but that doesn't make much sense. Another distinction is between *design-time* properties and *run-time only* properties. Design-time properties are declared in a `published` section of the class declaration. Anything that is declared in the `public` section is not available at design time—it is run-time only. All the read-only properties must be defined in the `public` section (or in the `protected` or `private` sections) because published properties must be *read-write*.

To see the value of a `published` property at design time or to change it, you can use the Object Inspector. This is the tool that Delphi's visual programming environment provides to give access to properties. At run time, you can access any `public` or `published` property by reading or writing its value.

note Remember that the Object Inspector lists only the design-time properties of a component, omitting the run-time only properties. Also, in Delphi 5 some properties can be hidden, if their category has been filtered out. For a complete list of the properties of a component, refer to the Delphi help files, not to the Object Inspector.

To summarize, along with the properties listed in the Object Inspector (design-time), there are other properties (run-time only), some of which can only be read (read-only). Note that you can usually assign a value to a property or read it, and you can even use properties in expressions, but you cannot always pass a property as a parameter to a procedure or method. This is because a property is not a memory location, so it cannot be used as a `var` parameter; it cannot be passed by reference.

Not all of the VCL classes have properties. Properties are present in components and in other subclasses of the `TPersistent` class, because properties usually can be streamed and saved to a file. A DFM file, in fact, is nothing but a collection of published properties of the components on the form. Delphi has extensive support for saving this kind of information, an advanced topic discussed in *Delphi Developers' Handbook* (Sybex, 1998)[124].

Adding Properties to Forms

Properties are a very sound OOP mechanism, a very well thought out application of the idea of encapsulation. Essentially, you have a name that hides the implementation of how to access the information of a class (either accessing the data directly or calling a method). In fact, using properties you end up with an interface that is unlikely to change. At the same time, if you only want to allow users access to some fields of your class, you can easily wrap those fields into properties instead of making them public. You have no further code to write (coding simple `Get` and `Set` methods is terribly boring), and you are still able to change the implementation of your class. Even if you replace the direct data access with method-based access, you won't have to change the source code that uses these properties at all. You'll only need to recompile it. Think of this as the concept of encapsulation raised to the maximum power!

Also keep in mind that using properties is very simple thanks to Code Completion. For this reason, every time you want to make some information of a form available to other forms, you should really use a property. Simply write in the form class declaration the property name and type:

```
property Clicks: Integer;
```

---

124 That book is now hard to find, given it was published over 25 years ago. The content of that book is still applicable today and th book is focused on Delphi RTL internals and writing components.

Then press Ctrl+Shift+C to activate Code Completion. You'll see the following effect:

```
type
  TForm1 = class(TForm)
  private
    FClicks: Integer;
    procedure SetClicks(const Value: Integer);
  public
    property Clicks: Integer
      read FClicks write SetClicks;
  end;

implementation

procedure TForm1.SetClicks(const Value: Integer);
begin
  FClicks := Value;
end;
```

This saves you a lot of typing, compared to the previous versions of Delphi, and it should make the use of properties the standard technique for accessing form data.

In my opinion, properties should also be used in the form classes to encapsulate the access to the components of a form. For example, if you have a main form with a status bar used to display some information (and with the `SimplePanel` property set to `True`), and you want to modify the text from a secondary form, you might be tempted to write:

```
Form1.StatusBar1.SimpleText := 'new text';
```

This is a standard practice in Delphi, but it's not a good one, because it doesn't provide any encapsulation of the form structure or components. If you have similar code in many places throughout an application, and you later decide to modify the user interface of the form (replacing `StatusBar` with another control or activating multiple panels), you'll have to fix the code in many places.

The alternative is to use a method or, even better, a property, to hide the specific control. Simply type

```
property StatusText: string
  read GetText write SetText;
```

and press the Ctrl+Shift+C combination again, to let Delphi add the definition of both methods for reading and writing the property:

```
function TForm1.GetText: string;
begin
  Result := StatusBar1.SimpleText;
```

```
end;

procedure TForm1.SetText(const Value: string);
begin
  StatusBar1.SimpleText := Value;
end;
```

In the other forms of the program, you can simply refer to the `StatusText` property of the form, and if the user interface changes, only the `Set` and `Get` methods of the property are affected.

Adding Properties to the TDate Class

In the previous chapter we developed the `TDate` class. Now we can extend it by using properties. This new example, DateProp, is basically an extension of the ViewD2 example from Chapter 2. Here is the new declaration of the class. It has some new methods (used to set and get the values of the properties) and four properties:

```
type
  TDate = class
  private
    fDate: TDateTime;
    function GetYear: Integer;
    function GetDay: Integer;
    function GetMonth: Integer;
    procedure SetDay (const Value: Integer);
    procedure SetMonth (const Value: Integer);
    procedure SetYear (const Value: Integer);
  public
    constructor Create; overload;
    constructor Create (y, m, d: Integer); overload;
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    procedure Increase (NumberOfDays: Integer = 1);
    procedure Decrease (NumberOfDays: Integer = 1);
    function GetText: string; virtual;
    property Day: Integer read GetDay write SetDay;
    property Month: Integer read GetMonth write SetMonth;
    property Year: Integer read GetYear write SetYear;
    property Text: string read GetText;
  end;
```

The `Year`, `Day`, and `Month` properties read and write their values using corresponding methods. Here are the two related to the `Month` property:

```
function TDate.GetMonth: Integer;
var
```

```
  y, m, d: Word;
begin
  DecodeDate (fDate, y, m, d);
  Result := m;
end;

procedure TDate.SetMonth(const Value: Integer);
begin
  if (Value < 1) or (Value > 12) then
    raise EDateOutOfRange.Create ('Invalid month');
  SetValue (Year, Value, Day);
end;
```

The call to `SetValue` performs the actual encoding of the date, raising an exception in case of an error. I've defined a custom exception class, which is raised every time a value is out of range:

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

The fourth property, `Text`, maps only to a read method. This function is declared as virtual, because it is replaced by the `TNewDate` subclass. There is no reason the `Get` or `Set` method of a property should not use late binding.

note What is important to acknowledge in this example is that the properties do not map directly to data. They are simply computed.

Having updated the class with the new properties, we can now update the example to use properties when appropriate. For example, we can use the `Text` property directly, and we can use some edit boxes to let the user read or write the values of the three main properties (as you can see in Figure 3.5). This happens when the Read button is pressed:

```
procedure TDateForm.BtnReadClick(Sender: TObject);
begin
  EditYear.Text := IntToStr (TheDay.Year);
  EditMonth.Text := IntToStr (TheDay.Month);
  EditDay.Text := IntToStr (TheDay.Day);
end;
```

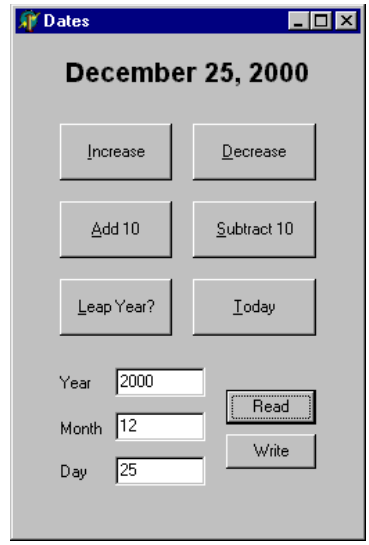


Figure 3.5: The updated form of the DateProp example at run time. Image from the original book.

The Write button does the reverse operation. You can write the code in either of the two following ways:

```
// direct use of properties
TheDay.Year := StrToInt (EditYear.Text);
TheDay.Month := StrToInt (EditMonth.Text);
TheDay.Day := StrToInt (EditDay.Text);

// update all values at once
TheDay.SetValue (StrToInt (EditMonth.Text),
  StrToInt (EditDay.Text),
  StrToInt (EditYear.Text));
```

The difference between the two approaches relates to what happens when the input doesn't correspond to a valid date. When we set each value separately, the program might change the year and then raise an exception and skip executing the rest of the code, so that the date is only partially modified. When we set all the values at once, either they are correct and are all set, or one is invalid and the date object retains the original value.

note The `SetValue` method of this class and the three properties have the same relationship as the `SetBounds` method of the `TControl` classes has with the `Left`, `Top`, `Width`, and `Height` properties. Actually, in some special circumstances the same problem described above arises with these positional properties of controls.

Events in Delphi

When a user does something with a component, such as clicking it, the component generates an event. Other events are generated by the system, in response to a method call or a change to one of that component's properties (or even a different component's). For example, if you set the focus on a component, the component currently having the focus loses it, triggering the corresponding event.

Technically, most Delphi events are triggered when a corresponding Windows message is received, although the events do not match the messages on a one-to-one basis. Delphi events tend to be higher-level than Windows messages, and Delphi provides a number of extra intercomponent messages.

From a theoretical point of view, an event is the result of a message sent to a window, and this window (or the corresponding component) can respond to the message. Following this approach, to handle the click event of a button, we would need to subclass the `TButton` class and add the new event handler.

In practice, creating a new class is too complex to be a reasonable solution. In Delphi, the event handler of a component usually is a method of the form that holds the component, not of the component itself. In other words, the component relies on its owner, the form, to handle its events. This technique is called *delegation,* and it is fundamental to the Delphi component-based model.

Events Are Properties

Another important concept is that events are properties. This means that to handle an event of a component, you assign a method to the corresponding event property, as we did in the CountOb2 example earlier in this chapter. When you double-click an event in the Object Inspector, a new method is added to the owner form and assigned to the proper event property of the component.

This is why it is possible for several events to share the same event handler or change an event handler at run time. To use this feature, you don't need much knowledge of the language. In fact, when you select an event in the Object Inspector, you can press the arrow button on the right of the event name to see a drop-down list of "compatible" methods—a list of methods having the same method pointer type. Using the Object Inspector, it is easy to select the same method for the same event of different components or for different, compatible events of the same component.

Adding an Event to the TDate Class

As we've added some properties to the `TDate` class, we can add one event. The event is going to be very simple. It will be called `OnChange`, and it can be used to warn the user of the component that the value of the date has changed. To define an event, we simply define a property corresponding to it, and we add some data to store the actual method pointer the event refers to. These are the new definitions added to the class:

```
type
  TDate = class
  private
    FOnChange: TNotifyEvent;
    ...
  protected
    procedure DoChange; dynamic;
    ...
  public
    property OnChange: TNotifyEvent
      read FonChange write FOnChange;
    ...
  end;
```

The property definition is actually very simple. A user of this class can assign a new value to the property and, hence, to the `FOnChange` private field. The class doesn't assign a value to this `FOnChange` field. It is the user of the component who does the assignment. The `TDate` class simply calls the method stored in the `FOnChange` field when the value of the date changes. Of course, the call takes place only if the event property has been assigned. The `DoChange` method (declared as a `dynamic` method as it is traditional with event firing methods) makes the test and the method call:

```
procedure TDate.DoChange;
begin
  if Assigned (FOnChange) then
    FOnChange (Self);
end;
```

The `DoChange` method in turn is called every time one of the values changes, as in the following method:

```
procedure TDate.SetValue (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
  // fire the event
  DoChange;
```

Now if we look at the program that uses this class, we can simplify its code considerably. First, we add a new custom method to the form class:

```
type
  TDateForm = class(TForm)
    ...
    procedure DateChange(Sender: TObject);
```

The code of this method simply updates the label with the current value of the `Text` property of the `TDate` object:

```
procedure TDateForm.DateChange;
begin
  LabelDate.Caption := TheDay.Text;
end;
```

This event handler is then installed in the `FormCreate` method:

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TDate.Init (7, 4, 1997);
  LabelDate.Caption := TheDay.Text;
  // assign the event handler for future changes
  TheDay.OnChange := DateChange;
end;
```

Well, this seems like a lot of work. Was I lying when I told you that the event handler would save us some coding? No. Now, after we've added some code, we can completely forget about updating the label when we change some of the data of the object. Here, as an example, is the handler of the `OnClick` event of one of the buttons:

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);
begin
  TheDay.Increase;
end;
```

The same simplified code is present in many other event handlers. Once we have installed the event handler, we don't have to remember to update the label continually. That eliminates a significant potential source of errors in the program. Also note that we had to write some code at the beginning because this is not a component installed in Delphi but simply a class. With a component, you simply select the event handler in the Object Inspector and write a single line of code to update the label. That's all. How difficult is it to write a new component in Delphi? It's actually so simple I'm going to show you how to do it in the next section.

---

note This is meant to be just a short introduction to the role of properties and events and to writing components. A basic understanding of these features is important for every Delphi programmer. If your aim is to write complex new components, you'll find a lot more information on all of these topics in Chapter 13.

Creating a TDate Component

The next step, actually a very simple one, is to turn our `TDate` class into a component. First, we have to inherit our class from the `TComponent` class, instead of the default `TObject` class. Here is the code:

```
type
  TDate = class (TComponent)
    ...
  public
    constructor Create (AOwner: TComponent); overload; override;
    constructor Create (y, m, d: Integer); reintroduce; overload;
```

As you can see, the second step was to add a new constructor to the class, overriding the default constructor for components to provide a suitable initial value. Because there is an overloaded version, we also need to use the `reintroduce` directive for it, to avoid a warning message from the compiler. The code of the new constructor simply sets the date to today's date, after calling the base class constructor:

```
constructor TDate.Create (AOwner: TComponent);
var
  Y, D, M: Word;
begin
  inherited Create (AOwner);
  // today...
  fDate := Date;
```

Having done this, we need to add to the unit that defines our class (the file `DATES.PAS` in the `DATECOMP` directory) a `Register` procedure. (Make sure this identifier start with an uppercase *R*, otherwise it won't be recognized.) This is required in order to add the component to Delphi's Components Palette. Simply declare the procedure, which requires no parameters, in the `interface` portion of the unit, and then write this code in the `implementation` section:

```
procedure Register;
begin
  RegisterComponents ('Md', [TDate]);
end;
```

This code adds the new component to the *Md* page of the Components Palette[125], creating the page if necessary. By the way, this is the same page I'll use for all the components built in the book.

---

125 Nowadays, the Palette pane hosts the components. The behavior described here remains the same.

The last step is to install the component. For this simple example we won't create a new package. Instead, we can install the component in the default Borland User's Components package (a file named `DCLUSR50.DPK` and stored in the `LIB` directory of Delphi). We'll see how to build new packages in Chapter 13.

To make the component available, select the Component ➢ Install Component menu item, choose the *Into existing package* page (this should be the default), select the `DCLUSR50.DPK` package filename[126] (again the default if you've never installed components), and enter the unit filename of the component, `DATES.PAS`. Now simply click OK and Delphi will update the package, compile it, and ask you to install it in Delphi (if you haven't already done so).

If you now move to the Components Palette, it should have a new *Md* page with the new component. This will be shown using the default icon for Delphi components. At this point you can place the component on the form of a new application and start manipulating its properties in the Object Inspector, as you can see in Figure 3.6. You can also handle the `OnChange` event in a much easier way than in the last example.

Besides trying to build your own sample application using this component (something I really suggest you do), you can now open the DateComp example, which is an updated version of the component we've built step-by-step over the last few sections of this chapter. This is basically a simplified version of the DateEvt example, because now the event handler is directly available in the Object Inspector.

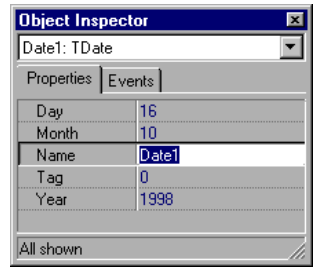


Figure 3.6: The properties of our new TDate component in the Object Inspector. Image from the original book.

126 The package is still called "dclusr.dpk" today. It's description, oddly enough, is "CodeGear User Components". Notice that in the first page of the dialog box you need to select the components source code file, while in the second you can pick the package you want to install it into.

note If you open the DateComp example before installing the new component, Delphi won't recognize the component as it opens the form and will give you an error message. You won't be able to compile the program or make it work until you install the new component.

Using Interfaces

Contrary to what happens in C++, the Delphi inheritance model doesn't support multiple inheritance. This means that each class can have only a single base class. The usefulness of multiple inheritance is a topic of heated debate. The absence of this construct in Delphi can be considered both a disadvantage (because you lose some of the power of C++) and an advantage (because you get a simpler language and fewer problems). My point is that Delphi's interfaces provide the flexibility and power of declaring support for multiple interfaces implemented on a class, while avoiding the problems of inheriting multiple implementations. Rather than get bogged down in this debate, I'll simply assume that it is useful to treat a single object from multiple "perspectives," to consider it a generic object of different base classes. But before I build an example following this principle, we have to introduce the role of interfaces in Object Pascal.

note The techniques covered in this section are used also to implement COM objects, and I'll cover them in more detail in Chapter 15. For the moment, let's consider them simply as language elements.

Declaring an Interface

Besides declaring abstract classes (classes with abstract methods), in Delphi you can also write a *purely abstract class;* that is, a sort of class with only virtual abstract methods. This is accomplished using a specific keyword, `interface`. For this reason we refer to these classes as *interfaces*. Technically, in fact, an interface is not a class, although it may resemble one. Interfaces are not classes, because they are considered a totally separate element, with its own common base interface, `IUnknown`[127], which has the same role as `TObject` for classes.

127 More recently the *IUnknown* interface has been renamed *IInterface*, to underline the fact you can use interface in Delphi even outside of the COM realm. The actual behavior of *IInterface*, though, is still identical to the previous one of *IUnknown*.

Borland introduced interfaces in Delphi 3 along with the support COM programming. If the interface language syntax may have been created to support COM, interfaces do not require COM. You can use interfaces to implement abstraction layers within your applications, without building COM server objects. For example, the Delphi IDE uses interfaces extensively in its internal architecture. In general, interfaces also have some distinctive advantages that can be useful for different types of programming:

- A class can inherit from a single base class, but it can also implement multiple interfaces. The drawback is that when a class implements an interface, it must provide the implementation for each of the methods of the interface.
- Interface type objects are reference-counted and automatically destroyed when there are no more references to the object. This mechanism is similar to how Delphi manages long strings and makes memory management almost automatic.
- The VCL already provides a few base classes to implement the basic behavior required by the `IUnknown` interface. The simplest one is the `TInterfacedObject` class.

note From a more general point of view, interfaces support a slightly different object-oriented programming model than classes. Objects implementing interfaces are subject to polymorphism for each of the interfaces they support. Indeed, the interface-based model is powerful. But having said that, I'm not interested in trying to assess which approach is better in each case. Certainly, interfaces favor encapsulation and provide a more loose connection between classes than inheritance.

Here is the syntax of the declaration of an interface (which, by convention, starts with the letter *I*):

```
type
  ICanFly = interface
    ['{10000000-0000-0000-0000-000000000000}']
    function Fly: string;
  end;
```

note To function properly, each interface requires a numeric ID, like the one above. In theory these should be unique GUIDs, generated in the Delphi editor by pressing Ctrl+Shift+G, but if you don't plan to export these objects, any number will do (more on GUIDs in Chapter 15). These GUIDs are required even if you don't plan exporting these classes, because they are used by the compiler to type-check interface types instead of the plain interface and class names.[128]

128 The GUID in the code snippet above is not a real one. I'd recommend you replace it in the code with an actual GUID, generated by pressing Ctrl+Shift+G, even if the code works anyway.

Once you've declared an interface, you can define a class to implement it, as in:

```
type
  TAirplane = class (TInterfacedObject, ICanFly)
    function Fly: string; virtual;
  end;
```

As mentioned, this class can derive from `TInterfacedObject` to inherit the implementation of the `IUnknown` methods. Although it is not compulsory to implement interface methods with virtual methods, this is a good approach to use if you want to be able to modify these methods in further sub-classes. An alternative technique is to re-declare the interface type in a derived class and rebind the interface methods to static methods declared in that class.

Now that we have defined an implementation of the interface, we can write as usual

```
var
  Airplane1: TAirplane;
begin
  Airplane1 := TAirplane.Create;
  Airplane1.Fly;
  Airplane1.Free;
end;
```

But we can also use an interface-type variable:

```
var
  Flyer1: ICanFly;
begin
  Flyer1 := TAirplane.Create;
  Flyer1.Fly;
end;
```

As soon as you assign an object to an interface variable, Delphi automatically checks to see whether the object implements that interface, using a special version of the `as` operator. You can explicitly express this operation as follows:

```
Flyer1 := TAirplane.Create as ICanFly;
```

Whether we use the direct assignment or the `as` statement, Delphi does one extra thing: it calls the `_AddRef` method of the object, increasing its reference count. At the same time, as soon as the `Flyer1` variable goes out of scope, Delphi calls the `_Release` method, which decreases the reference count, checks whether the reference count is zero, and if necessary, destroys the object. For this reason in the listing above, there is no code to free the object we've created[129].

---

129 There are many other techniques you can use with interfaces these days, including weak interfaces and unsafe ones. This is an advanced concept I cannot really cover in a footnote.

In other words, in Delphi objects referenced by interface variables are reference-counted, and they are automatically de-allocated when no interface variable refers to them any more.

note When using interface-based objects, you should generally access them only with object variables or only with interface variables. Mixing the two approaches breaks the reference counting scheme provided by Delphi and can cause memory errors that are extremely difficult to track. In practice, if you've decided to use interfaces, you should probably use exclusively interface-based variables.

Interface Properties, Delegation, Redefinitions

To demonstrate a few technical elements related to interfaces, I've written the IntfDemo example. This example is based on two different interfaces, `IWalker` and `IJumper`, defined as follows:

```
IWalker = interface
  ['{0876F200-AAD3-11D2-8551-CCA30C584521}']
  function Walk: string;
  function Run: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;

  property Position: Integer
    read GetPos write SetPos;
end;

IJumper = interface
  ['{0876F201-AAD3-11D2-8551-CCA30C584521}']
  function Jump: string;
  function Walk: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;

  property Position: Integer
    read GetPos write SetPos;
end;
```

Notice that the first interface defines also a property. An interface property is just a name mapped to a `read` and a `write` method. You cannot map an interface property to a field, simply because an interface cannot have a field.

Here comes a sample implementation of the `IWalker` interface. Notice that you don't have to define the property, only its access methods:

```
TRunner = class (TInterfacedObject, IWalker)
private
  Pos: Integer;
public
  function Walk: string;
  function Run: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
end;
```

The code is trivial, so I'm going to skip it (you can find it in the IntfDemo example). In a similar way, I've defined a class implementing the `IJumper` interface:

```
TJumperImpl = class (TInterfacedObject, IJumper)
private
  Pos: Integer;
public
  function Jump: string;
  function Walk: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
end;
```

Although this class isn't different from the other one, I'm going to use it in a different way. In the following class, `TMyJumper`, I don't want to repeat the implementation of the `IJumper` interface with similar methods. Instead, I want to delegate the implementation of that interface to a class already implementing it. This cannot be done through inheritance (we cannot have two base classes); instead, you can use specific features of the language interface delegation:

```
TMyJumper = class (TInterfacedObject, IJumper)
private
  fJumpImpl: IJumper;
public
  constructor Create;
  property Jumper: IJumper
    read fJumpImpl implements IJumper;
end;
```

This declaration indicates that the `IJumper` interface is implemented for the `TMyJumper` class by the `fJumpImpl` field. This field, of course, must actually implement all the methods of the interface. To make this work, you need to create a proper object for the field when a `TMyJumper` object is created:

```
constructor TMyJumper.Create;
begin
  fJumpImpl := TJumperImpl.Create;
end;
```

This example is simple, but in general, things get more complex as you start to modify some of the methods or add other methods that still operate on the data of the internal `fJumpImpl` object. This final step is demonstrated, along with other features, by the `TAthlete` class, which implements both the `IWalker` and `IJumper` interfaces:

```
TAthlete = class (TInterfacedObject, IWalker, IJumper)
private
  fJumpImpl: TJumperImpl;
public
  constructor Create;
  function Run: string; virtual;
  function Walk1: string; virtual;
  function IWalker.Walk = Walk1;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;

  property Jumper: TJumperImpl
    read fJumpImpl implements IJumper;
end;
```

One of the interfaces is implemented directly, whereas the other is delegated to the internal `fJumpImpl` object. Notice also that by implementing two interfaces, which have a method in common, we end up with a name clash. The solution is to rename one of the methods, with the statement

```
function IWalker.Walk = Walk1;
```

This declaration indicates that the class implements the `Walk` method of the `IWalker` interface with a method called `Walk1` (instead of with a method having the same name). Finally, in the implementation of all of the methods of this class, we need to refer to the Position property of the `fJumpImpl` internal object. By declaring a new implementation for the `Position` property, we'll end up with two positions for a single athlete, a rather odd situation. Here are a couple of examples:

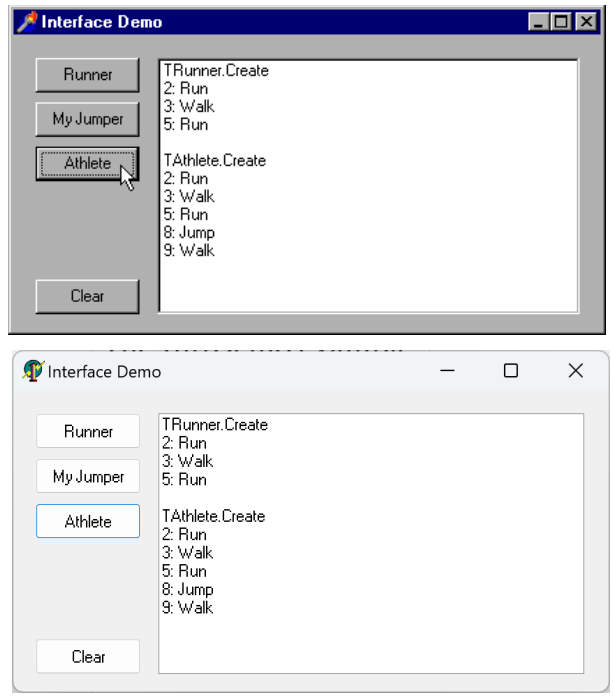
```
function TAthlete.GetPos: Integer;
begin
  Result := fJumpImpl.Position;
end;

function TAthlete.Run:string;
begin
  fJumpImpl.Position := fJumpImpl.Position + 2;
  Result := IntToStr (fJumpImpl.Position) + ': Run';
end;
```

You can further experiment with the IntfDemo example, which has a simple form with buttons to create and call methods of the various objects. Nothing fancy,

though, as you can see in Figure 3.7. Simply keep in mind that each call returns the position after the requested movement and a description of the movement itself.

Figure 3.7:
The IntfDemo example



An Example of Multiple Inheritance

After this example, let me move to a more complex series of interfaces. Suppose you have a hierarchy of classes related to animals. You can base the hierarchy on the standard taxonomic classifications (with categories such as mammals, birds, insects, and so on), or you can categorize them by capability (flying animals, quadrupeds or bipeds, meat eaters, and so on).

There is no easy way to express such a complex structure with single inheritance. You can use multiple inheritance if the language you are using supports this feature, or you can use interfaces. This is what I've done in my example, which represents a rather common study case for multiple inheritance. This program, named MultInh, has both a hierarchy of classes (representing the standard zoological classifications) and a hierarchy of interfaces (expressing the capabilities).

Both the hierarchy of classes and the hierarchy of interfaces actually use single inheritance. It is only when you look at how classes implement the various interfaces that the two hierarchies actually merge, as represented in Figure 3.8.

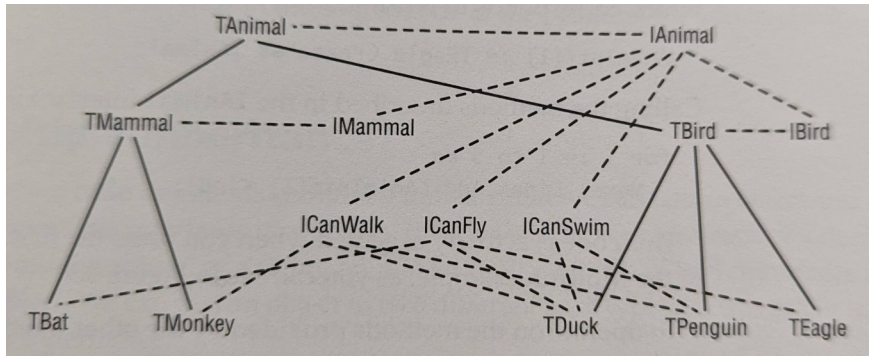


Figure 3.8: The complex relationships among the classes and interfaces of the MultInh example. Image from the original book, captured with a picture, as the original version got lost.

The declarations of these interfaces and their methods are quite long, so I've decided to skip them. Each of them has a specific GUID and defines one or more functions returning strings. The actual classes implement one or more of these interfaces, as depicted in Figure 3.8. Here are a couple of declarations:

```
type
  TBird = class (TAnimal, IBird)
    function LayEggs: string; virtual;
  end;

  TEagle = class (TBird, ICanFly)
    function Kind: string; override;
    function Fly: string; virtual;
  end;

  TPenguin = class (TBird, ICanWalk, ICanSwim)
    function Kind: string; override;
    function Walk: string; virtual;
    function Swim: string; virtual;
```

Now that we have designed this infrastructure, how can we use it? How do we create objects of these classes, and how can we use polymorphism in classes that implement multiple interfaces?

Interface Polymorphism

To use polymorphism with interfaces, I've declared and filled an array inside the form of the program:

```
private
  AnimIntf: array [1..5] of IAnimal;
```

The program extracts the `IAnimal` interface from newly created objects to initialize this array. This is done automatically by Delphi when you write

```
AnimIntf[1] := TEagle.Create;
```

which corresponds to writing

```
AnimIntf[1] := TEagle.Create as IAnimal;
```

Calling the methods described in the `IAnimal` interface is straightforward:

```
for I := 1 to 5 do
  Memo1.Lines.Add (AnimIntf[I].Kind);
```

This code is actually executed when you press the first button of the main form of the MultInh example, as you can see in Figure 3.9.

To operate on the methods provided by the other interface, we must first check to see whether any given object supports it. Because there is no `is` operator for interfaces[130], we can accomplish it by calling the `QueryInterface` method:

```
var
  Fly1: ICanFly;
begin
  AnimIntf[i].QueryInterface (ICanFly, Fly1);
  if Assigned (Fly1) then
    Memo1.Lines.Add (Fly1.Fly);
```

130 The *is* operators for interfaces has later been added to the language.

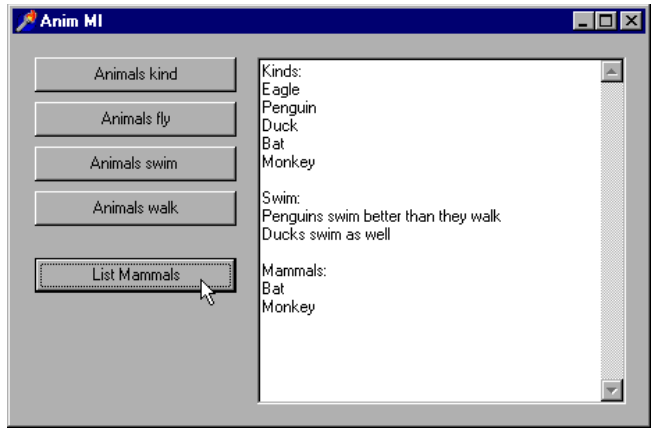


Figure 3.9: The simple user interface of the MultInh example. Image from the original book.

`QueryInterface` requires as parameters a variable for the return value and the type of interface to check for. Because it returns also an error code, we can also check this, as I've done in another case:

```
var
  Swim1: ICanSwim;
begin
  if AnimIntf[i].QueryInterface (
    ICanSwim, Swim1) <> E_NoInterface then
  Memo1.Lines.Add (Swim1.Swim);
```

We can also use the `as` statement using a `try-except` block, but this is not a solution I really like. (It is in the source code of the program for you to check, anyway.)

Is This Multiple Inheritance?

The last two code fragments combined indicate that we can use an object and cast it to the multiple interfaces it supports. In other words, we can consider a duck to be a swimming animal or a flying animal and call methods of both interfaces for a single object. We can cast an object to two different base types, so this really is like multiple inheritance.

What we don't get is the inheritance of the actual implementation of the methods; there is no code in the `ICanFly` interface, and if there were any code shared by all the "flying" objects, it would need to be reimplemented in each class that supports this interface. However, we already know that it is possible to define a single imple-

mentation class and delegate to it the implementation of an interface in many other classes, as I did in the previous example.

As I mentioned earlier, Borland added interfaces to Delphi to support Microsoft's COM, but they can really be used as an extra language feature. The biggest drawback is that interfaces must have an ID even for internal objects, because the type checking of interfaces depends on this number. The other minor problem is that there isn't an `is` operator to check whether an object supports a given interface, but we've seen it is very simple to mimic this behavior by calling `QueryInterface` with a single method call.

Summing up, does it really make sense to use interface types and variables in a program that doesn't need to support COM? If the program is designed around a complex hierarchy that might benefit from multiple inheritance, then the answer is yes. Considering the extra complexity of this design, however, you might disagree.

What's Next?

By reading this chapter, you might have had the impression that I've covered a number of unrelated topics. This was only partially the case. Class references, method pointers, properties, events, the `published` keyword, and exceptions are all language features upon which Delphi's Visual Component Library is built. Other topics, such as as class method or interfaces, are important additions to the language every Delphi developer should at least be familiar with.

Having covered the basics of OOP in the last chapter and all these language extensions in the current one, we can now focus on the structure of the VCL in the next chapter.

There is actually one extra step we've done in this chapter: we've built a first simple component, and we've installed it in the Delphi environment. This already demonstrates the fact that a component is actually an Object Pascal class that inherits from a specific base class, `TComponent`. *Delphi components are classes:* this apparently simple statement describes the nature of the Delphi programming model, underlying its differences with tools as Visual C++ or Visual Basic. The only other language coming close to Object Pascal in terms of components development is Java.

Chapter 4: VCL Programming Techniques

To simplify the work of programming, Delphi provides many powerful, ready-to-use functions and classes. It includes, for example, a number of standard routines. (The Help files no longer have a complete list of these routines, but you can find such a list at my `www.marcocantu.com` Web site[131].) Even larger and more important is Delphi's set of classes. Some of them are component classes, which show up in the Component Palette, while others are more general-purpose. This chapter focuses on the structure of the Delphi class library—known as the *Visual Component Library (VCL)*, although it includes more than components—and gives an overview of some general-purpose classes.

---

131 It's not there any more, and I doubt I'll be able to create an updated version

If you simply want to put the built-in components to work and don't care about the ins and outs of the VCL, you may want to skip this chapter for now and move on to Part II, which focuses on the use of components and other classes related to Windows, or to Part III, which covers database programming (including the standard data-aware components). Remember to come back to this chapter when you are ready to leverage your Delphi programming knowledge.

The TObject Class

At the heart of Delphi is a hierarchy of classes. Every class in the system is a subclass of the `TObject` class, so the whole hierarchy has a single root. This allows you to use the `TObject` data type as a replacement for the data type of any class type in the system.

For example, event handlers usually have a `Sender` parameter of type `TObject`. This simply means that the `Sender` object can be of any class, since every class is ultimately derived from `TObject`. The typical drawback of such an approach is that to work on the object, you need to know its data type. In fact, when you have a variable or a parameter of the `TObject` type you can apply to it only the methods and properties defined by `TObject`. If this variable or parameter happens to refer to an object of the `TButton` type, for example, you cannot directly access its `Caption` property. The solution to this problem lies in the fact that each object "knows" its actual class, and you can access this information using the `ClassType` and `ClassName` methods. For example, `ClassName` returns a string with the name of the class. Because it is a class method, you can apply it both to an object and to a class. Suppose you have defined a `TButton` class and a `Button1` object of that class. Then the following statements have the same effect:

```
Text := Button1.ClassName;
Text := TButton.ClassName;
```

There are occasions when you need to use the name of a class, but it can also be useful to retrieve a class reference to the class itself or to its base class. The class reference, in fact, allows you to operate on the class at run time (as we've seen in the last chapter), while the class name is just a string. We can get these class references with the `ClassType` and `ClassParent` methods. Once you have a class reference, you can use it as if it were an object—for example, to call the `ClassName` method.

Another method that might be useful is `InstanceSize`, which returns the run-time size of an object. (Although you might think that the `SizeOf` global function pro-

vides this information, that function actually returns the size of an object reference —a pointer, which is invariably four bytes—instead of the size of the object itself.)

There are other methods you can apply to any object (and also to any class or class references). Here is a partial list[132]:

`ClassName`	Returns a string with the name of the class.
`ClassNameIs`	Checks the class name.
`ClassParent`	Returns a class reference to the parent class.
`ClassInfo`	Returns a pointer to the internal Run Time Type Information (RTTI) of the class, discussed in *Delphi Developer's Handbook*.
`ClassType`	Returns a reference to the object's class (this cannot be applied directly to a class, only to an object).
`InheritsFrom`	Tests whether the class inherits (directly or indirectly) from a given base class (similar to the `is` operator).
`InstanceSize`	Returns the size of the object's data.

These methods of `TObject` are available for objects of every class, since `TObject` is the common ancestor class of every class. Here is how we can use these methods to access class information:

```
procedure TSenderForm.ShowSender(Sender: TObject);
begin
  Memo1.Lines.Add ('Class Name: ' +
    Sender.ClassName);

  if Sender.ClassParent <> nil then
    Memo1.Lines.Add ('Parent Class: ' +
      Sender.ClassParent.ClassName);

  Memo1.Lines.Add ('Instance Size: ' +
    IntToStr (Sender.InstanceSize));
```

The code checks to see whether the `ClassParent` is nil in case you are actually using an instance of the `TObject` type, which has no base type. You can use other methods to perform tests. For example, you can check whether the `Sender` object is of a specific type with the following code:

---

132 There have been several notable additions to the *TObject* class methods over the years, including *Equals*, *GetHashCode*, *QualifiedClassName*, *ToString*, *UnitName*. Some of these are virtual methods you can override in your derived classes. See docwiki.embarcadero.com/Libraries/en/System.TObject for more details.

```
if Sender.ClassType = TButton then ...
```

You can also check if the `Sender` parameter corresponds to a given object, with this test:

```
if Sender = Button1 then...
```

All these code fragments are part of the IfSender example.

Instead of checking for a particular class or object, you'll generally need to test the type compatibility of an object with a given class; that is, you'll need to check whether the class of the object is a given class or one of its subclasses. This lets you know whether you can operate on the object with the methods defined for the class. This test can be accomplished using the `InheritsFrom` method, which is also called when you use the `is` operator. The following two tests are equivalent:

```
if Sender.InheritsFrom (TButton) then ...
if Sender is TButton then ...
```

All these techniques are demonstrated by the IfSender example, which has a single event handler, called `ShowSender`, connected with the `OnClick` event of several controls: three buttons, a check box, and an edit box. One of the buttons is actually a Bitmap button, an object of a `TButton` subclass. You can see an example of the output of this program at run time in Figure 4.1.

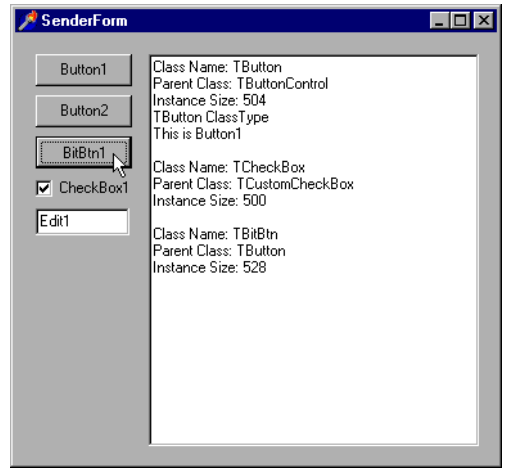


Figure 4.1: The output of the IfSender example. Image from the original book.

Showing Class Information

The IfSender example can be extended to show a complete list of base classes. Once you have a class reference, in fact, you can add all of its base classes to the `ListParent` list box with the following code:

```
with ListParent.Items do
begin
  Clear;
  while MyClass.ClassParent <> nil do
  begin
    MyClass := MyClass.ClassParent;
    Add (MyClass.ClassName);
  end;
end;
```

You'll notice that we use a class reference at the heart of the `while` loop, which tests for the absence of a parent class (so that the current class is `TObject`). Alternatively, we could have written the `while` statement in either of the following ways:

```
while not MyClass.ClassNameIs ('TObject') do...
while MyClass <> TObject do...
```

The code in the `with` statement referring to the `ListParent` list box is part of the ClassInfo example, which displays the list of parent classes and some other information about a few components of the VCL, basically those on the Standard page of the Component Palette. These components are manually added to a dynamic array holding classes and declared as:

```
private
  ClassArray: array of TClass;
```

When the program starts the array is used to show all the class names in a list box. Selecting an item of the list box triggers the visualization of its base classes, as you can see in the output of the program, in Figure 4.2.

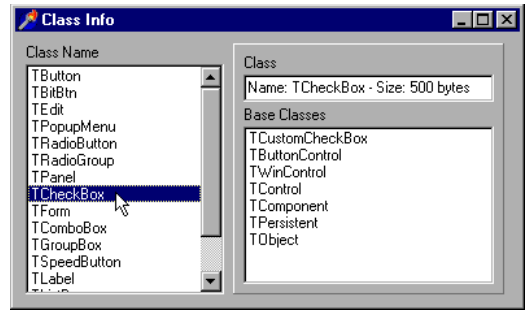


Figure 4.2: The output of the ClassInfo example. Image from the original book.

note As a further extension to this example, we might show all the base classes of the various components in a hierarchy. To do that, I've created the VclHierarchy Wizard, which you can find on my Web site.[133]

The VCL Hierarchy

The VCL defines a number of subclasses of `TObject`. Many of these classes are actually subclasses of other subclasses, forming a very complex hierarchy. Unless you are interested in developing new components, you'll usually use only the *terminal* classes of this hierarchy—the leaf nodes of the hierarchy tree. This is not really a precise description, as some of the leaf nodes can be further extended by deriving new components, and some of the classes in higher-level nodes can be instantiated directly.

note Delphi's documentation includes a large poster of the VCL class hierarchy. Although its size makes it a little cumbersome, this can be a precious reference for understanding the VCL class hierarchy. Again, you can also find a VCL class hierarchy on my Web site.[134]

We can divide the VCL hierarchy into three main areas: components, generic objects, and exceptions. Components can be modified visually in the Delphi IDE, typically using the Form Designer, while the other types of classes are referenced

133 While I have similar code available and in active use, this specific code is no longer on my site.

134 The poster was a great tool in the early days of the product. As the library kept growing and the printed documentation started reducing, it was removed. With the switch to digital distribution, there was later no point in considering it.

only in the source code. As a detailed description would take too much space, this chapter includes some general notes, mainly on components but also on some other important classes of the VCL.

Components

Components are the central elements of Delphi applications. When you write a program, you basically choose a number of components and define their interactions. That's all there is to Delphi visual programming.

There are different kinds of components in Delphi. Most components are included in the Component Palette, but some of them (including `TForm` and `TApplication`) are not. Technically, components are sub-classes of the `TComponent` class. As such, they can be streamed in a `DFM` file (since they inherit from the `TPersistent` class, which provides the information needed for streaming) and they may have published properties and events you can manipulate visually. We saw a simple example (DateComp) of building a component in the last chapter.

The part of the VCL hierarchy related to components is generally divided into three areas, as you can see in Figure 4.3[135]. These groups indicate components with a similar internal structure:

- *Controls* or *visual components* are all the classes that descend from `TControl`. Controls have a position and a size on the screen and show up in the form at design time in the same position they'll have at run time. Controls have two different specifications, window-based or graphical:
    - *Window-based controls* (also called *windowed controls*) are visual components based on an operating system window. From a technical point of view, this means that these controls have a window handle and descend from `TWinControl`. From a user perspective, windowed controls can receive the input focus and some of them can contain other controls. This is the biggest group of components in the Delphi VCL. We can further divide windowed controls in two groups: wrappers of Windows controls and custom controls.
    - *Graphical controls* (also called *nonwindowed controls*) are visual components that are not based on a window. Therefore, they have no handle, cannot receive the focus, and cannot contain

135 This division in terms of component groups and their role is still 100% applicable today.

other controls. These controls inherit from `TGraphicControl` and are painted by their parent form, which sends them mouse-related and other events. Examples of non-windowed controls are the Label and the SpeedButton components. There are just a few controls in this group, but they are critical to minimizing the use of system resources, particularly for components used often and in number, such as labels or toolbar buttons.

- *Non-visual components* are all the components that are not controls—all the classes that descend from `TComponent` but not from `TControl`. At design time, a non-visual component appears on the form as an icon (optionally with a caption below it). At run time, some of these components may be visible at times (for example, the standard dialog boxes), and others are always invisible (for example, the database table component). In other words, non-visual components are not visible themselves at run time, although they may manage something that is visual, such as a dialog box.

note You can simply move the mouse cursor over a control or component in the Form Designer to see a hint with its name and its class type. You can also use an environment option, Show Component Captions, to see the name of a non-visual component right under its icon.

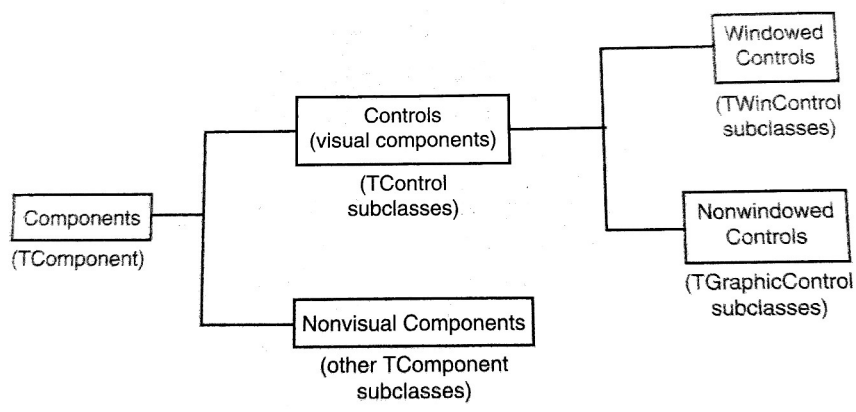


Figure 4.3: A graphical representation of the groups of components. Image based on a picture of the original printed book.

Windows Components

You might have asked yourself where the idea of using components for Windows programming came from. The answer is simple: Windows itself has some components, usually called controls. A *control* is technically a predefined window that has a specific behavior and some styles and is capable of responding to specific messages. These controls were the first step in the direction of component development. The second step was probably Visual Basic controls, and the third step is Delphi components.

note Actually, Microsoft's third step is its ActiveX, the designated successor of VBX controls. In Delphi you can use both ActiveX and native components, but if you look at the technology, Delphi components are really ahead of the ActiveX controls. Delphi components use OOP to its full extent, while ActiveX controls do not fully implement the concept of inheritance. I'll focus on the details of using and writing ActiveX controls in Chapter 16.

Windows 3.1 had six kinds of predefined controls, which were generally used in dialog boxes. Still used in Win32, they are buttons (push buttons, check boxes, and radio buttons), static labels, edit fields, list boxes, combo boxes, and scroll bars. Win32 adds a number of new predefined components, such as the list view, the status bar, the spin button, the progress bar, the tab control, and many others. Win32 developers can use the standard common controls provided by the system, and Delphi developers have the further advantage of having corresponding easy-to-use components.

The standard system controls are the basic components of each Windows application, regardless of the programming language used to write it, and are very well known by every Windows user. Delphi literally wraps these Windows predefined controls in some of its basic components. A Delphi wrapper class, for example `TEdit`, simply surfaces the capabilities of the underlying Windows control, making it easier to use. However, Delphi adds nothing to the capabilities of this control. In Windows 95/98 an edit or a memo control has a physical limit of 32KB of text, and this limit is retained by the Delphi component.

Why hasn't Borland overcome this limit? Why can't we change the color of a button[136]? Simply because by replacing a Windows control with a custom version, we would lose the close connection with the operating system. Suppose Microsoft

136 This isn't true any more, as now the VCL library now offers support for styling. The concept of relying on platform controls still applies, but styles offer a higher degree of visual customization for any platform control.

improves some of the controls in the next version of Windows. If we use our own version of the component, the application we build won't have the new features.

By using controls that are based on the operating system capabilities, instead, our programs can easily migrate through different versions of the OS and retain all the features provided by the specific version.

Of course, if you need a control that does something really different from the existing ones, you'll need to write your own custom controls, something the VCL itself does in the classes inheriting from `TCustomControl`. For example, a Delphi grid isn't related to any Windows control. All the classes in that portion of the VCL tree aren't directly related to Windows standard controls or Win32 common controls.

Note that wrapping an existing Windows is an effective way of reusing code and also helps reduce the size of your compiled code. Implementing yet another button control from scratch requires custom code in your application, while a wrapper around the OS-supplied button control requires less code and makes use of system code shared by all Windows applications.

Objects

Although the VCL is basically a collection of components, there are other classes that do not fit in this category, because they do not descend from `TComponent`. All the noncomponent classes are often identified (by the Delphi Help files and documentation, among others sources) as *objects*, although this is not a precise definition. There are two main uses for these classes. Generally, noncomponent classes define the data type of component properties, such as the `Picture` property of an image component (which is a `TGraphic` object) or the `Items` property of a list box (which is a `TStrings` object). These classes generally inherit from `TPersistent`, so they are *streamable*, and they can have sub-properties and even events.

The second use of noncomponent classes is a direct use. In the Delphi code you write, you can allocate and manipulate objects of these classes. You might do this for a number of purposes, including to store a copy of the value of a property in memory and modify it without changing the original component, to store a list of values, to write complex algorithms, and so on. You'll see several examples in this book that show how to use non-component classes directly.

There are several groups of non-component classes in the VCL[137]:

137 The list has been significantly extended over the years, but the core classes listed here are still available and relevant today.

- *Graphic-related objects* include `TBitmap`, `TBrush`, `TCanvas`, `TFont`, `TGraphic`, `TGraphicsObject`, `TIcon`, `TMetafile`, `TPen`, and `TPicture`.
- *Stream/file-related objects* include `TBlobStream`, `TFileStream`, `THandleStream`, `TIniFile`, `TMemoryStream`, `TFiler`, `TReader`, and `TWriter`.
- *Lists and collections* include `TList`, `TStrings`, `TStringList`, `TCollection`, `TCollectionItem` and the new container classes introduced by Delphi 5. We will focus on these classes in a later section of this chapter.
- *COM-related classes:* This is an important area of Delphi programming. COM-related classes are covered in Chapter 15.
- *Exception classes:* These are inherited from the `Exception` class. We discussed exception handling in Chapter 3, so I won't repeat the details here.

Common VCL Properties

Although each component has its own set of properties, you may have already noticed that some properties are common to all of them. Table 4.1 lists some of the common properties along with very short descriptions[138].

Table 4.1: Some Properties Available in Most Components

Property	Available For	Description
`Action`	Some controls	Identifies the Action object connected to the control (see Chapter 5 for details).
`Align`	Some controls	Determines how the control is aligned in its parent control area.
`Anchors`	Most controls	Indicates the side of the form the component is connected with (see Chapter 7 for an example).
`AutoSize`	Some controls	Indicates whether the control can determine its own size depending on its content.
`BiDiMode`	All controls	Provides support for languages written right to left (stands

138 This is still a very good list of the most relevant common properties.

		for BiDirectional Mode).
`BorderWidth`	Windowed controls	The width of the border.
`BoundsRect`	All controls	Defines the bounding rectangle of the control (run-time only).
`Caption`	Most controls	The caption of the control.
`ComponentCount`	All components	The number of components owned by the current one (run-time only and read-only).
`ComponentIndex`	All components	The position of the component in its owner's list of components (run-time only).
`Components`	All components	An array of the components owned by the current one (run-time only and read-only).
`Constraints`	All controls	Determines the maximum and minimum size of a control (or a form) during resizing operations.
`ControlCount`	All controls	The number of child controls of the current one (run-time only and read-only).
`Controls`	All controls	An array of the child controls of the current one (run-time only and read-only).
`Color`	Most controls	Indicates the color of the surface or the background.
`Ctrl3D`[139]	Most components	Determines whether the control has a three-dimensional look.
`Cursor`	All controls	The cursor used when the mouse pointer is over the control.
`DockSite`	Most windowed controls	Indicates whether the windowed control is a docking site. There are other properties related to this, including `DockClientCount`, `DockClients`, `UseDockManager`, and `DockManager`. Docking is discussed in Chapters 7 and 8.

139 This property is now obsolete.

`DragCursor`	Most controls	The cursor used to indicate that the control accepts dragging.
`DragKind`	Most controls	Lets you choose between dragging and docking, if the drag mode is automatic.
`DragMode`	Most controls	Determines whether the drag-and-drop behavior (allowing either dragging or docking, as specified in the `DragKind` property) will be activated automatically.
`Enabled`	All controls and some non-visual components	Determines whether the control is active or inactive (grayed).
`Font`	All controls	Determines the font of the text displayed inside the component.
`Handle`	All windowed controls	The handle of the system window used by the control (run-time only and read-only).
`Height`	All controls	The vertical size of the control.
`HelpContext`	All controls and the dialog components	A context number used to invoke the context-sensitive Help automatically.
`Hint`	All controls	The string used to display fly-by hints for the control.
`Left`	All controls	The horizontal coordinate of the upper-left corner of the component.
`Name`	All components	The unique name of an instance of the component, which can generally be used in the source code.
`Owner`	All components	Indicates the owner component (run-time only and read-only).
`Parent`	All controls	Indicates the parent control (run-time only).
`ParentColor`	Most controls	Determines if the component uses the same `Color` as the parent.
`ParentCtl3D`[140]	Most components	Determines whether the component uses the same `Ctrl3D` as the parent.

`ParentFont`	All controls	Determines whether the component uses the same `Font` as the parent.
`ParentShowHint`	All controls	Determines whether the component uses the same `ShowHint` as the parent.
`PopupMenu`	All controls	The pop-up menu used when the user right-clicks on the control.
`ShowHint`	All controls	Determines whether hints are enabled.
`Showing`	All controls	Determines whether the control is currently *showing* on the screen; that is, if all the controls in the parent chain have the `Visible` property set. In other words a control is `Showing` if it is `Visible`, its parent control is `Visible`, any parent control of the parent control is `Visible`, and so forth. (Run-time only and read-only.)
`TabOrder`	All windowed controls	Determines the control's tab order in its parent control.
`TabStop`	All windowed controls	Determines whether the user can move the control with the Tab key.
`Tag`	All components	A long integer available to store custom undefined data.
`Top`	All controls	The vertical coordinate of the upper-left corner of the component.
`UndockHeight`	Most controls	The height of the control when it is undocked.
`UndockWidth`	Most controls	The width of the control when it is undocked.
`Visible`	All controls	Determines whether the control is visible (provided its parent is also visible, as described in the `Showing` property).
`Width`	All controls	The horizontal size of the control.

Since there is inheritance among components, it is interesting to see in which ancestor classes the most common properties are introduced. You can look at Figure 4.4 for an overview of the properties introduced by the topmost classes of the VCL hier-

140 This is also irrelevant today

archy. The following sections provide basic descriptions of some of these common properties.

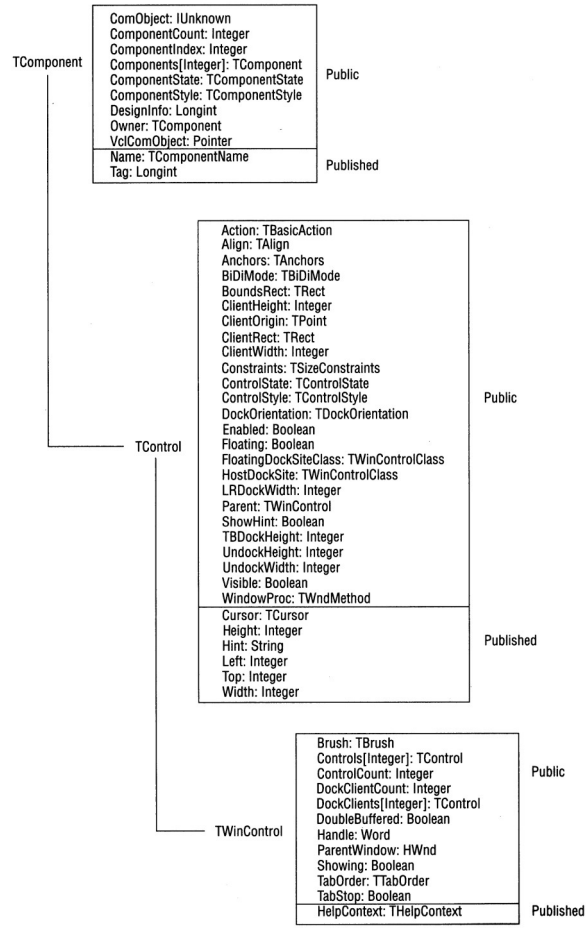


Figure 4.4: The properties introduced by the topmost classes of the VCL hierarchy and available in all of the subclasses. Image based on a picture of the original printed book.

The Name Property

Every component in Delphi should have a name. The name must be unique within the owner component, which is generally the form into which you place the component. This means that an application can have two different forms, each with a component with the same name, although you might want to avoid this practice to

prevent confusion. It is generally better to keep component names unique throughout an application.

Setting a proper value for the `Name` property is very important: If it's too long, you'll need to type a lot of code to use the object; if it's too short, you may confuse different objects. Usually the name of a component has a prefix with the component type; this makes the code more readable and allows Delphi to group components in the combo box of the Object Inspector, where they are sorted by name. There are three important elements related to the `Name` property of the components:

- First, the value of the `Name` property is used to define the name of the object in the declaration of the form class. This is the name you're generally going to use in the code to refer to the object. For this reason, the value of the name property must be a legal Pascal identifier.
- Second, if you set the `Name` property of a control before changing its `Caption` property, the new name is copied to the caption. That is, if the name and the caption are identical, then changing the name will also change the caption.
- Third, Delphi uses the name of the component to create the default name of the methods related to its events. If you have a `Button1` component, its default `OnClick` event handler will be called `Button1Click`, unless you specify a different name. If you later change the name of the component, Delphi will modify the names of the related methods accordingly. For example, if you change the name of the button to `MyButton`, the `Button1Click` method automatically becomes `MyButtonClick`.

The Components Array

Besides accessing a component by name, you can use the `Components` property of its owner, usually a form. Here is an example of the code you can use to add to a list box the names of all the components of a form (this code is actually part of the ChangeOwner example, presented in the next section):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  for I := 0 to ComponentCount - 1 do[141]
    ListBox1.Items.Add (Components [I].Name);
```

141 These days, you can also navigate the components owned by a component using a *for..in* loop.

```
end;
```

This code uses the `ComponentCount` property, which holds the total number of components owned by the current form, and the `Components` property, which is actually the list of the owned components. When you access a value from this list you get a value of the `TComponent` type. For this reason you can directly use only the properties common to all components, such as the `Name` property. To use properties specific to particular components, you have to use the proper type-downcast (`as`).

In Delphi, there are some components that are also component containers: the GroupBox, the Panel, the PageControl, and, of course, the Form component. When you use these controls, you can add other components inside them. In this case, the container is the parent of the components (as indicated by the `Parent` property), while the form is their owner (as indicated by the `Owner` property). You can use the `Controls` property of a form or group box to navigate the child controls, and you can use the `Components` property of the form to navigate all the owned components, regardless of their parent.

Using the `Components` property, we can always access each component of a form. If you need access to a specific component, however, instead of comparing each name with the name of the component you are looking for, you can let Delphi do this work, by using the `FindComponent` method of the form. This method simply scans the `Components` array looking for a name match.

The Owner Property

Every component usually has an owner. When a component is created at design time (or from the resulting DFM file) its owner will invariably be its form. When you create a component at run time, the owner is passed as a parameter to the `Create` constructor.

The owner is a read-only property, so you cannot change it. However, you can affect its value by calling the `InsertComponent` and `RemoveComponent` methods of the owner itself, passing the current component as parameter. Using these methods you can change a component's owner. However, you cannot apply them directly in an event handler of a form, as we attempt to do here:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RemoveComponent (Button1);
  Form2.InsertComponent (Button1);
end;
```

This code produces a memory access violation, because when you call `RemoveComponent`, Delphi disconnects the component from the form field (`Button1`), setting it to `nil`. The solution is to write a procedure like this:

```
procedure ChangeOwner (Component, NewOwner: TComponent);
begin
  Component.Owner.RemoveComponent (Component);
  NewOwner.InsertComponent (Component);
end;
```

This method (extracted from the ChangeOwner example) changes the owner of the component. It is called along with the simpler code used to change the parent component; the two commands combined move the button *completely* to another form, changing its owner:

```
procedure TForm1.ButtonChangeClick(Sender: TObject);
begin
  if Assigned (Button1) then
  begin
    // change parent
    Button1.Parent := Form2;
    // change owner
    ChangeOwner (Button1, Form2);
  end;
end;
```

The method checks whether the `Button1` field still refers to the control, because while moving the component, Delphi will set `Button1` to `nil`. You can see the effect of this code in Figure 4.5.

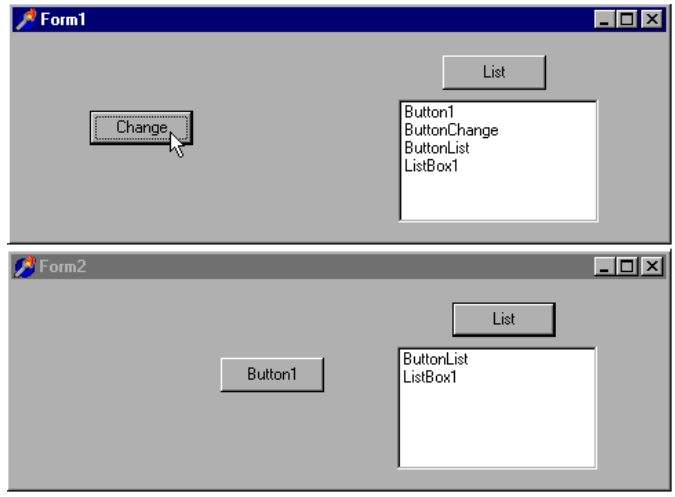


Figure 4.5: In the ChangeOwner example, pressing the Change button moves the Button1 component to the second form. Image from the original book.

To demonstrate that the `Owner` of the `Button1` component actually changes, I've added another feature to both forms. The List button fills the list box with the names of the components each form owns, using the procedure shown in the previous section. Press the two List buttons before and after moving the component, and you'll see what happens behind the scenes. As a final feature, the `Button1` component has a simple handler for its `OnClick` event, to display the caption of the owner form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage ('My owner is ' +
    ((Sender as TButton).Owner as TForm).Caption);
end;
```

Removing Form Fields

Every time you add a component to a form, Delphi adds its complete description, including all of its properties, to the DFM file. To the Pascal file, Delphi adds the corresponding field in the form class declaration. When the form is created, Delphi loads the DFM file and uses it to re-create all the components and set their properties back. Then it hooks the new object with the form field corresponding to its `Name` property.

For this reason, it is certainly possible to have a component without a name. If your application will not manipulate the component or modify it at run time, you can

remove the component name from the Object Inspector. Examples are a static label with fixed text, or a menu item, or even more obviously menu item separators. By blanking out the name, you'll remove the corresponding element from the form class declaration. This reduces the size of the form object (by only four bytes, the size of the object reference) and it reduces the DFM file by not including a useless string (the component name). Reducing the DFM also implies reducing the final EXE file size, even if only slightly.

note If you blank out component names, just make sure to leave at least one named component of each class used on the form so that the smart linker will link in the required code. If, as an example, you remove from a form all the fields referring to labels, the Delphi linker will remove the implementation of the `TLabel` class from the executable file. The effect is that when the system loads the form at run time, it is unable to create an object of an unknown class and issues an error indicating that the class is not available.

You can also keep the component name and manually remove the corresponding field of the form class. Even if the component has no corresponding form field, it is created anyway, although using it (through the `FindComponent` method of the form, for example) will be a little more difficult.

Hiding Form Fields[142]

Many OOP purists complain that Delphi doesn't really follow the encapsulation rules, because all of the components of a form are mapped to public fields and can be accessed from other forms and units. However, Delphi does that only as a default to help beginners learn to use the Delphi visual development environment quickly. A programmer can follow a different approach and use properties and methods to operate on forms. The risk, however, is that another programmer of the same team might inadvertently bypass this approach, directly accessing the components if they are left in the published section. The solution, which many programmers don't know about, is to move the components to the private portion of the class declaration.

As an example, I've taken a very simple form with an edit box, a button, and a list box. When the edit box contains text and the user presses the button, the text is added to the list box. When the edit box is empty, the button is disabled. This is the simple code of the HideComp example:

142 This section is still very relevant today, given Delphi's architecture in terms of the form class structure hasn't changed.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add (Edit1.Text);
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
  Button1.Enabled := Length (Edit1.Text) <> 0;
end;
```

I've listed these methods only to show you that in the code of a form we usually refer to the available components, defining their interactions. For this reason it seems impossible to get rid of the fields corresponding to the component. However, what we can do is hide them, moving them from the default published section to the private section of the form class declaration:

```
TForm1 = class(TForm)
  procedure Button1Click(Sender: TObject);
  procedure Edit1Change(Sender: TObject);
  procedure FormCreate(Sender: TObject);
private
  Button1: TButton;
  Edit1: TEdit;
  ListBox1: TListBox;
end;
```

Now if you run the program you'll get in trouble: The form will load fine, but because the private fields are not initialized, the events above will use `nil` object references. Delphi usually initializes the published fields of the form using the components created from the DFM file. What if we do it ourselves, with the following code?

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1 := FindComponent ('Button1') as TButton;
  Edit1 := FindComponent ('Edit1') as TEdit;
  ListBox1 := FindComponent ('ListBox1') as TListBox;
end;
```

It will *almost* work, but it generates a system error, similar to the one we discussed in the previous section. This time, the private declarations will cause the linker to link in the implementations of those classes, but the problem is that the streaming system needs to know the names of the classes in order to locate the class reference needed to construct the components while loading the DFM file.

The final touch we need is some registration code to tell Delphi at run time about the existence of the component classes we want to use. We should do this before the form is created, so I generally place this code in the initialization section of the unit:

```
initialization
  RegisterClasses ([TButton, TEdit, TListBox]);
```

Now the question is, is this really worth the effort? What we obtain is a higher degree of encapsulation, protecting the components of a form from other forms (and other programmers writing them). I have to say that replicating these steps for each and every form can be tedious, and I'd really like to have a wizard generating this code for me on the fly while I do the standard operations in Delphi. However, for a large project built according to the principles of object-oriented programming, I recommend you consider this or a similar technique[143].

Properties Related to Control Size and Position

Other important properties, common to all controls, are those related to size and position. The position of a control is determined by its `Left` and `Top` properties; its size by the `Height` and `Width` properties. Technically, all components have a position, because when you reopen an existing form at design time, you want to be able to see the icons for the non-visual components in exactly the position where you've placed them. This position is visible in the DFM file.

An important feature of the position of a component is that, like any other coordinate in Windows, it always relates to the client area of its parent component (which is the component indicated by its `Parent` property). For a form, the client area is the surface included within its borders (excluding the borders themselves). It would have been messy to work in screen coordinates, although there are some ready-to-use methods that convert the coordinates between the form and the screen and vice versa.

Note, however, that the coordinates of a control are always relative to the parent control, which is usually a form but can also be a panel or another *container* component. If you place a panel in a form, and a button in a panel, the coordinates of the button relate to the panel and not to the form containing the panel. In fact, in this case, the parent component of the button is the panel.

143 I've later build a Delphi Form Wizard, which can automate these steps. It is still available as part of my Cantools, see github.com/marcocantu/cantools.

Activation and Visibility Properties

There are two basic properties you can use to let the user activate or hide a component. The simplest is the `Enabled` property. When a component is disabled (when `Enabled` is set to `False`), there is usually some visual hint to specify this state to the user. At design time, the "disabled" property does not always have an effect, but at run time, disabled components are generally grayed.

For a more radical approach, you can completely hide a component, either by using the corresponding `Hide` method or by setting its `Visible` property to `False`. Be aware, however, that reading the status of the `Visible` property does not tell you if the control is actually visible. In fact, if the container of a control is hidden, even if the control is set to `Visible`, you cannot see it. For this reason, there is another property, `Showing`, which is a run-time and read-only property. You can read the value of `Showing` to know if the control is really visible to the user; that is, if it is visible, its parent control is visible, the parent control of the parent control is visible, and so on.

The Customizable Tag Property

The `Tag` property is a strange one, because it has no effect at all. It is merely an extra memory location, present in each component class, where you can store custom values. The kind of information stored and the way it is used are completely up to you.

It is often useful to have an extra memory location to attach information to a component without needing to define your component class. Technically, the `Tag` property stores a long integer[144] so that, for example, you can store the entry number of an array or list that corresponds to an object. Using typecasting, you can store in the `Tag` property a pointer, an object, or anything else that is four bytes wide. This allows a programmer to associate virtually anything with a component using its tag. We'll see how to use this property in several examples in future chapters, including the ODMenu examples in Chapter 5.

144 The property is now defined as *NativeInt*, so that its size with be different in a 32-bit or 64-bit application, matching the pointer size on each platform.

The User Interface: Color and Font

Two properties often used to customize the user interface of a component are `Color` and `Font`. There are several properties related to the color. The `Color` property itself usually refers to the background color of the component. Also, there is a `Color` property for fonts and many other graphic elements. Many components also have a `ParentColor` and a `ParentFont` property, indicating whether the control should use the same font and color as its parent component, which is usually the form. You can use these properties to change the font of each control on a form by setting only the `Font` property of the form itself.

When you set a font, either by entering values for the attributes of the property in the Object Inspector or by using the standard font selection dialog box, you can choose one of the fonts installed in the system. The fact that Delphi allows you to use all the fonts installed on your system has both advantages and drawbacks. The main advantage is that if you have a number of nice fonts installed, your program can use any of them. The drawback is that if you distribute your application, these fonts might not be available on your users' computers.

If your program uses a font that your user doesn't have, Windows will select some other font to use in its place. A program's carefully formatted output can be ruined by the font substitution. For this reason, you should probably rely only on standard Windows fonts (such as MS Sans Serif, System, Arial, Times New Roman, and so on). The alternative is to ship some fonts with your application, if the font's user license allows it.

There are a number of ways to set the value of a color. The type of this property is `TColor`. For properties of this type, you can choose a value from a series of predefined name constants or enter a value directly. The constants for colors include `clBlue`, `clSilver`, `clWhite`, `clGreen`, `clRed`, and many others. As a better alternative, you can use one of the colors used by Windows for system elements, such as the background of a window (`clWindow`), the color of the text of a highlighted menu (`clHightlightText`), the active caption (`clActiveCaption`), and the ubiquitous button face color (`clBtnFace`). All the color constants mentioned here are listed in Delphi's Help under the *TColor type* topic.

Another option is to specify a `TColor` as a number (a four-byte hexadecimal value) instead of using a predefined value. If you use this approach, you should know that the low three bytes of this number represent RGB color intensities for blue, green, and red, respectively. For example, the value `$00FF0000` corresponds to a pure blue color, the value `$0000FF00` to green, the value `$000000FF` to red, the value `$00000000`

to black, and the value `$00FFFFFF` to white. By specifying intermediate values, you can obtain any of the 16 million possible colors.

Instead of specifying these hexadecimal values directly, you should use the `RGB` function, which has three parameters, all ranging from 0 to 255. The first indicates the amount of red, the second the amount of green, and the last the amount of blue. Using the `RGB` function makes programs generally more readable than using a single hexadecimal constant.

note RGB is *almost* a Windows API function. It is defined by the Windows-related units and not by Delphi units, but a similar function does not exist in the Windows API. In C, there is a macro that has the same name and effect, so this is a welcome addition to the Pascal interface to Windows.

The highest-order byte of the `TColor` type is used to indicate which palette should be searched for the closest matching color, but palettes are too advanced a topic to discuss here. (Sophisticated imaging programs also use this byte to carry transparency information for each display element on the screen.) Regarding palettes and color matching, note that Windows sometimes replaces an arbitrary color with the closest available solid color, at least in video modes that use a palette. This is always the case with fonts, lines, and so on. At other times, Windows uses a dithering technique to mimic the requested color by drawing a tight pattern of pixels with the available colors. In 16-color (VGA) adapters[145] and at higher resolutions, you often end up seeing strange patterns of pixels of different colors and not the color you had in mind.

Common VCL Methods

Component methods are just like any other methods. There are procedures and functions you can call to perform the corresponding action. As mentioned earlier, you can often use methods to accomplish the same effect as reading or writing a property. Usually, the code is easier to read and understand when you use properties. However, not all methods have corresponding properties. Most of them are procedures, which execute an action instead of reading or writing a value. Again, some methods are available in all of the components; other methods are shared only by controls (visual components), and so on. Table 4.2 lists some common compo-

145 This tells you how old this book is, as this was current hardware back than!

nent methods. We'll see examples of using most of these methods throughout the book[146].

Table 4.2: Some Methods Available for Most VCL Components

Method	Available For	Description
`BeginDrag`	All controls	Starts manual dragging.
`BringToFront`	All controls	Puts the control in front of all others.
`CanFocus`	All controls	Determines whether the control will accept the keyboard input focus.
`ClientToScreen`	All controls	Translates client coordinates into screen coordinates.
`ContainsControl`	All controls	Determines whether a certain control is contained by the current one.
`Create`	All components	Creates a new instance (constructor).
`Destroy`	All components	Destroys the instance (destructor). You should actually call `Free`.
`Dragging`	All controls	Indicates whether the controls are being dragged.
`EndDrag`	All controls	Manually terminates dragging.
`ExecuteAction`	All components	Activates the action connected with the component.
`FindComponent`	All components	Returns the component in the `Components` array property having a given name (we've just used it in the HideComp example).
`FlipChildren`	All windowed controls	Moves child controls from the left side to the right side and vice versa. Used for supporting right-to-left languages (such as Arabic or Hebrew), along with the `IsRightToLeft` property.
`Focused`	All windowed controls	Determines whether the control has the focus.

146 These remains a fairly good list today, as well. Same for the list of events below.

`Free`	All components	Deletes the object from memory (forms should use the Release method).
`GetTextBuf`	All controls	Retrieves the text (or caption) of the control.
`GetTextLen`	All controls	Returns the length of the text (or caption) of the control.
`HandleAllocated`	All controls	Returns `True` if a system handle has been allocated for the control.
`HandleNeeded`	All controls	Allocates a corresponding system handle if one doesn't already exist.
`Hide`	All controls	Makes the control invisible (the same as setting the `Visible` property to `False`).
`InsertComponent`	All components	Adds a new element to the list of owned components.
`InsertControl`	All controls	Adds a new element to the list of controls that are the children of the current one.
`Invalidate`	All controls	Forces a repaint of the control.
`ManualDock`	All controls	Manually activates docking.
`ManualFloat`	All controls	Sets the docking control as a floating one.
`RemoveComponent`	All components	Removes a component from the `Components` list.
`ScaleBy`	All controls	Scales the control by a given percentage.
`ScreenToClient`	All controls	Translates screen coordinates into client coordinates.
`ScrollBy`	All controls	Scrolls the contents of the control.
`SendToBack`	All controls	Puts the control behind all the others.
`SetBounds`	All controls	Changes the position and size of the control (faster than accessing the related properties one by one).
`SetFocus`	All controls	Gives the input focus to the control.

`SetTextBuf`	All controls	Sets the text (or caption) of the control.
`Show`	All controls	Makes the control visible (the same as setting the `Visible` property to `True`).
`Update`	All controls	Immediately repaints the control, if there are pending painting requests.

Common VCL Events

Just as there is a set of properties common to all components, there are some events that are available for all of them. Table 4.3 provides short descriptions of these events. Again, this table is meant only as a starting point. You'll see examples using most of these events throughout the book.

Table 4.3: Some Events Available for Most Components

Event	Available For	Description
`OnCanResize`	Many controls	Occurs when the control is resized and allows you to stop the operation.
`OnChange`	Many components	Occurs when the object or its data change.
`OnClick`	Most controls	Occurs when the left mouse button is clicked over the component.
`OnContextPopupMenu`	All controls (new in Delphi 5)	Occurs when the user right-clicks the control. It allows you to do a different action than showing the attached popup menu.
`OnDblClick`	Many controls	Occurs when the user double-clicks with the mouse over the component.
`OnDockDrop`	Windowed controls	Occurs when the docking operation terminates over the current control.
`OnDockOver`	Windowed controls	Occurs when the user drags the mouse over the component during a docking operation.
`OnDragDrop`	Most controls	Occurs when a dragging operation terminates over the component; it is sent by the component that *received* the dragging operation.
`OnDragOver`	Most controls	Occurs when the user drags the mouse over the component.

`OnEndDock`	Most controls	Occurs when the docking operation of the current control terminates.
`OnEndDrag`	Most controls	Occurs when the dragging terminates; it is sent by the component that *started* the dragging operation.
`OnEnter`	All windowed controls	Occurs when the component is activated; that is, the component receives the focus.
`OnExit`	All windowed controls	Occurs when the component loses the focus.
`OnGetSiteInfo`	Windowed controls	Returns the control's docking information.
`OnKeyDown`	Some windowed controls	Occurs when the user presses a key on the keyboard; it is sent to the component with the input focus.
`OnKeyPress`	Some windowed controls	Occurs when the user presses a key; it is sent to the component with the input focus.
`OnKeyUp`	Some windowed controls	Occurs when the user releases a key; it is sent to the component with the input focus.
`OnMouseDown`	Most controls	Occurs when the user presses one of the mouse buttons; it is sent to the component under the mouse cursor.
`OnMouseMove`	Most controls	Occurs when the user moves the mouse over a component; it is sent to the component under the mouse cursor.
`OnMouseUp`	Most controls	Occurs when the user releases one of the mouse buttons; it is sent to the component under the mouse cursor.
`OnMouseWheel`, `OnMouseWheelDown`, `OnMouseWheelUp`	Windowed controls	Occur when the user rotates the mouse wheel or clicks on it as if it was a button.
`OnResize`	Most controls	Occurs when the resizing operation terminates.
`OnStartDock`	Most controls	Occurs when the user starts docking.
`OnStartDrag`	Most controls	Occurs when the user starts dragging; it is sent to the component *originating* the dragging operation.
`OnUnDock`	Windowed controls	Occurs when another control is undocked from the current one.

Understanding Frames

Chapter 1 introduced frames as one of the new features of Delphi 5. We've seen that you can create a new frame, place some components in it, write some event handlers for the components, and then add the frame to a form. In other words, a frame

is similar to a form, but it defines only a portion of a window, not a complete window. This is certainly not a feature worth a new construct. The totally new element of frames is that you can create multiple instances of a frame at design time and you can modify the class and the instance at the same time. This makes frames an effective tool for creating customizable composite controls at design time, something close to a visual component-building tool.

You are probably familiar with the Delphi concept of visual form inheritance (discussed in Chapter 2). You can work on both a base form and a derived form at design time, and any changes you make to the base form are propagated to the derived one, unless this overrides some property or event. With frames, you work on a class (as usual in Delphi), but the difference is that you can also customize one or more instances of the class created at design time. When you work on a form, you cannot change a property of the `TForm1` class for the `Form1` object at design time. With frames, you can.

Once you realize you are working with a class and one or more of its instances at design time, there is nothing more to understand about frames. In practice, frames are useful when you want to use the same group of components in multiple forms within an application. In this case, in fact, you can customize each of the instances at design time. Wasn't this already possible with component templates? It was, but component templates were based on the concept of copying and pasting some components and their code. There was no way to change the original definition of the template and see the effect in every place it was used. That is what happens with frames (and in a different way with visual form inheritance); changes to the original version (the class) are reflected in the copies (the instances).

There are many other uses of frames, which will become more apparent as Delphi programmers adopt this feature. Frames can be very useful when building multiple-page forms, as I'll demonstrate in Chapter 8.

Let's discuss a few more elements of frames with an example, called Frames2. This program has a frame with a list box, an edit box, and three buttons with simple code operating on the components. The frame also has a bevel aligned to its client area, because frames have no border. This is the definition of the frame in its own DFM file:

```
object FrameList: TFrameList
  Left = 0
  Top = 0
  Width = 202
  Height = 306
  TabOrder = 0
  object Bevel: TBevel
    Align = alClient
```

```
    Shape = bsFrame
  end
  object ListBox: TListBox...
  object Edit: TEdit
    Text = 'Some text'
  end
  object btnAdd: TButton
    Caption = '&Add'
    OnClick = btnAddClick
  end
  object btnRemove: TButton
    Caption = '&Remove'
    OnClick = btnRemoveClick
  end
  object btnClear: TButton
    Caption = '&Clear'
    OnClick = btnClearClick
  end
end
```

Of course, the frame has also a corresponding class, which looks like a normal form class:

```
type
  TFrameList = class(TFrame)
    ListBox: TListBox;
    Edit: TEdit;
    btnAdd: TButton;
    btnRemove: TButton;
    btnClear: TButton;
    Bevel: TBevel;
    procedure btnAddClick(Sender: TObject);
    procedure btnRemoveClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

What is different is that you can add the frame to a form. I've used two instances of the frame in the example (as you can see in Figure 4.6) and modified the behavior slightly. The first instance of the frame has the list box items sorted. When you change a property of a component of a frame, the DFM file of the hosting form will list the differences, as it does with visual form inheritance:

```
object FormFrames: TFormFrames
  Caption = 'Frames2'
  inline FrameList1: TFrameList
    Left = 8
    Top = 8
```

```
    inherited ListBox: TListBox
      Sorted = True
    end
  end
  inline FrameList2: TFrameList
    Left = 232
    Top = 8
    inherited btnClear: TButton
      OnClick = FrameList2btnClearClick
    end
  end
end
```

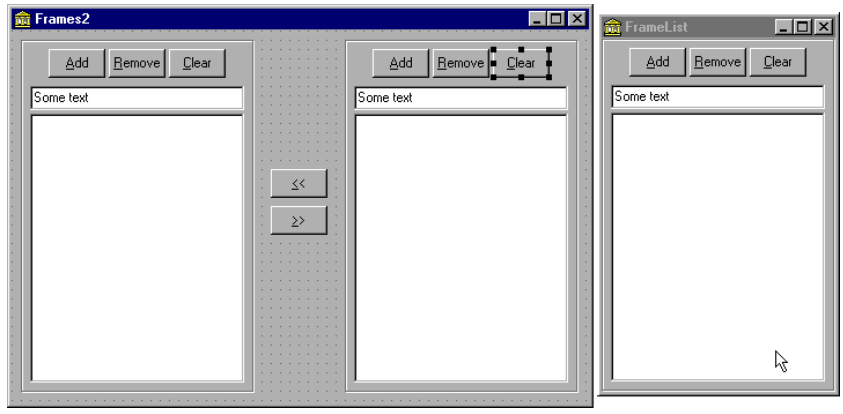


Figure 4.6: A frame and two instances of it at design time, in the Frames2 example. Image from the original book.

As you can see from the listing, the DFM file for a form that has frames uses a new DFM keyword, `inline`. The references to the modified components of the frame, instead, use the `inherited` keyword, although this term is used with an extended meaning. `inherited` here doesn't refer to a base class we are inheriting from, but to the class we are instancing (or inheriting) an object from. It was probably a good idea, though, to use an existing feature of visual form inheritance and apply it to the new context. The effect of this approach, in fact, is that you can use the Revert to Inherited command of the Object Inspector or of the form to cancel the changes and get back to the default value of properties.

Notice also that unmodified components of the frame class are not listed in the DFM file of the form using the frame, and that the form has two frames with different names, but the components on the two frames have the same name. In fact, these components are not owned by the form, but are owned by the frame. This implies that the form has to reference those components through the frame, as you can see in the code for the buttons that copy items from one list box to the other:

```
procedure TFormFrames.btnLeftClick(Sender: TObject);
begin
  FrameList1.ListBox.Items.AddStrings (
    FrameList2.ListBox.Items);
end;
```

Finally, besides modifying properties of any instance of a frame, you can change the code of any of its event handlers. If you double-click one of the buttons of a frame while working on the form (not on the stand-alone frame), Delphi will generate this code for you:

```
procedure TFormFrames.FrameList2btnClearClick(Sender: TObject);
begin
  FrameList2.btnClearClick(Sender);

end;
```

The line of code automatically added by Delphi corresponds to a call to the inherited event handler of the base class in visual form inheritance. This time, however, to get the default behavior of the frame we need to call an event handler and apply it to a specific instance—the frame object itself. The current form, in fact, doesn't include this event handler and knows nothing about it.

Whether you leave this call in place or remove it depends on the effect you are looking for. In the example I've decided to conditionally execute the default code, depending on the user confirmation:

```
procedure TFormFrames.FrameList2btnClearClick(Sender: TObject);
begin
  if MessageDlg ('OK to empty the list box?',
      mtConfirmation, [mbYes, mbNo], 0) = idYes then
    // execute standard frame code
    FrameList2.btnClearClick(Sender);
end;
```

note By the way, note that because the event handler has some code, leaving it empty and saving the form won't remove it as usual: in fact, it isn't empty! Instead, if you simply want to omit the default code for an event, you need to add at least a comment to it, to avoid it being automatically removed by the system!

Lists and Container Classes

It is often important to handle groups of components or objects. Besides using standard arrays and dynamic arrays, there are a few classes of the VCL that represent lists of other objects. These classes can be divided into three groups: simple lists, collections, and containers. The last group has been introduced in Delphi 5.

Lists are represented by the generic list of objects, `TList`, and by the two lists of strings, `TStrings` and `TStringList`:

- `TList` defines a list of pointers, which can be used to store objects of any class[147]. A `TList` is more flexible than a dynamic array, because it is expanded automatically, simply by adding new items to it. The advantage of dynamic arrays over a `TList`, instead, is that dynamic arrays allow you to indicate a specific type for contained objects and perform the proper compile-time type checking.
- `TStrings` is an abstract class to represent all forms of string lists, regardless of their storage implementations. This class defines an abstract list of strings. For this reason, `TStrings` objects are used only as properties of components capable of storing the strings themselves, such as a list box.
- `TStringList`, a subclass of `TStrings`, defines a list of strings with their own storage. You can use this class to define a list of strings in a program.

The second group, collections, contains only two classes, `TCollection` and `TCollectionItem`. `TCollection` defines a homogeneous list of objects, which are owned by the collection class. The objects in the collection must be descendants of the `TCollectionItem` class. If you need a collection storing specific objects, you have to create both a subclass of `TCollection` and a subclass of `TCollectionItem`. Collections are invariably used to specify values of properties of components. It is very unusual to work with collections directly inside programs. All these lists have a number of methods and properties. You can operate on lists using the array notation ("`[`" and "`]`") both to read and to change elements. There is a `Count` property, as well as typical access methods, such as `Add`, `Insert`, `Delete`, `Remove`, and search methods (for example, `IndexOf`).

`TStringList` and `TStrings` objects have both a list of strings and a list of objects associated with the strings. This opens up a number of different uses for these

147 Along with the introduction of the support for Generic programming in the Delphi language, the run-time library added a new *TList<T>* generic class, which can hold a list of objects of any specific class (and its sub-classes). Using a generic *TList<T>* makes applications more type safe and robust and it's highly recommended.

classes. For example, you can use them for dictionaries of associated objects or to store bitmaps or other elements to be used in a list box.

note The `TListbox` component actually uses a `TStringList` object when it needs to store strings while its window handle is invalid; it uses a different descendant of `TStrings` object when it finally associates with a Windows list box control, which stores its own strings.

The two classes of lists of strings also have ready-to-use methods to store or load their contents to or from a text file, `SaveToFile` and `LoadFromFile`. To loop through a list, you can use a simple `for` statement based on its index, as if the list were an array.

Using Lists of Objects

We can write an example focusing on the use of the generic `TList` class. When you need a list of any kind of data, you can generally declare a `TList` object, fill it with the data, and then access the data while casting it to the proper type. The ListDemo example demonstrates just this. It also shows the pitfalls of this approach[148]. Its form has a private variable, holding a list of dates:

```
private
 ListDate: TList;
```

This list object is created when the form itself is created:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  ListDate := TList.Create;
end;
```

A button of the form adds a random date to the list (of course, I've included in the project the unit containing the date component built in the previous chapter):

```
procedure TForm1.ButtonAddClick(Sender: TObject);
begin
  ListDate.Add (TDate.Create (1900 + Random (200),
    1 + Random (12), 1 + Random (30)));
end;
```

148 This pitfalls can be overcome using the generic *TList<T>*.

When you extract the items from the list, you have to cast them back to the proper type, as in the following method, which is connected to the List button (you can see its effect in Figure 4.7):

```
procedure TForm1.ButtonListDateClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add ((
      TObject(ListDate [I]) as TDate).Text);
end;
```

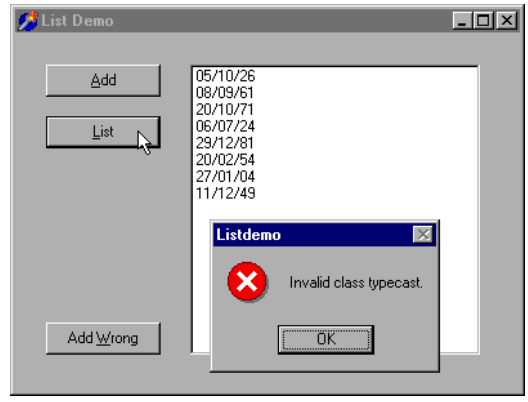


Figure 4.7: The list of dates shown by the ListDemo example. Image from the original book.

At the end of the code above, before we can do an `as` downcast, we first need to hard-cast the pointer returned by the `TList` into a `TObject` reference. This kind of expression can result in an invalid typecast exception, or it can generate a memory error when the pointer is not a reference to an object[149].

To demonstrate that things can indeed go wrong, I've added one more button, which adds a `TButton` object to the list:

```
procedure TForm1.ButtonWrongClick(Sender: TObject);
begin
  // add a button to the list
  ListDate.Add (Sender);
end;
```

149 Again, this can be addressed by using the generic class *TList<T>* based on the specific type of elements we want to add to the list.

If you click this button and then update one of the lists, you'll get an error. Finally, remember that when you destroy a list of objects, you should remember to destroy all of the objects of the list first. The ListDemo program does this in the `FormDestroy` method of the form:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ListDate.Count - 1 do
    TObject(ListDate [I]).Free;
  ListDate.Free;
end;
```

Delphi 5 Container Classes

Delphi 5 introduces a new series of container classes, defined in the Contnrs unit. These classes extend the `TList` classes, by adding the idea of ownership and defining specific extraction rules (mimicking stacks and queues). The basic difference between `TList` and the new `TObjectList`[150] class is that the latter is defined as a list of `TObject` objects, not a list of pointers. Even more important, however, is the fact that if the object list has the `OwnsObjects` property set to `True`, it automatically deletes an object when it is replaced by another one and deletes each object when the list itself is destroyed. Here's a list of all the new container classes:

- The `TObjectList` class I've already described represents a list of objects, eventually owned by the list itself.
- The inherited class `TComponentList` represents a list of components, with full support for destruction notification (an important safety feature when two components are connected using their properties; that is, when a component is the value of a property of another component).
- The `TClassList` class is a list of class references. It inherits from `TList` and requires no destruction.
- The classes `TStack`[151] and `TObjectStack` represent lists of pointers and objects, from which you can only extract elements starting from the last one you've inserted. A stack follows the LIFO order (Last In, First Out). The typical methods

150 There is now also a generic version, *TObjectList<T>*, available in the *System.Generics.Collections* unit.

151 Or the better equivalent *TStack<T>* in the *System.Generics.Collections* unit.

of a stack are Push for insertion, Pop for extraction, and Peek to preview the first item without removing it. You can still use all the methods of the base class, `TList`.

- The classes `TQueue`[152] and `TObjectQueue` represent lists of pointers and objects, from which you always remove the *first* item you've inserted (FIFO: First In, First Out). The methods of these classes are the same as those of the stack classes but behave differently.

note Unlike the `TObjectList`, the `TObjectStack` and the `TObjectQueue` do not own the inserted objects and will not destroy those objects left in the data structure when it is destroyed. You can simply `Pop` all the items, destroy them once you're finished using them, and then destroy the container.

To demonstrate the use of these classes, I've modified the earlier ListDate example into the new Contain example. First, I changed the type of the `ListDate` variable to `TObjectList`. In the `FormCreate` method, I've modified the list creation to the following code, which activates the list ownership:

```
ListDate := TObjectList.Create (True);
```

At this point, we can simplify the destruction code, as applying `Free` to the list will automatically free the dates it holds.

I've also added to the program a stack and a queue object, filling each of them with numbers. One of the form's two buttons displays a list of the numbers in each container, and the other removes the last item (displayed in a message box):

```
procedure TForm1.btnQueueClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to Stack.Count - 1 do
  begin
    ListBox1.Items.Add (IntToStr (Integer (Queue.Peek)));
    Queue.Push(Queue.Pop);
  end;
  ShowMessage ('Removed: ' + IntToStr (Integer (Stack.Pop)));
end;
```

By pressing the two buttons, you can see that calling Pop for each container returns the last item. The difference is that the `TQueue` class inserts elements at the beginning, and the `TStack` class inserts them at the end.

152 Or the better equivalent TQueue<T> in the *System.Generics.Collections* unit.

Type-Safe Containers and Lists

Containers and lists have a problem: They are not type safe, as I've shown in both examples by adding a button object to a list of dates. To ensure that the data in a list is homogenous, you can check the type of the data you extract before you insert it, but as an extra safety measure you might also want to check the type of the data while extracting it. However, adding run-time type checking slows down a program and is risky—a programmer might fail to check the type in some cases.

To solve both problems, you can create specific list classes for given data types and fashion the code from the existing `TList` or `TObjectList` classes (or another container class). There are two approaches to accomplish this[153]:

- Derive a new class from the list class and customize the `Add` method and the access methods, which relate to the `Items` property. This is also the approach used by Borland for the container classes, which all derive from `TList`.
- Create a brand-new class that contains a `TList` object, and map the methods of the new class to the internal list using proper type checking. This approach defines a wrapper class, a class that "wraps" around an existing one to provide a different or limited access to its methods (in our case, to perform a type conversion).

I've implemented both solutions in the DateList example, which defines lists of `TDate` objects. In the listing below you'll find the declaration of the two classes, the inheritance-based `TDateListI` class and the wrapper class `TDateListW`.

```
type
// inheritance based
TDateListI = class (TObjectList)
protected
  procedure SetObject (Index: Integer; Item: TDate);
  function GetObject (Index: Integer): TDate;
public
  function Add (Obj: TDate): Integer;
  procedure Insert (Index: Integer; Obj: TDate);
  property Objects [Index: Integer]: TDate
    read GetObject write SetObject; default;
end;
// wrapper based
TDateListW = class(TObject)
private
  FList: TObjectList;
```

153 There is now a much better and easier approach, which is using the generic container classes in the *System.Generics.Collections* unit.

```
  function GetObject (Index: Integer): TDate;
     procedure SetObject (Index: Integer; Obj: TDate);
  function GetCount: Integer;
public
  constructor Create;
  destructor Destroy; override;
  function Add (Obj: TDate): Integer;
  function Remove (Obj: TDate): Integer;
  function IndexOf (Obj: TDate): Integer;
  property Count: Integer read GetCount;
  property Objects [Index: Integer]: TDate
    read GetObject write SetObject; default;
end;
```

Obviously, the first class is simpler to write—it has fewer methods, and they simply call the inherited ones. The good thing is that a `TDateListI` object can be passed to parameters expecting a `TList`. The problem is that the code that manipulates an instance of this list via a generic `TList` variable will not be calling the specialized methods, because they are not virtual and might end up adding to the list objects of other data types.

Instead, if you decide not to use inheritance, you end up writing a lot of code, because you need to reproduce each and every one of the original `TList` methods, simply calling the methods of the internal `FList` object. The drawback is that the `TDateListW` class is not type compatible with `TList`, which limits its usefulness. It can't be passed as parameter to methods expecting a `TList`.

Both of these approaches provide good type checking. After you've created an instance of one of these list classes, you can add only objects of the appropriate type, and the objects you extract will naturally be of the correct type. This is demonstrated by the DateList example. This program has a few buttons, a combo box to let a user choose which of the lists to show, and a list box to show the actual values of the list. The program stretches the lists by trying to add a button to the list of `TDate` objects. To add an object of a different type to the `TDateListI` list, we can simply convert the list to its base class, `TList`. This might accidentally happen if you pass the list as a parameter to a method that expects a base class object. In contrast, for the `TDateListW` list to fail we must explicitly cast the object to `TDate` before inserting it, something a programmer should never do:

```
procedure TForm1.ButtonAddButtonClick(Sender: TObject);
begin
  ListW.Add (TDate(TButton.Create (nil)));
  TList(ListI).Add (TButton.Create (nil));
  UpdateList;
end;
```

The `UpdateList` call triggers an exception, displayed directly in the list box, because I've used an `as` type cast in the custom list classes. A wise programmer should never write the above code.

To summarize, writing a custom list for a specific type makes a program much more robust. Writing a wrapper list instead of one that's based on inheritance tends to be a little safer, although it requires more coding.

---

note Instead of rewriting wrapper-style list classes for different types, you can use my List Template Wizard, discussed in *Delphi Developer's Handbook* and available on my Web site.[154]

What's Next?

As we have seen in this chapter, Delphi includes a full-scale class library that is just as complete as Microsoft's MFC C++ class library. Delphi's VCL, of course, is much more component-oriented, and its classes offer a higher-level abstraction over the Windows API than the C++ libraries usually do.

To use components, you only need a clear understanding of the terminal nodes of the VCL hierarchy; that is, the components that show up in the Component Palette plus a few others. You really don't need a deeper knowledge of the VCL internals to use components; this knowledge is only necessary when you write new components or modify existing ones.

This chapter ends Part I of the book, which has covered the foundations of Delphi programming. Part II is fully devoted to examples of the use of the various components. We'll start in Chapter 5 with the advanced use of traditional Windows controls and menus, cover the `TForm` class in Chapter 6, and then examine toolbars, status bars, dialog boxes, and MDI applications in later chapters.

---

154 This is not available any more, given it's pretty much useless after the introduction of generics to the Delphi language and of generic collections in the RTL.

Chapter 5: Advanced Use Of The Standard Components

Now that you've been introduced to the Delphi environment and have seen an overview of the Object Pascal language and the Visual Component Library, we are ready to delve into the second part of the book: the use of components. This is really what Delphi is about. Visual programming using components is the key feature of this development environment.

Delphi comes with a number of ready-to-use components. I will not describe every component in detail, examining each of its properties and methods. If you need this

information, you can find it easily in the Help system. The aim of Part II of this book is to show you how to use some of the advanced features offered by the Delphi predefined components to build applications.

I'll start by trying to list all the various component alternatives you have, since choosing the right component is often a way to get into a project faster. This chapter presents the components in the Standard page of the Component Palette and some of the Win32 controls.

Opening the Component Tool Box

So you want to write a Delphi application[155]. You open a new Delphi project and find yourself faced with a large number of components. The problem is that for every operation there are multiple alternatives. For example, you can show a list of values using a list box, a combo box, a radio group, a string grid, a list view, or even a tree view if there is a hierarchical order. Which one should you use? That is difficult to say. There are many considerations, depending on what you want your application to do. For this reason I've provided a highly condensed summary of alternative options for a few common tasks.

note For some of the controls described in the following sections Delphi also includes a data-aware version, usually indicated by the DB prefix. As you'll see in Chapter 9, the DB version of a control typically serves a role similar to that of its "standard" equivalent; but the properties and the ways you use it are often quite different. For example, in an Edit control you use the `Text` property, while in a DBEdit component you access the `Value` of the related field object.

The Text Input Component

Although a form or a component can handle keyboard input directly, using the `OnKeyPress` event, this isn't a common operation. Windows provides ready-to-use controls you can use to get string input and even build a simple text editor. Delphi has several slightly different components in this area.

155 At the time of this book, using VCL for the UI was the only option. These days you can choose between VCL and FireMonkey, which has similar UI controls but is based on a completely different architecture. FireMonkey is not covered in this book, which is focused on VCL and Windows programming, because that's what was available in the Delphi 5 timeframe.

The Edit Component

The Edit component allows the user to enter a single line of text[156]. (You can also display a single line of text with a Label or a StaticText control, but these components are generally used only for fixed text or program-generated output, not for input.) The Edit component uses the `Text` property, whereas many other controls use the `Caption` property to refer to the text they display. The only condition you can impose on user input is the number of characters to accept. If you want to accept only specific characters, you can handle the `OnKeyPress` event of the edit box. For example, we can write a method that tests whether the character is a number or the Backspace key (which has a numerical value of 8). If it's not, we change the value of the key to the null character (`#0`), so that it won't be processed by the edit control and will produce a warning beep:

```
procedure TForm1.Edit1KeyPress(
  Sender: TObject; var Key: Char);
begin
  // check if the key is a number or backspace
  if not (Key in ['0'..'9', #8]) then
  begin
    Key := #0;
    Beep;
  end;
end;
```

The MaskEdit Component

To customize the input of an edit box further, you can use the MaskEdit component, which has an `EditMask` property. This is a string indicating for each character whether it should be uppercase, lowercase, or a number, and other similar conditions. You can see the editor of the `EditMask` property in Figure 5.1.

note You can display any property's editor by selecting the property in the Object Inspector and clicking the ellipsis (...) button.

The Input Mask editor allows you to enter a mask, but it also asks you to indicate a character to be used as a placeholder for the input and to decide whether to save the *literals* present in the mask, together with the final string. For example, you can choose to display the parentheses around the area code of a phone number only as

156 There is now also a NumberBox component, which is specific meant for the input of numeric values, including integers, floating point numbers, and currency. It's a much better solution comapred to the code in the snippet below to make an edit accept only numeric characters.

an input hint or to save them with the string holding the resulting number. These two entries in the Input Mask editor correspond to the last two fields of the mask (separated by semicolons).

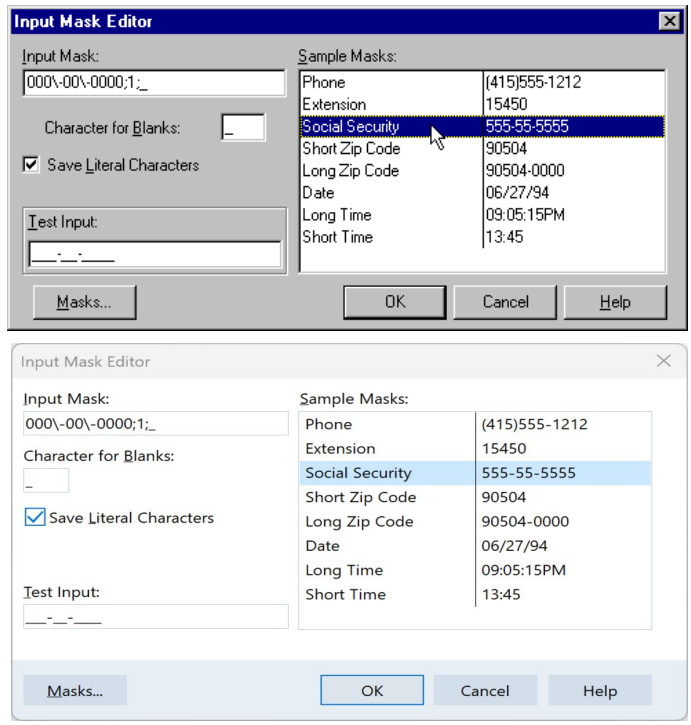


Figure 5.1: The MaskEdit component's EditMask property editor. Images captured in Delphi 5 and Delphi 12.

note Pressing the Masks button of the Mask Editor lets you choose predefined input masks for different countries.

The Memo and RichEdit Components

Both of the controls discussed so far allow a single line of input. The Memo component, by contrast, can host several lines of text but (on the Win95/98 platforms) still retains the 16-bit Windows 32KB text limit and allows only a single font for the entire text. You can work on the text of the memo line by line (using the `Lines` string list) or access the entire text at once (using the `Text` property).

If you want to host a large amount of text or change fonts and paragraph alignments, you should use the RichEdit control, a Win32 common control based on the

RTF document format. You can find an example of a complete editor based on the RichEdit component among the sample programs that ship with Delphi. (The example is named RichEdit, too.)

The RichEdit component has a `DefAttributes` property indicating the default styles and a `SelAttributes` property indicating the style of the current selection. These two properties are not of the `TFont` type, but they are compatible with fonts, so we can use the `Assign` method to copy the value, as in the following code fragment:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
  begin
    FontDialog1.Font.Assign (RichEdit1.DefAttributes);
    if FontDialog1.Execute then
      RichEdit1.SelAttributes.Assign (FontDialog1.Font);
  end;
end;
```

Selecting Options

There are two standard Windows controls that allow the user to choose different options, as well as controls for grouping sets of options.

The CheckBox and RadioButton Components

The first is the *check box*, which corresponds to an option that can be selected regardless of the status of other check boxes. Setting the `AllowGrayed` property of the check box allows you to display three different states (selected, not selected, and grayed), which alternate as a user clicks on the check box.

The second type of control is the *radio button*, which corresponds to an exclusive selection. Two radio buttons on the same form or inside the same radio group container cannot be selected at the same time, and one of them should always be selected (as programmer, you are responsible for selecting one of the radio buttons at design time).

The GroupBox Components

To host several groups of radio buttons, you can use a GroupBox control to hold them together, both functionally and visually. To build a group box with radio buttons, simply place the GroupBox component on a form and then add the radio buttons to the group box.

You can handle the radio buttons individually, but it's easier to navigate through the array of controls owned by the group box, as discussed in the previous chapter. Here is a small code excerpt used to get the text of the selected radio button of a group:

```
var
  I: Integer;
  Text: string;
begin
  for I := 0 to GroupBox1.ControlCount - 1 do
    if (GroupBox1.Controls[I] as TRadioButton).Checked then
      Text := (GroupBox1.Controls[I] as TRadioButton).Caption;
```

The RadioGroup Component

Delphi has a similar component that can be used specifically for radio buttons, the RadioGroup component. A RadioGroup is a group box with some radio button *clones* painted inside it. The term *clone* in this context refers to the fact that the RadioGroup component is a single control, a single window, with elements similar to radio buttons painted on its surface.

Using the radio group is generally easier than using the group box, since the various items are part of a list, as in a list box. This is how you can get the text of the selected item:

```
Text := RadioGroup1.Items [RadioGroup1.ItemIndex];
```

Technically, a RadioGroup uses fewer resources and less memory, and it should be faster to create and paint. Also, the RadioGroup component can automatically align its radio buttons in one or more columns (as indicated by the `Columns` property), and you can easily add new choices at run time, by adding strings to the `Items` string list. By contrast, adding new radio buttons to a group box would be quite complex.

Lists

When you have many selections, radio buttons are not appropriate. The usual number of radio buttons is no more than five or six, to avoid cluttering the user interface; when you have more choices, you can use a list box or one of the other controls that display lists of items and allow the selection of one of them.

The ListBox Component

The selection of an item in a list box uses the `Items` and `ItemIndex` properties as in the code shown above for the RadioGroup control. If you need access to the text of selected list box items often, you can write a small wrapper function like this:

```
function SelText (List: TListBox): string;
var
  nItem: Integer;
begin
  nItem := List.ItemIndex;
  if nItem >= 0 then
    Result := List.Items [nItem]
  else
    Result := '';
end;
```

Another important feature is that by using the ListBox component, you can choose between allowing only a single selection, as in a group of radio buttons, and allowing multiple selections, as in a group of check boxes. You make this choice by specifying the value of the `MultiSelect` property. There are two kinds of multiple selections in Windows and in Delphi list boxes: *multiple selection* and *extended selection*. In the first case a user selects multiple items simply by clicking on them, while in the second case the user can use the Shift and Ctrl keys to select multiple consecutive or nonconsecutive items. This second choice is determined by the `ExtendedSelect` property.

For a multiple-selection list box, a program can retrieve information about the number of selected items by using the `SelCount` property, and it can determine which items are selected by examining the `Selected` array. This array of `Boolean` values has the same number of entries as the list box. For example, to concatenate all the selected items into a string, you can scan the `Selected` array as follows:

```
var
  SelItems: string;
  nItem: Integer;
begin
  SelItems := '';
  for nItem := 0 to ListBox1.Items.Count - 1 do
    if ListBox1.Selected [nItem] then
      SelItems := SelItems + ListBox1.Items[nItem] + ' ';
```

The ComboBox Component

List boxes take up a lot of screen space, and they offer a fixed selection. That is, a user can choose only among the items in the list box and cannot enter any choice that the programmer did not specifically foresee.

You can solve both problems by using a ComboBox control, which combines an edit box and a drop-down list. The behavior of a ComboBox component changes a lot depending on the value of its `Style` property. The `csDropDown` style defines a typical combo box, which allows direct editing and displays a list box on request, the `csDropDownList` style defines a combo box that does not allow editing (but uses the keystrokes to select an item), and the `csSimple` style defines a combo box that always displays the list box below it.

Note also that accessing the text of the selected value of a ComboBox is easier than doing the same operation for a list box, since you can simply use the `Text` property. A useful and common trick for combo boxes is to add a new element to the list when a user enters some text and presses the Enter key. The following method first tests whether the user has pressed that key, by looking for the character with the numeric (ASCII) value of 13. It then tests to make sure the text of the combo box is not empty and is not already in the list—if its position in the list is less than zero. Here is the code:

```
procedure TForm1.ComboBox1KeyPress(Sender: TObject; var Key: Char);
begin
  // if the user presses the Enter key
  if Key = Chr (13) then
    with ComboBox3 do
      if (Text <> '') and (Items.IndexOf (Text) < 0) then
        Items.Add (Text);
end;
```

The CheckListBox Component

Another extension of the list box control is represented by the CheckListBox component, a list box with each item preceded by a check box (as you can see in Figure 5.2). A user can select a single item of the list, but can also click on the check boxes to toggle their status. This makes the CheckListBox a very good component for multiple selections or for highlighting the status of a series of independent items (as in a series of check boxes).

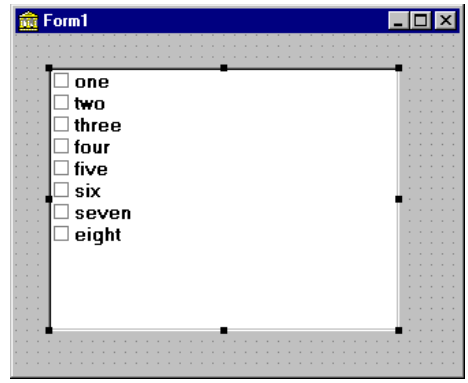


Figure 5.2: The user interface of the CheckListBox control, basically a list of check boxes. Image from the original book.

To check the current status of each item, you can use the `Checked` and the `State` array properties (use the latter if the check boxes can be grayed). Delphi 5 introduces the `ItemEnabled` array property, which you can use to enable or disable each item of the list. We'll use the CheckListBox in the DragList example, later on in this chapter.

note Most of the list-based controls share a common and important feature. Each item of the list has an associated 32-bit value, usually indicated by the `TObject` type. This value can be used as a tag for each list item, and it's very useful for storing additional information along with each item. This approach is connected to a specific feature of the native Windows list box control, which offers four bytes of extra storage for each list box item. We'll use this feature in the ODList example later on in this chapter.

The ListView and TreeView Components

If you want an even more sophisticated list, you can use the ListView Win32 common control, which will make the user interface of your application look very modern. This component is slightly more complex to use, as described toward the end of this chapter. Other alternatives for listing values are the TreeView common control, which shows items in a hierarchical output, and the StringGrid control, which shows multiple elements for each line. The string grid control is described in Chapter 22, "Graphics in Delphi".[157]

If you use the common controls in your application, users will already know how to interact with them, and they will regard the user interface of your program as up to

157 This was originally a bonus chapter available as a separate download on the publisher web site, but it's now part of this ebook.

date. TreeView and ListView are the two key components of Windows Explorer, and you can assume that many users will be familiar with them, even more than with the traditional Windows controls.

Ranges

Finally, there are a few components you can use to select values in a range. Ranges can be used for numeric input and for selecting an element in a list.

The ScrollBar Component

The stand-alone ScrollBar control is the original component of this group, but it is seldom used by itself. Scroll bars are usually associated with other components, such as list boxes and memo fields, or are associated directly with forms. In all these cases, the scroll bar can be considered part of the surface of the other components. For example, a form with a scroll bar is actually a form that has an area resembling a scroll bar painted on its border, a feature governed by a specific Windows style of the form window. By *resembling,* I mean that it is not technically a separate window of the ScrollBar component type. These “fake” scroll bars are usually controlled in Delphi using specific properties of the form and the other components hosting them.

The TrackBar and ProgressBar Components

Direct use of the ScrollBar component is quite rare, especially with the TrackBar component introduced with Windows 95, which is used to let a user select a value in a range. Among Win32 common controls there is the companion ProgressBar control, which allows the program to output a value in a range, showing the progress of a lengthy operation.

The UpDown Component

Another related control is the UpDown component, which is usually connected to an edit box so that the user can either type a number in it or increase and decrease the number using the two small arrow buttons. To connect the two controls, you set the `Associate` property of the UpDown component. Nothing prevents you from using the UpDown component as a stand-alone control, displaying the current value in a label or in some other way.

The PageScroller Component

The Win32 PageScroller control is a container allowing you to scroll the internal control. For example, if you place a toolbar in the page scroller and the toolbar is larger than the available area, the PageScroller will display two small arrows on the side. Pressing these arrows will scroll the internal area. This component can be used as a scrollbar, but it also partially replaces the ScrollBox control.

The ScrollBox Component

The ScrollBox control represents a region of a form, which can scroll independently from the rest of the surface. For this reason the ScrollBox has two scrollbars used to move the embedded components. You can easily place other components inside a ScrollBox, as you do with a panel. In fact, a ScrollBox is basically a panel with scroll bars to move its internal surface, an interface element used in many Windows applications. When you have a form with many controls and a toolbar or status bar, you might use a ScrollBox to cover the central area of the form, leaving its toolbars and status bars outside of the scrolling region. By relying on the scrollbars of the form, in fact, you might allow the user to move the toolbar or status bar out of view, a very odd situation.

Dragging from One Component to Another

Now that you've been introduced to the standard controls, we'll examine a couple of general techniques: dragging and focus handling. Let me start with a simple example of dragging, called DragList. The form of this example, shown in Figure 5.3 at run time, contains a ListBox and a CheckListBox. You can drag items from one control to the other. It also has an edit box you can use to enter new items and drag them to either list. If you run the program, you'll see that there is also a rule: Lists cannot have duplicated items. This means we have to check whether the item is already in the list before inserting it.

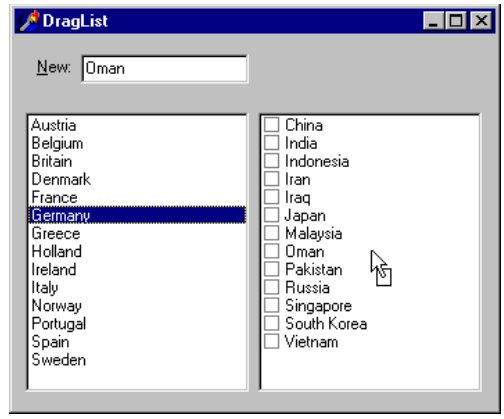


Figure 5.3: The form of the DragList example at run time, during a dragging operation. Image from the original book.

The two list boxes use the `dmAutomatic` value for the `DragMode` property (with the `DragKind` property left to the default value `dkDrag`). For the edit box, by contrast, we have to use manual dragging to let the edit box behave as usual when a user clicks on it. For this reason, as a user presses the mouse button over the edit box, we must initiate the dragging operation, delaying it as indicated by the first parameter of the `BeginDrag` method:

```
procedure TDragForm.Edit1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Edit1.BeginDrag (False, 10);
end;
```

The two lists share the same handler for the `OnDragOver` event, which is used to determine whether the control accepts dragging from a given source. In this handler, when the user is dragging from the edit box, the program checks to see whether the text is already in the list and disallows the dragging operation if it is. We can easily write a single event handler for both controls because they inherit from the same base class, `TCustomListBox`:

```
procedure TDragForm.ListDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  Accept := True;
  // if the source is the edit and the items
  // is already in the destination list, reject it
  if (Source = Edit1) and
    ((Sender as TCustomListBox).Items.IndexOf (Edit1.Text) >= 0) then
    Accept := False;
end;
```

The handlers of the `OnDragDrop` events, however, are quite different, so I've decided to separate them. The list box allows only a single item to be selected, while the list check box can have multiple selected items; this makes the code quite different in the two cases. What can be shared is the code to add an item to a list only if it is not already there. I've added this shared code to a method of the form, which is called by both event handlers:

```
function TDragForm.AddNotDup (List: TCustomListBox;
  Text: string): Boolean;
begin
  // return if the string was not already in the list
  Result := List.Items.IndexOf (Text) < 0;
  if Result then
    List.Items.Add (Text);
end;
```

The code for the two drag-drop methods is quite simple. For the check list box, the program copies the text of the edit box or that of the selected list box item and removes it from the source:

```
procedure TDragForm.CheckListBox1DragDrop(Sender,
  Source: TObject; X, Y: Integer);
var
  nItem: Integer;
begin
  if Source = Edit1 then
    // copy the text of the edit box
    CheckListBox1.Items.Add (Edit1.Text)
  else if Source = ListBox1 then
  begin
    // copy if not duplicate
    nItem := ListBox1.ItemIndex;
    if AddNotDup (CheckListBox1, ListBox1.Items [nItem]) then
      // remove source item
      ListBox1.Items.Delete (nItem);
  end;
end;
```

For the list box, we have to scan all the items of the check list box to see which one is selected. Since we want to delete the items we copy, we must do this operation in reverse order, because deleting an item changes the position of the items that follow it:

```
procedure TDragForm.ListBox1DragDrop(Sender,
  Source: TObject; X, Y: Integer);
var
  I: Integer;
begin
  if Source = Edit1 then
    // copy the text of the edit box
```

```
    ListBox1.Items.Add (Edit1.Text)
  else if Source = CheckListBox1 then
  begin
    // copy all the selected items (unless duplicate)
    // and delete them (using reverse order!)
    for I := CheckListBox1.Items.Count - 1 downto 0 do
      if CheckListBox1.Checked [I] then
      begin
        if AddNotDup (ListBox1, CheckListBox1.Items [I]) then
          CheckListBox1.Items.Delete (I);
      end;
  end;
end;
```

note We'll see an example of dragging operations within a TreeView control at the end of this chapter.

Handling the Input Focus

Using the `TabStop` and `TabOrder` properties available in most controls, you can specify the order in which controls will receive the input focus when the user presses the Tab key. Instead of setting the tab order property of each component of a form manually, you can use the shortcut menu of the Form Designer to activate the Edit Tab Order dialog box, as shown in Figure 5.4.

Besides these basics settings, it is important to know that each time a component receives or loses the input focus, it receives a corresponding `OnEnter` or `OnExit` event. This allows you to fine-tune and customize the order of the user operations. Some of these techniques are demonstrated by the InFocus example, which creates a fairly typical password-login window. Its form has three edit boxes with labels indicating their meaning, as shown in Figure 5.5. At the bottom of the window is a status area with prompts guiding the user. Each item needs to be entered in sequence.

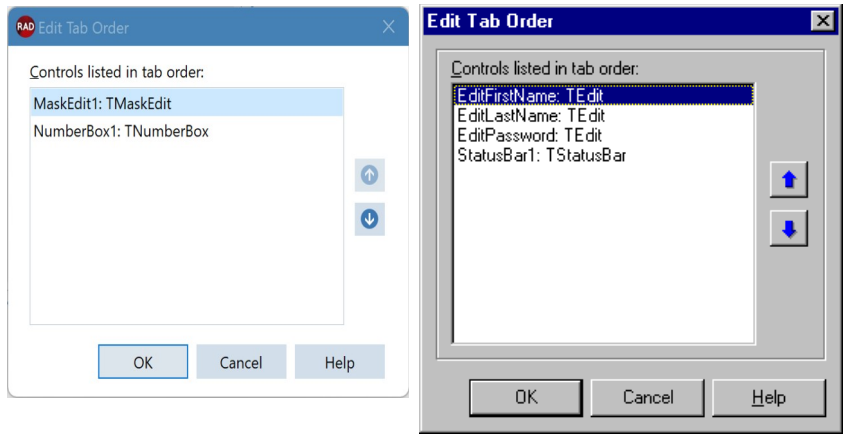


Figure 5.4: The Edit Tab Order dialog box. Images captured in Delphi 5 and Delphi 12.

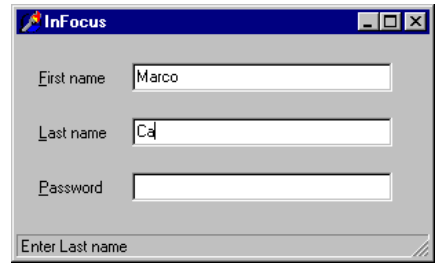


Figure 5.5: The InFocus example at run time. Image from the original book.

For the output of the status information I've used the StatusBar component, with a single output area (obtained by setting its `SimplePanel` property to `True`). Here is a summary of the properties for this example. Notice the `&` character in the labels, indicating a shortcut key, and the connection of these labels with corresponding edit boxes (using the `FocusControl` property):

```
object FocusForm: TFocusForm
  ActiveControl = EditFirstName
  Caption = 'InFocus'
  object Label1: TLabel
    Caption = '&First name'
    FocusControl = EditFirstName
  end
  object EditFirstName: TEdit
    OnEnter = GlobalEnter
    OnExit = EditFirstNameExit
  end
  object Label2: TLabel
```

```
    Caption = '&Last name'
    FocusControl = EditLastName
  end
  object EditLastName: TEdit
    OnEnter = GlobalEnter
  end
  object Label3: TLabel
    Caption = '&Password'
    FocusControl = EditPassword
  end
  object EditPassword: TEdit
    PasswordChar = '*'
    OnEnter = GlobalEnter
  end
  object StatusBar1: TStatusBar
    SimplePanel = True
  end
end
```

The program is very simple and does only two operations. The first is to identify, in the status bar, the edit control that has the focus. It does this by handling the controls' `OnEnter` event, possibly using a single generic event handler to avoid repetitive code. In the example, instead of storing some extra information for each edit box, I've checked each control of the form to determine which label is connected to the current edit box (indicated by the `Sender` parameter):

```
procedure TFocusForm.GlobalEnter(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ControlCount - 1 do
    // if the control is a label
    if (Controls [I] is TLabel) and
      // and the label is connected to the current edit box
      (TLabel(Controls[I]).FocusControl = Sender) then
    // copy the text leaving off the initial & character
    StatusBar1.SimpleText := 'Enter ' +
      Copy (TLabel(Controls[I]).Caption, 2, 1000);
end;
```

The second event handler of the form relates to the `OnExit` event of the first edit box. If the control is left empty, it refuses to release the input focus and sets it back before showing a message to the user. The methods also look for a given input value, automatically filling the second edit box and moving the focus directly to the third one:

```
procedure TFocusForm.EditFirstNameExit(Sender: TObject);
begin
  if EditFirstName.Text = '' then
  begin
```

```
    // don't let the user get out
    EditFirstName.SetFocus;
    MessageDlg ('First name is required',
      mtError, [mbOK], 0);
  end
  else if EditFirstName.Text = 'Admin' then
  begin
    // fill the second edit and jump to the third
    EditLastName.Text := 'Admin';
    EditPassword.SetFocus;
  end;
end;
```

Working with Menus

Working with menus and menu items is generally quite simple. This section offers only some very brief notes and a few more advanced examples. The first thing to keep in mind about menu items is that they can serve different purposes:

- **Commands** are menu items used to execute an action.
- **State-setters** are menu items used to toggle an option on and off, to change the state of a particular element. These commands usually have a check mark on the left to indicate they are active.
- **Radio items** have a round check mark and are grouped to represent alternative selections, like radio buttons. To obtain radio menu items, simply set the `RadioItem` property to `True` and set the `GroupIndex` property for the alternative menu items to the same value.
- **Dialog menu items** cause a dialog box to appear and are usually indicated by an ellipsis (three dots) after the text.

As you enter new elements in the Menu Designer, Delphi creates a new component for each menu item and lists it in the Object Inspector (although nothing is added to the form). To name each component, Delphi uses the caption you enter and appends a number (so that *Open* becomes `Open1`). Because Delphi removes spaces and other special characters in the caption when it creates the name, and the menu item separators are set up using a hyphen as caption, these items would have an empty name. For this reason Delphi adds the letter N to the name, appending the number and generating items called `N1`, `N2`, and so on.

note Do not use the `Break` property, which is used to lay out a pull-down menu on multiple columns. The `mbMenuBarBreak` value indicates that this item will be displayed in a second or subsequent line, the `mbMenuBreak` value that this item will be added to a second or subsequent column of the pull-down.

Accelerator Keys in Delphi 5

In Delphi 5 you don't need to enter the `&` character in the `Caption` of a menu item; it provides an automatic accelerator key if you omit one. The Delphi 5 automatic accelerator key system can also figure out if you have entered conflicting accelerator keys and fix them on the fly. This doesn't mean you should stop adding custom accelerator keys with the `&` character, because the automatic system simply uses the first available letter, and it doesn't follow the default standards. You might also find better mnemonic keys than those chosen by the automatic system.

This new Delphi 5 feature is controlled by the `AutoHotkeys` property, which is available in the main menu component and in each of the pull-down menus and menu items. In the main menu, this property defaults to `maAutomatic`, while in the pull-downs and menu items it defaults to `maParent`, so that the value you set for the main menu component will be used automatically by all the subitems, unless they have a specific value of `maAutomatic` or `maManual`.

The engine behind this system is the `RethinkHotkeys` method of the `TMenuItem` class, and the companion `InternalRethinkHotkeys`. There is also a `RethinkLines` method, which checks whether a pull-down has two consecutive separators, or begins or ends with a separator. In all these cases the separator is automatically removed.

One of the reasons Delphi includes this new feature is the new ITE (Integrated Translation Environment)[158]. When you need to translate the menu of an application, it is convenient if you don't have to deal with the accelerator keys, or at least if you don't have to worry about whether two items on the same menu conflict. Having a system that can automatically resolve similar problems is definitely an advantage. Another motivation was Delphi's IDE itself. With all the dynamically loaded packages that install menu items in the IDE main menu or in pop-up menus, and with different packages loaded in different versions of the product, it's next to impossible

158 The VCL translation support has recently been removed as an official feature of the product and is only available as an additional download in the GetIt Package Manager. The foundations of the concepts remain valid and can be applicable to other, third-party, translation tools.

to get non-conflicting accelerator-key selections in each menu. That is why this mechanism isn't a wizard that does static analysis of your menus at design time; it was created to deal with the real problem of managing menus created dynamically at run time.

note This new feature is certainly very handy, but because it is active by default, it can break existing code. I had to modify two of this chapter's program examples from the previous edition of the book, just to avoid run-time errors caused by this change. As we'll see later, the problem is that I use the caption in the code, and the extra & broke my code. The change was quite simple, though, as all I had to do was to set the `AutoHotkeys` property of the main menu component to `maManual`.

Pop-Up Menus and the OnContextPopup Event

Besides the MainMenu component, you can use the similar PopupMenu component. This is typically displayed when the user right-clicks a component that uses the given pop-up menu as the value for its `PopupMenu` property.

However, besides connecting the pop-up menu to a component with the corresponding property, you can call its `Popup` method, which requires the position of the pop-up in screen coordinates. The proper values can be obtained by converting a local point to a screen point with the `ClientToScreen` method of the local component, in this code fragment a label:

```
procedure TForm1.Label3MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ScreenPoint: TPoint;
begin
  // if some condition applies...
  if Button = mbRight then
  begin
    ScreenPoint := Label3.ClientToScreen (
      Point (X, Y));
    PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
  end;
end;
```

An alternative approach provided by Delphi 5 is the use of the `OnContextMenu` event. This brand-new event fires when a user right-clicks on a component, exactly what we've traced above with the test `if Button = mbRight`. The advantage is that the same event is also fired in response to a Shift+F10 key combination, as well as by any other user input methods defined by Windows Accessibility options or hard-

ware (including the shortcut menu key of some Windows-compatible keyboards). We can use this event to fire a pop-up menu with little code:

```
procedure TFormPopup.Label1ContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
var
  ScreenPoint: TPoint;
begin
  // add dynamic items
  PopupMenu2.Items.Add (NewLine);
  PopupMenu2.Items.Add (NewItem (TimeToStr (Now),
    0, False, True, nil, 0, ''));
  // show popup
  ScreenPoint := ClientToScreen (MousePos);
  PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
  Handled := True;
  // remove dynamic items
  PopupMenu2.Items [4].Free;
  PopupMenu2.Items [3].Free;
end;
```

This example adds some dynamic behavior to the shortcut menu, adding a temporary item indicating when the pop-up menu is displayed. This is not particularly useful, but I've done it to highlight that if you need to display a plain pop-up menu, you can easily use the PopupMenu property of the control in question or one of its parent controls. Handling the `OnContextMenu` event makes sense only when you want to do some extra processing.

The `Handled` parameter is initialized to `False`, so that if you do nothing in the event handler, the normal pop-up menu processing will occur. If you do something in your event handler to replace the normal pop-up menu processing (such as popping up a dialog or a customized menu, as in this case), you should set `Handled` to `True` and the system will stop processing the message. Setting `Handled` to `True` should be fairly rare, as you'll generally handle the `OnContextPopup` to dynamically create or customize the pop-up menu, but then you can let the default handler actually show the menu.

The handler of an `OnContextPopup` event isn't limited to displaying a pop-up menu. It can do any other operation, such as directly display a dialog box. Here is an example of a right-click operation used to change the color of the control:

```
procedure TFormPopup.Label2ContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
begin
  ColorDialog1.Color := Label2.Color;
  if ColorDialog1.Execute then
    Label2.Color := ColorDialog1.Color;
  Handled := True;
```

```
end;
```

All the code snippets of this section are available in the simple CustPop example.

Creating Menu Items Dynamically

Besides defining the structure of a menu with the Menu Designer and modifying the status of the items using the `Checked`, `Visible`, and `Caption` properties, you can create an entire menu or portions of one at run time. This makes sense, for example, when you have many repetitive items, or when the menu items depend on some system configuration or user permissions.

The basic idea is that each object of the `TMenuItem` class—which Delphi uses for both menu items and pull-down menus—contains a list of menu items. Each of these items has the same structure, in a kind of recursive way. A pull-down menu has a list of submenus, and each sub-menu has a list of sub-menus, each with its own list of submenus, and so on. The properties you can use to explore the structure of an existing menu are `Items`, which contains the actual list of menu items, and `Count`, which contains the number of subitems. Adding new menu items or entire pull-down menus to an existing menu is fairly easy, particularly if you can write a single event handler for all of them.

This is demonstrated by the DynaMenu example, which also illustrates the use of menu check marks, radio items, and many other features of menus that aren't described in detail in the text. As soon as you start this program, it creates a new pull-down with menu items used to change the font size of a big label hosted by the form. Instead of creating a bunch of menu items with captions indicating sizes ranging from 8 to 48, you can let the program do this repetitive work for you.

The new pull-down menu should be inserted in the `Items` property of the `MainMenu1` component. You can calculate the position by asking the MainMenu component for the previous pull-down menu:

```
procedure TFormColorText.FormCreate(Sender: TObject);
var
  PullDown, Item: TMenuItem;
  Position, I: Integer;
begin
  // create the new pull-down menu
  PullDown := TMenuItem.Create (Self);
  PullDown.AutoHotkeys := maManual;
  PullDown.Caption := '&Size';
  PullDown.OnClick := SizeClick;
  // compute the position and add it
```

```
  Position := MainMenu1.Items.IndexOf (Options1);
  MainMenu1.Items.Insert (Position + 1, PullDown);
  // create menu items for various sizes
  I := 8;
  while I <= 48 do
  begin
    // create the new item
    Item := TMenuItem.Create (Self);
    Item.Caption := IntToStr (I);
    // make it a radio item
    Item.GroupIndex := 1;
    Item.RadioItem := True;
    // handle click and insert
    Item.OnClick := SizeItemClick;
    PullDown.Insert (PullDown.Count, Item);
    I := I + 4;
  end;
  // add extra item at the end
  Item := TMenuItem.Create (Self);
  Item.Caption := 'More...';
  // make it a radio item
  Item.GroupIndex := 1;
  Item.RadioItem := True;
  // handle it by showing the font selection dialog
  Item.OnClick := Font1Click;
  PullDown.Insert (PullDown.Count, Item);
end;
```

As you can see in the code above, the menu items are created in a `while` loop, setting the radio item style and calling the `Insert` method with the number of items as a parameter to add each item at the end of the pull-down. At the end, the program adds one extra item, which is used to set a different size than those listed. The `OnClick` event of this last menu item is handled by the `Font1Click` method (also connected to a specific menu item), which displays the font selection dialog box. You can see the dynamic menu in Figure 5.6.

note Because the program uses the `Caption` of the new items dynamically, we should either disable the `AutoHotkeys` property of the main menu component, or disable this feature for the pull-down menu we are going to add (and thus automatically disable it for the menu items). This is what I've done in the code above by setting the `AutoHotkeys` property of the dynamically created pull-down component to `maManual`. An alternative approach is to let the menu display the automatic captions and then call the new `StripHotkeys` function before converting then caption to a number. There is also a new `GetHotkey` function, which returns the *active* character of the caption.

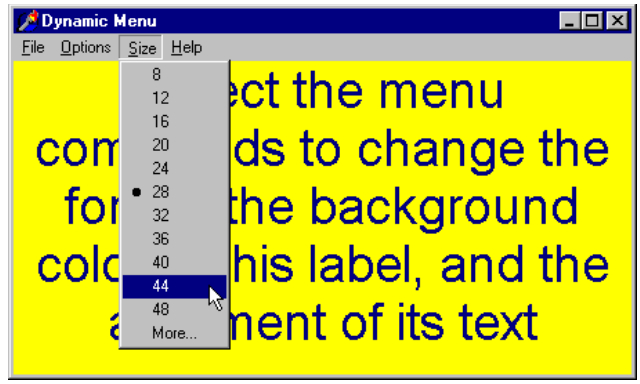


Figure 5.6: The Size pull-down menu of the DynaMenu example is created at run time, along with all of its menu items. Image from the original book.

The handler for the `OnClick` event of these dynamically created menu items uses the caption of the `Sender` menu item to set the size of the font:

```
procedure TFormColorText.SizeItemClick(Sender: TObject);
begin
  with Sender as TMenuItem do
    Label1.Font.Size := StrToInt (Caption);
end;
```

This code doesn't set the proper radio item mark next to the selected item, because the user can select a new size also by changing the font. The proper radio item is checked in the `OnClick` event handler of the entire pull-down menu, which is connected just after the pull-down is created and activated just before showing the pull-down. The code scans the items of the pull-down menu (the `Sender` object) and checks whether the caption matches the current `Size` of the font. If no match is found, the program checks the last menu item, to indicate that a different size is active:

```
procedure TFormColorText.SizeClick (Sender: TObject);
var
  I: Integer;
  Found: Boolean;
begin
  Found := False;
  with Sender as TMenuItem do
  begin
    // look for a match, skipping the last item
    for I := 0 to Count - 2 do
      if StrToInt (Items [I].Caption) =
        Label1.Font.Size then
      begin
        Items [I].Checked := True;
        Found := True;
```

```
        System.Break; // skip the rest of the loop
      end;
    if not Found then
      Items [Count - 1].Checked := True;
  end;
end;
```

When you want to create a menu or a menu item dynamically, you can use the corresponding components, as I've done in the DynaMenu example. As an alternative, you can also use some global functions available in the Menus unit: `NewMenu`, `NewPopupMenu`, `NewSubMenu`, `NewItem`, and `NewLine`.

Using Menu Images

In Delphi it is very easy to improve a program's user interface by adding images to menu items. This is becoming common in Windows applications and it is very nice that Borland has added all the required support, making the development of graphical menu items trivial.

All you have to do is add an image list control to the form, add a series of bitmaps to the image list, connect the image list to the menu using its `Images` property, and set the proper `ImageIndex` property for the menu items. You can see the effect of these simple operations in Figure 5.7. (You can also associate a bitmap with the menu item directly, using the `Bitmap` property.)

note Delphi 5 makes the definition of images for menus more flexible, by allowing you to associate an image list with any specific pull-down menu (and even a specific menu item) using the new `SubMenuImages` property. Having a specific and smaller image list for each pull-down menu, instead of one single huge image list for the entire menu, allows for more run-time customization of an application.

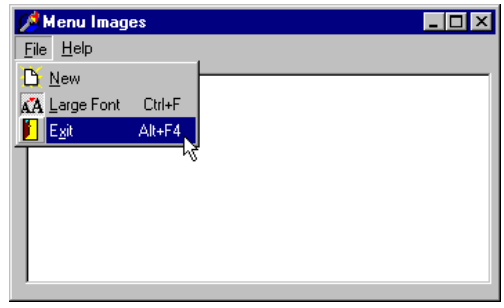
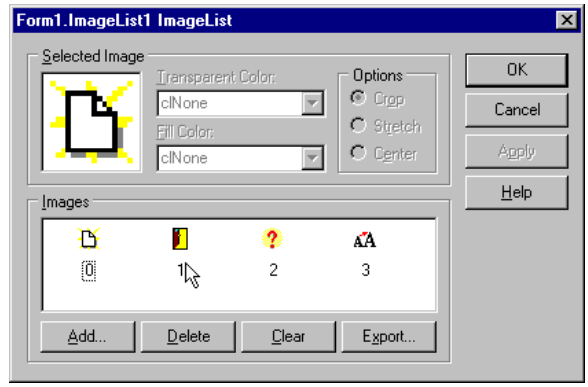


Figure 5.7: The simple graphical menu of the MenuImg example. Image from the original book.

To create the image list you can double-click on the component, activating the corresponding editor (shown in Figure 5.8), and then import existing bitmap or icon files. You can actually prepare a single large bitmap and let the image editor divide it according to the `Height` and `Width` properties of the ImageList component, which refer to the size of the individual bitmaps in the list.

Figure 5.8: The Image List editor, with the bitmaps of the MenuImg example. Image from the original book.



note As an alternative, you can use the series of images that ship with Delphi[159] and are stored by default in the `Program Files/Common Files/Borland Shared/Images/Buttons` directory. Each bitmap contains both an "enabled" and a "disabled" image. As you import them, the Image List editor will ask you whether to split them in two, a suggestion you should accept. This operation adds to the image list a normal image and a disabled one, which is not generally used (as it can be built automatically when needed). For this reason I generally delete the disabled part of the bitmap from the Image List.

The program's code is very simple. The only element I want to emphasize is that if you set the `Checked` property of a menu item with an image instead of displaying a check mark, the item paints its image as sunken. You can see this in the Large Font menu of the MenuImg example in Figure 5.7. Here is the code for that menu item selection:

```
procedure TForm1.LargeFont1Click(Sender: TObject);
begin
  if Memo1.Font.Size = 8 then
    Memo1.Font.Size := 12
  else
```

159 These images are no longer available. The GetIt Package manager offers a nice collection of images, called Icons8 (licensed under Creating Commons), but you can find many others available online.

```
    Memo1.Font.Size := 8;
  // changes the image style near the item
  LargeFont1.Checked := not LargeFont1.Checked;
end;
```

Customizing the System Menu

In some circumstances, it is interesting to add menu commands to the system menu itself, instead of (or besides) having a menu bar. This might be useful for secondary windows, toolboxes, windows requiring a large area on the screen, and "quick-and-dirty" applications. Adding a single menu item to the system menu is straightforward:

```
AppendMenu (GetSystemMenu (Handle, FALSE),
  MF_SEPARATOR, 0, '');
AppendMenu (GetSystemMenu (Handle, FALSE),
  MF_STRING, idSysAbout, '&About...');
```

This code fragment (extracted from the `OnCreate` event handler of the SysMenu example) adds a separator and a new item to the system menu item. The `GetSystemMenu` API function, which requires as a parameter the handle of the form, returns a handle to the system menu. The `AppendMenu` API function is a general-purpose function you can use to add menu items or complete pull-down menus to any menu (the menu bar, the system menu, or an existing pull-down menu). When adding a menu item, you have to specify its text and a numeric identifier. In the example I've defined this identifier as:

```
const
  idSysAbout = 100;
```

Adding a menu item to the system menu is easy, but how can we handle its selection? Selecting a normal menu generates the `wm_Command` Windows message. This is handled internally by Delphi, which activates the `OnClick` event of the corresponding menu item component. The selection of system menu commands, instead, generates a `wm_SysCommand` message, which is passed by Delphi to the default handler. Windows usually needs to do something in response to a system menu command.

We can intercept this command and check to see whether the command identifier (passed in the `CmdType` field of the `TWmSysCommand` parameter) of the menu item is our `idSysAbout`. Since there isn't a corresponding event in Delphi, we have to define a new message-response method for the form class:

```
public
  procedure WMSysCommand (var Msg: TMessage);
    message wm_SysCommand;
```

The code of this procedure is not very complex. We just need to check whether the command is our own and call the default handler:

```
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
begin
  if Msg.CmdType = idSysAbout then
    ShowMessage ('Mastering Delphi: SysMenu example');
  inherited;
end;
```

To build a more complex system menu, instead of adding and handling each menu item as we have just done, we can follow a different approach. Just add a MainMenu component to the form, create its structure (any structure will do), and write the proper event handlers. Then reset the value of the `Menu` property of the form, removing the menu bar.

Now we can add some code to the SysMenu example to add each of the items from the hidden menu to the system menu. This operation takes place when the button of the form is pressed. The corresponding handler uses generic code that doesn't depend on the structure of the menu we are appending to the system menu:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  // add a separator
  AppendMenu (GetSystemMenu (Handle, FALSE), MF_SEPARATOR, 0, '');
  // add the main menu to the system menu
  with MainMenu1 do
    for I := 0 to Items.Count - 1 do
      AppendMenu (GetSystemMenu (Self.Handle, FALSE),
        mf_Popup, Items[I].Handle, PChar (Items[I].Caption));
  // disable the button
  Button1.Enabled := False;
end;
```

note This code uses the expression `Self.Handle` to access the handle of the form. This is required because we are currently working on the `MainMenu1` component, as specified by the `with` statement.[160]

160 This is, in fact, a very good reason to avoid the use of the with statement in the first place. In retrospective, I don't like the fact I was encouraging this and I really don't like this code snippet. I decided to keep it offers me a good opportunity to explain this is not good code.

The menu flag used in this case, `mf_Popup`, indicates that we are adding a pull-down menu. In this function call the fourth parameter is interpreted as the handle of the pull-down menu we are adding (in the previous example we passed the identifier of the menu, instead). Since we are adding to the system menu items with sub-menus, the final structure of the system menu will have two levels, as you can see in Figure 5.9.

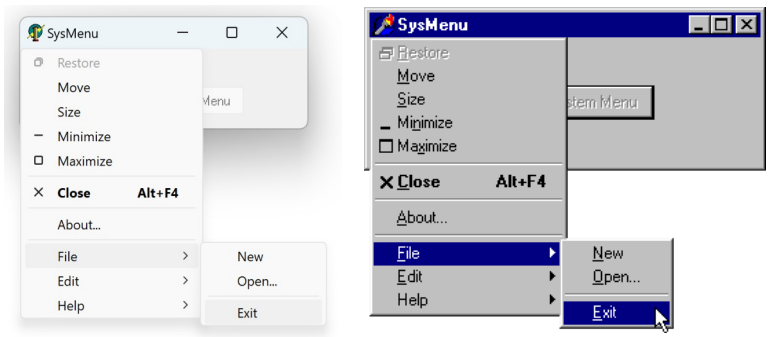


Figure 5.9: The second-level system menu items of the SysMenu example are the result of copying a complete main menu to the system menu. Images captured in Delphi 5 and Delphi 12.

note The Windows API uses the terms *pop-up menu* and *pull-down menu* interchangeably. This is really odd, because most of us use the terms to mean different things. Pop-up menus are shortcut menus, and pull-down menus are the secondary menus of the menu bar. Apparently, Microsoft uses the terms in this way because the two elements are implemented with the same kind of internal windows; and the fact that they are two distinct user-interface elements is probably something that was later conceptually built over a single basic internal structure.

Once you have added the menu items to the system menu, you need to handle them. Of course, you can check for each menu item in the `WMSysCommand` method, or you can try building a smarter approach. Since in Delphi it is easier to write a handler for the `OnClick` event of each item, we can look for the item corresponding to the given identifier in the menu structure. Delphi helps us by providing a `FindItem` method.

When (and if) we have found a main menu item that corresponds to the item selected in the system menu, we can call its `Click` method (which invokes the `OnClick` handler). Here is the code I've added to the `WMSysCommand` method:

```
var
  Item: TMenuItem;
begin
  ...
  Item := MainMenu1.FindItem (Msg.CmdType, fkCommand);
```

```
if Item <> nil then
  Item.Click;
```

In this code, the `CmdType` field of the message structure that is passed to the `WMSysCommand` procedure holds the command of the menu item being called.

note You can also use a simple `if` or `case` statement to handle one of the system menu's predefined menu items that have special codes for this identifier, such as `sc_Close`, `sc_Minimize`, `sc_Maximize`, and so on. For more information, you can see the description of the `wm_SysCommand` message in the Windows API Help file.

This application works but has one glitch. If you click the right mouse button over the Taskbar icon representing the application, you get a plain system menu (actually different from the default one). The reason is that this system menu belongs to a different window, the window of the `Application` global object. I'll discuss the `Application` object, and update this example to make it work with the Taskbar button, in Chapter 6.

The ActionList Component[161]

As explained in the previous chapter, Delphi's event architecture is very open: You can write a single event handler and connect it to the `OnClick` events of a toolbar button and a menu. You can also connect the same event handler to different buttons or menu items, as the event handler can use the `Sender` parameter to refer to the object that fired the event by using the `Sender` parameter. It's a little more difficult to synchronize the status of toolbar buttons and menu items. If you have a menu item and a toolbar button that both toggle the same option, every time the option is toggled, you must both add the check mark to the menu item and change the status of the button to show it pressed.

To overcome this problem, Delphi 4 introduced an event-handling architecture based on actions. An action (or command) both indicates the operation to do when a menu item or button is clicked and determines the status of all the elements connected to the action. The connection of the action with the user interface of the

161 This is a fundamental feature of the VCL architecture, still incredibly modern and still largely unused by Delphi developers. I want to underline the fact this was a great idea and it remains very important today to move form a pure RAD visual development to a much more flexible architecture based on visual design, but separating the UI from the application logic.

linked controls is very important and should not be underestimated, because it is where you can get the real advantages of this architecture.

note If you have ever written code using the MFC class library of Visual C++, you'll recognize that a Delphi action maps to both a command and a `CCommandUpdateUI` object. The Delphi architecture is more flexible, though, because it can be extended by sub-classing the action classes.

There are many players in this event-handling architecture. The central role is certainly played by the action objects. Action objects have a name, like any other component, and they have other properties that will be applied to the linked controls (called action clients). These properties include the `Caption`, the graphical representation (`ImageIndex`), the status (`Checked`, `Enabled`, and `Visible`), and the user feedback (`Hint` and `HelpContext`). The base class for an action object is `TBasicAction`. There is a `TAction` class, but it inherits from `TCustomAction`, which derives from `TContainedAction`, which in turn descends from `TBasicAction`, a `TComponent` subclass.

Each action object is connected to one or more client objects through an ActionLink object. Multiple controls, possibly of different types, can share the same action object, as indicated by their `Action` property. Technically, the ActionLink objects maintain a bidirectional connection between the client object and the action. The ActionLink object is required because the connection works in both directions. An operation on the object (such as a click) is forwarded to the action object and results in a call to its `OnExecute` event; an update to the status of the action object is reflected in the connected client controls. In other words, one or more client controls can create an ActionLink, which registers itself with the action object.

You should not set the properties of the client controls you connect with an action, because the action will override the property values of the client controls. For this reason you should generally write the actions first and then create the menu items and buttons you want to connect with them. Note also that when an action has no `OnExecute` handler, the client control is automatically disabled (or grayed), unless the `DisableIfNoHandler` property is set to `False`.

The client controls connected to actions are usually menu items and various types of buttons (push buttons, check boxes, radio buttons, speed buttons, toolbar buttons, and the like), but nothing prevents you from creating new components that hook into this architecture. Component writers can even define new actions and new link action objects.

Besides a client control, some actions can also have a target component. Some predefined actions hook to a specific target component (for examples, see the coverage of the DataSet components in the Chapter 9 section "Looking for Records in a

Table"). Other actions automatically look for a target component in the form that supports the given action, starting with the active control.

Finally, the action objects are held by an ActionList component, the only class of this architecture that shows up on the Component Palette. The action list receives the execute actions that aren't handled by the specific action objects, firing the `OnExecuteAction`. If even the action list doesn't handle the action, Delphi calls the `OnExecuteAction` event of the `Application` object. The ActionList component has a special editor you can use to create a number of actions, as you can see in Figure 5.10.

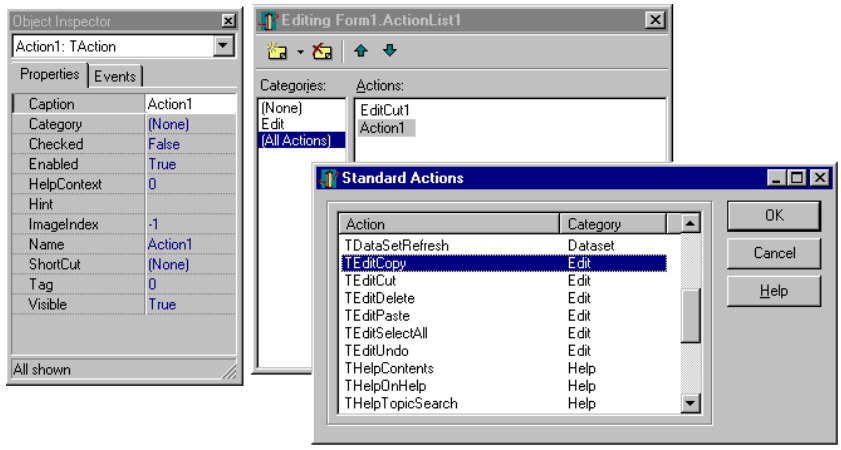


Figure 5.10: The ActionList component editor, with a list of predefined actions you can use. Image from the original book.

In the editor, actions are displayed in different groups, as indicated by their `Category` property. By simply setting this property to a brand-new value, you instruct the editor to introduce a new category. These categories are basically logical groups, although in some cases a group of actions can work only on a specific type of target component. You might want to define a category for every pull-down menu or group them in some other logical way.

With the action list editor, you can create a brand new action or choose one of the existing actions registered in the system. These are listed in a secondary dialog box, as shown in Figure 5.10. There are many predefined actions, which can be divided into logical groups[162]:

162 There are many additional groups added to the predefined list of actions, including DataSnap Client, Dialog, File, Format (for RichEdit operations), Internet, Search, Tab, and Tools. There are almost 70 predefined actions in Delphi 12.

- **Edit actions**, illustrated in the next example. They include Cut, Copy, and Paste actions.
- **MDI window actions**, which will be demonstrated in Chapter 8, as we examine the Multiple Document Interface approach. They include all the most common MDI operations: Arrange, Cascade, Close, Tile, and Minimize all.
- **Dataset actions**, which relate to database tables and queries and will be discussed in Chapter 11. There are many dataset actions, representing all the main operations you can perform on a dataset.
- **Help actions,** which allow you to activate the contents page or index of the Help file attached to the application.

note You can also define new custom actions and register them in Delphi's IDE, as we'll see in Chapter 13.

Besides handling the `OnExecute` event of the action and changing the status of the action to affect the user interface of the client controls, an action can also handle the `OnUpdate` event, which is activated when the application is idle. This gives you the opportunity to check the status of the application or the system and change the user interface of the controls accordingly. For example, the standard PasteEdit action enables the client controls only when there is some text in the Clipboard.

Actions in Practice

Now that you understand the main ideas behind this very important Delphi feature, let's try out an example. The program is called Actions and demonstrates a number of features of the action architecture.

I began building it by placing a new ActionList component in its form and adding the three standard edit actions and a few custom ones. The form also has a panel with some speed buttons, a main menu, and a Memo control (the automatic target of the edit actions). This is the list of the actions, extracted from the DFM file:

```
object ActionList1: TActionList
  Images = ImageList1
  object ActionCopy: TEditCopy
    Category = 'Edit'
    Caption = '&Copy'
    Hint = 'Copy'
    ImageIndex = 1
```

```
    ShortCut = <Ctrl+C>
  end
  object ActionCut: TEditCut
    Category = 'Edit'
    Caption = 'Cu&t'
    Hint = 'Cut'
    ImageIndex = 0
    ShortCut = <Ctrl+X>
  end
  object ActionPaste: TEditPaste
    Category = 'Edit'
    Caption = '&Paste'
    Hint = 'Paste'
    ImageIndex = 2
    ShortCut = <Ctrl+V>
  end
  object ActionNew: TAction
    Category = 'File'
    Caption = '&New'
    Hint = 'New'
    ImageIndex = 3
    ShortCut = <Ctrl+N>
    OnExecute = ActionNewExecute
  end
  object ActionExit: TAction
    Category = 'File'
    Caption = 'E&xit'
    Hint = 'Exit'
    ImageIndex = 5
    ShortCut = <Alt+F4>
    OnExecute = ActionExitExecute
  end
  object NoAction: TAction
    Category = 'Test'
    Caption = '&No Action'
    Hint = 'No Action'
  end
  object ActionCount: TAction
    Category = 'Test'
    Caption = '&Count Chars'
    Hint = 'Count Characters'
    ImageIndex = 6
    OnExecute = ActionCountExecute
    OnUpdate = ActionCountUpdate
  end
  object ActionBold: TAction
    Category = 'Edit'
    Caption = '&Bold'
    Hint = 'Bold'
    ImageIndex = 4
    ShortCut = <Ctrl+B>
    OnExecute = ActionBoldExecute
  end
```

```
  object ActionEnable: TAction
    Category = 'Test'
    Caption = '&Enable NoAction'
    Hint = 'Enable No Action'
    OnExecute = ActionEnableExecute
  end
  object ActionSender: TAction
    Category = 'Test'
    Caption = 'Test &Sender'
    Hint = 'Test Sender'
    OnExecute = ActionSenderExecute
  end
end
```

note The shortcut keys are stored in the DFM files using virtual key numbers, which also include values for the Ctrl and Alt keys. In this and other listings throughout the book I've replaced the numbers with the literal values, enclosing them in angle brackets.

All of these actions are connected with the items of a MainMenu component and some of them also with the buttons of a Toolbar control (more on the Toolbar control in Chapter 7). Notice that the images selected in the ActionList control affect the actions in the editor only, as you can see in Figure 5.11. For the images of the ImageList to show up also in the menu items and in the toolbar buttons, you must also select the image list in the MainMenu and in the Toolbar components.

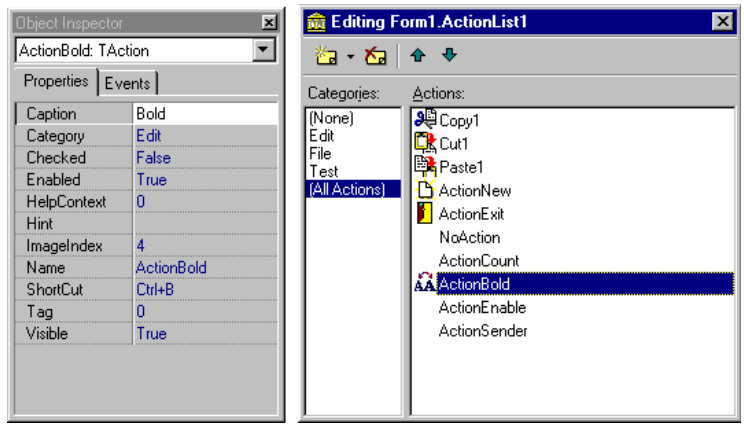


Figure 5.11: The ActionList editor of the Actions example. Image from the original book.

The three predefined actions for the Edit menu don't have associated handlers, but these special objects have internal code to perform the related action on the active edit or memo control. These actions also enable and disable themselves, depending on the content of the Clipboard and on the existence of selected text in the active

edit control. Most other actions have custom code, except for the `NoAction` object. Having no code, the menu item and the button connected with this command are disabled, even if the `Enabled` property of the action is set to `True`.

I've added to the example, and to the Test menu, another action that enables the menu item connected to the `NoAction` object:

```
procedure TForm1.ActionEnableExecute(Sender: TObject);
begin
  NoAction.Enabled := True;
  NoAction.DisableIfNoHandler := False;
  ActionEnable.Enabled := False;
end;
```

Simply setting `Enabled` to `True` will produce the effect for only a very short time, unless you set the `DisableIfNoHandler` property, as discussed in the previous section. Once this operation is done, I disable the current action, since there is no need to issue the same command again.

This is different from an action you can toggle, such as the Edit ➢ Bold menu item and the corresponding speed button. Here is the code of the Bold action:

```
procedure TForm1.ActionBoldExecute(Sender: TObject);
begin
  with Memo1.Font do
    if fsBold in Style then
      Style := Style - [fsBold]
    else
      Style := Style + [fsBold];
  // toggle status
  ActionBold.Checked := not ActionBold.Checked;
end;
```

The `ActionCount` object has very simple code, but it demonstrates an `OnUpdate` handler; when the memo control is empty, it is automatically disabled. We could have obtained the same effect by handling the `OnChange` event of the memo control itself, but in general it might not always be possible or easy to determine the status of a control simply by handling one of its events. Here is the code of the two handlers of this action:

```
procedure TForm1.ActionCountExecute(Sender: TObject);
begin
  ShowMessage ('Characters: ' + IntToStr (
    Length (Memo1.Text)));
end;

procedure TForm1.ActionCountUpdate(Sender: TObject);
begin
  ActionCount.Enabled := Memo1.Text <> '';
```

```
end;
```

Finally, I've added a special action to test the sender object of the action event handler and get some other system information. Besides showing the object class and name, I've added code that accesses the action list object. I've done this mainly to show that you can access this information and how to do it:

```
procedure TForm1.ActionSenderExecute(Sender: TObject);
begin
  Memo1.Lines.Add (
    'Sender class: ' + Sender.ClassName);
  Memo1.Lines.Add (
    'Sender name: ' + (Sender as TComponent).Name);
  Memo1.Lines.Add (
    'Category: ' + (Sender as TAction).Category);
  Memo1.Lines.Add (
    'Action list name: ' + (Sender as TAction).ActionList.Name );
  end;
```

You can see the output of this code in Figure 5.12, along with the user interface of the example. Notice that the `Sender` is not the menu item you've selected, even if the event handler is connected to it. The `Sender` object, which fires the event, is the action, which intercepts the user operation.

Finally, keep in mind that you can also write handlers for the events of the ActionList object itself, which play the role of global handlers for all the actions of the list (something I haven't done in the example).

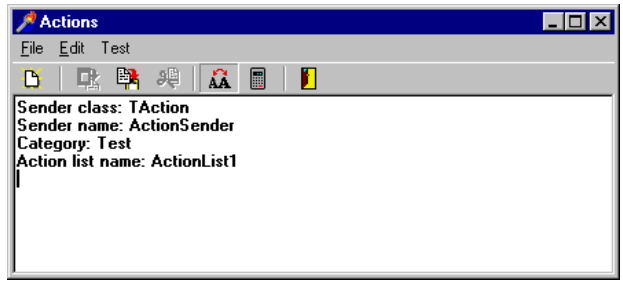


Figure 5.12: The Actions example, with a detailed description of the Sender of an Action object's OnExecute event. Image from the original book.

Owner-Draw Controls

Let's return briefly to menu graphics. Besides using an ImageList to add glyphs to the menu items, you can turn a menu into a completely graphical element, using the owner-draw technique. The same technique also works for other controls, such as list boxes. In Windows, the system is usually responsible for painting buttons, list boxes, edit boxes, menu items, and similar elements. Basically these controls know how to paint themselves. As an alternative, however, the system allows the owner of these controls, generally a form, to paint them. This technique, available for buttons, list boxes, combo boxes, and menu items, is called *owner-draw*.

In Delphi the situation is slightly more complex. The components can take care of painting themselves in this case (as in the `TBitBtn` class for bitmap buttons) and possibly activate corresponding events. Basically, the system sends the request for painting to the owner (usually the form), and the form forwards the event back to the proper control, firing its event handlers.

note Most of the Win32 common controls have support for the owner-draw technique, generally called custom drawing. You can fully customize the appearance of a ListView, a TreeView, a TabControl, a PageControl, a HeaderControl, a StatusBar, and a ToolBar. In Delphi 5 the ToolBar, ListView and TreeView controls also support *advanced* custom drawing, a more fine-tuned drawing capability introduced by Microsoft in the latest versions of the Win32 common controls library. The downside to owner-draw is that when the Windows user interface style changes in the future (and it always does), your owner-draw controls that fit in perfectly with the current user interface styles will look outdated and out of place. Since you are creating a custom user interface, you'll need to keep it updated yourself. By contrast, if you use the standard output of the controls, your applications will automatically adapt to a new version of such controls.

Owner-Draw Menu Items

Delphi makes the development of graphical menu items quite simple compared to the traditional approach of the Windows API. You set the `OwnerDraw` property of a menu item component to `True` and handle its `OnMeasureItem` and `OnDrawItem` events.

In the `OnMeasureItem` event you can determine the size of the menu items. This event handler is activated once for each menu item when the pull-down menu is displayed and has two reference parameters you can set:

```
procedure ColorMeasureItem (Sender: TObject;
```

```
  ACanvas: TCanvas; var Width, Height: Integer);
```

The other parameter, `ACanvas`, is typically used to determine the height of the current font.

In the `OnDrawItem` event you paint the actual image. This event handler is activated every time the item has to be repainted. This happens when Windows first displays the items and each time the status changes; for example, when the mouse moves over an item, it should become highlighted. In fact, to paint the menu items, we have to consider all the possibilities, including drawing the highlighted items with specific colors, drawing the check mark if required, and so on. Luckily enough the Delphi event passes to the handler the `Canvas` where it should paint, the output rectangle, and the status of the item (selected or not):

```
procedure ColorDrawItem(Sender: TObject;
  ACanvas: TCanvas; ARect: TRect; Selected: Boolean);
```

In the ODMenu example I'll handle the highlighted color, but skip other advanced aspects (such as the check marks). I've set the `OwnerDraw` property of the menu and written handlers for some of the menu items. To write a single handler for each event of the three color-related menu items, I've set their `Tag` property to the value of the actual color in the `OnCreate` event handler of the form:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Blue1.Tag := clBlue;
  Red1.Tag := clRed;
  Green1.Tag := clGreen;
end;
```

This makes the handler of the actual `OnClick` event of the items quite straightforward:

```
procedure TForm1.ColorClick(Sender: TObject);
begin
  ShapeDemo.Brush.Color :=
    (Sender as TComponent).Tag
end;
```

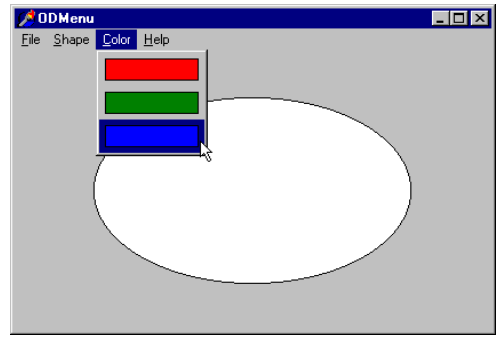
The handler of the `OnMeasureItem` event doesn't depend on the actual items, but uses a fixed value (different from the handler of the other pull-down):

```
procedure TForm1.ColorMeasureItem(Sender: TObject;
  ACanvas: TCanvas; var Width, Height: Integer);
begin
  Width := 80;
  Height := 30;
end;
```

The most important portion of the code is in the handlers of the `OnDrawItem` events. For the color, we use the value of the tag to paint a rectangle of the given color, as you can see in Figure 5.13. Before doing this, however, we have to fill the background of the menu items (the rectangular area passed as a parameter) with the standard color for the menu (`clMenu`) or the selected menu items (`clHighlight`):

```
procedure TForm1.ColorDrawItem(Sender: TObject;
  ACanvas: TCanvas; ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
  ACanvas.FillRect (ARect);
  // show the color
  ACanvas.Brush.Color := (Sender as TComponent).Tag;
  InflateRect (ARect, -5, -5);
  ACanvas.Rectangle (ARect.Left, ARect.Top,
    ARect.Right, ARect.Bottom);
end;
```

Figure 5.13: The owner-draw menu of the ODMenu example. Image from the original book.



The three handlers for this event of the Shape pull-down menu items are all different, although they use similar code:

```
procedure TForm1.Ellipse1DrawItem(Sender: TObject; ACanvas: TCanvas;
  ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
  ACanvas.FillRect (ARect);
  // draw the ellipse
```

```
  ACanvas.Brush.Color := clWhite;
  InflateRect (ARect, -5, -5);
  ACanvas.Ellipse (ARect.Left, ARect.Top,
    ARect.Right, ARect.Bottom);
end;
```

note To accommodate the increasing number of states in the Windows 2000 user interface style, Delphi 5 includes a new `OnAdvancedDrawItem` event for menus.

A ListBox of Colors

As we have just seen for menus, list boxes have an owner-draw capability, which means a program can paint the items of a list box. The same support is provided for combo boxes. To create an owner-draw list box, we set its `Style` property to `lbOwnerDrawFixed` or `lbOwnerDrawVariable`. The first value indicates that we are going to set the height of the items of the list box by specifying the `ItemHeight` property and that this will be the height of each and every item. The second owner-draw style indicates a list box with items of different heights. In this case the component will trigger the `OnMeasureItem` event for each item, to ask the program for their heights.

In the ODList example, I'll stick with the first, simpler, approach. The example stores color information along with the items of the list box and then draws the items in colors (instead of using a single color for the whole list). Here are the properties of the components of the main form of this example:

```
object ODListForm: TODListForm
  Caption = 'Owner-draw Listbox'
  OnCreate = FormCreate
  object ListBox1: TListBox
    Align = alClient
    Font.Charset = ANSI_CHARSET
    Font.Color = clBlack
    Font.Height = -32
    Font.Name = 'Arial'
    Font.Style = [fsBold]
    ItemHeight = 16
    ParentFont = False
    Sorted = True
    Style = lbOwnerDrawFixed
    OnDblClick = ListBox1DblClick
    OnDrawItem = ListBox1DrawItem
  end
  object ColorDialog1: TColorDialog...
```

```
end
```

Notice the value of the `TextHeight` attribute of the form, which indicates the number of pixels required to display text. This is the value we should use for the `ItemHeight` property of the list box. An alternative solution is to compute this value at run time, so that if we later change the font at design time we don't have to remember to set the height of the items accordingly.

note I've just described `TextHeight` as an *attribute* of the form, not a property. And in fact it isn't a property but a local value of the form. If it is not a property, you might ask, how does Delphi save it in the `DFM` file? Well, the answer is that Delphi's streaming mechanism is based on properties plus special *property-clones* created by the `DefineProperties` method. You can refer to the Delphi Help file or to *Delphi Developer's Handbook* for information about this advanced topic.

Since `TextHeight` is *not* a property, although it is listed in the form description, we cannot access it directly. Studying the VCL source code, I found that this value is computed by calling a private method of the form, `GetTextHeight`. Since it is private, we cannot call this function. What we can do is to duplicate its code (which is actually quite simple) in the `FormCreate` method of the form, after selecting the font of the list box:

```
Canvas.Font := ListBox1.Font;
ListBox1.ItemHeight := Canvas.TextHeight('0');
```

The next thing we have to do is add some items to the list box. Since this is a list box of colors, we want to add color names to the `Items` of the list box and the corresponding color values to the `Objects` data storage related to each item of the list. Instead of adding the two values separately, I've written a procedure to add new items to the list:

```
procedure TODListForm.AddColors (Colors: array of TColor);
var
  I: Integer;
begin
  for I := Low (Colors) to High (Colors) do
    ListBox1.Items.AddObject (
      ColorToString (Colors[I]),
      TObject(Colors[I]));
end;
```

This method uses an open-array parameter, an array of an undetermined number of elements of the same type. (See the online tutorial *Essential Pascal* at `www.marcocantu.com` if you are unfamiliar with this language construct.) For each item passed as a parameter, we add the name of the color to the list, and we add its value to the related data, by calling the `AddObject` method. To obtain the string cor-

responding to the color, we call the Delphi `ColorToString` function. This returns a string containing either the corresponding color constant, if any, or the hexadecimal value of the color. The color data is added to the list box after casting its value to the `TObject` data type (a four-byte reference), as required by the `AddObject` method.

note Besides `ColorToString`, which converts a color value into the corresponding string with the identifier or the hexadecimal value, there is also a Delphi function to convert a properly formatted string into a color, `StringToColor`.

In the ODList example this method is called in the `OnCreate` event handler of the form (after previously setting the height of the items):

```
AddColors ([clRed, clBlue, clYellow, clGreen, clFuchsia, clLime,
  clGray, RGB (213, 23, 123), RGB (0, 0, 0),
  clAqua, clNavy, clOlive, clPurple, clTeal]);
```

The code used to draw the items is not particularly complex. We simply retrieve the color associated with the item, set it as the color of the font, and then draw the text:

```
procedure TODListForm.ListBox1DrawItem(Control: TWinControl;
  Index: Integer; Rect: TRect; State: TOwnerDrawState);
begin
  with Control as TListbox do
  begin
    // erase
    Canvas.FillRect(Rect);
    // draw item
    Canvas.Font.Color := TColor (Items.Objects [Index]);
    Canvas.TextOut(Rect.Left, Rect.Top, Listbox1.Items[Index]);
   end;
end;
```

The system already sets the proper background color, so the selected item is displayed properly even without any extra code on our part. You can see an example of the output of this program at startup in Figure 5.14. The example also allows you to add new items, by double-clicking on the list box:

```
procedure TODListForm.ListBox1DblClick(Sender: TObject);
begin
  if ColorDialog1.Execute then
    AddColors ([ColorDialog1.Color]);
end;
```

If you try using this capability, you'll notice that some colors you add are turned into color names (one of the Delphi color constants) while others are converted into hexadecimal numbers.

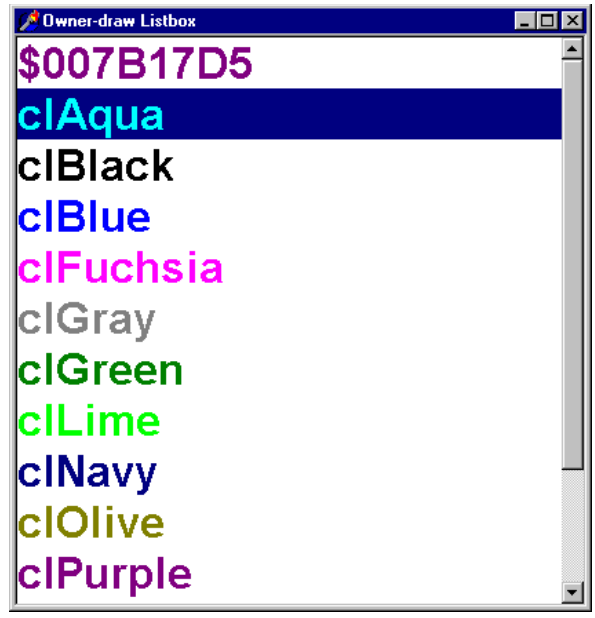


Figure 5.14: The output of the ODList example, with a colored owner-draw list box. Image from the original book.

ListView and TreeView

Although using an owner-draw list box is quite simple, this kind of list box is often replaced by the more powerful ListView and TreeView controls. Again, these two controls are part of the Win32 common controls, stored in the ComCtl32.DLL library.

Microsoft has kept expanding this library over the last two years, adding new controls such as the calendar and the coolbar, all available since Delphi 4, and extending the existing ones. Some of the versions of the library (distributed in particular along with the numerous versions of Microsoft Internet Explorer) have created compatibility problems with the controls, although the situation has apparently become more stable over the last year.

Some of these controls are complex, can be customized in a number of ways, and even support custom drawing features. Here I'll show you a couple of simple examples of the use of the TreeView and ListView components. In Chapters 7 and 8 we'll

use other common controls. In any case, I cannot provide extensive coverage of all of the features of these controls, which would require too much space.

A Graphical Reference List

When you use a ListView component, you can provide bitmaps both indicating the status of the element (for example, the selected item) and describing the contents of the item in a graphical way.

How do we connect the images to a list or tree? We need to refer to the ImageList component we've already used for the images of the menu. A ListView can actually have three image lists, one for the large icons (the `LargeImages` property), one for the small icons (the `SmallImages` property), and one used for the state of the items (the `StateImages` property).

To define the images of the RefList example, however, I used an alternative approach: I created a single big bitmap (16 x 80 pixels for five small images and 32 x 160 pixels for five large images) with all the images inside. Figure 5.15 shows these two bitmaps in the Delphi Image Editor[163]. Then I added the bitmap to a resource file and wrote some code to load it all at once (not one image at a time).

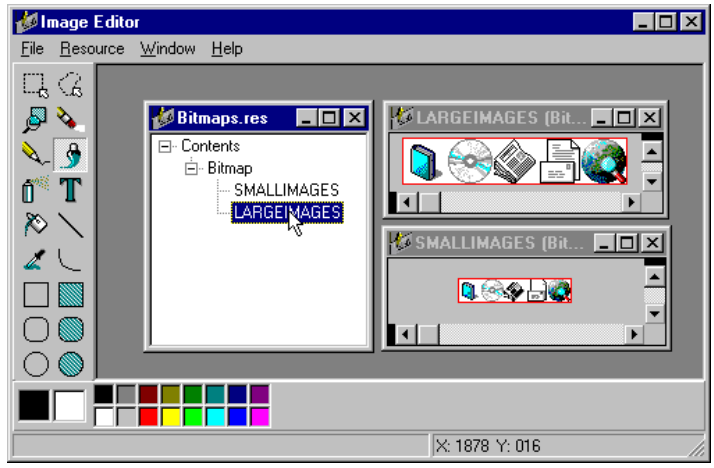


Figure 5.15: All the images of the ListView of the RefList example are stored in two bitmaps. Image from the original book (the Image Editor isn't available with Delphi any more).

I created two ImageList components at run time. As you can see in the parameter of the `Create` constructor, I assigned the form as their owner, so that I don't have to

163 This image editing tool is not available any more.

manually destroy them at the end. Here is the code of the handler for the first part of the form's `OnCreate` event:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  ImageList1, ImageList2: TImageList;
begin
  // load the large images
  ImageList1 := TImageList.Create (self);
  ImageList1.Height := 32;
  ImageList1.Width := 32;
  ImageList1.ResourceLoad (rtBitmap,
    'LargeImages', clWhite);
  ListView1.LargeImages := ImageList1;

  // load the small images
  ImageList2 := TImageList.Create (self);
  ImageList2.ResourceLoad (rtBitmap,
    'SmallImages', clWhite);
  ListView1.SmallImages := ImageList2;
```

Each of the items of the ListView has an `ImageIndex`, which refers to its image in the list. For this to work properly, the elements in the two image lists should follow the same order. When you have a fixed image list, you can add items to it using Delphi's ListView Item Editor, which is connected to the `Items` property. You can see an example of the use of this editor in Figure 5.16. In this editor you can define items and so-called subitems. The subitems are displayed only in the detailed view (when you set the `vsReport` value of the `ViewStyle` property) and are connected with the titles set in the `Columns` property.

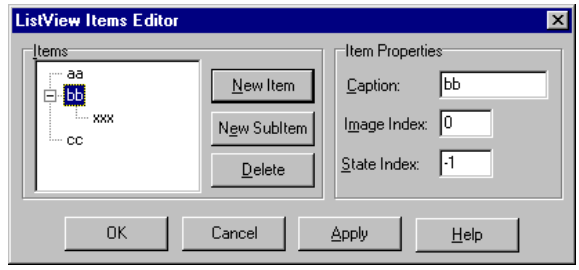


Figure 5.16: The ListView Item Editor. Image from the original book.

In my RefList example (a simple list of references to books, magazines, CD-ROMs, and Web sites) the items are stored to a file, since users of the program can edit the content of the list, which are automatically saved as the program exits. This way, edits made by the user become persistent.

Saving and loading the contents of a ListView is not trivial, since the `TListItems` type doesn't have an automatic mechanism to save the data. As an alternative simple approach, I've copied the data to and from a string list, using a custom format. The string list can then be saved to a file and reloaded with a single command.

The file format is simple, as you can see in the following saving code. For each item of the list, the program saves the caption on one line, the image index on another line (prefixed by the `@` character), and the subitems on the following lines, indented with a tab character:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I, J: Integer;
  List: TStringList;
begin
  // store the items
  List := TStringList.Create;
  try
    for I := 0 to ListView1.Items.Count - 1 do
    begin
      // save the caption
      List.Add (ListView1.Items[I].Caption);
      // save the index
      List.Add ('@' + IntToStr (ListView1.Items[I].ImageIndex));
      // save the subitems (indented)
      for J := 0 to ListView1.Items[I].SubItems.Count - 1 do
        List.Add (#9 + ListView1.Items[I].SubItems [J]);
    end;
    List.SaveToFile (
      ExtractFilePath (Application.ExeName) + 'Items.txt');
  finally
    List.Free;
  end;
end;
```

The items are then reloaded in the second part of the `FormCreate` method:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  List: TStringList;
  NewItem: TListItem;
  I: Integer;
begin
  ...
  // load the items
  ListView1.Items.Clear;
  List := TStringList.Create;
  try
    List.LoadFromFile (
      ExtractFilePath (Application.ExeName) + 'Items.txt');
    for I := 0 to List.Count - 1 do
```

```
      if List [I][1] = #9 then
        NewItem.SubItems.Add (Trim (List [I]))
      else if List [I][1] = '@' then
        NewItem.ImageIndex := StrToIntDef (List [I][2], 0)
      else
      begin
        // a new item
        NewItem := ListView1.Items.Add;
        NewItem.Caption := List [I];
      end;
  finally
    List.Free;
  end;
end;
```

The program has a menu you can use to choose one of the different views supported by the ListView control, and to add check boxes to the items, as in a CheckListBox. You can see some of the various combinations of these styles in Figure 5.17[164].

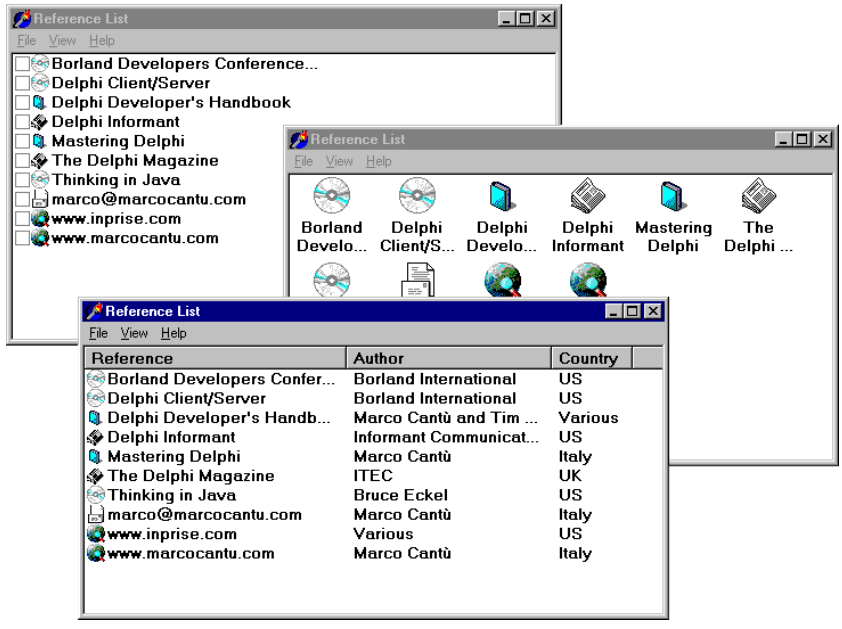


Figure 5.17: Different examples of the output of a ListView component of the RefList program, obtained by changing the ViewStyle property and adding the check boxes. Image from the original book.

Another important feature, which is common in the detailed or report view of the control, is to let a user sort the items on one of the columns. To accomplish this

164 The content of this demo, that is the list of books and magazine, it really a blast form the past. You can notice even the company web site, inprise.com!

requires three operations. The first is to set the `SortType` property of the ListView to `stBoth` or `stData`. In this way, the ListView will operate the sorting not based on the captions, but calling the `OnCompare` event for each two items it has to sort. Since we want to do the sorting on each of the columns of the detailed view, we also handle the `OnColumnClick` event (which takes place when the user clicks on the column titles in the detailed view, but only if the `ShowColumnHeaders` property is set to `True`). Each time a column is clicked, the program saves the number of that column in the `nSortCol` private field of the form class:

```
procedure TForm1.ListView1ColumnClick(Sender: TObject;
  Column: TListColumn);
begin
  nSortCol := Column.Index;
  ListView1.AlphaSort;
end;
```

Then, in the third step, the sorting code uses either the caption or one of the subitems according to the current sort column:

```
procedure TForm1.ListView1Compare(Sender: TObject;
  Item1, Item2: TListItem;
  Data: Integer; var Compare: Integer);
begin
  if nSortCol = 0 then
    Compare := CompareStr (Item1.Caption, Item2.Caption)
  else
    Compare := CompareStr (Item1.SubItems [nSortCol - 1],
       Item2.SubItems [nSortCol - 1]);
end;
```

The final features I've added to the program relate to mouse operations. When the user left-clicks an item, the RefList program shows a description of the selected item. Right-clicking the selected item sets it in edit mode, and a user can change it (keep in mind that the changes will automatically be saved when the program terminates). Here is the code for both operations, in the `OnMouseDown` event handler of the ListView control:

```
procedure TForm1.ListView1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  strDescr: string;
  I: Integer;
begin
  // if there is a selected item
  if ListView1.Selected <> nil then
    if Button = mbLeft then
    begin
      // create and show a description
      strDescr := ListView1.Columns [0].Caption + #9 +
```

```
        ListView1.Selected.Caption + #13;
      for I := 1 to ListView1.Selected.SubItems.Count do
        strDescr := strDescr + ListView1.Columns [I].Caption + #9 +
          ListView1.Selected.SubItems [I-1] + #13;
      ShowMessage (strDescr);
    end
    else if Button = mbRight then
      // edit the caption
      ListView1.Selected.EditCaption;
end;
```

Although it is not feature-complete, this example shows some of the potential of the ListView control. I've also activated the Windows 98 "hot-tracking" feature, which lets the list view highlight and underline the item under the mouse, as Figure 5.18 demonstrates. The relevant properties of the ListView can be seen in its textual description:

```
object ListView1: TListView
  Align = alClient
  Columns = <
    item
      Caption = 'Reference'
      Width = 230
    end
    item
      Caption = 'Author'
      Width = 180
    end
    item
      Caption = 'Country'
      Width = 80
    end>
  Font.Height = -13
  Font.Name = 'MS Sans Serif'
  Font.Style = [fsBold]
  FullDrag = True
  HideSelection = False
  HotTrack = True
  HotTrackStyles = [htHandPoint, htUnderlineHot]
  SortType = stBoth
  ViewStyle = vsList
  OnColumnClick = ListView1ColumnClick
  OnCompare = ListView1Compare
  OnMouseDown = ListView1MouseDown
end
```

This program is actually quite interesting, and I'll further extend it in Chapter 8, adding a dialog box to it.

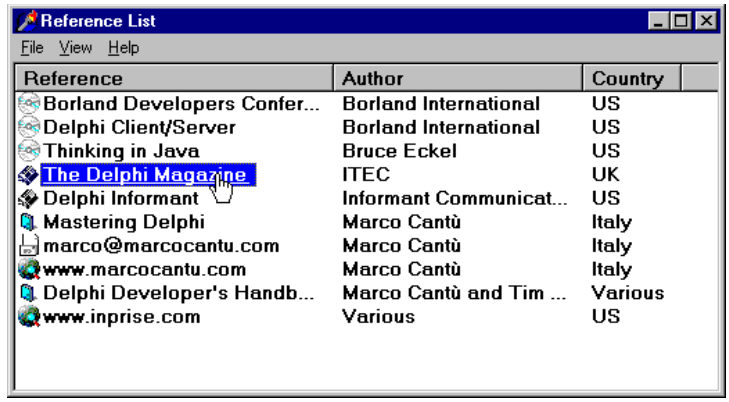


Figure 5.18: The new hot-tracking feature of the ListView control. Notice that the items are sorted by author. Image from the original book.

A Tree of Data

Now that we've seen an example based on the ListView, we can close the chapter by looking at the TreeView control. The TreeView has a user interface that is flexible and powerful (with support for editing and dragging elements). It is also standard, because it is the user interface of the Windows Explorer. There are a number of properties and various ways to customize the bitmap of each line or of each type of line.

To define the structure of the nodes of the TreeView at design time, you can use the TreeView `Items` property editor (see Figure 5.19). In this case, however, I've decided to load it in the TreeView data at startup, in a way similar to the last example.

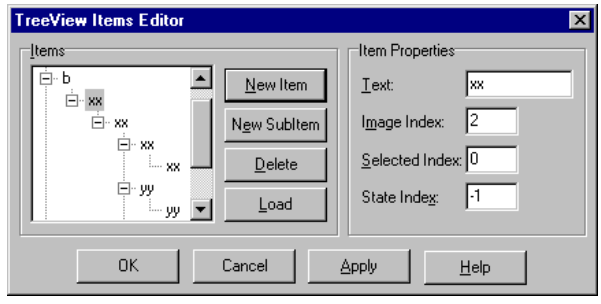


Figure 5.19: The TreeView Items property editor. Image from the original book.

The `Items` property of the TreeView component has many member functions you can use to alter the hierarchy of strings. For example, we can build a two-level tree with the following lines:

```
var
  Node: TTreeNode;
begin
  Node := TreeView1.Items.Add (nil, 'First level');
  TreeView1.Items.AddChild (Node, 'Second level');
```

Using these two methods (`Add` and `AddChild`) we can build a complex structure at run time. But how do we load the information? Again, you can use a StringList at run time, load a text file with the information, and parse it.

However, since the TreeView control has a `LoadFromFile` method, the DragTree example uses the following simpler code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TreeView1.LoadFromFile (
    ExtractFilePath (Application.ExeName) + 'TreeText.txt');
end;
```

The `LoadFromFile` method basically loads the data in a string list and checks the level of each item by looking at the number of tab characters. (If you are curious, see the `TTreeStrings.GetBufStart` method, which you can find in the ComCtrls unit in the VCL source code included in Delphi.) By the way, the data I've prepared for the TreeView is the organizational chart of a multinational company.

Besides loading the data, the program saves it when it terminates, making the changes persistent. It also has a few menu items to customize the font of the TreeView control and change some other simple settings. The specific feature I've implemented in this example is support for dragging items and entire subtrees. I've set the `DragMode` property of the component to `dmAutomatic` and written the event handlers for the `OnDragOver` and `OnDragDrop` events.

In the first of the two handlers, the program makes sure the user is not trying to drag an item over a child item (which would be moved along with the item, leading to an infinite recursion):

```
procedure TForm1.TreeView1DragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
var
  TargetNode, SourceNode: TTreeNode;
begin
  TargetNode := TreeView1.GetNodeAt (X, Y);
  // accept dragging from itself
  if (Source = Sender) and (TargetNode <> nil) then
```

```
  begin
    Accept := True;
    // determines source and target
    SourceNode := TreeView1.Selected;
    // look up the target parent chain
    while (TargetNode.Parent <> nil) and
        (TargetNode <> SourceNode) do
      TargetNode := TargetNode.Parent;
    // if source is found
    if TargetNode = SourceNode then
      // do not allow dragging over a child
      Accept := False;
  end
  else
    Accept := False;
end;
```

The effect of this code is that (except for the particular case we need to disallow) a user can drag an item of the TreeView over another one, as shown in Figure 5.20. Writing the actual code for moving the items is simple, because the TreeView control provides the support for this operation, through the `MoveTo` method of the `TTreeNode` class:

```
procedure TForm1.TreeView1DragDrop(Sender,
  Source: TObject; X, Y: Integer);
var
  TargetNode, SourceNode: TTreeNode;
begin
  TargetNode := TreeView1.GetNodeAt (X, Y);
  if TargetNode <> nil then
  begin
    SourceNode := TreeView1.Selected;
    SourceNode.MoveTo (TargetNode, naAddChildFirst);
    TargetNode.Expand (False);
    TreeView1.Selected := TargetNode;
  end;
end;
```

note Among the Demos shipping with Delphi, there is an interesting one showing a custom-draw TreeView control. The example is in the CustomDraw sub-directory.[165]

165 The CustomDraw example is no longer available as part of the official Delphi demos.

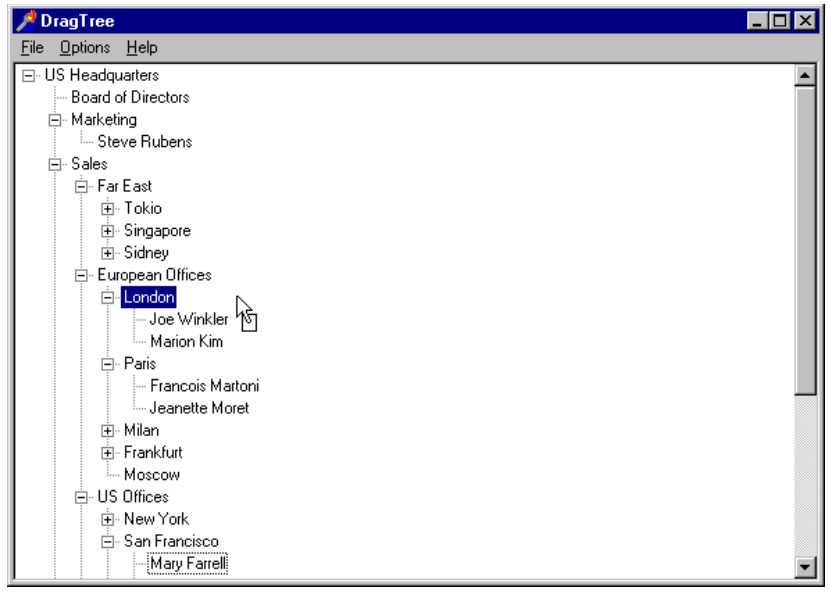


Figure 5.20: The DragTree example during a dragging operation

What's Next?

In this chapter, we have started to explore some of the basic components available in Delphi. These components correspond to the standard Windows controls and some of the Win32 common controls, and they are extremely common in applications. We have also seen how to create main menus and pop-up menus, and we've seen how to add extra graphics to some of these controls.

We also explored a very important and still little-used component, the ActionList, and its architecture for handling menu-item and toolbar-button events. We'll get back to this topic in other examples, and we'll cover the standard MDI and dataset actions in the related chapters.

The next step, however, is to explore in depth one of the most common elements of Delphi programming: forms. We've already used forms many times, but there are still plenty of new features to discuss, including some quite important ones.

Chapter 6: Forms, Windows, And Applications

If you've read the previous chapter, you should now be able to use Delphi's standard components in your applications. So let's turn our attention to the central element of development in Delphi: the form. We have used forms since the initial chapters, but I've never described in detail what you can do with a form, which properties you can use, or which methods of the `TForm` class are particularly interesting.

This chapter looks at some of the properties and styles of forms and at sizing and positioning them. I'll also introduce applications with multiple forms and cover the global VCL objects that handle the interaction among them, `Screen` and `Application`. I'll also devote some time to input on a form, both from the keyboard and the mouse. Let me start this chapter with a general, theoretical discussion of forms and windows.

Forms versus Windows

In Windows, most elements of the user interface are windows. For this reason, in Delphi most components are also based on windows—most of them, but not all. Of course, this is not what a user perceives. The distinction is not obvious, so you should consider the following definitions carefully. Then we can make some further observations.

- From a user standpoint, a window is a portion of the screen surrounded by a border, having a caption and usually a system menu, that can be moved on the screen, closed, and at times also minimized and maximized. Windows can be moved on the screen or inside other windows, as in MDI (Multiple Document Interface) applications. These user windows can be divided into two general categories: main windows and dialog boxes.
- Technically speaking, a window is an entry in an internal memory area of the Windows system, often corresponding to an element visible on the screen, that has some associated code. One of the Windows system libraries contains a list of all the windows that have been built by every application and assigns to each of them a unique number (usually known as a *handle*). Some of these windows are perceived as such by users (see the first definition above), others have the role of controls or visual components, others are temporarily created by the system (for example, to show a pull-down menu), and still others are created by the application but remain hidden from the user.

The common denominator of all windows is that they are known by the Windows system and refer to a function for their behavior; each time something happens in the system, a notification message is sent to the proper window, which responds by executing some code. Each window of the system, in fact, has an associated function (generally called its *window procedure*), which handles the various messages the window is interested in.

In a Delphi application, the system converts these lower-level messages into events. But at times, as we have already seen in some examples, we handle low-level messages directly in a form. Delphi allows us to work at a higher level than the system, making application development much easier.

note The memory area of the Windows system allocated to listing all the windows that have been built is limited. Building too many windows reduces the so-called *system resources*. Windows 3.1 had a severe limit to the number of windows available in the system. In Windows 95 and 98, this limit has been greatly enlarged, and in Windows NT it doesn't even exist. Once there are too many windows in the system (including all the controls and hidden windows), you cannot create even one more window, something that will block most applications. This is why, in Delphi, there are a number of non-windowed components, including labels. This approach lets you save a lot of this system memory without having to worry (or even know) about it. As already mentioned in Chapter 4, graphical non-windowed controls have also other advantages, including faster creation and redraw and less overhead overall[166].

With these general definitions in mind, we can now move back to Delphi and try to understand the role of forms. Forms represent windows from a user standpoint and can be used to build main windows, MDI windows, and dialog boxes. Their behavior is defined mostly by the code written for them but also by a couple of very important properties, `FormStyle` and `BorderStyle`, which we'll explore shortly. Many other components are based on windows, but only forms appear to be windows from a user's point of view. The other windowed components, or controls, can be considered windows only according to the technical definition.

As an example, simply create a new application and place a button in it, save the files in a directory with default names, and run the program. Using the WinSight tool supplied with Delphi[167], you can see the list of the windows of the system; notice in particular the windows created by the application, as shown in Figure 6.1. These include the following windows:

- A main window, the form, with the title *Form1*. It is an overlapped window of class `TForm1`.
- A child window, the button inside the form, with the title *Button1*. This is a child window of class `TButton`.
- A hidden main window, the application window, entitled *Project1*. This is a pop-up window of class `TApplication`.

Notice that the names in brackets in WinSight, which are internal names of the system, correspond to the names of the classes of the Delphi components.

166 If if the Windows handle limit is not relevant today, using graphical controls with no handle still offers advantages, for controls with limited user interaction and not mapped to platform controls.

167 WinSight is not available as part of the product, and I haven't been able to find it online. A newer, similar tools is Spy++ by Microsoft.

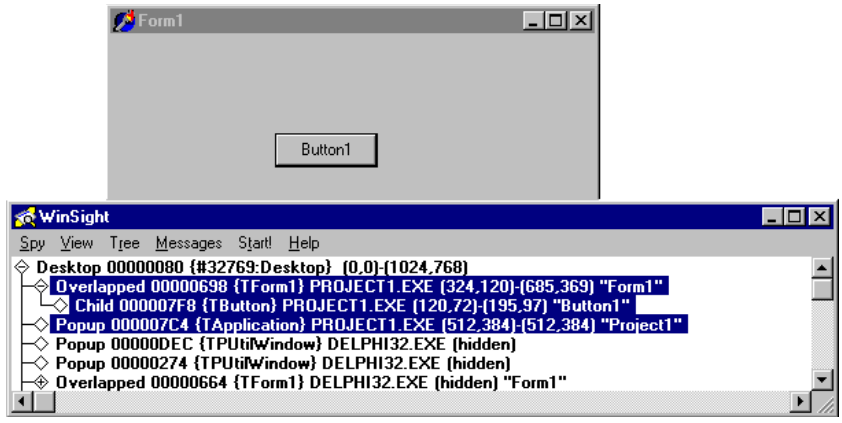


Figure 6.1: The windows of a simple application as they appear in WinSight. Image from the original book.

Overlapped, Pop-Up, and Child Windows

To understand the role of the various windows of this program, we need to look at some technical elements related to the Windows environment. These are not simple concepts, but they are worth knowing about.

1. Each window you create has one of three general styles that determine its behavior. These styles are overlapped, pop-up, and child:
Overlapped windows are main windows of the application, which behave as you would probably expect.

2. *Pop-up* windows are often used for dialog boxes and message boxes and can be considered a holdover from older versions of the system. In fact, in Windows 1, the windows were not overlapped but tiled, and only the pop-up windows could cover other windows. Pop-up windows are generally very similar to overlapped windows.

3. The third group, *child* windows, was originally used for controls inside a dialog box. You can use this style for any window that cannot move outside the client area of the parent window. The obvious extension is to use child windows to build MDI applications; but this behavior is not automatic.

It is important to note that, technically speaking, only child windows can have a parent. Any other window, however, can have an owner. An *owner* is a window that has a continuous message exchange with the windows it owns—for example, when the window is reduced to an icon, when it is activated, and so on. Usually a parent is also the owner, but it forces its child to live inside its client area. The child windows don't use screen coordinates; instead, they use the

client area coordinates of their parent window—to display themselves they borrow pixels not from the screen but from their parent window.

Notice that the Windows API uses the same term (*Parent*) to indicate both the parent and the owner. Even the `GetParent` API function can return both items. Within the system, however, the two handles (that of the parent window and that of the owner window) are stored separately. This is indeed very odd, and it causes a lot of confusion.

In Delphi, all forms are overlapped windows, including dialog boxes, and the form owns all the windowed components (the controls) you place inside it. However, their parent can be either the form or one of the special *container* components, such as the GroupBox or the Panel. When you place a radio button inside a group box, the group box is its parent, but the form is its owner. What about pop-up windows? In Delphi, they are used for the hidden application window, the drop-down list of custom combo boxes, and hint windows. In the system, they are used for message boxes and pop-up or pull-down menus, just to mention two examples.

The `Parent` property of a control indicates what is responsible for displaying it. When you drop a component into a form in the designer, the form will become both parent and owner. When you create the control at run time, you'll need to set the owner (using the Create constructor parameter), but you must also set the `Parent` property, or the control won't be visible.

The Application Is a Window

From the analysis of the WinSight information, you might have noticed that the program has an extra window for the application. Similarly, in the last chapter, we saw that items added to the system menu of the main form were not added to the Taskbar icon, as well. The application window, in fact, is hidden from sight but appears on the Taskbar. This is why Delphi names the window *Form1* and the corresponding Taskbar icon *Project1*[168].

The window related to the `Application` object—the application window—serves to keep together all the windows of an application. The fact that all the top-level forms of a program have this hidden owner window, for example, is fundamental when the

168 There is a a significant change in the VCL pertaining to what its displayed in the task bar. You have the option to use the main form rather than the Application hidden window. This is important today, since the taskbar button offers a form preview and other related features. The default project source code adds by default the line "*Application.MainFormOnTaskbar := True;*" which does what the name implies: It displays the main form rather then the Application handle in the taskbar. When you open an old Delphi application (like those in the original source code of this book), that code is missing and you get the old behavior.

application is activated. In fact, when the windows of your program are behind other windows, clicking on one window in your application will bring all of your application's windows to the front. In other words, the hidden application window is used to connect the different forms of the application. Actually the application window is not hidden, because that would affect its behavior; it simply has zero height and width, and therefore it is not visible.

When you create a new, blank application, Delphi generates a code for the project file, which includes the following[169]:

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

This code uses the global object `Application`, which is of class `TApplication` and is defined by the VCL in the Forms unit. This object is indeed a component, although you cannot set its properties using the Object Inspector. The properties include the name of the executable file (`ExeName`), the `Title` of the application (by default, the name of the executable file without the extension), and the `Icon` displayed in the Taskbar. You can see the application's `Title` in the Windows Taskbar[170]. The same name appears when you scan the running applications with the Alt+Tab keys. To avoid a discrepancy between the two titles, you can change the application's title at design time, in the Application page of the Project Options dialog box. Or at run time, you can copy the form's caption to the title of the application with this code[171]:

```
Application.Title := Form1.Caption;
```

You can also set other properties of the global `Application` object using the same dialog box. To handle the events of the `Application` object, until Delphi 4 you had to write the code manually. Delphi 5, instead, includes the new ApplicationEvents component, specifically intended to handle events of the `Application` object. Beside the easier connection of event handlers at design time, the advantage of using this new component lies in the fact it allows for multiple handlers. If you simply place two instances of the ApplicationEvents component in two different forms, each of them can handle the same event, and both event handlers will be executed. In other words, multiple ApplicationEvents components can chain the handlers.

---

169 As mentioned in the previous note, there is now an extra line, "*Application.MainFormOnTaskbar := True;*".

170 That is, unless you set *MainFormOnTaskbar* to True.

171 This isn't recommended any more, given the better alternative available.

Some of these application-wide events, including `OnActivate`, `OnDeactivate`, `OnMinimize`, and `OnRestore`, allow you to keep track of the status of the application. Other events are forwarded to the application by the controls receiving them, as `OnActionExecute`, `OnActionUpdate`, `OnHelp`, `OnHint`, `OnShortCut`, and `OnShowHint`. Finally, there is the `OnException` global exception handler we've used in Chapter 3, the `OnIdle` event used for background computing and the `OnMessage` event, which fires whenever a message is posted to any of the windows or windowed controls of the application.

In most applications, you don't care about the application window, apart from setting its `Title` and icon and handling some of its events. There are some simple operations you can do anyway. Setting the `ShowMainForm` property to `False` in the project source code indicates that the main form should not be displayed at start-up. Inside a program, instead, you can use the `MainForm` property of the `Application` object to access the main form, which is the first form created in the program.

Displaying the Application Window

There is no better proof that there is indeed a window for the `Application` object than to display it. Actually, we don't need to show it—we just need to resize it and set a couple of window attributes, such as the presence of a caption and a border. We can perform these operations by using Windows API functions on the window indicated by the `Handle` property of the `Application` object:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldStyle: Integer;
begin
  // add border and caption to the app window
  OldStyle := GetWindowLong (
    Application.Handle, gwl_Style);
  SetWindowLong (Application.Handle, gwl_Style,
    OldStyle or ws_ThickFrame or ws_Caption);
  // set the size of the app window
  SetWindowPos (Application.Handle,
    0, 0, 0, 200, 100, swp_NoMove or swp_NoZOrder);
end;
```

The two `GetWindowLong` and `SetWindowLong` API functions are used to access the system information related to the window. In this case, we are using the `gwl_Style` parameter to read or write the styles of the window, which include its border, title, system menu, border icons, and so on. The code above gets the current styles and adds (using an `or` statement) a standard border and a caption to the form. As we'll

see later in this chapter, you seldom need to use these low-level API functions in Delphi, because there are properties of the `TForm` class that have the same effect. We need this code here because the application window is not a form.

Executing this code displays the project window, as you can see in Figure 6.2[172]. Although there's no need to implement something like this in your own programs, running this program will reveal the relationship between the application window and the main window of a Delphi program. This is a very important starting point if you want to understand the internal structure of Delphi applications.

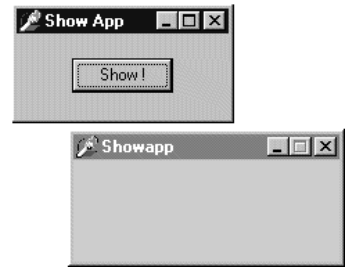


Figure 6.2: The hidden application window revealed by the ShowApp program. Image from the original book.

note In Windows, the minimize and maximize operations are associated by default with system sounds and a visual animated effect. Applications built with Delphi up to version 4 didn't play the sounds or show the visual effect (unless you write some specific code). Delphi 5 applications, instead, produce the sound and display the visual effect by default. Simply recompile your programs and they'll exhibit this extra feature! Technically, the reason this didn't happen in earlier releases is that the main form's minimize and maximize system messages were not being passed to the default window procedure, where Windows implements system sound behavior, was to avoid an unwanted animation effect in the Taskbar. Having found a fix for this problem in Delphi 5, the default behavior has been restored by passing the messages to the operating system.[173]

172 This still happens today with Delphi 12 and Windows 11. If you try it, the application window on screen looks really weird. The point, of course, is just to make you see it exists, it has no role in an actual application.

173 Even if for different reasons, VCL main forms minimize and maximize without using the most common Windows APIs calls for this operations, therefore missing some of the standard effects. I don't think this is a significant issue.

The Application System Menu

Unless you write a very odd program like the example we've just looked at, users will only see the application window in the Taskbar. There, they can activate the window's system menu by right-clicking on it. As I mentioned, when discussing the system menu in the last chapter, an application's menu is not the same as that of the main form. In the SysMenu example in Chapter 5, I added custom items to the system menu of the main form. Now in the SysMenu2 example, I want to customize the system menu of the application window in the Taskbar.

First we have to add the new items to the system menu of the application window when the program starts. Here is the updated code of the `FormCreate` method:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // add a separator and a menu item to the system menu
  AppendMenu (GetSystemMenu (Handle, FALSE),
    MF_SEPARATOR, 0, '');
  AppendMenu (GetSystemMenu (Handle, FALSE),
    MF_STRING, idSysAbout, '&About...');
  // add the same items to the application system menu
  AppendMenu (GetSystemMenu (Application.Handle, FALSE),
    MF_SEPARATOR, 0, '');
  AppendMenu (GetSystemMenu (Application.Handle, FALSE),
    MF_STRING, idSysAbout, '&About...');
end;
```

The first part of the code adds the new separator and item to the system menu of the main form. The other two calls add the same two items to the application's system menu, simply by referring to `Application.Handle`. This is enough to display the updated system menu, as you can see by running this program. The next step is to handle the selection of the new menu item.

To handle form messages, we can simply write new event handlers or message-handling methods. We cannot do the same with the application window, simply because inheriting from the `TApplication` class is quite a complex issue. Most of the time we can simply handle the `OnMessage` event of this class, which is activated for every message the application retrieves from the message queue.

To handle the `OnMessage` event of the global `Application` object, simply add an ApplicationEvents component to the main form, and define a handler for the `OnMessage` event of this component. In this case, we simply need to handle the `wm_SysCommand` message, and we only need to do that if the `wParam` parameter indicates that the user has selected the menu item we've just added, `idSysAbout`:

```
procedure TForm1.ApplicationEvents1Message(var Msg: tagMSG;
```

```
  var Handled: Boolean);
begin
  if (Msg.Message = wm_SysCommand) and
    (Msg.wParam = idSysAbout) then
  begin
    ShowMessage ('Mastering Delphi: SysMenu2 example');
    Handled := True;
  end;
end;
```

This method is very similar to the one used to handle the corresponding system menu item of the main form:

```
procedure WMSysCommand (var Msg: TWMSysCommand);
  message wm_SysCommand;
...
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
begin
  // handle a specific command
  if Msg.CmdType = idSysAbout then
    ShowMessage ('Mastering Delphi: SysMenu2 example');
  inherited;
end;
```

Activating Applications and Forms

To show how the activation of forms and applications works, I've written a simple, self-explanatory example called ActivApp. This example has two forms. Each form has a Label component (`LabelForm`) used to display the status of the form. The program uses text and color for this, as the handlers of the `OnActivate` and `OnDeactivate` events of the first form demonstrate:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  LabelForm.Caption := 'Form2 Active';
  LabelForm.Color := clRed;
end;

procedure TForm1.FormDeactivate(Sender: TObject);
begin
  LabelForm.Caption := 'Form2 Not Active';
  LabelForm.Color := clBtnFace;
end;
```

The second form has a similar label and similar code. The main form also displays the status of the entire application. It uses an ApplicationEvents component to handle the `OnActivate` and `OnDeactivate` events of the Application object. These two

event handlers are similar to the two listed previously, with the only difference being that they modify the text and color of a second label of the form.

If you try running this program, you'll see whether this application is the active one and, if so, which of its forms is the active one. By looking at the output (see Figure 6.3) and listening for the beep, you can understand how each of the activation events is triggered by Delphi. Run this program and play with it for a while to understand how it works. We'll get back to other events related to the activation of forms in a while.

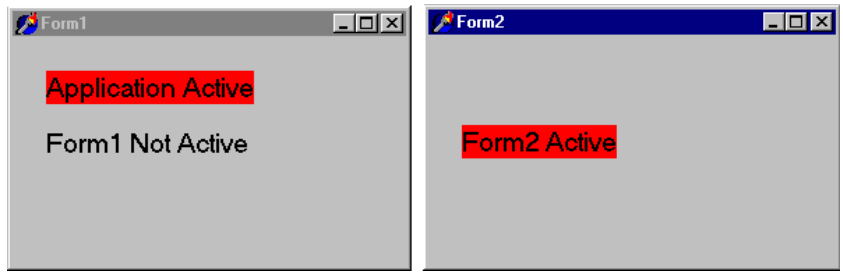


Figure 6.3: The ActivApp example shows whether the application is active and which of the application's forms is active. Image from the original book.

Setting Form and Border Styles

Among the properties of a form, two of them determine the fundamental rules of its behavior: `FormStyle` and `BorderStyle`. The first of these two special properties allows you to choose between a normal SDI (Single Document Interface) and one of the windows that make up an MDI (Multiple Document Interface) application[174].

These are the possible values of the `FormStyle` property:

- `fsNormal`: The form is a normal SDI window or a dialog box.
- `fsMDIChild`: The form is an MDI child window.
- `fsMDIForm`: The form is an MDI parent window—that is, the frame window of the MDI application.

174 MDI has long been deprecated by Microsoft, but some VCL developers still use it. That's why Delphi in version 12 overhauled MDI with quality and features improvements .

- `fsStayOnTop`: The form is an SDI window, but it always remains on top of all other windows except for any that also happen to be *stay-on-top* windows.

Because an application that conforms to the Multiple Document Interface standard needs windows of two different kinds (frame and child), two values of the `FormStyle` property are involved. To build an MDI application, you can use the standard application template or look at Chapter 8, which focuses on the MDI in detail. For now, though, it might be interesting to explore the use of the `fsStayOnTop` style.

To create a topmost form (a form whose window is always on top), you need only set the `FormStyle` property, as indicated above. This property has two different effects, depending on the kind of form you apply it to:

- The main form of an application will remain in front of every other application (unless other applications have the same topmost style, too).
- A secondary form will remain in front of any other form of the application it belongs to. The windows of other applications are not affected, though.

The Border Style

The second property of a form is `BorderStyle`. This property refers to a visual element of the form, but it has a much more profound influence on the behavior of the window, as you can see in Figure 6.4.

At design time, the form is always shown using the default value of the `BorderStyle` property, `bsSizeable`. This corresponds to a Windows style known as *thick frame*. When a main window has a thick frame around it, a user can resize it by dragging its border. This is made clear by the special *resize* cursors (with the shape of a double-pointer arrow) displayed when the user moves the mouse onto this thick window border.[175]

175 The user interface of windows borders has changed a lot in Windows over recent editions. While the technical foundations described here are still the same, the UI and behavior are significantly different.

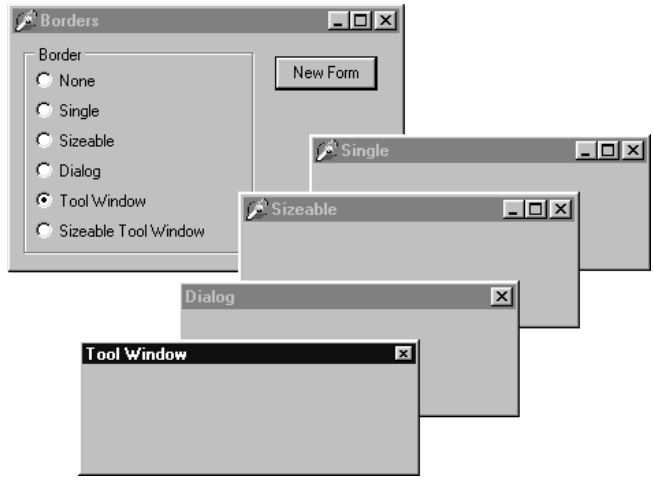


Figure 6.4: Sample forms with the different border styles, created by the Borders example. Image from the original book.

A second important choice for this property is `bsDialog`. If you select it, the form uses as its border the typical dialog-box frame—a thick frame that doesn't allow resizing. In addition to this graphical element, note that if you select the `bsDialog` value, the form becomes a dialog box. This involves a number of changes. For example, the items on its system menu are different, and the form will ignore some of the elements of the `BorderIcons` set property.

note Setting the `BorderStyle` property at design time produces no visible effect. In fact, several component properties do not take effect at design time, because they would prevent you from working on the component while developing the program. For example, how could you resize the form with the mouse if it were turned into a dialog box? When you run the application, though, the form will have the border you requested.

There are four more values we can assign to the `BorderStyle` property. The style `bsSingle` can be used to create a main window that's not resizable. Many games and applications based on windows with controls (such as data-entry forms) use this value, simply because resizing these forms makes no sense. Enlarging a form to see an empty area or reducing its size to make some components less visible often doesn't help a program's user (although Delphi's automatic scroll bars partially solve the last problem). The value `fsNone` is used only in very special situations and inside other forms. You'll never see an application with a main window that has no border or caption (except maybe as an example in a programming book to show you that it makes no sense).

The last two values, `bsToolWindow` and `bsSizeToolWin`, are related to the specific Win32 extended style `ws_ex_ToolWindow`. This style turns the window into a floating toolbox, with a small title font and close button. This style should not be used for the main window of an application.

To test the effect and behavior of the different values of the `BorderStyle` property, I've written a simple program called Borders. You've already seen its output, in Figure 6.4. However, I suggest you run this example and experiment with it for a while to understand all the differences in the forms.

The main form of this program contains only a radio group and a button. There is also a secondary form, with no components and the `Position` property set to `poDefaultPosOnly`. This affects the initial position of the secondary form we'll create by pressing the button. (I'll discuss the `Position` property later in this chapter.)

The code of the program is very simple. When you press the button, a new form is dynamically created, depending on the selected item of the radio group:

```
procedure TForm1.BtnNewFormClick(Sender: TObject);
var
  NewForm: TForm2;
begin
  NewForm := TForm2.Create (Application);
  NewForm.BorderStyle := TFormBorderStyle (
    BorderRadioGroup.ItemIndex);
  NewForm.Caption := BorderRadioGroup.Items[
    BorderRadioGroup.ItemIndex];
  NewForm.Show;
end;
```

This code actually uses a trick: it casts the number of the selected item into the `TFormBorderStyle` enumeration. This works because I've given the radio buttons the same order as the values of this enumeration:

```
type
  TFormBorderStyle = (bsNone, bsSingle, bsSizeable,
    bsDialog, bsToolWindow, bsSizeToolWin);
```

The `BtnNewFormClick` method then copies the text of the radio button to the caption of the secondary form. This program refers to `TForm2`, the secondary form defined in a secondary unit of the program, saved as `SECOND.PAS`. For this reason, to compile the example, you must add the following lines to the `implementation` section of the unit of the main form:

```
uses
  Second;
```

note Whenever you need to refer to another unit of a program, place the corresponding `uses` statement in the `implementation` portion instead of the `interface` portion if possible. This speeds up the compilation process, results in cleaner code (because the units you include are separate from those included by Delphi), and never generates circular references between different units. To accomplish this, you can also use the File ➢ Use Unit menu command.

The Border Icons

Another important element of a form is the presence of icons on its border[176]. By default, a window has a small icon connected to the system menu, a Minimize button, a Maximize button, and a Close button on the far right. You can set different options using the `BorderIcons` property, a set with four possible values: `biSystemMenu`, `biMinimize`, `biMaximize`, and `biHelp`.

note The `biHelp` border icon enables the "What's this?" Help. When this style is included and the `biMinimize` and `biMaximize` styles are excluded, a question mark appears in the form's title bar. If you click on this question mark and then click on a component inside the form (but not on the form itself!), Delphi activates the Help about that object inside a pop-up window. This is demonstrated by the BIcons example, which has a simple Help file with a page connected to the `HelpContext` property of the button in the middle of the form.

The BIcons example demonstrates the behavior of a form with different border icons and shows how to change this property at run time. The form of this example is very simple: it has only a menu, with a pull-down containing four menu items, one for each of the possible elements of the set of border icons. I've written a single method, connected with the four commands, that reads the check marks on the menu items to determine the value of the `BorderIcons` property. This code is therefore also a good exercise in working with sets:

```
procedure TForm1.SetIcons(Sender: TObject);
var
  BorIco: TBorderIcons;
begin
  (Sender as TMenuItem).Checked :=
    not (Sender as TMenuItem).Checked;
  if SystemMenu1.Checked then
    BorIco := [biSystemMenu]
```

176 The ability of customizing the border has been largely expanded over time. The VCL library includes a TitleBarPanel component offering complete control on the title bar, as you can place other controls on it (effectively displaying the in the title bar). This is the component the Delphi IDE also uses to host buttons, a combo boxe, and a search box it its title bar.

```
  else
    BorIco := [];
  if MaximizeBox1.Checked then
    Include (BorIco, biMaximize);
  if MinimizeBox1.Checked then
    Include (BorIco, biMinimize);
  if Help1.Checked then
    Include (BorIco, biHelp);
  BorderIcons := BorIco;
end;
```

While running the BIcons example, you can easily set and remove the various visual elements of the form's border. You'll immediately see that some of these elements are closely related: if you remove the system menu, all of the border icons will disappear; if you remove either the Minimize or the Maximize button, it will be grayed; if you remove both these buttons, they will disappear. Notice also that in these last two cases, the corresponding items of the system menu are automatically disabled. This is the standard behavior for any Windows application. When the Maximize and Minimize buttons have been disabled, you can activate the Help button. As a shortcut to obtain this effect, you can press the button inside the form. Also, you can click on the button after pressing the Help Menu icon to see a Help message, as you can see in Figure 6.5.

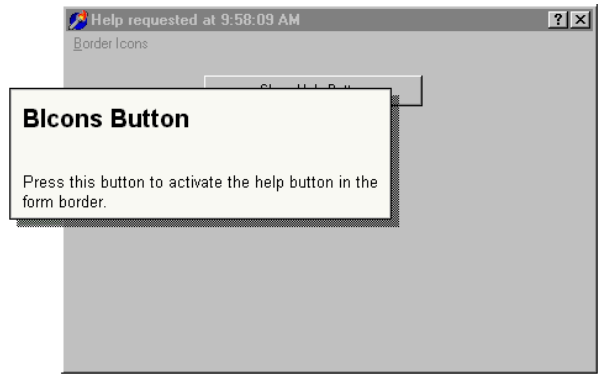


Figure 6.5: The Help button displayed by the BIcons example. By dragging the Help cursor over the button you get the Help displayed in the figure. Image from the original book.

As an extra feature, the program also displays the time that the Help was invoked in the caption, by handling the `OnHelp` event of the form. This effect is visible in the figure.

Setting More Window Styles

The border style and border icons are indicated by two different Delphi properties, which can be used to set the initial value of the corresponding user interface elements. We have seen that besides changing the user interface, these properties affect the behavior of a window. It is important to know that these border-related properties and the `FormStyle` property mainly correspond to different settings in the *style* and *extended style* of a window. These two terms reflect two parameters of the `CreateWindowEx` API function Delphi uses to create forms.

It is important to acknowledge this, because Delphi allows you to modify these two parameters freely by overriding the `CreateParams` virtual method:

```
public
  procedure CreateParams (
    var Params: TCreateParams); override;
```

This is the only way to use some of the peculiar window styles that are not directly available through form properties. For a list of window styles and extended styles, see the API Help under the topics *CreateWindow* and *CreateWindowEx*. You'll notice that the Win32 API has a number of styles for these functions, including those related to tool windows.

To show how to use this approach, I've written the NoTitle example, which lets you create a program with a custom caption. First we have to remove the standard caption but keep the resizing frame by setting the corresponding styles:

```
procedure TForm1.CreateParams (var Params: TCreateParams);
begin
  inherited CreateParams (Params);
  Params.Style := (Params.Style or ws_Popup) and
    not ws_Caption;
end;
```

note Besides changing the style and other features of a window when it is created, you can change them at run time, although some of the settings do not take effect. To change most of the creation parameters at run time, you can use the `SetWindowLong` API function, which allows you to change the internal information of a window. The companion `GetWindowLong` function can be used to read the current status. Two more functions, `GetClassLong` and `SetClassLong`, can be used to read and modify class styles (the information of the `WindowClass` structure of `TCreateParams`). You'll seldom need to use these low-level Windows API functions in Delphi, unless you write advanced components.

To remove the caption, we need to change the overlapped style to a pop-up style; otherwise, the caption will simply stick. Now how do we add a custom caption? I've

placed a label aligned to the upper border of the form and a small button on the far end. You can see this effect at run time in Figure 6.6.

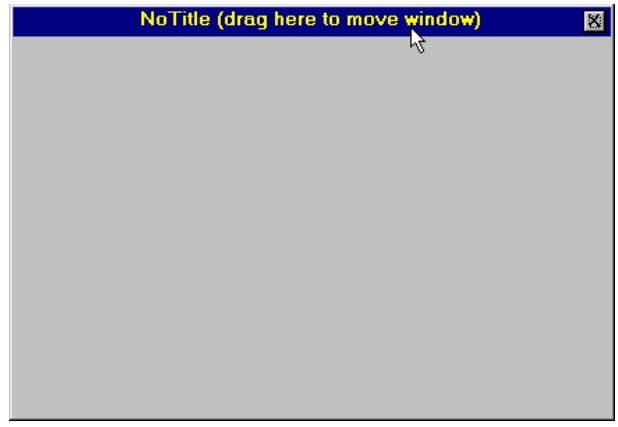


Figure 6.6: The NoTitle example has no real caption but a fake one made with a label. Image from the original book.

To make the fake caption work, we have to tell the system that a mouse operation on this area corresponds to a mouse operation on the caption. This can be done by intercepting the `wm_NCHitTest` Windows message, which is frequently sent to Windows to determine where the mouse currently is. When the hit is in the client area and on the label, we can pretend the mouse is on the caption by setting the proper result:

```
procedure TForm1.HitTest (var Msg: TWmNCHitTest);
  // message wm_NcHitTest
begin
  inherited;
  if (Msg.Result = htClient) and (Msg.YPos <
      Label1.Height + Top + GetSystemMetrics (sm_cyFrame)) then
    Msg.Result := htCaption;
end;
```

The `GetSystemMetrics` API function used in the listing above is used to query the operating system about the size of the various visual elements[177]. It is important to make this request every time (and not cache the result) because users can customize most of these elements by using the Appearance tab of the Desktop options (in Control Panel) and other Windows settings. The small button, instead, has a call to the `Close` method in its `OnClick` event handler. The button is kept in its position even when the window is resized by using the `[akTop,akRight]` value for the `Anchors`

177 These days the VCL intercepts the *GetSystemMetrics* API to make is DPI aware.

property. The form also has size constraints, so that a user cannot make it too small, as described in the "Form Constraints" section later in this chapter.

Scaling Forms

When you create a form with a number of components, it is common to make the form nonresizable to avoid having some of the components fall outside the visible portions of the form. This is not a big problem, because Delphi automatically adds scroll bars to the form so you can reach every control easily. (Form scrolling is one of the subjects of Chapter 7.)

Be aware of this problem when you create a big form: if you build a form on a high-resolution screen, it might be bigger than the available screen size on your end-user's systems. This is a pity, and it is more common that you might expect. If you can, never build a form larger than 640 x 480 pixels[178].

If you have to build a bigger form and using scroll bars is not a solution, Delphi has some nice scaling features. There are two basic techniques:

- The form's `ScaleBy` method allows you to scale the form and each of its components. You can use this method at startup, after you've determined the screen resolution, or in response to a specific request by the user.
- The `PixelsPerInch` and `Scaled` properties allow Delphi to resize an application automatically when the application is run with a different system font size, often because of a different screen resolution[179]. Of course, you can change the values of these properties manually, as described in the next section, and let the system scale the form only when you want.

---

note Form scaling is calculated based on the difference between the font height at run time and the font height at design time. Scaling ensures that edit and other controls are large enough to display their text using the user's font preferences without clipping the text. The form scales as well, as we will see later on, but the main point is to make edit and other controls readable.

---

178 This size makes little sense with today's screens. Also this entire sections ignores the issues with HiDPI Windows configurations, system scaling, and all of the options that have been dded to the operating system and the VCL to handle these scenarios. I won't call out these differences for each of the references in this section that are subject to these changes, or I could add a footnote for almost each line!

179 This is mostly true for HiDPI configurations.

In both cases, to make the form scale its window, be sure to also set the `AutoScroll` property to `False`. Otherwise, the contents of the form will be scaled, but the form border itself will not.

Manual Form Scaling

Any time you want to scale a form, including its components, you can use the `ScaleBy` method, which has two integer parameters, a multiplier and a divisor—it's a fraction. You can apply the same method to a single component. For example, with this statement

```
ScaleBy (3, 4);
```

the size of the current form is divided by 4 and multiplied by 3; that is, the form is reduced to three-quarters of its original size. Generally, it is easier to use percentage values. The same statement can be written this way:

```
ScaleBy (75, 100);
```

When you scale a form, all the proportions are maintained, but if you go below or above certain limits, the text strings can alter their proportions slightly. If you reduce the size of a form too much, most of the components will become unusable or even disappear completely. The problem is that in Windows, components can be placed and sized only in whole pixels, while scaling almost always involves multiplying by fractional numbers. So any fractional portion of a component's origin or size will be truncated.

To avoid similar problems, you should let the user perform only a limited number of scaling operations or re-create the form from scratch before each new scaling so that round-off errors do not accumulate.

---

note If you apply the `ScaleBy` method to a form, the form won't actually be scaled. Only the components inside the form will change their size. As I mentioned before, to overcome this problem, you should disable the form's `AutoScroll` property. What is the relationship between scaling and scrolling? My guess is that if scrolling is enabled, the component can be moved outside the form's visible area without many problems; otherwise, the form is resized, too.

I've built a simple example, Scale, to show how you can scale a form manually, responding to a request by the user. The form of this application (see Figure 6.7) has two buttons, a label, an edit box, and an UpDown control.

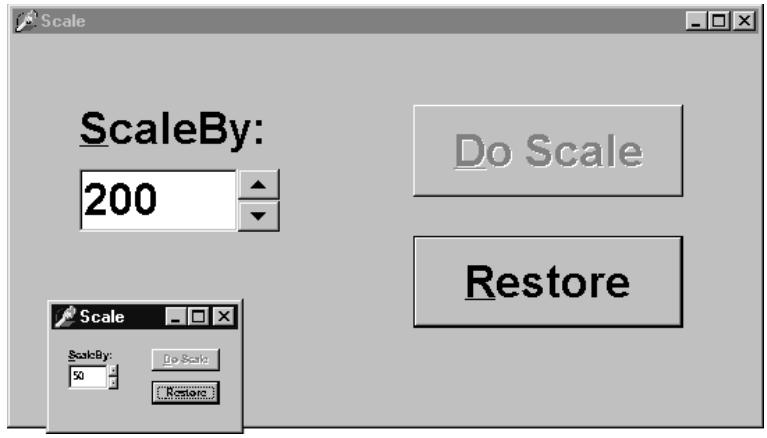


Figure 6.7: The form of the Scale example after a scaling with 50 and 200. Image from the original book.

The UpDown component connects to the edit box, using its `Associate` property. With this setting, a user can type numbers in the box or click on the two small arrows to increase or decrease the number in the edit box by a fixed amount (indicated by the `Increment` property of the UpDown component). To extract the input value, you can use the `Text` property of the edit box or the `Position` of the UpDown control. You can also prevent input errors by setting the Min and Max properties of the UpDown, as I've done in this example:

```
object UpDown1: TUpDown
  Associate = Edit1
  Min = 30
  Max = 300
  Increment = 10
  Position = 100
  Wrap = False
end
```

When you press the ScaleButton button, the current input value is used to determine the scaling percentage of the form:

```
procedure TForm1.ScaleButtonClick(Sender: TObject);
begin
  AmountScaled := UpDown1.Position;
  ScaleBy (AmountScaled, 100);
  UpDown1.Height := Edit1.Height;
  ScaleButton.Enabled := False;
  RestoreButton.Enabled := True;
end;
```

This method stores the current input value in the form's `AmountScaled` private field and enables the Restore button, disabling the one that was pressed. Later, when the

user presses the Restore button, the opposite scaling takes place, by calling `ScaleBy(100, AmountScaled)`. In both cases, I've added a line of code to set the `Height` of the UpDown component to the same `Height` as the edit box it is attached to. This prevents small differences between the two.

note If you want to scale the text of the form properly, including the captions of components, the items in list boxes, and so on, you should use TrueType fonts exclusively. The system font (MS Sans Serif) doesn't scale well. The font issue is important because the size of many components depends on the text height of their captions, and if the caption does not scale well, the component might not work properly. For this reason, in the Scale example I've used an Arial font.[180]

Automatic Form Scaling

Instead of playing with the `ScaleBy` method, you can ask Delphi to do the work for you. When Delphi starts, it asks the system for the display configuration and stores the value in the `PixelsPerInch` property of the `Screen` object, a special global object of the VCL, available in any application.

`PixelsPerInch` sounds like it has something to do with the pixel resolution of the screen, but unfortunately, it doesn't. If you change your screen resolution from 640 x 480 to 800 x 600 to 1024 x 768 or even 1600 x 1280[181], you will find that Windows reports the same `PixelsPerInch` value in all cases, unless you change the system font. What `PixelsPerInch` really refers to is the screen pixel resolution that the currently installed system font was designed for. When the end user changes the system font scale, usually to make menus and other text easier to read, the user will expect all applications to honor those settings. An application that does not reflect the user desktop preferences will look out of place and, in extreme cases, may be unusable to visually impaired users who rely on very large fonts and high-contrast color schemes.

The most common `PixelPerInch` values are 96 (small fonts) and 120 (large fonts), but other values are possible. Windows 98 even allows the user to set the system font size to an arbitrary scale[182]. At design time, the `PixelsPerInch` value of the screen, which is a read-only property, is copied to every form of the application.

180 I'd say this is now hardly the case any more, as the Windows OS had many improvements in this area.

181 Which is still very small compared to a 4K monitor... again, some of the core concepts still apply, but a lot has changed in Windows and in the VCL in this area.

182 This has now become a very commonly used feature, with many user setting their display at 150% or 200% scalcing.

Delphi then uses the value of `PixelsPerInch`, if the `Scaled` property is set to `True`, to resize the form when the application starts.

As I've already mentioned, both automatic scaling and the scaling performed by the `ScaleBy` method operate on components by changing the size of the font. The size of each control, in fact, depends on the font it uses. With automatic scaling, the value of the form's `PixelsPerInch` property (the design-time value) is compared to the current system value (indicated by the corresponding property of the `Screen` object), and the result is used to change the font of the components on the form. Actually, to improve the accuracy of this code, the final height of the text is compared to the design-time height of the text, and its size is adjusted if they do not match.

Thanks to Delphi automatic support, the same application running on a system with a different system font size automatically scales itself, without any specific code. The application's edit controls will be the correct size to display their text in the user's preferred font size, and the form will be the correct size to contain those controls. Although automatic scaling has problems in some special cases, if you comply with the following rules, you should get good results[183]:

- Set the `Scaled` property of forms to `True`. (This is the default.)
- Use only TrueType fonts.
- Use Windows small fonts (96dpi) on the computer you use to develop the forms.
- Set the `AutoScroll` property to `False`, if you want to scale the form and not just the controls inside it. (AutoScroll defaults to True, so don't forget to do this step.)
- Set the form position either near the upper-left corner or in the center of the screen (with the `poScreenCenter` value) to avoid having an out-of-screen form. Form position is discussed in the next section.

Setting the Form's Position and Size

In addition to `PixelsPerInch`, there are more run-time properties you can set to control the appearance of a form. The `Position` property indicates the initial position of the form on the screen when it is first created. The default `poDesigned` value

183 Add to this using PerMonivotrv2 configuration as a must have, along with enabling themes. There is a lot more to be said, but covering modern HiDPI and proper forms and controls scaling and positioning will require an entire chapter, not a footnote.

indicates that the form will appear where you designed it and use the positional and size properties of the form. Some of its other choices (`poDefault`, `poDefaultPosOnly`, and `poDefaultSizeOnly`) depend on a feature of the system: using a specific flag, Windows can position and/or size new windows using a cascade layout. Finally, with the `poScreenCenter` value, the form is displayed in the center of the screen, with the size you set at design time.

note The default positions are ignored when the form has a dialog border style.

The second parameter that affects the initial size and position of a window is its *state*. You can use the `WindowState` property at design time to display a maximized or minimized window at startup. This property, in fact, can have only three values: `wsNormal`, `wsMinimized`, and `wsMaximized`. The meaning of this property is intuitive. If you set a minimized window state, it will be properly displayed in the Windows Taskbar.

Of course, you can maximize or minimize a window at run time, too. Simply changing the value of the `WindowState` property to `wsMaximized` or to `wsNormal` produces the expected effect. Setting the property to `wsMinimized`, however, creates a minimized window that is placed over the Taskbar, not within it. This is not the expected action for a main form, but that for a secondary form! The simple solution to this problem is to call the `Minimize` method of the `Application` object. There is also a `Restore` method in the `TApplication` class that you can use when you need to restore a form, although most often the user will do this using the Restore command of the system menu.

The Size of a Form and Its Client Area

At design time, there are two ways to set the size of a form: by setting the value of the `Width` and `Height` properties or by dragging its borders. At run time, if the form has a resizable border, the user can resize it (producing the `OnResize` event).

However, if you look at a form's properties in source code or in the online Help, you can see that there are two properties referring to its width and two referring to its height. `Height` and `Width` refer to the size of the form, including the borders; `ClientHeight` and `ClientWidth` refer to the size of the internal area of the form, excluding the borders, the caption, scroll bars (if any), and the menu bar. The client area of the form is the surface you can use to place components on the form, to create output, and to receive user input.

Since you might be interested in having a certain available area, at times it makes sense to set the client size of a form instead of its global size. This is straightforward, because as you set one of the two client properties, the corresponding form property changes accordingly. When you modify the value of `ClientHeight`, the value of `Height` immediately changes.

note In Windows, it is also possible to create output and receive input from the nonclient area of the form—that is, its border. Painting on the border and getting input when you click on it are complex issues. If you are interested, look in the Help file at the description of such Windows messages as `wm_NCPaint`, `wm_NCCalcSize`, and `wm_NCHitTest` and the series of nonclient messages related to the mouse input, such as `wm_NCLButtonDown`. The difficulty of this approach is in combining your code with the default Windows behavior. However, Delphi lets you process these low-level Windows messages without any problem, something that most visual programming environments do not allow at all.

Form Constraints

When you choose a re-sizable border for a form, users can generally resize the form as they like and also maximize it to full screen. Windows informs you that the form's size has changed with the `wm_Size` message, which generates the `OnResize` event. `OnResize` takes place after the size of the form has already been changed. Modifying the size again in this event (if the user has reduced or enlarged the form too much) would be silly. A preventive approach is better suited to this problem.

Delphi provides a specific property for forms and also for all controls: the `Constraints` property. Simply setting the sub-properties of the `Constraints` property to the proper maximum and minimum values creates a form that cannot be resized beyond those limits. Here is an example:

```
object Form1: TForm1
  Width = 242
  Height = 162
  Constraints.MaxHeight = 300
  Constraints.MaxWidth = 300
  Constraints.MinHeight = 150
  Constraints.MinWidth = 150
end
```

Notice that as you set up the `Constraints` property, it has an immediate effect even at design time, changing the size of the form if it is outside the permitted area.

Delphi also uses the maximum constraints for maximized windows, producing an awkward effect. For this reason, you should generally disable the Maximize button

of a window that has a maximum size. There are cases in which maximized windows with a limited size make sense—this is the behavior of Delphi's main window.

note The `Constraints` property plays an even more important role for controls and for their docking operations, as we'll see in Chapter 7.

In case you need to change constraints at run time, you can also consider using two specific events, `OnCanResize` and `OnContrainedResize`. The first of the two can also be used to disable resizing a form or control in given circumstances.

Creating Forms

Up to now we have ignored the issue of form creation. We know that when the form is created, we receive the `OnCreate` event and can change or test some of the initial form's properties or fields. The statement responsible for creating the form is in this project's source file (or `DPR` file, available through the Project menu command):

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

To skip the automatic form creation, you can either modify this code or use the Forms page of the Project Options dialog box (see Figure 6.8). In this dialog box, you can decide whether the form should be automatically created. If you disable the automatic creation, the project's initialization code becomes the following:

```
begin
  Applications.Initialize;
  Application.Run;
end.
```

If you now run this program, nothing happens. It terminates immediately because no main window is created. So what is the effect of the call to the application's `CreateForm` method? It creates a new instance of the form class passed as the first parameter and assigns it to the variable passed as the second parameter.

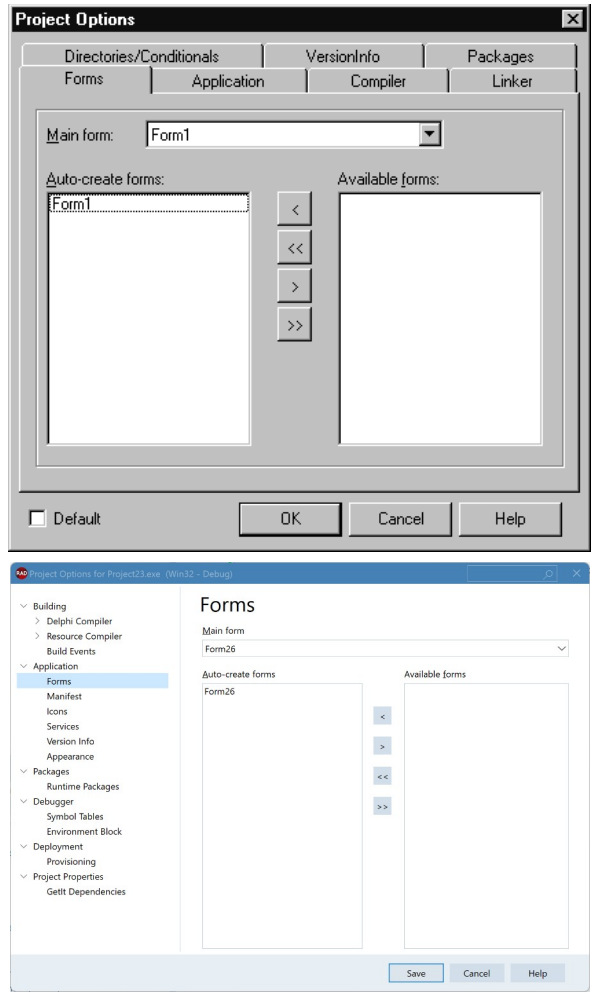


Figure 6.8: The Forms page of the Delphi Project Options dialog box. Images captured in Delphi 5 and Delphi 12.

Something else happens behind the scenes. When `CreateForm` is called, if there is currently no main form, the current form is assigned to the application's `MainForm` property. For this reason, the form indicated as *Main form* in the dialog box shown in Figure 6.8 corresponds to the first call to the application's `CreateForm` method (that is, when several forms are created at start-up).

The same holds for closing the application. Closing the main form terminates the application, regardless of the other forms. If you want to perform this operation from the program's code, simply call the `Close` method of the main form, as we've done several times in past examples.

note In Delphi 5, you can (finally) control the automatic creation of secondary forms by using the Auto Create Forms checkbox on the Preferences page of the Environment Options dialog box[184].

Delphi Form Creation Order

Regardless of the manual or automatic creation of forms, when a form is created, there are many events you can intercept. Form-creation events are fired in the following order:

1. `OnCreate` indicates that the form is being created.
2. `OnShow` indicates that the form is being displayed. Besides main forms, this event happens after you set the `Visible` property of the form to `True` or call the `Show` or `ShowModal` methods. This event is fired again if the form is hidden and then displayed again.
3. `OnActivate` indicates that the form becomes the active form within the application. This event is fired every time you move from another form of the application to the current one, as we saw in the section "Activating Applications and Forms."
4. Other events, including `OnResize` and `OnPaint`, indicate operations always done at start-up but then repeated many times.

As you can see in the list above, every event has a specific role apart from form initialization, except for the `OnCreate` event, which is guaranteed to be called only once as the form is created.

However, there is an alternative approach to adding initialization code to a form: overriding the constructor. This is usually done as follows:

```
constructor TForm1.Create(AOwner: TComponent);
begin
  inherited Create (AOwner);
  // extra initialization code
end;
```

Before the call to the `Create` method of the base class, the properties of the form are still not loaded, and the internal components are not available. For this reason the standard approach is to call the base class constructor first and then do the custom operations.

Now the question is whether these custom operations are executed before or after the `OnCreate` event is fired. The answer depends on the value of the `OldCreateOrder`

184 This is in the Tools | Options dialog box under User Interface | Form Designer.

property of the form, introduced in Delphi 4 for backward compatibility with past versions of Delphi[185]. (This property is part of the Legacy category, which in Delphi 5 is hidden by default.) By default, for a new project, all of the code in the constructor is executed before the `OnCreate` event handler. In fact, this event handler is not activated by the base class constructor but by its `AfterConstruction` method, a sort of constructor introduced for compatibility with C++Builder.

note To study the creation order and the potential problems, you can examine the CreatOrd program. This program has an `OnCreate` event handler, which creates a list box control dynamically. The constructor of the form can access to this list box or not depending on the value of the `OldCreateOder` property.

Tracking Forms with the Screen Object

We have already explored some of the properties and events of the `Application` object. Other interesting global information about an application is available through the `Screen` object, whose base class is `TScreen`. This object holds information about the system display (the screen size and the screen fonts) and also about the current set of forms in a running application. For example, you can display the screen size and the list of fonts by writing:

```
Label1.Caption := IntToStr (Screen.Width) + 'x' +
  IntToStr (Screen.Height);
ListBox1.Items := Screen.Fonts;
```

`TScreen` also reports the number and resolution of monitors in a multimonitor system. What I want to focus on now, however, is the list of forms held by the `Forms` property of the `Screen` object, the topmost form indicated by the `ActiveForm` property, and the related `OnActiveFormChange` event. Note that the forms the `Screen` object references are the forms of the application and not those of the system.

These features are demonstrated by the Screen example, which maintains a list of the current forms in a list box. This list must be updated each time a new form is created, an existing form is destroyed, or the active form of the program changes. To see how this works, you can create a number of secondary forms by clicking on the button labeled New:

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

185 This *OldCreateOrder* property was recently removed, after having been deprecated for a very long time. As indicated here, it was added for Delphi 4 migration, and after another 20 versions of the product, the team felt it was time to remove it.

```
var
  NewForm: TSecondForm;
begin
  // create a new form, set its caption, and run it
  NewForm := TSecondForm.Create (Self);
  Inc (nForms);
  NewForm.Caption := 'Second ' + IntToStr (nForms);
  NewForm.Show;
end;
```

One of the key portions of the program is the `OnCreate` event handler of the form, which fills the list a first time and then connects a handler to the `OnActiveFormChange` event:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FillFormsList (Self);
  // set the secondary forms counter to 0
  nForms := 0;
  // set an event handler on the screen object
  Screen.OnActiveFormChange := FillFormsList;
end;
```

The code used to fill the Forms list box is inside a second procedure, `FillFormsList`, which is also installed as an event handler for the `OnActiveFormChange` event of the `Screen` object:

```
procedure TMainForm.FillFormsList (Sender: TObject);
var
  I: Integer;
begin
  FormsLabel.Caption := 'Forms: ' +
    IntToStr (Screen.FormCount);
  FormsListBox.Clear;
  // write class name and form title to the list box
  for I := 0 to Screen.FormCount - 1 do
    FormsListBox.Items.Add (Screen.Forms[I].ClassName +
      ' - ' + Screen.Forms[I].Caption);
  ActiveLabel.Caption := 'Active Form : ' +
    Screen.ActiveForm.Caption;
end;
```

note It is very important that you remove the `OnActiveFormChange` event handler before exiting the application; that is, before the main form is destroyed. Otherwise, the code will be executed when no list box exists, and you'll get a system error. The solution is to handle the `OnClose` event of the main form and assign `nil` to `Screen.OnActiveFormChange`.

The `FillFormsList` method fills the list box and sets a value for the two labels above it to show the number of forms and the name of the active one. When you click on

the New button, the program creates an instance of the secondary form, gives it a new title, and displays it. The Forms list box is updated automatically because of the handler we have installed for the `OnActiveFormChange` event. Figure 6.9 shows the output of this program when some secondary windows have been created.

note The program always updates the text of the `ActiveLabel` above the list box to show the currently active form, which is always the same as the first one in the list box.

The secondary forms each have a Close button you can select to remove them. The program handles the `OnClose` event, setting the `Action` parameter to `caFree`, so that the form is actually destroyed when it is closed. This code closes the form, but it doesn't update the list of the windows properly. The system moves the focus to another window first, firing the event that updates the list, and destroys the old form only after this operation.

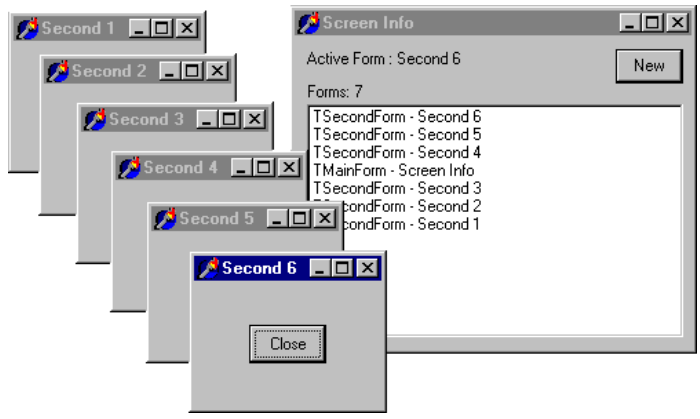


Figure 6.9: The output of the Screen example with a number of secondary forms. Image from the original book.

The first idea I had to update the windows list properly is to introduce a delay, posting a user-defined Windows message. Because the posted message is queued and not handled immediately, if we send it at the last possible moment of life of the secondary form, the main form will receive it when the other form is destroyed.

The trick is to post the message in the `OnDestroy` event handler of the secondary form. To accomplish this, we need to refer to the `MainForm` object, by adding a `uses` statement in the implementation portion of this unit. I've posted a `wm_User` message, which is handled by a specific `message` method of the main form, as shown here:

```
public
```

```
  procedure ChildClosed (var Message: TMessage);
    message wm_User;
```

Here is the code for this method:

```
procedure TMainForm.ChildClosed (var Message: TMessage);
begin
  FillFormsList (self);
end;
```

The problem here is that if you close the main window before closing the secondary forms, the main form exists, but its code cannot be executed anymore. To avoid another system error (an Access Violation Fault), you need to post the message only if the main form is not closing. But how do you know that? One way is to add a flag to the `TMainForm` class and change its value when the main form is closing, so that you can test the flag from the code of the secondary window.

This is a good solution—so good that the VCL already provides something similar. There is a barely documented `ComponentState` property. It is a Pascal set that includes (among other flags) a `csDestroying` flag, which is set when the form is closing. Therefore, we can write the following code:

```
procedure TSecondForm.FormDestroy(Sender: TObject);
begin
  if not (csDestroying in MainForm.ComponentState) then
    PostMessage (MainForm.Handle, wm_User, 0, 0);
end;
```

With this code, the list box always lists all of the forms in the application. Note that you need to disable the automatic creation of the secondary form by using the Forms page of the Project Options dialog box.

After giving it some thought, however, I found an alternative and much more Delphi-oriented solution. Every time a component is destroyed, it tells its owner about the event by calling the `Notification` method defined in the `TComponent` class. Because the secondary forms are owned by the main one, as specified in the code of the `NewButtonClick` method, we can override this method and simplify the code. In the form class, simply write

```
protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
```

Here is the code of the method:

```
procedure TMainForm.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
```

```
  inherited Notification(AComponent, Operation);
  if Showing and (AComponent is TForm) then
      FillFormsList;
end;
```

You'll find the complete code of this version in the Screen2 directory.

note In case the secondary forms were not owned by the main one, we could have used the `FreeNotification` method to get the secondary form to notify the main form when they are destroyed. `FreeNotification` receives as parameter the component to notify when the current component is destroyed. The effect is a call to the `Notification` method coming from a component other than the owned ones. `FreeNotification` is generally used by component writers to safely connect components on different forms or data modules.

The last feature I've added to both versions of the program is a simple one. When you click on an item in the list box, the corresponding form is activated, using the `BringToFront` method:

```
procedure TMainForm.FormsListBoxClick(Sender: TObject);
begin
  Screen.Forms [FormsListBox.ItemIndex].BringToFront;
end;
```

Nice—well, almost nice. If you click on the list box of an inactive form, the main form is activated first, and the list box is rearranged, so you might end up selecting a different form than you were expecting. If you experiment with the program, you'll soon realize what I mean. This minor glitch in the program is an example of the risks you face when you dynamically update some information and let the user work on it at the same time.

Closing a Form

When you close the form using the `Close` method or by the usual means (Alt+F4, the system menu, or the Close button), the `OnCloseQuery` event is called. In this event, you can ask the user to confirm the action, particularly if there is unsaved data in the form. Here is a simple scheme of the code you can write:

```
procedure TForm1.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if MessageDlg ('Are you sure you want to exit?',
      mtConfirmation, [mbYes, mbNo], 0) = idNo then
    CanClose := False;
end;
```

If `OnCloseQuery` indicates that the form should still be closed, the `OnClose` event is called. The third step is to call the `OnDestroy` event, which is the opposite of the `OnCreate` event and is generally used to deallocate objects related to the form and free the corresponding memory.

note To be more precise, the `BeforeDestruction` method generates an `OnDestroy` event before the `Destroy` destructor is called. That is, unless you have set the `OldCreateOrder` property to `True`, in which case Delphi uses a different closing sequence.

So what is the use of the intermediate `OnClose` event? In this method, you have another chance to avoid closing the application, or you can specify alternative "close actions." The method, in fact, has an `Action` parameter passed by reference. You can assign the following values to this parameter:

- `caNone`: The form is not allowed to close. This corresponds to setting the `CanClose` parameter of the `OnCloseQuery` method to `False`.
- `caHide`: The form is not closed, just hidden. This makes sense if there are other forms in the application; otherwise, the program terminates. This is the default for secondary forms, and it's the reason I had to handle the `OnClose` event in the previous example to actually close the secondary forms.
- `caFree`: The form is closed, freeing its memory, and the application eventually terminates if this was the main form. This is the default action for the main form and the action you should use when you create multiple forms dynamically (if you want to remove the Windows and destroy the corresponding Delphi object as the form closes).
- `caMinimize`: The form is not closed but only minimized. This is the default action for MDI child forms, as we'll see in Chapter 8.

note When a user shuts down Windows, the `OnCloseQuery` event is activated, and a program can use it to stop the shut-down process. In this case, the `OnClose` event is not called even if `OnCloseQuery` sets the `CanClose` parameter to `True`.

Form Input

Having discussed some special capabilities of forms, I'll now move to a very important topic: user input in a form. If you decide to make limited use of components,

you might write complex programs as well, receiving input from the mouse and the keyboard. In this chapter, I'll only introduce this topic. More about graphics can be found in Chapter 22, "Graphics in Delphi."

Supervising Keyboard Input

Generally, forms don't handle keyboard input directly. If a user has to type something, your form should include an edit component or one of the other input components. If you want to handle keyboard shortcuts, you can use those connected with menus (possibly using a hidden pop-up menu).

At other times, however, you might want to handle keyboard input in particular ways for a specific purpose. What you can do in these cases is turn on the `KeyPreview` property of the form. Then, even if you have some input controls, the form's `OnKeyPress` event will always be activated for any keyboard-input operation. The keyboard input will then reach the destination component, unless you stop it in the form by setting the character value to zero (not the character *0,* but the value 0 of the character set, indicated as `#0`).

The example I've built to demonstrate this, KPreview, has a form with no special properties (not even `KeyPreview`), a radio group with four options, and some edit boxes, as you can see in Figure 6.10.

By default the program does nothing special, except when the different radio buttons are used to enable the key preview:

```
procedure TForm1.RadioPreviewClick(Sender: TObject);
begin
  KeyPreview := RadioPreview.ItemIndex <> 0;
end;
```

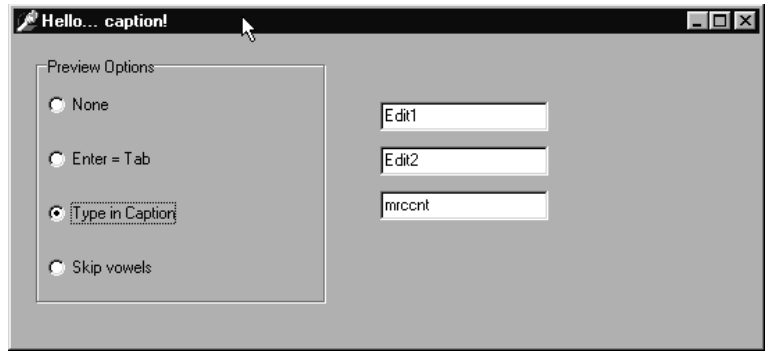


Figure 6.10: The KPreview program allows you to type into the caption of the form (among other things). Image from the original book.

Now we'll start receiving the `OnKeyPress` events, and we can do one of the three actions requested by the three special buttons of the radio group. The action depends on the value of the `ItemIndex` property of the radio group component. This is the reason the event handler is based on a `case` statement:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  case RadioPreview.ItemIndex of
    ...
```

In the first case, if the value of the `Key` parameter is `#13`, which corresponds to the Enter key, we disable the operation (setting `Key` to zero) and then mimic the activation of the Tab key. There are many ways to accomplish this, but the one I've chosen is quite particular. I send the `CM_DialogKey` message to the form, passing the code for the Tab key (`VK_TAB`):

```
1: // Enter = Tab
  if Key = #13 then
  begin
    Key := #0;
    Perform (CM_DialogKey, VK_TAB, 0);
  end;
```

note The `CM_DialogKey` message is an internal undocumented Delphi message, something that is really beyond the scope of this book but is discussed in other texts, including my own *Delphi Developer's Handbook* (Sybex, 1998).[186]

186 Not an easy book to find, thee days, but I have to say it has a lot of advanced content still valid and interesting today, along with areas of the libraries that have been significantly modified.

To type in the caption of the form, the program simply adds the character to the current `Caption`, as you can see in Figure 6.10. There are two special cases. When the Backspace key is pressed, the last character of the string is removed (by copying to the `Caption` all the characters of the current `Caption` but the last one). When the Enter key is pressed, the program stops the operation, by resetting the `ItemIndex` property of the radio group control. Here is the code:

```
2: // type in caption
begin
  if Key = #8 then // backspace: remove last char
    Caption := Copy (Caption, 1,
      Length (Caption) - 1)
  else if Key = #13 then // enter: stop operation
    RadioPreview.ItemIndex := 0
  else // anything else: add character
    Caption := Caption + Key;
  Key := #0;
end;
```

Finally, if the last radio item is selected, the code checks whether the character is a vowel (by testing for its inclusion in a constant *vowel set*). In this case, the character is skipped altogether:

```
3: // skip vowels
  if Key in ['a', 'e', 'i', 'o', 'u',
      'A', 'E', 'I', 'O', 'U'] then
    Key := #0;
```

Getting Mouse Input

When a user presses one of the mouse buttons over a form (or over a component, by the way), Windows sends the application some messages. Delphi defines some events you can use to write code that responds to these messages. The two basic events are as follows:

- `OnMouseDown` is received when one of the mouse buttons is pressed.
- `OnMouseUp` is received when one of the buttons is released.

Another fundamental system message is related to mouse movement. The event is `OnMouseMove`. Although it should be easy to understand the meaning of the three messages—down, up, and move—the question that might arise is, how do they relate to the `OnClick` event we have often used up to now?

We have used the `OnClick` event for components, but it is also available for the form. Its general meaning is that the left mouse button has been pressed and

released on the same window or component. However, between these two actions, the cursor might have been moved outside the area of the window or component, while the left mouse button was held down. If you press the mouse button at a certain position and then move it away and release it, no click is involved. In this case, the window receives *only* a down message, some move messages, and an up message. Another difference is that the click event relates only to the left mouse button.

The Mouse Buttons

Most of the mouse types connected to a Windows PC have two mouse buttons, and some even have three. Usually we refer to these buttons as the left mouse button, which is the most used; the right mouse button; and the middle mouse button:

- The left mouse button is *the* mouse button. It is used to select elements on screen, to give menu commands, to click buttons, to select and move elements (*dragging*), to select and activate (usually with a double-click), and so on.
- The right mouse button is used for local pop-up menus. Many applications used this approach in the past, but Windows 95 has made local menus the standard effect of right-clicking.
- The middle button is seldom used because most users either don't have it or don't have a proper software driver. Some CAD programs use the middle button. If you want to support this button, it should be optional (or else you should be ready to provide your customers with a free three-button mouse and the corresponding driver).

Keep in mind that users can customize their mouse buttons, switching the left and right buttons and turning a single click on the middle button into a double-click of the left button. When you refer to events related to a mouse button in your code, what matters is not the physical button but rather its meaning.

note Beyond the three traditional mouse buttons, there are now some mouse devices with a "button wheel" instead of the middle button. Users typically use the wheel for scrolling (causing an `OnMouseWheel` event), but they can also press it (generating the `OnMouseWheelDown` and `OnMouseWheelUp` events). The up and down messages are similar to the mouse button messages, whereas the `OnMouseWheel` event is devoted to handling the scrolling operations. Mouse wheel events are automatically converted into scrolling events.

Using Windows without a Mouse

A user should always be able to use any Windows application without the mouse. This is not an option; it is a Windows programming rule. Of course, an application

might be easier to use with a mouse, but that should never be mandatory. In fact, there are users who for various reasons might not have a mouse connected, such as travelers with a small laptop and no space, workers in industrial environments, and bank clerks with a number of other peripherals around.

There is another reason, already mentioned in this chapter in respect to the menu, to support the keyboard: Using the mouse is nice, but it tends to be slower. If you are a skilled touch typist, you won't use the mouse to drag a word of text; you'll use shortcut keys to copy and paste it, without moving your hands from the keyboard.

For all these reasons, you should always set up a proper tab order for a form's components, remember to add keys for buttons and menu items for keyboard selection, use shortcut keys on menu commands, and so on. An exception to this rule might be a graphics program. However, be aware that you can use even a program such as Microsoft Paint without the mouse—although I don't recommend it.

The Parameters of the Mouse Events

Since I'm going to build a graphics program, I will focus only on the use of the mouse. The first event we need to consider for the first minimal version of the MouseOne program is `OnMouseDown`. The related method has a number of parameters, as shown in the following declaration:

```
procedure TShapesForm.FormMouseDown (
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

In addition to the usual `Sender` parameter, there are four more parameters:

- `Button` indicates which of the three mouse buttons has been pressed. Possible values are `mbRight`, `mbLeft`, and `mbCenter`. These are exclusive values because the purpose of this parameter is to determine which button generated the message.
- `Shift` indicates which *mouse-related keys* were pressed when the event occurred. These mouse-related keys are Alt, Ctrl, and Shift, plus the mouse buttons themselves. This parameter is of a set type since several keys (and mouse buttons) might be pressed at the same time. This means you should test for a condition using the `in` expression, not for equality.
- `X` and `Y` indicate the coordinates of the position of the mouse, in *client area* coordinates of the current window (a form or a control). The origin of the *x*-and *y*-axes of these coordinates is the upper-left corner of the client area of the window receiving the event (again, a form or a control).

Using this information, it is very simple to draw a small circle in the position of a left mouse button-down event:

```
procedure TForm1.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
    Canvas.Ellipse (X-10, Y-10, X+10, Y+10);
end;
```

note To draw on the form, we use a very special property: `Canvas`. A `TCanvas` object has two distinctive features: it holds a collection of drawing tools (such as a pen, a brush, and a font) and it has a number of drawing methods, which use the current tools. This kind of direct drawing code in this example is not correct, because the on-screen image is not persistent: moving another window over the current one will clear its output. The next example demonstrates the Windows "store-and-draw" approach.

Dragging and Drawing with the Mouse

To demonstrate a few of the mouse techniques discussed so far, I've built a simple example based on a form without any component and called MouseOne. The first feature of this program is that it displays in the `Caption` of the form the current position of the mouse:

```
procedure TMouseForm.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
end;
```

You can use this simple feature of the program to better understand how the mouse works. Make this test: run the program (this simple version or the complete one) and resize the windows on the desktop so that the form of the MouseOne program is behind another window and inactive but with the title visible. Now move the mouse over the form, and you'll see that the coordinates change. This means that the `OnMouseMove` event is sent to the application even if its window is not active, and it proves what I have already mentioned: mouse messages are always directed to the window under the mouse. The only exception is the mouse capture operation I'll discuss in this same example.

Besides showing the position in the title of the window, the MouseOne example can track mouse movements by painting small pixels on the form if the user keeps the Shift key pressed. (Again this direct painting code produces nonpersistent output.)

```
procedure TMouseForm.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
  if ssShift in Shift then
    // mark points in yellow
    Canvas.Pixels [X, Y] := clYellow;
end;
```

The real feature of this example, however, is the direct mouse dragging support. Contrary to what you might think, Windows has no system support for dragging, which is implemented in the VCL by means of lower-level mouse events and operations. (An example of dragging from one control to another was discussed in the last chapter.) In the VCL, forms cannot originate dragging operations, so in this case we are obliged to use the low-level approach. The aim of this example is to draw a rectangle from the initial position of the dragging operation to the final one, giving the users some visual clue of the operation they are doing.

The idea behind dragging is quite simple. The program receives a sequence of button-down, mouse-move, and button-up messages. When the button is pressed, dragging begins, although the real actions take place only when the user moves the mouse (without releasing the mouse button) and when dragging terminates (when the button-up message arrives).

The problem with this basic approach is that it is not reliable. A window usually receives mouse events only when the mouse is over its client area; so if the user presses the mouse button, moves the mouse onto another window, and then releases the button, the second window will receive the button-up message.

There are two solutions to this problem. One (seldom used) is mouse clipping. Using a Windows API function (namely `ClipCursor`), you can force the mouse not to leave a certain area of the screen. When you move it outside the specified area, it stumbles against an invisible barrier. The second and more common solution is to capture the mouse. When a window captures the mouse, all the subsequent mouse input is sent to that window. This is the approach we will use for the MouseOne example.

The code of the example is built around three methods: `FormMouseDown`, `FormMouseMove`, and `FormMouseUp`. Pressing the left mouse button over the form starts the process, setting the `fDragging` Boolean field of the form (which indicates that dragging is in action in the other two methods). The method also uses a `TRect`

variable used to keep track of the initial and current position of the dragging. Here is the code:

```
procedure TMouseForm.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    fDragging := True;
    SetCapture (Handle);
    fRect.Left := X;
    fRect.Top := Y;
    fRect.BottomRight := fRect.TopLeft;
    Canvas.DrawFocusRect (fRect);
  end;
end;
```

An important action of this method is the call to the `SetCapture` API function. Now even if a user moves the mouse outside of the client area, the form still receives all mouse-related messages. You can see that for yourself by moving the mouse toward the upper-left corner of the screen; the program shows negative coordinates in the caption.

When dragging is active and the user moves the mouse, the program draws a dotted rectangle corresponding to the actual position. Actually, the program calls the `DrawFocusRect` method twice. The first time this method is called, it deletes the current image, thanks to the fact that two consecutive calls to `DrawFocusRect` simply reset the original situation. After updating the position of the rectangle, the program calls the method a second time:

```
procedure TMouseForm.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
  if fDragging then
  begin
    // remove and redraw the dragging rectangle
    Canvas.DrawFocusRect (fRect);
    fRect.Right := X;
    fRect.Bottom := Y;
    Canvas.DrawFocusRect (fRect);
  end
  else
    if ssShift in Shift then
      // mark points in yellow
      Canvas.Pixels [X, Y] := clYellow;
end;
```

When the mouse button is released, the program terminates the dragging operation by calling the `ReleaseCapture` API function and by setting the value of the `fDragging` field to `False`:

```
procedure TMouseForm.FormMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if fDragging then
  begin
    ReleaseCapture;
    fDragging := False;
    Invalidate;
  end;
end;
```

The final call, `Invalidate`, triggers a painting operation and executes the following `OnPaint` event handler:

```
procedure TMouseForm.FormPaint(Sender: TObject);
begin
  Canvas.Rectangle (fRect.Left, fRect.Top,
    fRect.Right, fRect.Bottom);
end;
```

This makes the output of the form persistent, even if you hide it behind another form. Figure 6.11 shows a previous version of the rectangle and a dragging operation in action.

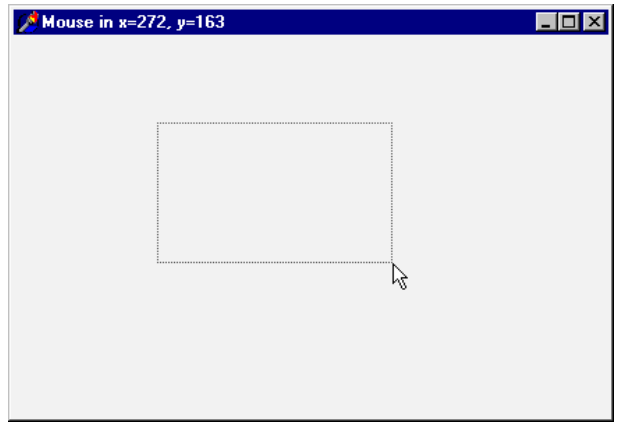


Figure 6.11: The MouseOne example uses a dotted line to indicate, during a dragging operation, the final area of a rectangle. Image from the original book.

Painting in Windows

Why do we need to handle the `OnPaint` event to produce a proper output, and why can we not paint directly over the form canvas? It depends on Windows' default behavior. As you draw on a window, Windows does *not* store the resulting image. When the window is covered, its contents are usually lost[187].

The reason for this behavior is simple: to save memory. Windows assumes it's "cheaper" in the long run to redraw the screen using code than to dedicate system memory to preserving the display state of a window. It's a classic memory versus CPU cycles trade-off. A color bitmap for a 300 x 400 image at 256 colors requires about 120KB. By increasing the color count or the number of pixels, you can easily have full-screen bitmaps of about 1MB and reach 4MB of memory for a 1280 x 1024 resolution at 16 million colors. If storing the bitmap was the default choice, running half a dozen simple applications would require at least 8MB of memory, if not 16MB, just for remembering their current output[188].

In the general case you want to have a consistent output for your applications, there are two techniques you can use. The general solution is to store enough data about the output to be able to reproduce it when the systems sends a *painting* requested. An alternative approach is to save the output of the form in a bitmap while you produce it, by placing an Image component over the form and drawing on the canvas of this image component.

The first technique, painting, is the common approach to handling output in Windows, aside from particular graphics-oriented programs that store the form's whole image in a bitmap. The approach used to implement painting has a very descriptive name: *store and paint*. In fact, when the user presses a mouse button or performs any other operation, we need to store the position and other elements; then, in the painting method, we use this information to actually paint the corresponding image.

The idea of this approach is to let the application repaint its whole surface under any of the possible conditions. If we provide a method to redraw the contents of the form, and if this method is automatically called when a portion of the form has been hidden and needs repainting, we will be able to re-create the output properly.

---

187 Core Windows painting concepts haven't changed at all over the years.

188 Needless to say some of the memory usage observations makes little sense in today's world, although we get inquiries about issues with multi-gigabytes bitmaps, which make me wonder if developers have an idea of the fact memory is finite anyway, even if so much larger.

Since this approach takes two steps, we must be able to execute these two operations in a row, asking the system to repaint the window—without waiting for the system to ask for this. You can use several methods to invoke repainting: `Invalidate`, `Update`, `Repaint`, and `Refresh`. The first two correspond to the Windows API functions, while the latter two have been introduced by Delphi.

- The `Invalidate` method informs Windows that the entire surface of the form should be repainted. The most important thing is that `Invalidate` does *not* enforce a painting operation immediately. Windows simply stores the request and will respond to it only after the current procedure has been completely executed and as soon as there are no other events pending in the system. Windows deliberately delays the painting operation because it is one of the most time-consuming operations. At times, with this delay, it is possible to paint the form only after a number of changes have taken place, avoiding a number of consecutive calls to the (slow) paint method.
- The `Update` method asks Windows to update the contents of the form, repainting it immediately. However, remember that this operation will take place only if there is an *invalid area*. This happens if the `Invalidate` method has just been called or as the result of an operation by the user. If there is no invalid area, a call to `Update` has no effect at all. For this reason, it is common to see a call to `Update` just after a call to `Invalidate`. This is exactly what is done by the two Delphi methods, `Repaint` and `Refresh`.
- The `Repaint` method calls `Invalidate` and `Update` in sequence. As a result, it activates the `OnPaint` event immediately. There is a slightly different version of this method called `Refresh`. For a form the effect is the same; for components it might be slightly different.

When you need to ask the form for a repaint operation, you should generally call `Invalidate`, following the standard Windows approach. This is particularly important when you need to request this operation frequently, because if Windows takes too much time to update the screen, the requests for repainting can be accumulated into a simple repaint action. The `wm_Paint` message in Windows is a sort of low-priority message. To be more precise, if a request for repainting is pending but other messages are waiting, the other messages are handled before the system actually performs the paint action.

On the other hand, if you call `Repaint` several times, the screen must be repainted each time before Windows can process other messages, and because paint operations are computationally intensive, this can actually make your application less responsive. There are times, however, when you want the application to repaint a

surface as quickly as possible. In these less-frequent cases, calling `Repaint` is the way to go.

note Another important consideration is that during a paint operation Windows redraws only the so-called *update region*, to speed up the operation. For this reason if you invalidate only a portion of a window, only that area will be repainted. To accomplish this you can use the `InvalidateRect` and `InvalidateRegion` functions. Actually, this feature is a double-edged sword. It is a very powerful technique, which can improve speed and reduce the flickering caused by frequent repaint operations. On the other hand, it can also produce incorrect output. A typical problem is when only some of the areas affected by the user operations are actually modified, while others remain as they were even if the system executes the source code that is supposed to update them. In fact, if a painting operation falls outside the update region, the system ignores it, as if it were outside the visible area of a window.

What's Next?

In this chapter we've explored some important form properties. Now we know how to handle the size and position of a form, how to resize it, and how to get mouse input and paint over it. We've also discussed in detail the role of two global objects, `Application` and `Screen`, and we've built applications with multiple forms. In Chapter 8, we'll extend this to cover dialog boxes in more detail.

Other chapters in the book will describe topics related to forms. In particular, Chapter 22, which was originally a bonus chapter available as a separate download; Chapter 7, the use of toolbars, status bars, and scrolling forms; Chapter 8, building a dialog box, forms with multiple pages, and MDI applications. As you can see from this list, forms play a central role in Delphi programming, and we still have to explore a number of topics related to them.

Chapter 7: Building A User Interface

One of the distinctive features of many Windows applications is the presence of a toolbar at the top of the window and a status bar at its bottom[189]. The toolbar usually contains a number of small buttons the user can click to give commands or to toggle options on and off. At times, a toolbar can also contain combo boxes, edit boxes, or other controls. The toolbars of the current generation of big applications usually can be moved to the left or right of the window, or even hidden and turned into a *toolbox,* a small floating window with an array of buttons.

---

189 This is still a common UI, although many alternatives emerged and became popular over the years. Modern apps tend to reduce the visible UI elements, which makes them nicer aesthetically, but often not so easy to use.

More complex applications tend to have multiple toolbars the user can configure. In Delphi you can use either the native ControlBar component or the Win32 CoolBar control[190], originally introduced by Microsoft Internet Explorer, for this purpose.

Toolbars account only for the first part of this chapter, which also covers the docking support that was introduced in Delphi 4 and presents examples of splitting forms, resizing controls dynamically, and scrolling the content of a form. These topics are not particularly complex, but it is worth examining their key concepts briefly.

The Toolbar Control

In early versions of Delphi, toolbars had to be created using panels and speed buttons, as briefly described in "Building a Toolbar with a Panel" later in this chapter. Starting with version 3, Delphi introduced a specific Toolbar component, which encapsulates the corresponding Win32 common control. This component provides a toolbar, with its own buttons, and it has some extended capabilities.

You've already seen examples of the Toolbar component in the Chapter 5 discussion of actions. To use this component, you place it on a form and then use the component editor (the shortcut menu activated by a right mouse button click) to create a few buttons and separators. You can see an example of a Toolbar component under construction in Figure 7.1.

The Toolbar is populated with objects of the `TToolButton` class. These are *internal* objects, just as a `TMenuItem` is an internal object of a MainMenu component. These objects have a fundamental property, `Style`, which determines their behavior[191]:

- The `tbsButton` style indicates a standard push button.
- The `tbsCheck` style indicates a button with the behavior of a check box, or that of a radio button if the button is `Grouped` with the others in its block (determined by the presence of separators).

190 The Coolbar controls, while still available, is rarely used these days. I've kept the coverage, though.

191 There are now two further toolbar button styles, `tbsTextButton` and `tbsWholeDropDown`.

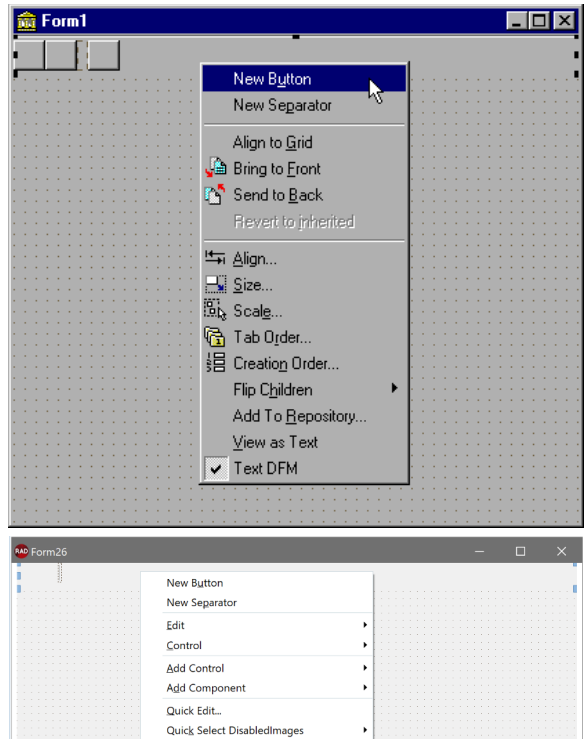


Figure 7.1: To create a toolbar, you can place the corresponding component on a form and then use its shortcut menu to add buttons and separators. Images captured in Delphi 5 and Delphi 12.

- The `tbsDropDown` style indicates a drop-down button, a sort of combo box. The drop-down portion can be easily implemented in Delphi by connecting a PopupMenu control to the `DropdownMenu` property of the control.
- The `tbsSeparator` and `tbsDivider` styles indicate separators with no or different vertical lines (depending on the `Flat` property of the toolbar).

To create a graphic toolbar, you can add an ImageList component to the form[192], load some bitmaps into it, and then connect the ImageList with the `Images` property of the toolbar. By default the images will be assigned to the buttons in the order they appear, but you can change this quite easily by setting the `ImageIndex` property of each toolbar button. You can prepare further ImageLists for special conditions of

192 Instead of using an image list, it's now recommended to use an ImageCollection and a VirtualImageList. The combination of these controls allows your application to select the correct image depending on the HighDPI resolution the application is running on. In the ImageCollection you can provide multiple set of images for different resolutions, or the component can create those automatically for you by resizing the available ones. By using the old approach explained here, you can end up with toolbars having very small images on HighDPI.

the buttons and assign them to the `DisabledImages` and `HotImages` properties of the toolbar. The first group is used for the disabled buttons, the second for the button currently under the mouse. This is the effect introduced by Microsoft Internet Explorer.

In a nontrivial application, you would generally add an ActionList component, particularly if you plan to have a menu with options that duplicate toolbar buttons (for example, a File ➢ Save menu option and a Save button). In this case you'll attach very little behavior to the toolbar buttons, as their properties and events will be managed by the action components. For example, you can obtain a toolbar button that toggles between a "selected" and an "unselected" state, like a check box. You obtain this effect by toggling the value of the `Checked` property of the action every time this is executed. In this case there is no need to set up the toolbar button with the `tbsCheck` style, as the code will determine the requested behavior.

The Toolbar and the ActionList of an Editor

In the MdEdit1 example, I've built a menu and a toolbar around a RichEdit control, providing the first step of an RTF (Rich Text File) editor I'll expand further in this and future chapters.

The application is based on an ActionList component, which includes actions for file handling and Clipboard support, and handles font and paragraph attributes. My aim is not to build a full-featured editor, or to investigate each and every feature of the RichEdit common control. I simply want to show how to build the user interface of a program, for which purpose it is valuable to work with a useful example. Rather than discuss all of the features of the program, I'll only highlight the points related to the current discussion. For a more detailed description of the code, you can open the "MdEdit Basics" RTF document available with the source code of the project.

The toolbar of the MdEdit1 example has most of its buttons connected to actions, which are available in a single ActionList component used to handle also all of the menu items. Only the last button, which has the `tbsDropDown` style, is handled directly and not through an action. Here is the structure of the toolbar:

```
object ToolBar1: TToolBar
  AutoSize = True
  Flat = True
  Images = Images
  object ToolButton1: TToolButton
    Action = acNew
  end
  object ToolButton2: TToolButton
```

```
    Action = acOpen
  end
  ...
  object ToolButton17: TToolButton
    DropdownMenu = SizeMenu
    ImageIndex = 13
    Style = tbsDropDown
    OnClick = ToolButton17Click
  end
end
```

The last button is connected to a PopupMenu component (called `SizeMenu`). This is all you have to do to make it display the list of items when the down arrow is selected, as you can see in Figure 7.2. Because the button can also be clicked, I've provided an event handler, which increases the size of the selected text.

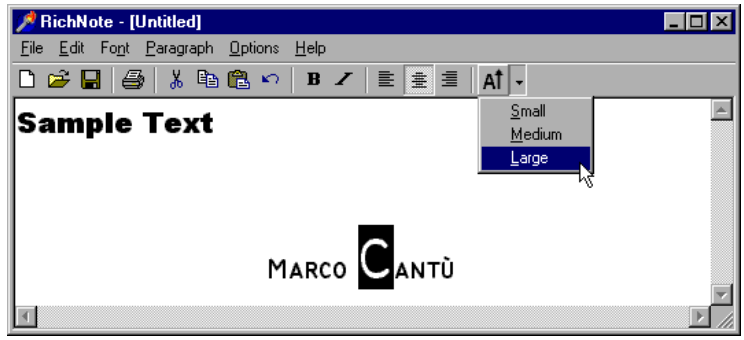


Figure 7.2: The toolbar of the MdEdit1 example has a drop-down button connected to a pop-up menu. Image from the original book.

The three paragraph-alignment buttons have their `Grouped` property set to `True`, forming a group (as they are enclosed between two separators). This is required because the program checks the action corresponding to the current style, in the `OnUpdate` event of the action list, but it fails to uncheck the other two actions. The user interface behavior of the menu items is determined by their `RadioItem` style and that of the toolbar buttons with the grouping and the `AllowAllUp` property.

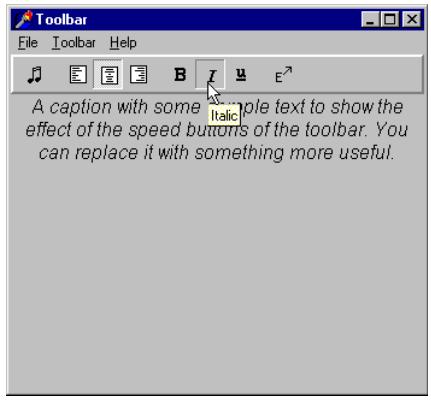
Building a Toolbar with a Panel

Before the toolbar control was available in Delphi, the standard approach for building a toolbar was to use a panel aligned to the top of the form and place a number of SpeedButton components inside it. A *speed button* is a lightweight graphical control (consuming no Windows resources); it cannot receive the input focus, it has no tab order, and it is faster to create and paint than a bitmap button.

Speed buttons can behave like push buttons, check boxes, or radio buttons, and they can have different bitmaps depending on their status. To make a group of speed buttons work like radio buttons, just place some speed buttons on the panel, select all of them, and give the same value to each one's `GroupIndex` property. All the buttons having the same `GroupIndex` become mutually exclusive selections. One of these buttons should always be selected, so remember to set the `Down` property to `True` for one of them at design time or as soon as the program starts.

By setting the `AllowAllUp` property, you can create a group of mutually exclusive buttons, each of which can be *up*—that is, a group from which the user can select one option or leave them all unselected. As a special case, you can make a speed button work as a check box, simply by defining a group (the `GroupIndex` property) that has only one button and that allows it to be deselected (the `AllowAllUp` property).

Finally, you can set the `Flat` property of all the SpeedButton components to `True`, obtaining a more modern user interface. If you are interested in this approach, you can look at the PanelBar example, illustrated here:



The use of SpeedButton controls is becoming less common. Besides the fact that the Toolbar control is very handy and definitely more standard, speed buttons have two big problems. First, each of them requires a specific bitmap and cannot use one from an image list (unless you write some complex code). Second, speed buttons don't work very well with actions, because some properties such as the `Down` state do not map directly.

A Combo Box in a Toolbar

We can extend this example by adding a combo box to the toolbar. A number of common applications use combo boxes in toolbars to show lists of styles, fonts, font sizes, and so on. Because we've used a drop-down button for the font size, we can add a combo box to allow rapid selection of the font family. This is simple to accomplish, as the Toolbar control is a full-featured control container; you can directly

take an edit box, a combo box, and other controls and place them inside the toolbar. Figure 7.3 shows the MdEdit2 application, with its font-selection combo box.

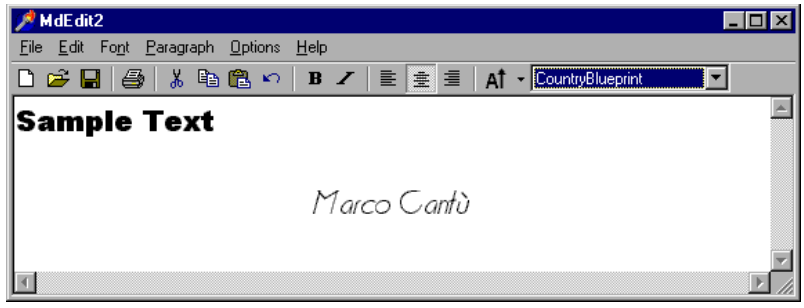


Figure 7.3: The MdEdit2 example at run time. Image from the original book.

The combo box in the toolbar is initialized in the `FormCreate` method, which extracts the screen fonts available in the system:

```
ComboFont.Items := Screen.Fonts;
ComboFont.ItemIndex := ComboFont.Items.IndexOf (
  RichEdit.Font.Name)
```

The combo box initially displays the name of the default font used in the RichEdit control, which is set at design time. This value is recomputed each time the current selection changes, using the font of the selected text:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
  ComboFont.ItemIndex :=
    ComboFont.Items.IndexOf (RichEdit.SelAttributes.Name)
end;
```

When a new font is selected from the combo box, the reverse action takes place. The text of the current combo box item is assigned as the name of the font for any text selected in the RichEdit control:

```
procedure TFormRichNote.ComboFontClick(Sender: TObject);
begin
  RichEdit.SelAttributes.Name :=
    ComboFont.Text;
end;
```

Toolbar Hints

Another common element in toolbars is the *fly-by hint*, also called *balloon help*—some text that briefly describes the button currently under the cursor. This text is usually displayed in a yellow box after the mouse cursor has remained steady over a button for a set amount of time. To add hints to an application's toolbar, simply set its `ShowHints` property to `True`.

I want to use the `Caption` of each action as its hint, so I could simply copy them all at run time, instead of setting each at design time. The problem is that the captions include the ampersand character used for the menu shortcuts. We can solve this by removing those extra characters with the new `StripHotKey` function in the Menus unit. Here is the code:

```
procedure TFormRichNote.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  ...
  // move captions to hints, removing the &
  for I := 0 to ActionList.ActionCount - 1 do
    (ActionList.Actions[I] as TAction).Hint :=
      StripHotKey ((ActionList.Actions[I] as TAction).Caption);
end;
```

As you can see in Figure 7.4, the hints also include a string showing the shortcut associated with each menu item, as a reminder to the user. This is a default behavior you can disable by setting the `HintShortCuts` property of the `Application` object. This global object controls the hints with other properties and some methods and events. For example, you can change the `HintColor`, `HintPause`, `HintHidePause`, and `HintShortPause` properties. The MdEdit2 example allows a user to customize the hint background color by selecting a specific menu item (Options ➢ Hint Color), with the following event handler:

```
procedure TFormRichNote.acHintColorExecute (Sender: TObject);
begin
  ColorDialog.Color := Application.HintColor;
  if ColorDialog.Execute then
    Application.HintColor := ColorDialog.Color;
end;
```

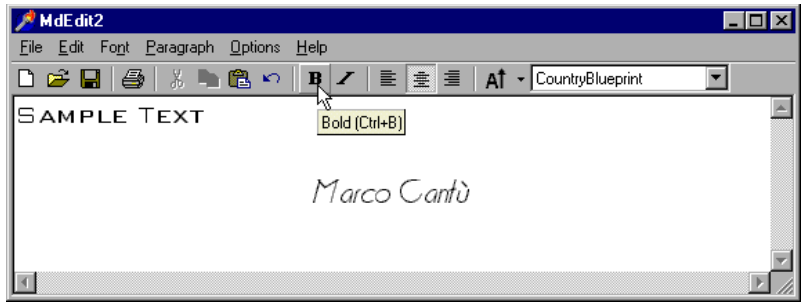


Figure 7.4: The hints displayed by the MdEdit2 example. Image from the original book.

note As an alternative, you can change the hint color by handling the `OnShowHint` property of the `Application` object. This handler can change the color of the hint just for specific controls. The `OnShowHint` event is used in the following CustHint example.

Customizing the Hints

Just as we have added hints to an application's toolbar, we can add hints to forms or to the components of a form. For a large control, the hint will show up near the mouse cursor. In some cases, it is important to know that a program can freely customize how hints are displayed[193].

The simplest thing you can do is change the value of the `HintColor` property of the `Application` object (as in the previous example) and the three properties related to the hint pause: `HintPause`, `HintHidePause`, and `HintShortPause`. The first defines how long the cursor should remain on a component before hints are displayed, the second how long the hint will be displayed, and the third how long the system should wait to display a hint if another hint has just been displayed.

To obtain more control over hints, you can customize them even further by assigning a method to the application's `OnShowHint` event. You need to either hook them up manually or—better—add an ApplicationEvents component to the form and handle its `OnShowHint` event.

193 There is now also a BalloonHint component you can use to create hints with a more complex UI structure and including more information. These new hints require Windows themes to be active.

The method you have to define has some interesting parameters, such as a string with the text of the hint, a `Boolean` flag for its activation, and a structure with further information:

```
TShowHintEvent = procedure (
  var HintStr: string;
  var CanShow: Boolean;
  var HintInfo: THintInfo) of object;
```

Each of the parameters is passed by reference, so you have a chance to change it. The last parameter is a structure, containing a reference to the control, the position of the hint, its color, and other information:

```
THintInfo = record
  HintControl: TControl;
  HintPos: TPoint;
  HintMaxWidth: Integer;
  HintColor: TColor;
  CursorRect: TRect;
  CursorPos: TPoint;
end;
```

You can modify the values of this structure; for example, you can change the position of the hint window before it is displayed. This is what I've done in the CustHint example, which shows the hint of the label at the center of its area. Here is what you can write to show the hint for the big label in the center of its surface:

```
procedure TForm1.ShowHint (var HintStr: string;
  var CanShow: Boolean; var HintInfo: THintInfo);
begin
  with HintInfo do
    if HintControl = Label1 then
      HintPos := HintControl.ClientToScreen (Point (
        HintControl.Width div 2, HintControl.Height div 2));
end;
```

The code has to retrieve the center of the generic control (the `HintInfo.HintControl`) and then convert its coordinates to screen coordinates, applying the `ClientToScreen` method to the control itself.

We can further update the CustHint example in a different way. The RadioGroup control in the form has three radio buttons. However, these are not stand-alone components, but simply radio button clones painted on the surface of the radio group. What if we want to add a hint for each of them?

The `CursorRect` field of the `THintInfo` record can be used for this purpose. It indicates the area of the component that the cursor can move over without disabling the hint. When the cursor moves outside this area, Delphi hides the hint window. If we specify a different text for the hint and a different area for each of the radio buttons,

we can in practice provide three different hints. Since computing the actual position of each radio button isn't easy, I've simply divided the surface of the radio group into as many equal parts as there are radio buttons. The text of the radio button (not the selected item, but the item under the cursor) is then added to the text of the hint:

```
procedure TForm1.ShowHint (var HintStr: string;
  var CanShow: Boolean; var HintInfo: THintInfo);
var
  RadioItem, RadioHeight: Integer;
  RadioRect: TRect;
begin
  with HintInfo do
    if HintControl = Label1 ... // as before
    else
    if HintControl = RadioGroup1 then
    begin
      RadioHeight := (RadioGroup1.Height) div
        RadioGroup1.Items.Count;
      RadioItem := CursorPos.Y div RadioHeight;
      HintStr := 'Choose the ' +
        RadioGroup1.Items [RadioItem] + ' button';
      RadioRect := RadioGroup1.ClientRect;
      RadioRect.Top := RadioRect.Top +
        RadioHeight * RadioItem;
      RadioRect.Bottom := RadioRect.Top + RadioHeight;
      // assign the hints rect and pos
      CursorRect := RadioRect;
    end;
end;
```

The final part of the code builds the rectangle for the hint, starting with the rectangle corresponding to the client area of the component and moving its `Top` and `Bottom` values to the proper section of the `RadioGroup1` component. The resulting effect is that each radio button of the radio group appears to have a specific hint, as shown in Figure 7.5.

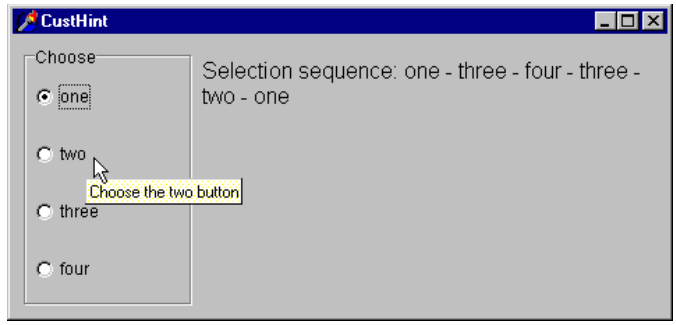


Figure 7.5: The RadioGroup control of the CustHint example shows a different hint, depending on the radio button the mouse is over. Image from the original book.

Toolbar Containers

Most modern applications have multiple toolbars, generally hosted by a specific container. Microsoft Internet Explorer, the various standard business applications, and the Delphi IDE all use this general approach. However, they each implement it differently. Delphi has two ready-to-use toolbar containers, the CoolBar and the ControlBar components. They have differences in their user interface, but the biggest one is that the CoolBar is a Win32 common control, part of the operating system, while the ControlBar is a VCL-based component.

Both components can host toolbar controls, as well as some extra elements, such as combo boxes and other controls. Actually, a toolbar can also replace the menu of an application, as we'll see later on.

We'll investigate the two components in the next two sections, but I want to emphasize here (without getting too far ahead of myself) that I generally favor the use of the ControlBar. It is based on the VCL (and not subject to upgrade along with each minor release of Microsoft Internet Explorer), and its user interface is nicer and more similar to that of common office applications.

A Really Cool Toolbar

The CoolBar component is basically a collection of `TCoolBand` objects. Unlike the toolbar buttons, these objects do not appear as stand-alone objects in the form, but are simply a collection of subitems. They appear in the Object Inspector only when

you select the editor of the CoolBar's `Bands` property, as you can see in Figure 7.6. You create one or more bands and then set their attributes.

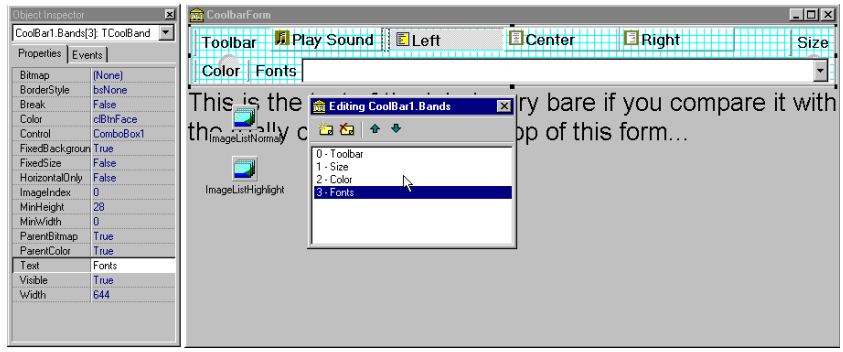


Figure 7.6: The property editor of the CoolBar component's Bands property works in conjunction with the Object Inspector. Image from the original book.

You can customize the CoolBar component in many ways: You can set a bitmap for its background, add some bands using the editor for the `Bands` property, and then assign to each band an existing component or component container. You can use any window-based control (not graphic controls), but only some of them will show up properly. If you want to have a bitmap on the background of the CoolBar, for example, you need to use partially transparent controls.

The typical component used in a CoolBar is the Toolbar (which can be made completely transparent), but combo boxes, edit boxes, and animation controls are also quite common. This is often inspired by the user interface of the Internet Explorer, the first Microsoft application featuring the CoolBar component.

You can place one band on each line or all of them on the same line. Each one would use a part of the available surface, and it would be automatically enlarged when the user clicks on its title. It is easier to use this new component than to explain it. Try it yourself or follow the description below, in which we build a new version of our continuing toolbar example based on a CoolBar control. You can see the form displayed by this application at run time in Figure 7.7.

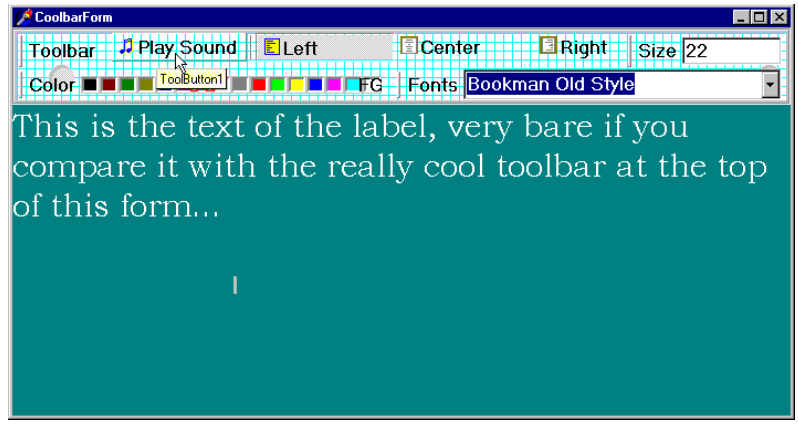


Figure 7.7: The form of the CoolBar example at run time. Image from the original book.

The CoolBar example has a `TCoolBar` component with four bands, two for each of the two lines. The first band includes a subset of the toolbar of the previous example, this time adding an ImageList for the highlighted images. The second has an edit box used to set the font of the text; the third has a ColorGrid component, used to choose the font color and that of the background. The last band has a ComboBox control with the available fonts.

The ControlBar

The user interface of the CoolBar component is really very nice, and Microsoft is increasingly using it in its applications[194]. However, the Windows CoolBar control has had many different and incompatible versions, as Microsoft has released different versions of the common control library with different versions of the Internet Explorer. Some of these versions "broke" existing programs built with Delphi.

note It is interesting to note that Microsoft applications generally don't use the common control libraries. Word and Excel use their own internal versions of the common controls, and VB uses an OCX, not the common controls directly. Part of the reason that Borland had so much trouble with the common controls is that it uses them more (and in more ways) than even Microsoft does.

194 This was true at the time. Microsoft later moved towards using the Ribbon control even outside of of Office, where it was originally introduced. The Ribbon control is a common replacement of toolbars.

For this reason, Borland introduced (in Delphi 4) a toolbar container called the ControlBar. A control bar hosts several controls, as a CoolBar does, and offers a similar user interface that lets a user drag items and reorganize the toolbar at run time. A good example of the use of the ControlBar control is Delphi's own toolbar, but Microsoft applications use a very similar user interface.

The ControlBar is a control container, and you build it just by placing other controls inside it, as you do with a panel. Every control placed in the bar gets its own dragging area (a small panel with two vertical lines, on the left of the control), as you can see in Figure 7.8. For this reason, you should generally avoid placing specific buttons inside the ControlBar, but rather add further containers with buttons inside them. Rather than using a panel, you should generally use one ToolBar control for every section of the toolbar.

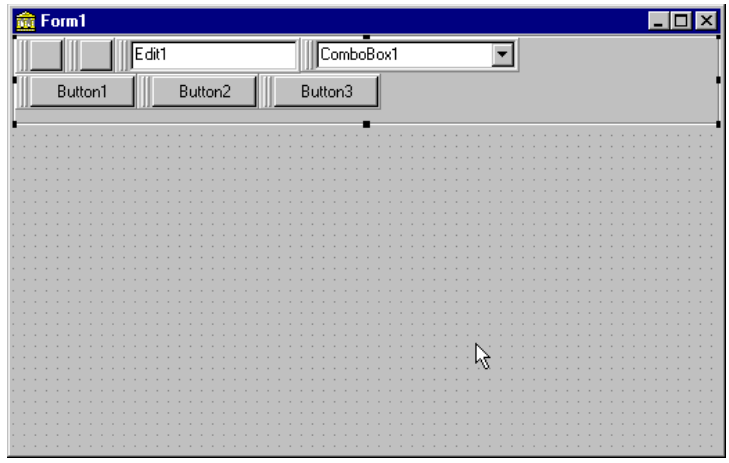


Figure 7.8: The ControlBar is a container that allows a user to drag all the elements, using the special drag bar on the side. Notice that each button gets a separate drag bar, something you'll generally try to avoid. Image from the original book.

The MdEdit3 example is another version of the RichEdit demo we've developed throughout this chapter. I've basically grouped the buttons into three toolbars (instead of a single one) and left the combo box as a stand-alone control. All these components are inside a ControlBar, so that a user can arrange them at will, as you can see in Figure 7.9 and in the following DFM listing:

```
object ControlBar1: TControlBar
  Align = alTop
  AutoSize = True
  ShowHint = True
  object ToolBarFile: TToolBar
    AutoSize = True
    EdgeBorders = []
```

```
    EdgeInner = esNone
    EdgeOuter = esNone
    Flat = True
    Images = Images
    Wrapable = False
    object ToolButton1: TToolButton
      Action = acNew
    end
    // more buttons
  end
  object ToolBarEdit: TToolBar
    // similar properties
    object ToolButton6: TToolButton
      Action = acCut
    end
    // more buttons
  end
  object ToolBarFont: TToolBar
    // ...
  end
  object ComboFont: TComboBox
    Hint = 'Font Family'
    Style = csDropDownList
    Font.Height = -11
    Font.Name = 'Arial'
    ItemHeight = 14
    ParentFont = False
    Sorted = True
    OnClick = ComboFontClick
  end
end
```

Notice in the listing that to obtain the standard effect, you have to disable the edges of the toolbar controls and set their style to flat. Sizing all the controls alike, so that you obtain one or two rows of elements of the same height, is not as easy as it might seem at first. Some controls have automatic sizing or various constraints. In particular, to make the combo box the same height as the toolbars, you have to tweak the type and size of its font. Resizing the control itself has no effect.

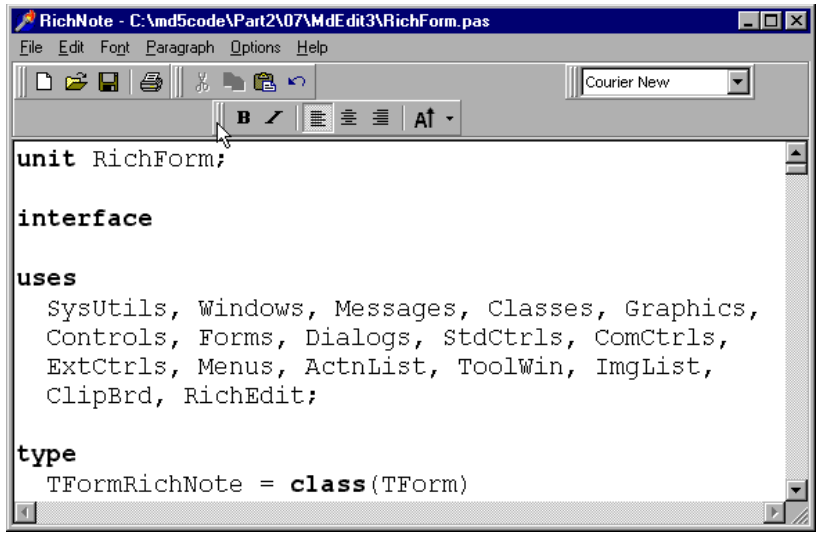


Figure 7.9: The MdEdit3 example at run time, while a user is rearranging the toolbars in the control bar. Image from the original book.

The ControlBar also has a shortcut menu that allows you to show or hide each of the controls currently inside it. Instead of writing code specific to this example, I've implemented a more generic (and reusable) solution. The shortcut menu, called `BarMenu`, is empty at design time and is populated when the program starts:

```
procedure TFormRichNote.FormCreate(Sender: TObject);
var
  I: Integer;
  mItem: TMenuItem;
begin
  ...
  // populate the control bar menu
  for I := 0 to ControlBar.ControlCount - 1 do
  begin
    mItem := TMenuItem.Create (Self);
    mItem.Caption := ControlBar.Controls [I].Name;
    mItem.Tag := Integer (ControlBar.Controls [I]);
    mItem.OnClick := BarMenuClick;
    BarMenu.Items.Add (mItem);
  end;
```

The `BarMenuClick` procedure is a single event handler that is used by all of the items of the menu and uses the `Tag` property of the `Sender` menu item to refer to the element of the ControlBar associated with the item in the `FormCreate` method:

```
procedure TFormRichNote.BarMenuClick(Sender: TObject);
var
  aCtrl: TControl;
```

```
begin
  aCtrl := TControl ((Sender as TComponent).Tag);
  aCtrl.Visible := not aCtrl.Visible;
end;
```

Finally, the `OnPopup` event of the menu is used to refresh the check mark of the menu items:

```
procedure TFormRichNote.BarMenuPopup(Sender: TObject);
var
  I: Integer;
begin
  // update the menu checkmarks
  for I := 0 to BarMenu.Items.Count - 1 do
    BarMenu.Items [I].Checked :=
      TControl (BarMenu.Items [I].Tag).Visible;
end;
```

A Menu in a Control Bar

If you look at the user interface of the Delphi development environment, you can see that a ControlBar also hosts the application's menu, which can be dragged in the same way as the toolbars and the Component Palette[195]. How can we add a menu to the ControlBar of our application?

The menu of the form cannot be placed inside the ControlBar, but we can add another new toolbar control to host it. This control should have the `ShowCaptions` property and the `Flat` property set to `True`. Then you should add as many tool buttons as there are pull-down menus, set their `AutoSize` and `Grouped` properties to `True`, and connect each tool button with the proper pull-down menu using the `MenuItem` property.

---

note Borland has made available a free TMenuBar component on its Web site (in the Delphi Downloads area). This component connects directly with a MainMenu component, doing all the required settings automatically.[196]

Once more, instead of doing all of these operations at design time, we can automate the creation of as many buttons as requested by the menu, adding more code to the `FormCreate` method:

---

195 This is still the case today.

196 This extensions has been later integrated in the VCL library. You can now add a menu to the control bar.

```
// create the buttons of the menu toolbar
ToolSize := 0;
for I := MainMenu.Items.Count - 1 downto 0 do
begin
  tb := TToolButton.Create (ToolBarMenu);
  tb.Parent := ToolBarMenu;
  tb.AutoSize := True;
  tb.Grouped := True;
  tb.Caption := MainMenu.Items[I].Caption;
  tb.MenuItem := MainMenu.Items[I];
  Inc (ToolSize, tb.Width);
end;
// size the menu toolbar
ToolBarMenu.Width := ToolSize;
// hide the standard menu, using the form's Menu property
Menu := nil;
```

Notice that you have to disconnect the menu from the form, by removing the value of the form's `Menu` property, which is automatically set as you place the menu component in the form. The result is a menu inside the ControlBar, as you can see in Figure 7.10.

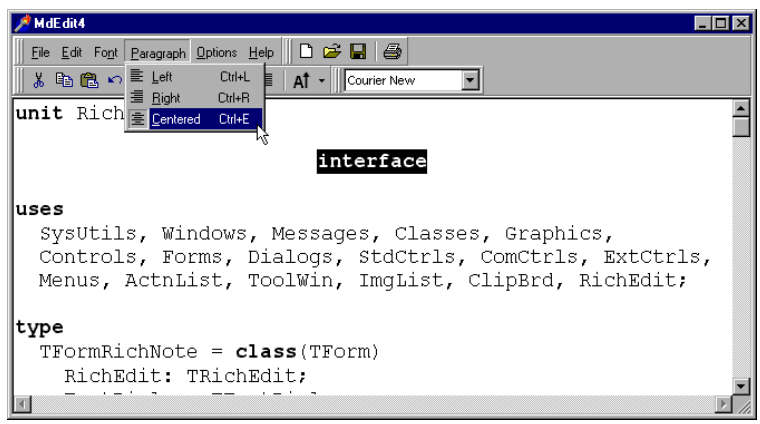


Figure 7.10: The MdEdit4 example shows how to place a menu inside a toolbar based on the ControlBar component. Image from the original book.

Creating a Status Bar

Building a status bar is even simpler than building a toolbar. Delphi includes a specific StatusBar component, based on the corresponding Windows common control. This component can be used almost as a panel when its `SimplePanel` property is set

to `True`. In this case you can use the `SimpleText` property to output some text. The real advantage of this component, however, is that it allows you to define a number of subpanels just by activating the editor of its `Panels` property. (You can also display this property editor by double-clicking on the status bar control.) Each subpanel has its own graphical attributes, which you can customize using the editor. Another feature of the status bar component is the "size grip" area added to the lower-right corner of the bar, which is useful for resizing the form itself. This is a typical element of the Windows user interface, and you can control it with the `SizeGrip` property.

There are a number of uses for a status bar. The most common is to display information about the menu item currently selected by the user. Besides this, a status bar often displays other information about the status of a program: the position of the cursor in a graphical application, the current line of text in a word processor, the status of the lock keys, the time and date, and so on.

Menu Hints in the Status Bar

A new version of the editor, MdEdit5, has a status bar capable of displaying the description of the current menu item, the status of the Caps Lock key, and the current editing position. The StatusBar component of this example has four panels. Although we're going to display text on only three of them, we need to define the fourth in order to delimit the area of the third panel. The last panel, in fact, is always large enough to cover the remaining surface of the status bar.

To show information on a panel, you simply use its `Text` property, generally using an expression like this:

```
StatusBar1.Panels[1].Text := 'message';
```

The panels are not independent components, so you cannot access them by name. A good solution to improve the readability of the program is to define a constant for each panel you want to use, and then use these constants when referring to the panels. The MdEdit5 example defines the following constants:

```
const
  sbpMessage = 0;
  sbpCaps = 1;
  sbpPosition = 2;
```

Now we have to populate the panels of the status bar with the proper text. First, we want to display a hint message for the menu items and toolbar buttons. To obtain this effect, you need to take two steps. First, input a string as a `Hint` property of each

action of the ActionList component. This hint will be used both as a fly-by hint for toolbar buttons and as a status bar message when the cursor is over the button or the menu item is selected. Actually, we can use the `Hint` property to specify different strings for the two cases, by writing a string divided into two portions by a separator, the | character. For example, you might enter the following as the value of the `Hint` property:

```
'Help|Activate the help of the application'
```

The first portion of the string, *Help*, is used by fly-by hints, the second portion by the status bar. You can see an example of this effect in Figure 7.11.

note When the hint for a control is made up of two strings, you can use the `GetShortHint` and `GetLongHint` methods to extract the first (short) and second (long) substrings from the string you pass as a parameter, which is usually the value of the `Hint` property.

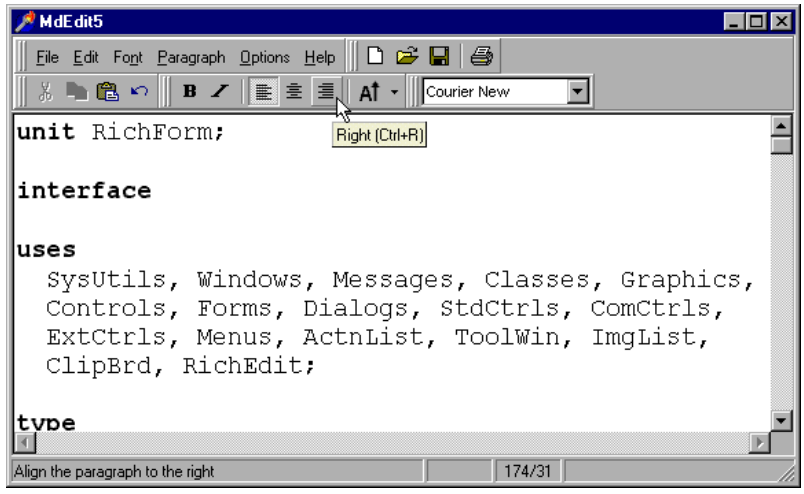


Figure 7.11: The status bar of the MdEdit5 example displays (among other information) a description of the current button or menu item. The two portions of the Hint property are displayed in the status bar and as a fly-by hint. Image from the original book.

To obtain the hint in the status bar, we have to write some code to handle the application's `OnHint` event. To avoid adding a new method to the form manually and then assign it to the `OnHint` event of the `Application` object, we can add to the form the ApplicationEvents component, and handle its event at design time.

The `ShowHint` procedure copies the current value of the application's `Hint` property, which temporarily contains a copy of the selected item's hint, to the status bar:

```
procedure TFormRichNote.ShowHint(Sender: TObject);
begin
```

```
  StatusBar1.Panels[sbpMessage].Text := Application.Hint;
end;
```

This is all you need to do to display a hint indicating the effect of a menu in the status bar.

To display the status of the Caps Lock key, or of any other key, you have to call the `GetKeyState` API function, which returns a state number. If the low-order bit of this number is set (that is, if the number is odd), then the key is pressed. When do we check this state? We can do it every time the user presses a key on the form, when the application is idle, or we can add a timer and make the check every 5 seconds. This second approach has an advantage, because the user might press the Caps Lock key while working with a different application, and this should be indicated on the status bar of our program, too. However, using a timer makes the response to pressing the key quite slow, while speeding up the timer might slow down the program. So I've decided to write a simple procedure, called `CheckCapslock`, and then call it both in the `OnUpdate` event handler of the ActionList component (called when the application has some idle time) and in the `OnTimer` event handler of a timer component I've added to the form:

```
procedure TFormRichNote.CheckCapslock;
begin
  if Odd (GetKeyState (VK_CAPITAL)) then
    StatusBar1.Panels[sbpCaps].Text := 'CAPS'
  else
    StatusBar1.Panels[sbpCaps].Text := '';
end;
```

Finally, the program uses the third panel to display the current cursor position (measured in lines and characters per line) every time the selection changes. Because the `CaretPos` values are zero-based (that is, the upper-left corner is line 0, character 0), I've decided to add one to each value, to make them more reasonable for a casual user:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
  ...
  // update the position in the status bar
  StatusBar.Panels[sbpPosition].Text := Format ('%d/%d',
    [RichEdit.CaretPos.Y + 1, RichEdit.CaretPos.X + 1]);
end;
```

Scrolling a Form

When you build a simple application, a single form might hold all of the components you need. As the application grows, however, you may need to squeeze in the components, increase the size of the form, or add new forms.

If you reduce the space occupied by the components, you might add some capability to resize them at run time, possibly splitting the form into different areas. If you choose to increase the size of the form, you might use scroll bars to let the user move around in a form that is bigger than the screen.

---

note If you choose to add a new form, you can create secondary forms and dialog boxes, create forms with multiple pages, or use the MDI approach (as described in the next chapter).

Adding a scroll bar to a form is simple. In fact, you don't need to do anything. If you place a number of components in a big form and then reduce its size, a scroll bar will be added to the form automatically, as long as you haven't changed the value of the `AutoScroll` property from its default of `True`.

Along with `AutoScroll`, forms have two properties, `HorzScrollBar` and `VertScrollBar`, which can be used to set several properties of the two `TFormScrollBar` objects associated with the form. The `Visible` property indicates whether the scroll bar is present, the `Position` property determines the initial status of the scroll thumb, and the `Increment` property determines the effect of clicking one of the arrows at the ends of the scroll bar. The most important property, however, is `Range`.

The `Range` property of a scroll bar determines the virtual size of the form in one direction, not the actual range of values of the scroll bar. At first, this might be somewhat confusing. Here is an example to clarify how the `Range` property works. Suppose you need a form that will host a number of components and will therefore need to be 1000 pixels wide. We can use this value to set the "virtual range" of the form, changing the range of the horizontal scroll bar. See Figure 7.12 for an illustration of the virtual size of a form implied by the range of a scroll bar. If the width of the client area of the form is smaller than 1000 pixels, a scroll bar will appear. Now you can start using it at design time to add new components in the "hidden" portion of the form.

The `Position` property of the scroll bar ranges from 0 to 1000 minus the current size of the client area. For example, if the client area of the form is 300 pixels wide, you can scroll 700 pixels to see the far end of the form (the thousandth pixel).

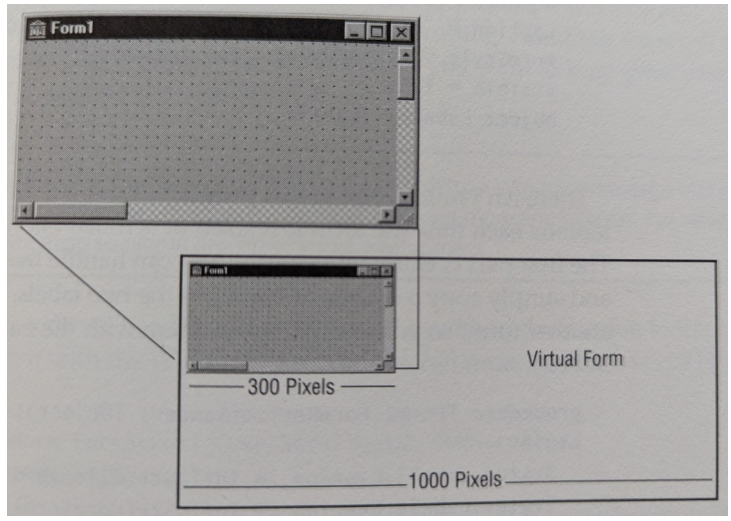


Figure 7.12: A representation of the virtual size of a form implied by the range of a scroll bar. Image based on a picture of the original printed book.

The Scroll Testing Example

I've built an example, Scroll1, which has a virtual form of 1000 pixels. To accomplish this, I simply set the range of the horizontal scroll bar to 1000:

```
object Form1: TForm1
  Width = 458
  Height = 368
  HorzScrollBar.Range = 1000
  VertScrollBar.Range = 305
  AutoScroll = False
  Caption = 'Scrolling Form'
  OnResize = FormResize
  ...
```

The form of this example has been filled with a number of meaningless list boxes, and I could have obtained the same scroll bar range by placing the rightmost list box so that its position (`Left`) plus its size (`Width`) would equal 1000.

The interesting part of the example is the presence of a toolbox window displaying the status of the form and of its horizontal scroll bar. This second form has four labels; two with fixed text and two with the actual output. Besides this, the secondary form (called `Status`) has a `bsToolWindow` border style and is a topmost window. You should also set its `Visible` property to `True`, to have its window automatically displayed at startup:

```
object Status: TStatus
  BorderIcons = [biSystemMenu]
  BorderStyle = bsToolWindow
  Caption = 'Status'
  FormStyle = fsStayOnTop
  Visible = True
  object Label1: TLabel...
  ...
```

There isn't much code in this program. Its aim is to update the values in the toolbox each time the form is resized or scrolled (as you can see in Figure 7.13). The first part is extremely simple. You can handle the `OnResize` event of the form and simply copy a couple of values to the two labels. The labels are part of another form, so you need to prefix them with the name of the form instance, `Status`:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  Status.Label3.Caption := IntToStr(ClientWidth);
  Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;
```

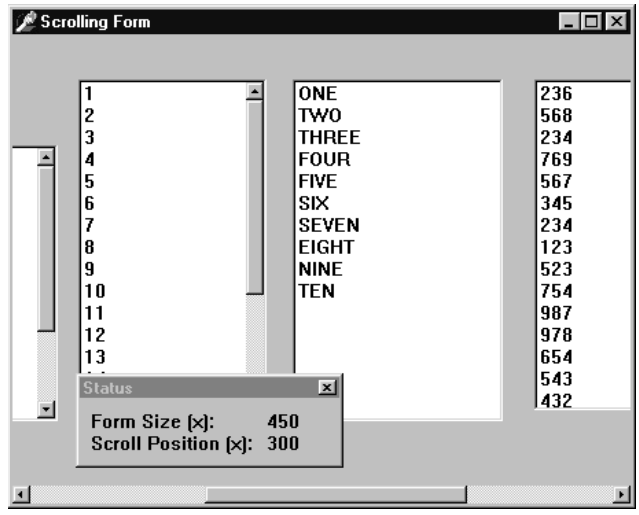


Figure 7.13: The output of the Scroll1 example. Image from the original book.

If we wanted to change the output each time the user scrolls the contents of the form, we could not use a Delphi event handler, because there isn't an `OnScroll` event for forms (although there is one for stand-alone ScrollBar components). Omitting this event makes sense, because Delphi forms handle scroll bars automatically in a powerful way. In Windows, by contrast, scroll bars are extremely low-level elements, requiring a lot of coding. Handling the scroll event makes sense only in

special cases, such as when you want to keep track precisely of the scrolling operations made by a user.

note Once again, what I really like in Delphi is that handling a Windows message that is not supported by the environment requires only one more line of code. I've never seen something so nice in any other visual environment.

Here is the code we need to write. First, add a method declaration to the class and associate it with the Windows horizontal scroll message (`wm_HScroll`):

```
  public
    procedure FormScroll (var ScrollData: TWMScroll);
      message wm_HScroll;
```

Then write the code of this procedure, which is almost the same as the code of the `FormResize` method we've seen before:

```
procedure TForm1.FormScroll (var ScrollData: TWMScroll);
begin
  inherited;
  Status.Label3.Caption := IntToStr(ClientWidth);
  Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;
```

It's important to add the call to `inherited`, which activates the method related to the same message in the base class form. The `inherited` keyword in Windows message handlers calls the method of the base class we are overriding, which is the one associated with the corresponding Windows message (even if the procedure name is different). Without this call, the form won't have its default scrolling behavior; that is, it won't scroll at all.

Automatic Scrolling

The scroll bar's `Range` property can seem strange until you start to use it consistently. When you think about it a little, you'll start to understand the advantages of the "virtual range" approach. First of all, the scroll bar is automatically removed from the form when the client area of the form is big enough to accommodate the virtual size; and when you reduce the size of the form, the scroll bar is added again.

This feature becomes particularly interesting when the `AutoScroll` property of the form is set to `True`. In this case, the extreme positions of the rightmost and lower controls are automatically copied into the `Range` properties of the form's two scroll bars. Automatic scrolling works well in Delphi. In the last example, the virtual size

of the form would be set to the right border of the last list box. This was defined with the following attributes:

```
object ListBox6: TListBox
  Left = 832
  Width = 145
end
```

Therefore, the horizontal virtual size of the form would be `977` (which is the sum of the two above values). This number is automatically copied into the `Range` field of the `HorzScrollBar` property of the form, unless you change it manually to have a bigger form (as I've done for the Scroll1 example, setting it to 1000 to leave some space between the last list box and the border of the form). You can see this value in the Object Inspector, or make the following test: Run the program, size the form as you like, and move the scroll thumb to the rightmost position. When you add the size of the form and the position of the thumb, you'll always get 1000, the virtual coordinate of the rightmost pixel of the form, whatever the size.

Scrolling and Form Coordinates

We have just seen that forms can automatically scroll their components. But what happens if you paint directly on the surface of the form? Some problems arise, but their solution is at hand. Suppose that we want to draw some lines on the virtual surface of a form, as shown in Figure 7.14.

Since you probably do not own a monitor capable of displaying 2000 pixels on each axis, you can create a smaller form, add two scroll bars, and set their `Range` property, as I've done in the Scroll2 example. Here is the textual description of the form:

```
object Form1: TForm1
  HorzScrollBar.Range = 2000
  VertScrollBar.Range = 2000
  ClientHeight = 336
  ClientWidth = 472
  OnPaint = FormPaint
end
```

If we simply draw the lines using the virtual coordinates of the form, the image won't display properly. In fact, in the `OnPaint` response method, we need to compute the virtual coordinates ourselves. Fortunately, this is easy, since we know that the virtual `X1` and `Y1` coordinates of the upper-left corner of the client area correspond to the current positions of the two scroll bars:

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

```
var
  X1, Y1: Integer;
begin
  X1 := HorzScrollBar.Position;
  Y1 := VertScrollBar.Position;

  // draw a yellow line
  Canvas.Pen.Width := 30;
  Canvas.Pen.Color := clYellow;
  Canvas.MoveTo (30-X1, 30-Y1);
  Canvas.LineTo (1970-X1, 1970-Y1);
// and so on ...
```

As a better alternative, instead of computing the proper coordinate for each output operation, we can call the `SetWindowOrgEx` API to move the origin of the coordinates of the `Canvas` itself. This way, our drawing code will directly refer to virtual coordinates but will be displayed properly:

```
procedure TForm2.FormPaint(Sender: TObject);
begin
  SetWindowOrgEx (Canvas.Handle,
    HorzScrollbar.Position,
    VertScrollbar.Position, nil);

  // draw a yellow line
  Canvas.Pen.Width := 30;
  Canvas.Pen.Color := clYellow;
  Canvas.MoveTo (30, 30);
  Canvas.LineTo (1970, 1970);

  // and so on ...
  ...
```

This is the version of the program you'll find in the source code you've downloaded. Try using the program and commenting out the `SetWindowOrgEx` call to see what happens if you don't use virtual coordinates: You'll find that the output of the program is not correct—it won't scroll, and the same image will always remain in the same position, regardless of scrolling operations.

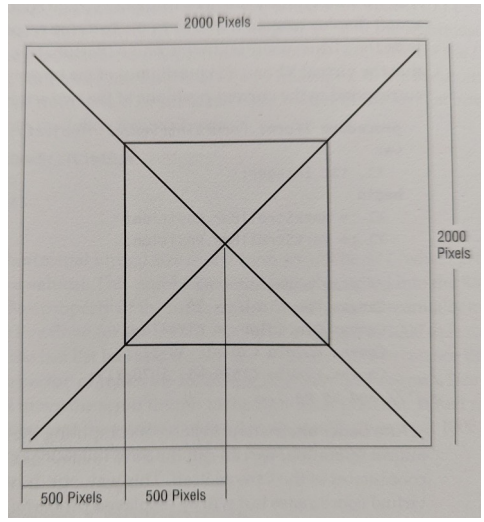


Figure 7.14: The lines to draw on the virtual surface of the form. Image based on a picture of the original printed book.

Form-Splitting Techniques

There are several ways to implement form-splitting techniques in Delphi, but the simplest approach is to use the Splitter component, found in the Additional page of the Component Palette. To make it more effective, the splitter can be used in combination with the `Constraints` property of the controls it relates to. As we'll see in the Split1 example, this allows us to define maximum and minimum positions of the splitter and of the form.

To build this example, simply place a ListBox component in a form; then add a Splitter component, a second ListBox, another Splitter, and finally a third ListBox component. The form also has a simple toolbar based on a panel.

By simply placing these two splitter components, you give your form the complete functionality of moving and sizing the controls it hosts at run time. The `Width`, `Beveled`, and `Color` properties of the splitter components determine their appearance, and in the Split1 example you can use the toolbar controls to change them. Another relevant property is `MinSize`, which determines the minimum size of the components of the form. During the splitting operation (see Figure 7.15), a line marks the final position of the splitter, but you cannot drag this line over a certain limit. The behavior of the Split1 program is not to let controls become too small. An

alternative technique is to set the new `AutoSnap` property of the splitter to `True`. This property will make the splitter hide the control when its size goes below the `MinSize` limit.

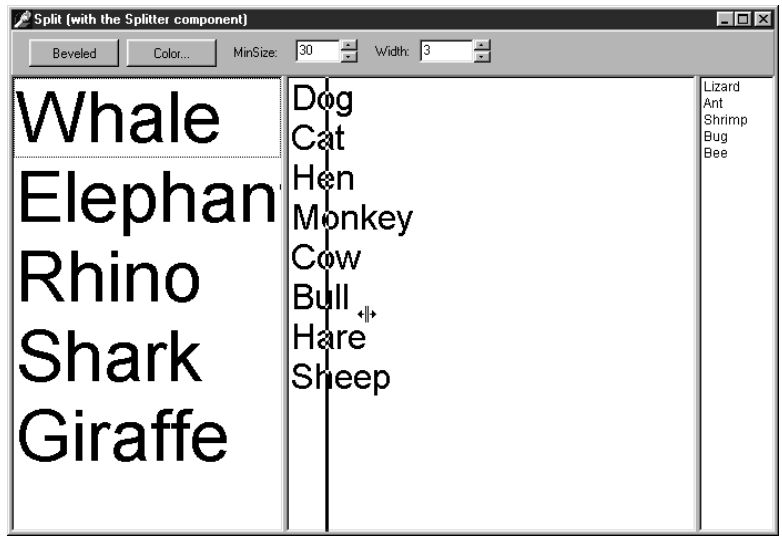


Figure 7.15: The splitter component of the Split1 example determines the minimum size for each control on the form, even those not adjacent to the splitter itself. Image from the original book.

I suggest you try using the Split1 program, so that you'll fully understand how the splitter affects its adjacent controls and the other controls of the form.

Even if I've set the `MinSize` property, a user of this program can reduce the size of its entire form to a minimum, hiding some of the list boxes. If you test the Split2 version of the example, instead, you'll get better behavior. In Split2 I've set some `Constraints` for the ListBox controls, as for example:

```
object ListBox1: TListBox
  Constraints.MaxHeight = 400
  Constraints.MinHeight = 200
  Constraints.MinWidth = 150
```

The size constraints are applied only as you actually resize the controls, so to make this program work in a satisfactory way, you have to set the `ResizeStyle` property of the two splitters to `rsUpdate`. This value indicates that the position of the controls is updated for every movement of the splitter, not only at the end of the operation. If you select the `rsLine` or the new `rsPattern` values, instead, the splitter simply draws a line in the required position, checking the `MinSize` property but not the constraints of the controls.

note The Splitter component in Delphi 5 has a new property, `AutoSnap`. When you set this to `True`, the splitter will completely hide the neighboring control when the size of that control is below the minimum set for it in the Splitter component.

Horizontal Splitting

The Splitter component can also be used for horizontal splitting, instead of the default vertical splitting. However, this approach is a little more complicated. Basically you can place a component on a form, align it to the top, and then place the splitter on the form. By default, it will be left-aligned. Choose the `alTop` value for the `Align` property, and then resize the component manually, by changing the `Height` property in the Object Inspector (or by resizing the component).

You can see a form with a horizontal splitter in the SplitH example. This program has two memo components you can open a file into, and it has a splitter dividing them, defined as:

```
object Splitter1: TSplitter
  Cursor = crVSplit
  Align = alTop
  OnMoved = Splitter1Moved
end
```

When you double-click on a memo, the program loads a text file into it (notice the structure of the `with` statement):

```
procedure TForm1.MemoDblClick(Sender: TObject);
begin
  with Sender as TMemo, OpenDialog1 do
    if Execute then
      Lines.LoadFromFile (FileName);
end;
```

The program features a status bar, which keeps track of the current height of the two memo components. It handles the `OnMoved` event of the splitter (the only event of this component) to update the text of the status bar. The same code is executed whenever the form is resized:

```
procedure TForm1.Splitter1Moved(Sender: TObject);
begin
  StatusBar1.Panels[0].Text := Format (
    'Upper Memo: %d - Lower Memo: %d',
    [MemoUp.Height, MemoDown.Height]);
end;
```

You can see the effect of this code by looking at Figure 7.16, or by running the SplitH example.

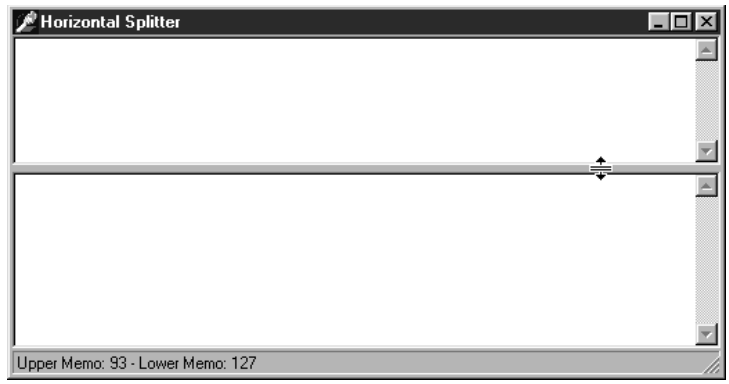


Figure 7.16: The status bar of the SplitH example indicates the position of the horizontal splitter component.

Splitting with a Header

An alternative to using splitters is to use the standard HeaderControl compo-nent. If you place this control on a form, it will be automatically aligned with the top of the form. Then you can add the three list boxes to the rest of the client area of the form. The first list box can be aligned on the left, but this time you cannot align the second and third list box as well. The problem is that the sections of the header can be dragged outside the visible surface of the form. If the list boxes use automatic alignment, they cannot move outside the visible surface of the form, as the program requires.

The solution is to define the sections of the header, using the specific editor of the `Sections` property. This property editor allows you to access the various subobjects of the collection, changing various settings. You can set the caption and alignment of the text; the current, minimum, and maximum size of the header; and so on. Setting the limit values is a powerful tool, and it replaces the `MinSize` property of the splitter or the constraints of the list boxes we've used in past examples. You can see the output of this program, named HdrSplit, in Figure 7.17.

We need to handle two events: `OnSectionResize` and `OnSectionClick`. The first handler simply resizes the list box connected with the modified section (determined by associating numbers with the `ImageIndex` property of each section and using it to determine the name of the list box control):

```
procedure TForm1.HeaderControl1SectionResize(
  HeaderControl: THeaderControl; Section: THeaderSection);
var
  List: TListBox;
begin
  List := FindComponent ('ListBox' + IntToStr (
    Section.ImageIndex)) as TListBox;
  List.Width := Section.Width;
end;
```

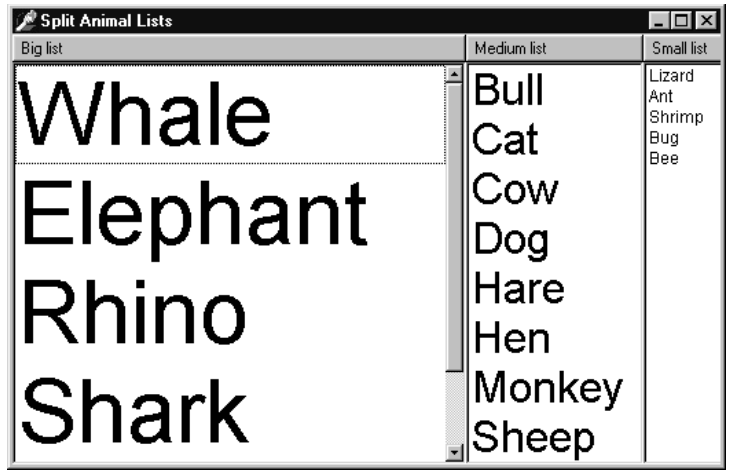


Figure 7.17: The output of the HdrSplit example. Image from the original book.

Along with this event, we need to handle the resizing of the form, using it to synchronize the list boxes with the sections, which are all resized by default:

```
procedure TForm1.FormResize(Sender: TObject);
var
  I: Integer;
  List: TListBox;
begin
  for I := 0 to 2 do
  begin
    List := FindComponent ('ListBox' + IntToStr (
      HeaderControl1.Sections[I].ImageIndex)) as TListBox;
    List.Left := HeaderControl1.Sections[I].Left;
    List.Width := HeaderControl1.Sections[I].Width;
  end;
end;
```

After setting the height of the list boxes, this method simply calls the previous one, passing parameters that we won't use in this example. The second method of the

HeaderControl, called in response to a click on one of the sections, is used to sort the contents of the corresponding list box:

```
procedure TForm1.HeaderControl1SectionClick(
  HeaderControl: THeaderControl; Section: THeaderSection);
var
  List: TListBox;
begin
  List := FindComponent ('ListBox' + IntToStr (
    Section.ImageIndex)) as TListBox;
  List.Sorted := not List.Sorted;
end;
```

Of course, this code doesn't provide the common behavior of sorting the elements when you click on the header and then sorting them in the reverse order if you click again. To implement this, you should write your own sorting algorithm. Finally, the HdrSplit example uses a new feature for the header control. It sets the `DragReorder` property to enable dragging operations to reorder the header sections. When this operation is performed, the control fires the `OnSectionDrag` event, where you can exchange the positions of the list boxes. This event fires before the sections are actually moved, so I have to use the coordinates of the other section:

```
procedure TForm1.HeaderControl1SectionDrag(Sender: TObject;
FromSection,
  ToSection: THeaderSection; var AllowDrag: Boolean);
var
  List: TListBox;
begin
  List := FindComponent ('ListBox' + IntToStr (
    FromSection.ImageIndex)) as TListBox;
  List.Left := ToSection.Left;
  List.Width := ToSection.Width;

  List := FindComponent ('ListBox' + IntToStr (
    ToSection.ImageIndex)) as TListBox;
  List.Left := FromSection.Left;
  List.Width :=fromSection.Width;
end;
```

Control Anchors

In this chapter I've described how you can use alignment and splitters to create nice and flexible user interfaces, which adapt to the current size of the form, giving users the maximum freedom. Delphi also supports right and bottom anchors. Before this

feature was introduced in Delphi 4, every control placed on a form had coordinates relative to the top and bottom sides, unless it was aligned to the bottom or right sides. Aligning is good for some controls but not all of them, particularly buttons.

By using anchors, you can make the position of a control relative to any side of the form. For example, to have a button anchored to the bottom-right corner of the form, place the button in the required position and set its `Anchors` property to `[akRight, akBottom]`. When the form size changes, the distance of the button from the anchored sides is kept fixed. In other words, if you set these two anchors and remove the two defaults, the button will remain in the bottom-right corner.

On the other hand, if you place a large component such as a Memo or a ListBox in the middle of a form, you can set its `Anchors` property to include all four sides. This way the control will behave as an aligned control, growing and shrinking with the size of the form, but there will be some margin between it and the form sides.

note Anchors, like constraints, work both at design time and at run time; so you should set them up as early as possible, to benefit from this feature while you're designing the form as well as at run time.

As an example of both approaches, you can try out the Anchors application, which has two buttons on the bottom-right corner and a list box in the middle. As shown in Figure 7.18, the controls automatically move and stretch as the form size changes. To make this form work properly, you must also set its `Constraints` property; otherwise, as the form becomes too small the controls can overlap or disappear.

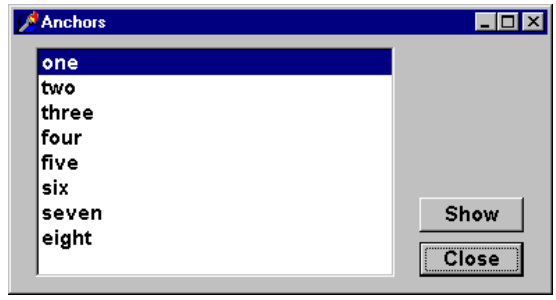


Figure 7.18: The controls of the Anchors example move and stretch automatically as the user changes the size of the form. No code is needed to move the controls, only a proper use of the Anchors property. Image from the original book.

note If you remove all of the anchors, or two opposite ones (for example, left and right), the resize operations will cause the control to float. The control keeps its current size, and the system adds or removes the same number of pixels on each side of it. This can be defined as a centered anchor, because if the component is initially in the middle of the form it will keep that position. In any case, if you want a centered control, you should generally use both opposite anchors, so that if the user makes the form larger the control size will grow as well. In the case just presented, in fact, making the form larger leaves a small control in its center.

Docking Toolbars and Controls

Another feature added in Delphi 4 was the support for *dockable* toolbars and controls[197]. In other words, you can create a toolbar and move it to any of the sides of a form, or even move it freely on the screen, undocking it. However, setting up a program properly to obtain this effect is not as easy as it sounds.

First of all, Delphi's docking support is connected with container controls, not with forms. A panel, a ControlBar, and other containers (technically, any control derived from `TWinControl`) can be set up as dock targets by enabling their `DockSite` property. You can also set the `AutoSize` property of these containers, so that they'll show up only if they actually hold a control.

To be able to drag a control (an object of any `TControl`-derived class) into the dock site, simply set its `DragKind` property to `dkDock` and its `DragMode` property to `dmAutomatic`. This way, the control can be dragged away from its current position into a new docking container. To undock a component and move it to a special form, you can set its `FloatingDockSiteClass` property to `TCustomDockForm` (to use a predefined stand-alone form with a small caption).

All the docking and undocking operations can be tracked by using special events of the component being dragged (`OnStartDock` and `OnEndDock`) and the component that will receive the docked control (`OnDragOver` and `OnDragDrop`). These docking events are very similar to the dragging events available in earlier versions of Delphi.

There are also commands you can use to accomplish docking operations in code and to explore the status of a docking container. Every control can be moved to a different location using the `Dock`, `ManualDock`, and `ManualFloat` methods. A container has a `DockClientCount` property, indicating the number of docked controls, and a `DockClients` property, with the array of these controls.

197 Docking remains a core feature of the VCL library and the Delphi IDE uses it heavily.

Moreover, if the dock container has the `UseDockManager` property set to `True`, you'll be able to use the `DockManager` property, which implements the `IDockManager` interface. This interface has many features you can use to customize the behavior of a dock container, even including support for streaming its status.

As you can see from this brief description, docking support in Delphi is based on a large number of properties, events, methods and objects (such as dock zones and dock trees)—more features than we have room to explore in detail. The next example introduces the main features you'll generally need.

Docking Toolbars in ControlBars

The MdEdit6 example is the final version of the RichEdit editor presented in this chapter. This new version has a second ControlBar at the bottom of the form, which accepts dragging one of the toolbars in the ControlBar at the top. Since both toolbar containers have the `AutoSize` property set to `True`, they are automatically removed when the host contains no controls. To let users drag the toolbars with the same anchor used for moving them inside the container, remember to set the `AutoDrag` property of the ControlBars, as well.

You can see an example of the program at run time in Figure 7.19. The components inside the control bar at the top have their `DragKind` property set to `dkDock`. However, the menu toolbar cannot be moved outside of its container, because we want to keep it close to the typical position of a menu bar. The combo box can be dragged, but we don't want to let a user dock it in the lower control bar. We implement the second constraint in the control bar's `OnDockOver` event handler, by accepting the docking operation only for toolbars:

```
procedure TFormRichNote.ControlBarLowerDockOver(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean);
begin
  Accept := Source.Control is TToolbar;
end;
```

Figure 7.19: The MdEdit6 example allows you to dock the toolbars (and the menu) at the top or bottom of the form or to leave them floating. Image from the original book.

Next, we want to have a border for the lower control bar, but only when it hosts some components, so that we don't see the empty border (as the control bar resizes itself to a very thin line when it is empty). To accomplish this, we can add the border whenever a control is dropped onto the bar (`OnDockDrop`) and remove it when the last control is being undocked (`OnUnDock`). To determine the number of controls, we can use the `DockClientCount` property, which is updated after the undocking is completed, so its value is still 1 when the last control is being undocked:

```
procedure TFormRichNote.ControlBarLowerDockDrop(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer);
begin
  ControlBarLower.BevelKind := bkTile;
end;

procedure TFormRichNote.ControlBarLowerUnDock(Sender: TObject;
  Client: TControl; NewTarget: TWinControl; var Allow: Boolean);
begin
  if ControlBarLower.DockClientCount = 1 then
    ControlBarLower.BevelKind := bkNone;
end;
```

This excerpt from the form's DFM file shows the properties related to docking support:

```
object FormRichNote: TFormRichNote
  object RichEdit: TRichEdit...
  object ControlBar: TControlBar
    AutoSize = True
    object ToolBarFile: TToolBar
      DragKind = dkDock
      DragMode = dmAutomatic
    end
    object ToolBarEdit: TToolBar
      DragKind = dkDock
```

```
      DragMode = dmAutomatic
    end
    object ToolBarFont: TToolBar
      DragKind = dkDock
      DragMode = dmAutomatic
    end
    object ComboFont: TComboBox
      DragKind = dkDock
      DragMode = dmAutomatic
    end
    object ToolBarMenu: TToolBar...
  end
  object StatusBar: TStatusBar...
  object ControlBarLower: TControlBar
    BevelKind = bkNone
    OnDockDrop = ControlBarLowerDockDrop
    OnDockOver = ControlBarLowerDockOver
    OnUnDock = ControlBarLowerUnDock
  end  ...
end
```

note When you move one of the toolbars to the automatically created floating form, you might be tempted to set it back by closing the floating form. This doesn't work, as the floating form is removed along with the toolbar it contains. However, you can use the shortcut menu of the top-most ControlBar to show this hidden toolbar.

Controlling Docking Operations

Delphi provides many events and methods that give you a lot of control over docking operations, including a dock manager. To explore some of these features, try out the DockTest example, a test bed for docking operations. The program assigns the `FloatingDockSiteClass` property of a Memo component to `TForm2`, so that you can design specific features and add them to the floating frame that will host the control when it is floating, instead of using an instance of the default `TCustomDockForm` class.

Another feature of the program is that it handles the `OnDockOver` and `OnDockDrop` events of a dock host panel to display messages to the user, such as the number of controls currently docked:

```
procedure TForm1.Panel1DockDrop(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer);
begin
  Caption := 'Docked: ' + IntToStr (Panel1.DockClientCount);
end;
```

In the same way, the program also handles the main form's docking events. Another control, a list box, has a shortcut menu you can invoke to perform docking and undocking operations in code, without the usual mouse dragging:

```
procedure TForm1.DocktoPanel1Click(Sender: TObject);
begin
  // dock to the panel
  ListBox1.ManualDock (Panel1, Panel1, alBottom);
end;

procedure TForm1.DocktoForm1Click(Sender: TObject);
begin
  // dock to the current form
  ListBox1.Dock (Self, Rect (200, 100, 100, 100));
end;

procedure TForm1.Floating1Click(Sender: TObject);
begin
  // toggle the floating status
  if ListBox1.Floating then
    ListBox1.ManualDock (Panel1, Panel1, alBottom)
  else
    ListBox1.ManualFloat (Rect (100, 100, 200, 300));
  Floating1.Checked := ListBox1.Floating;
end;
```

The final feature of the example is probably the most interesting one: Every time the program closes, it saves the current docking status of the panel, using the dock manager support. When the program is reopened, it reapplies the docking information, restoring the previous configuration of the windows. The program does this only with the panel, so the other floating windows will be displayed in their original positions. Here is the code for saving and loading:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  FileStr: TFileStream;
begin
  if Panel1.DockClientCount > 0 then
  begin
    FileStr := TFileStream.Create (DockFileName,
      fmCreate or fmOpenWrite);
    try
      Panel1.DockManager.SaveToStream (FileStr);
    finally
      FileStr.Free;
    end;
  end
  else
    // remove the file
    DeleteFile (DockFileName);
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
var
  FileStr: TFileStream;
begin
  // reload the settings
  DockFileName := ExtractFilePath (Application.Exename) +
    'dock.dck';
  if FileExists (DockFileName) then
  begin
    FileStr := TFileStream.Create (DockFileName, fmOpenRead);
    try
      Panel1.DockManager.LoadFromStream (FileStr);
    finally
      FileStr.Free;
    end;
  end;
  Panel1.DockManager.ResetBounds (True);
end;
```

There are many more features you can test, but the DockTest program already tries to do too many things, some of which conflict. For example, automatic alignments don't work terribly well with the docking manager's code for restoring. I suggest you take this program and explore its behavior, extending it to support the type of user interface you prefer.

note Remember that although docking panels make an application look nice, some users get confused by the fact that their toolbars might disappear or be in a different position than they are used to. Don't overuse the docking features, or some of your inexperienced users may get lost.

What's Next?

In this chapter, we have examined a series of topics related to toolbars and forms: the definition of a toolbar and a status bar; and ways to scroll, split, and drag forms. Although these may seem very diverse topics, they all relate to the development of a modern user interface for a form.

You can consider this chapter the first step toward building professional applications. We will take other steps in the following chapters; but you already know enough to make your programs similar to some best-selling Windows applications, which may be very important for your clients. Now that the elements of the main form of our programs are properly set up, we can consider adding secondary forms and dialog boxes. This is the topic of the next chapter, although we have already

seen how simple it is to add a second form to a program to build a toolbox. In the next chapter we'll also explore multiple-page forms, another important addition to the toolkit of any developer who wants to create a modern user interface.

Chapter 8: Using Multiple Forms

Up to this point, most of the programs in this book have consisted of single forms. Usually, applications have a main window, some floating toolboxes or palettes, and a number of dialog boxes that can be invoked through menu commands or command buttons. More complex applications might have an MDI structure—a frame window with a number of child windows inside its client area. The development of MDI applications will be discussed briefly at the end of this chapter, after we focus on building dialog boxes and applications with multiple forms.

Dialog Boxes versus Forms

Before presenting examples of applications with multiple forms or user-defined dialog boxes, let me begin with a general description of these two alternatives. When you write a program, there is really no big difference between a dialog box and a

second form, aside from the border, the border icons, and other user-interface elements you can customize.

What users associate with a dialog box is the concept of a *modal window*—a window that takes the focus and must be closed before the user can move back to the main window. This is true for message boxes and usually for dialog boxes, as well. However, you can also have nonmodal—or *modeless*—dialog boxes. So if you think that dialog boxes are just modal forms, you are on the right track, but your description is not precise. In Delphi (as in Windows), you can have modeless dialog boxes and modal forms. We have to consider two different elements:

- The form's border and its user interface determine whether it looks like a dialog box.
- The use of two different methods (`Show` or `ShowModal`) to display the second form determines its behavior (modeless or modal).

Adding a Second Form to a Program

To add a second form to an application, you simply click on the New Form button on the Delphi toolbar or use the File ➢ New Form menu command. As an alternative you can select File ➢ New, move to the Forms or Dialogs page, and choose one of the available form templates or form wizards.

If you have two forms in a project, you can use the Select Form or the Select Unit button of the Delphi toolbar to navigate through them at design time. You can also choose which form is the main one and which forms should be automatically created at start-up using the Forms page of the Project Options dialog box. This information is reflected in the source code of the project file.

---

note Secondary forms are automatically created in the project source-code file depending on a new Delphi 5 setting, which is the Auto Create Forms check box of the Preferences page of the Environment Options dialog box. Although automatic creation is the simplest and most reliable approach for novice developers and quick-and-dirty projects, I suggest that you disable this check box for any serious development. When your application contains hundreds of forms, you really shouldn't have them all created at application start-up. Create instances of secondary forms when and where you need them, and free them when you're done.

Once you have prepared the secondary form, you can simply set its `Visible` property to `True`, and both forms will show up as the program starts. In general, the secondary forms of an application are left "invisible" and are then displayed by calling the `Show` method (or setting the `Visible` property at run time). If you use the

`Show` function, the second form will be displayed as modeless, so you can move back to the first one while the second is still visible. To close the second form, you might use its system menu or click a button or menu item that calls the `Close` method. As we saw in Chapter 6, the default close action (see the `OnClose` event) for a secondary form is simply to hide it, so the secondary form is not destroyed when it is closed. It is kept in memory (again, not always the best approach) and is available if you want to show it again[198].

Creating Secondary Forms at Run Time

Unless you create the forms when the program starts, you'll need to check whether a form exists and create it if necessary. The simplest case is when you want to create multiple copies of the same form at run time. In the MultiWin example, I've done this by writing the following code:

```
procedure TForm1.btnMultipleClick(Sender: TObject);
begin
  with TForm3.Create (Application) do
    Show;
end;
```

Every time you click the button, a new copy of the form is created. Notice that I don't use the `Form3` global variable, because it doesn't make much sense to assign this variable a new value every time you create a new form object. The important thing, however, is not to refer to the global `Form3` object in the code of the form itself or in other portions of the application. The `Form3` variable, in fact, will invariably be a pointer to `nil`, so you should actually remove it from the unit to avoid any confusion.

---

note In the code of a form, you should never explicitly refer to the form by using the global variable that Delphi sets up for it. For example, suppose that in the code of `TForm3` you refer to `Form3.Caption`. If you create a second object of the same type (the class `TForm3`), the expression `Form3.Caption` will invariably refer to the caption of the form object referenced by the `Form3` variable, which might not be the current object executing the code. To avoid this problem, refer to the `Caption` property in the form's method to indicate the caption of the current form object, and use the `Self` keyword when you need a specific reference to the object of the current form. To avoid any problem when creating multiple copies of a form, I suggest removing the global form object from the interface portion of the unit declaring the form. This global variable is required only for the automatic form creation.

---

198 All of this still applies 100% today.

When you create multiple copies of a form dynamically, remember to destroy each form object as is it closed, by handling the corresponding event:

```
procedure TForm3.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
end;
```

Failing to do so will result in a lot of memory consumption, because all the forms you create (both the windows and the Delphi objects) will be kept in memory and simply hidden from view.

Now let us focus on the dynamic creation of a form, in a program that accounts for only one copy of the form at a time. Creating a modal form is quite simple, because the dialog box can be destroyed when it is closed, with code like this:

```
procedure TForm1.btnModalClick(Sender: TObject);
var
  Modal: TForm4;
begin
  Modal := TForm4.Create (Application);
  try
    Modal.ShowModal;
  finally
    Modal.Free;
  end;
end;
```

Because the `ShowModal` call can raise an exception, you should write it in a `finally` block to make sure the object will be deallocated. Usually this block also includes code that initializes the dialog box before displaying it and code that extracts the values set by the user before destroying the form. The final values are read-only if the result of the `ShowModal` function is `mrOK`, as we'll see in the next example.

The situation is a little more complex when you want to display only one copy of a modeless form. In fact, you have to create the form, if it is not already available, and then show it:

```
procedure TForm1.btnSingleClick(Sender: TObject);
begin
  if not Assigned (Form2) then
    Form2 := TForm2.Create (Application);
  Form2.Show;
end;
```

With this code the form is created the first time it is required and then is kept in memory, visible on the screen or hidden from view. To avoid using up memory and

system resources unnecessarily, you'll want to destroy the secondary form when it is closed. You can do that by writing a handler for the `OnClose` event:

```
procedure TForm2.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
  // important: set pointer to nil!
  Form2 := nil;
end;
```

Notice that after we destroy the form, the global `Form2` variable is set to `nil`. Without this code, closing the form would destroy its object, but the `Form2` variable would still refer to the original memory location. At this point, if you try to show the form once more with the `btnSingleClick` method shown earlier, the `if not Assigned` test will succeed, as it simply checks whether the `Form2` variable is `nil`. The code fails to create a new object, and the `Show` method, invoked on a nonexistent object, will result in a system memory error.

As an experiment, you can generate this error by removing the last line of the listing above. As we have seen, the solution is to set the `Form2` object to `nil` when the object is destroyed, so that properly written code will "see" that a new form has to be created before using it. Again, experimenting with the MultiWin example can prove useful to test various conditions. I haven't illustrated any screens from this example because the forms it displays are quite bare (totally empty except for the main form, which has three buttons).

note Setting the form variable to `nil` makes sense—and works—if there is to be only one instance of the form present at any given instant. If you want to create multiple copies of a form, you'll have to use other techniques to keep track of them. Also keep in mind that in this case we cannot use the new Delphi 5 `FreeAndNil` procedure, because we cannot call Free on `Form2`. The reason is that we cannot destroy the form before its event handlers have finished executing.

Merging Form Menus

Another feature of modeless forms is worth mentioning. Although every form of an application can have its own menu bar, you can also use Delphi's menu merging technique to move the items of the secondary form's menu to the main form's menu bar. This technique is very useful in MDI applications but less interesting for modeless forms, as this behavior can confuse the user.

In this technique, the application's main window has a menu bar, as usual. The other forms have a menu bar with the `AutoMerge` property enabled, so their menu bar won't be displayed in the form but will instead be merged with the one from the main window. These are the rules for menu merging: Each pull-down menu has a `GroupIndex` property. When menu bars are merged, the pull-down menus are arranged as follows:

- If two elements of the different menu bars have the same `GroupIndex`, those of the original menu are removed.
- Elements are ordered by ascending `GroupIndex` values.

Creating a Dialog Box

I stated earlier in this chapter that a dialog box is not very different from other forms. There is a very simple trick to build a dialog box instead of a form. Just select the `bsDialog` value for the form's `BorderStyle` property. With this simple change, the interface of the form becomes like that of a dialog box, with no system icon, no Minimize or Maximize boxes, and a system menu you can activate by right-clicking over the caption. Of course, such a form has the typical thick dialog box border, which is nonresizable.

Once you have built a dialog box form, you can display it as a modal or modeless window using the two usual show methods (`Show` and `ShowModal`). Modal dialog boxes, however, are more common than modeless ones. This is exactly the reverse of forms; modal forms should generally be avoided since a user won't expect them. The following table lists the complete schema of the various combinations of styles:

Window Type	Modal	Modeless
Form	Never used	Usual, in SDI applications
Dialog box	Most common kind of secondary form	Used, but not very common

To avoid using too many secondary forms, you can build multipage forms, as discussed later in this chapter. Another alternative is to use MDI forms, also covered later in this chapter.

The Dialog Box of the RefList Example

In Chapter 5 we explored the RefList program, which used a ListView control to display references to books, magazines, Web sites, and more. In the RefList2 version

I'll simply add to the basic version of that program a dialog box, used in two different circumstances: adding new items to the list and editing existing items. You can see the form of the dialog box in Figure 8.1 and its textual description in the following listing (detailed because it has many interesting features, so I suggest you read this code with care):

```
object FormItem: TFormItem
  Caption = 'Item'
  Color = clBtnFace
  Position = poScreenCenter
  object Label1: TLabel
    Caption = '&Reference:'
    FocusControl = EditReference
  end
  object EditReference: TEdit...
  object Label2: TLabel
    Caption = '&Type:'
    FocusControl = ComboType
  end
  object ComboType: TComboBox
    Style = csDropDownList
    Items.Strings = (
      'Book'
      'CD'
      'Magazine'
      'Mail Address'
      'Web Site')
  end
  object Label3: TLabel
    Caption = '&Author:'
    FocusControl = EditAuthor
  end
  object EditAuthor: TEdit...
  object Label4: TLabel
    Caption = '&Country:'
    FocusControl = EditCountry
  end
  object EditCountry: TEdit...
  object BitBtn1: TBitBtn
    Kind = bkOK
  end
  object BitBtn2: TBitBtn
    Kind = bkCancel
  end
end
```

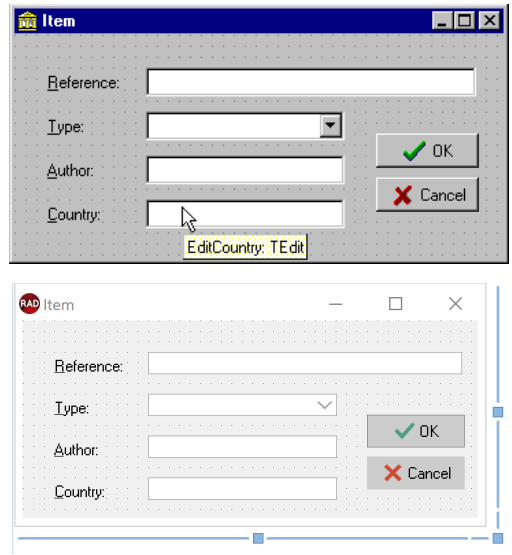


Figure 8.1: The form of the dialog box of the RefList2 example at design time. Images captured in Delphi 5 and Delphi 12.

note The items of the combo box in this dialog describe the available images of the image list, so that a user can select the type of the item and the system will show the corresponding glyph. An even better option would have been to show those glyphs in the combo box, along with their descriptions.

As I mentioned, this dialog box is used in two different cases. The first takes place as the user selects File ➢ Add Items from the menu:

```
procedure TForm1.AddItems1Click(Sender: TObject);
var
  NewItem: TListItem;
begin
  FormItem.Caption := 'New Item';
  FormItem.Clear;
  if FormItem.ShowModal = mrOK then
  begin
    NewItem := ListView1.Items.Add;
    NewItem.Caption := FormItem.EditReference.Text;
    NewItem.ImageIndex := FormItem.ComboType.ItemIndex;
    NewItem.SubItems.Add (FormItem.EditAuthor.Text);
    NewItem.SubItems.Add (FormItem.EditCountry.Text);
  end;
end;
```

Besides setting the proper caption of the form, this procedure needs to initialize the dialog box, as we are entering a brand-new value. If the user clicks OK, however, the

program adds a new item to the list view and sets all its values. To empty the edit boxes of the dialog, the program calls the custom `Clear` method, which resets the text of each edit box control:

```
procedure TFormItem.Clear;
var
  I: Integer;
begin
  // clear each edit box
  for I := 0 to ControlCount - 1 do
    if Controls [I] is TEdit then
      TEdit (Controls[I]).Text := '';
end;
```

Editing an existing item requires a slightly different approach. First, the current values are moved to the dialog box before it is displayed. Second, if the user clicks OK, the program modifies the current list item instead of creating a new one. Here is the code:

```
procedure TForm1.ListView1DblClick(Sender: TObject);
begin
  if ListView1.Selected <> nil then
  begin
    // dialog initialization
    FormItem.Caption := 'Edit Item';
    FormItem.EditReference.Text := ListView1.Selected.Caption;
    FormItem.ComboType.ItemIndex := ListView1.Selected.ImageIndex;
    FormItem.EditAuthor.Text := ListView1.Selected.SubItems [0];
    FormItem.EditCountry.Text := ListView1.Selected.SubItems [1];

    // show it
    if FormItem.ShowModal = mrOK then
    begin
      // read the new values
      ListView1.Selected.Caption := FormItem.EditReference.Text;
      ListView1.Selected.ImageIndex := FormItem.ComboType.ItemIndex;
      ListView1.Selected.SubItems [0] := FormItem.EditAuthor.Text;
      ListView1.Selected.SubItems [1] := FormItem.EditCountry.Text;
    end;
  end;
end;
```

You can see the effect of this code in Figure 8.2. Notice that the code used to read the value of a new item or modified one is similar. In general, you should try to avoid this type of duplicated code and possibly place the shared code statements in a method added to the dialog box. In this case, the method could receive as parameter a `TListItem` object and copy the proper values into it.

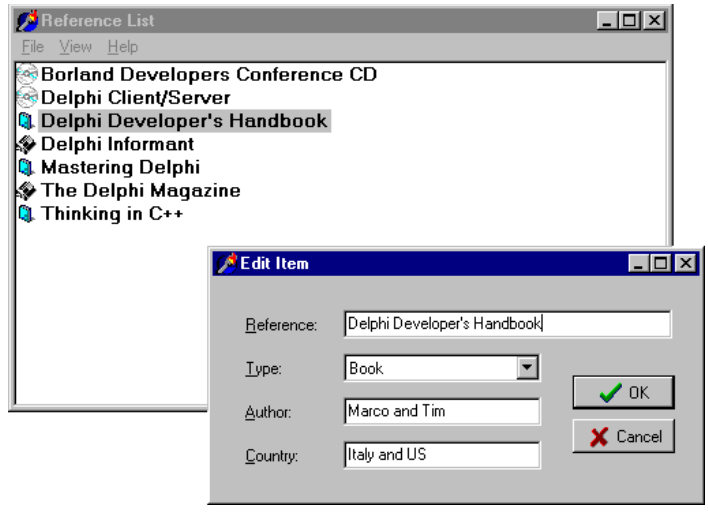


Figure 8.2: The dialog box of the RefList2 example used in edit mode

note What happens internally when the user clicks on the OK or Cancel buttons of the dialog box? A modal dialog box is closed by setting its `ModalResult` property, and it returns the value of this property. You can indicate the return value by setting the `ModalResult` property of the button. When the user clicks on the button, its `ModalResult` value is copied to the form, which closes the form and returns the value as the result of the `ShowModal` function.

A Modeless Dialog Box

The second example of dialog boxes shows a more complex modal dialog box that uses the standard approach as well as a modeless dialog box. The main form of the DlgApply example has five labels with names, as you can see in Figure 8.3 and by viewing the source code you've downloaded.

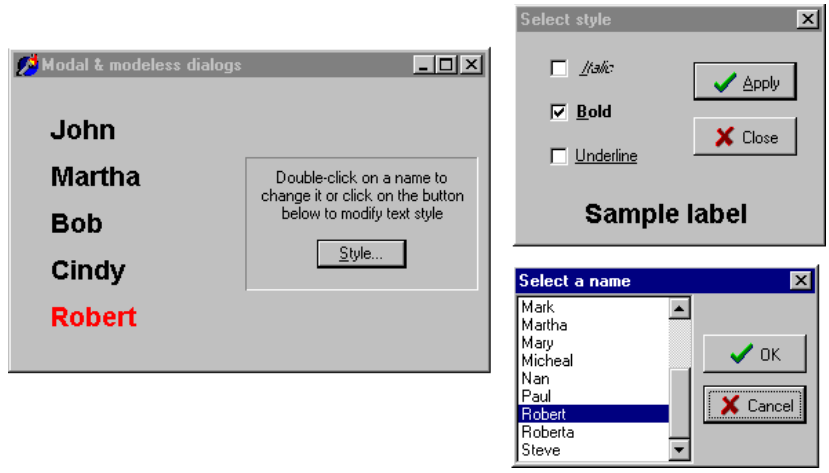


Figure 8.3: The three forms (a main form and two dialog boxes) of the DlgApply example at run time. Images from the original book.

If the user clicks on a name, its color turns to red; if the user double-clicks on it, the program displays a modal dialog box with a list of names to choose from. If the user clicks on the Style button, a modeless dialog box appears, allowing the user to change the font style of the main form's labels. The five labels of the main form are connected to two methods, one for the `OnClick` event and the second for the `OnDoubleClick` event. The first method turns the last label a user has clicked on to red, resetting to black all the others (which have the `Tag` property set to `1`, as a sort of group index). Notice that the same method is associated with all of the labels:

```
procedure TForm1.LabelClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ComponentCount - 1 do
   if (Components[I] is TLabel) and
       (Components[I].Tag = 1) then
     TLabel (Components[I]).Font.Color := clBlack;
  // set the color of the clicked label to red
  (Sender as TLabel).Font.Color := clRed;
end;
```

The second method common to all of the labels is the handler of the `OnDoubleClick` event. The `LabelDoubleClick` method selects the `Caption` of the current label (indicated by the `Sender` parameter) in the list box of the dialog and then shows the modal dialog box. If the user closes the dialog box by clicking on OK and an item of the list is selected, the selection is copied back to the label's caption:

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

```
begin
  with ListDial.Listbox1 do
  begin
    // select the current name in the list box
    ItemIndex := Items.IndexOf (Sender as TLabel).Caption);
    // show the modal dialog box, checking the return value
    if (ListDial.ShowModal = mrOk) and (ItemIndex >= 0) then
      // copy the selected item to the label
      (Sender as TLabel).Caption := Items [ItemIndex];
end;
```

note Notice that all the code used to customize the modal dialog box is in the `LabelDoubleClick` method of the main form. The form of this dialog box has no added code.

The modeless dialog box, by contrast, has a lot of coding behind it. The main form simply displays the dialog box when the Style button is clicked (notice that the button caption ends with three dots to indicate that it leads to a dialog box), by callings its `Show` method. You can see the dialog box running in Figure 8.3 above.

Two buttons, Apply and Close, replace the OK and Cancel buttons in a modeless dialog box. (The fastest way to obtain these buttons is to select the `bkOK` or `bkCancel` value for the `Kind` property and then edit the `Caption`.) At times, you may see a Cancel button that works as a Close button, but the OK button in a modeless dialog box usually has no meaning. Instead, there might be one or more buttons that perform specific actions on the main window, such as Apply, Change Style, Replace, Delete, and so on.

If the user clicks on one of the check boxes of this modeless dialog box, the style of the sample label's text at the bottom changes accordingly. You accomplish this by adding or removing the specific flag that indicates the style, as in the following `OnClick` event handler:

```
procedure TStyleDial.ItalicCheckBoxClick(Sender: TObject);
begin
  if ItalicCheckBox.Checked then
    LabelSample.Font.Style :=
      LabelSample.Font.Style + [fsItalic]
  else
    LabelSample.Font.Style :=
      LabelSample.Font.Style - [fsItalic];
end;
```

When the user selects the Apply button, the program copies the style of the sample label to each of the form's labels, rather than considering the values of the check boxes:

```
procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);
```

```
begin
  Form1.Label1.Font.Style := LabelSample.Font.Style;
  Form1.Label2.Font.Style := LabelSample.Font.Style;
  ...
```

As an alternative, instead of referring to each label directly, you can look for it by calling the `FindComponent` method of the form, passing the label name as a parameter, and then casting the result to the `TLabel` type. The advantage of this approach is that we can create the names of the various labels with a `for` loop:

```
procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 5 do
    (Form1.FindComponent ('Label' + IntToStr (I)) as TLabel).
       Font.Style := LabelSample.Font.Style;
end;
```

note The `ApplyBitBtnClick` method could also be written by scanning the `Controls` array in a loop, as I've already done in other examples. I decided to use the `FindComponent` method, instead, to show you a new technique.

This second version of the code is certainly slower, because it has more operations to do, but you won't notice the difference, because it is very fast anyway. Of course, this second approach is also more flexible; if you add a new label, you only need to fix the higher limit of the `for` loop, provided all the labels have consecutive numbers. Notice that when the user clicks on the Apply button, the dialog box does not close. Only the Close button has this effect. Consider also that this dialog box needs no initialization code because the form is not destroyed, and its components maintain their status each time the dialog box is displayed.

Windows Common Dialogs

Besides building your own dialog boxes, Delphi allows you to use some default dialog boxes of different kinds. Some are predefined by Windows, others are simple dialog boxes (such as message boxes) displayed by a Delphi routine. The Delphi Component Palette contains a page of dialog box components. Each of these dialog

boxes—known as *Windows common dialogs*—is defined in the system library `ComDlg32.DLL`[199].

I have already used some of these dialog boxes in several examples in the previous chapters, so you are probably familiar with them. Basically, you need to put the corresponding component on a form, set some of its properties, run the dialog box (with the `Execute` method, returning a `Boolean` value), and retrieve the properties that have been set while running it. To help you experiment with these dialog boxes, I've built the CommDlg test program. I won't discuss the program in detail nor show its simple but lengthy source code in the book. As always, you can find this code among the downloaded files.

What I want to do is simply highlight some key and nonobvious features of the common dialog boxes, and let you study the source code of the example for the details:

- The Open Dialog Component[200] can be customized by setting different file extensions filters, using the `Filter` property, which has a handy editor and can be assigned directly with a string like `Text File (*.txt)|*.txt`. Another handy feature is to let the dialog check whether the extension of the selected file matches the default extension, by checking the `ofExtensionDifferent` flag of the `Options` property after executing the dialog. Finally, this dialog allows multiple selections by setting its `ofAllowMultiSelect` option. In this case you can get the list of the selected files by looking at the `Files` string list property.
- The SaveDialog component is used in similar ways and has similar properties, although you cannot select multiple files, of course.
- The OpenPictureDialog and SavePictureDialog components provide similar features but have a customized form, which shows a preview of an image. Of course, it makes sense to use them only for opening or saving graphical files.
- The FontDialog component can be used to show and select from all types of fonts, fonts useable on both the screen and a selected printer (wysiwyg), or only TrueType fonts. You can show the portion related to the special effects or hide it,

---

199 Oddly, this is still the name of the library even in the 64-bit version of Windows. All of the ideas and most of the code in this ebook should equally apply to the Win64 target that Delphi offers today. For non-Windows targets, instead, you cannot use the VCL library, but have to switch to the similar FireMonkey library.

200 There are new versions of these dialog boxes available in Windows. They can be enable in Delphi by using Windows themes (the default for new applications), however some of the extended features are available only when using the newer FileOpenDialog and FileSaveDialog components. There is also a new TaskDialog now available in the VCL, mapped to another relatively new Windows API.

and obtain other different versions by setting its `Options` property. You can also activate an Apply button simply by providing an event handler for its `OnApply` event and using the `fdApplyButton` option. A Font dialog box with an Apply button (see Figure 8.4) behaves almost like a modeless dialog box (but isn't one).

- The ColorDialog component is used with different options, to show the dialog fully open at first or to prevent it from opening fully. These settings are the `cdFullOpen` or `cdPreventFullOpen` values of the `Options` property.

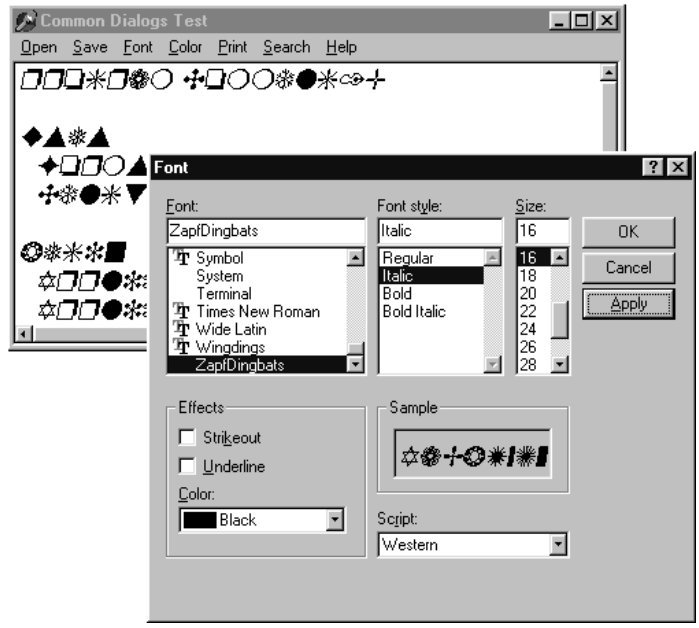


Figure 8.4: The Font selection dialog box with an Apply button. Image from the original book.

- The Find and Replace dialog boxes are truly modeless dialogs, but you have to implement the find and replace functionality yourself, as I've partially done in the CommDlg example. The custom code is connected to the buttons of the two dialog boxes by providing the `OnFind` and `OnReplace` events.

A Parade of Message Boxes

The Delphi message boxes and input boxes are another set of predefined dialog boxes. There are basically six Delphi procedures and functions you can use to display simple dialog boxes[201]:

- The `MessageDlg` function shows a customizable message box, with one or more buttons and usually a bitmap. We have used this function quite often in previous examples.
- The `MessageDlgPos` function is similar to the `MessageDlg` function. The difference is that the message box is displayed in a given position, not in the center of the screen.
- The `ShowMessage` procedure displays a simpler message box, with the application name as the caption, and just an OK button. The `ShowMessageFmt` procedure is a variation of `ShowMessage`, which has the same parameters as the `Format` function. It corresponds to calling `Format` inside a call to `ShowMessage`.
- The `ShowMessagePos` procedure does the same, but you also indicate the position of the message box.
- The `MessageBox` method of the `Application` object allows you to specify both the message and the caption; you can also provide various buttons and features. This is a simple and direct encapsulation of the `MessageBox` function of the Windows API, which passes as a main window parameter the handle of the `Application` object. This handle is required to make the message box behave like a modal window.
- The `InputBox` function asks the user to input a string. You provide the caption, the query, and a default string.

The `InputQuery` function asks the user to input a string, too. The only difference between this and the `InputBox` function is in the syntax. The `InputQuery` function has a `Boolean` return value that indicates whether the user has clicked on OK or Cancel.

To demonstrate some of the message boxes available in Delphi, I've written another sample program, with a similar approach to the preceding CommDlg example. In the MBParade example, you have a high number of choices (radio buttons, check boxes, edit boxes, and spin edit controls) to set before you press one of the buttons

201 As already mentioned in a previous note, there is now also a TaskDialog component in the VCL, mapped to a specific, relatively new, Windows API.

that displays a message box. You can get a better idea of the program by looking at its form in Figure 8.5.

Expandable Dialog Boxes

Some dialog boxes display a number of components for the user to work with. At times, you can divide them into logical pages, which Delphi supports through the PageControl component (discussed later in this chapter). At other times, you can temporarily hide some dialog box controls to help first-time users of your application. Another alternative is to increase the size of the dialog box to host new controls when the user presses a More button[202].

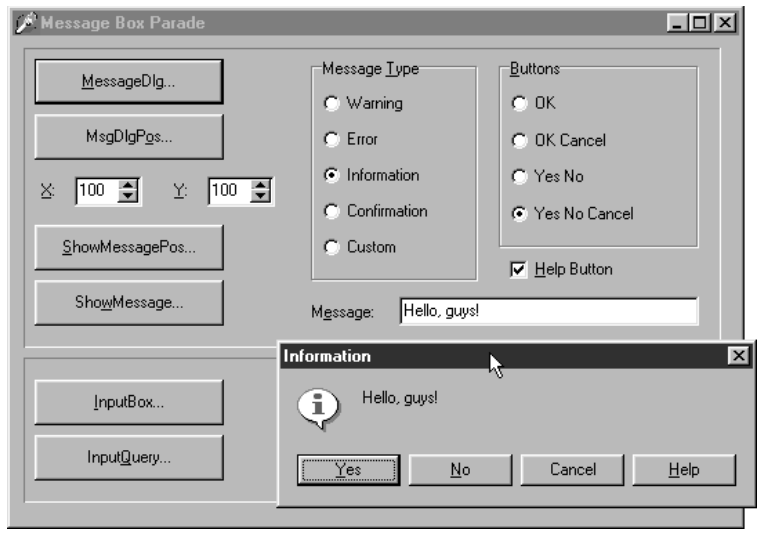


Figure 8.5: The main form of the MBParade example, with a sample message box. Image from the original book.

I'll use this approach to create the simple dialog box in the More example. First of all, we need to create the dialog box and add some simple controls, a More button (see Figure 8.6), and two check boxes labeled *italic* and *bold,* which are in a panel placed outside the design-time surface of the form. In practice, once you have added the panel with some controls in it, you need to resize the dialog box so that the new panel is outside the visible surface of the form and set the `AutoScroll` property of the form to `False`. The panel is not visible, because I've removed its borders, and

202 I have to admin this UI style of expanding the size of a dialog box is way less frequent today.

makes the program more flexible, as you can add more controls to the hidden portion of the dialog without changing the source code: simply place them on the panel.

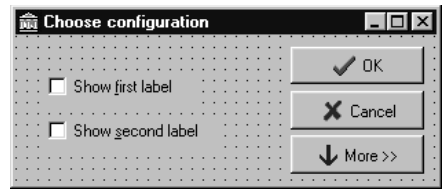


Figure 8.6: The dialog box of the More example at design time. Some of the components are invisible because they are beyond the border. Image from the original book.

The panel should be hidden; otherwise, the user might press the Tab key and move onto its controls even if they are not visible. As an alternative, you might disable its `TabStop` property. These properties (`Visible` or `TabStop`) are then set to `True` when the form is enlarged.

Now, in addition to the standard code required to move values from the main form to the dialog, we need to write some code to resize the form when a user clicks on the More button. To prepare the resizing effect, we need a couple of fields in the form (named `OldHeight` and `NewHeight`) to store the two different heights of the client area of the form. We can set up their values when the form is first created:

```
procedure TConfigureDialog.FormCreate(Sender: TObject);
begin
  OldHeight := ClientHeight;
  NewHeight := PanelMore.Top + PanelMore.Height;
end;
```

I determined the new height by adding to the height of the panel its position. The real dialog box resizing takes place when the More button is pressed. Here is a first version:

```
procedure TConfigureDialog.btnMoreClick(Sender: TObject);
begin
  PanelMore.Visible := True;
  btnMore.Enabled := False;
  ClientHeight := NewHeight;
end;
```

The result it produces is shown in Figure 8.7. If you want a more spectacular effect, you might increase the height a pixel at a time instead of setting the final value at once. If you write a `for` loop, increasing the client height and repainting the form each time, the new controls will appear with a nice effect, only a little slower. The last line of the `btnMoreClick` method above becomes

```
for I := ClientHeight to NewHeight do
begin
  ClientHeight := I;
  Update;
end;
```

Each time the dialog box is activated (`OnFormActivate` event), we reset its height, hide the panel (to avoid letting the user Tab to its controls), and enable the More button:

```
procedure TConfigureDialog.FormActivate(Sender: TObject);
begin
  ClientHeight := OldHeight;
  btnMore.Enabled := True;
  PanelMore.Visible := False;
end;
```

This code is required so that each time the dialog box is displayed it starts in the default small configuration.

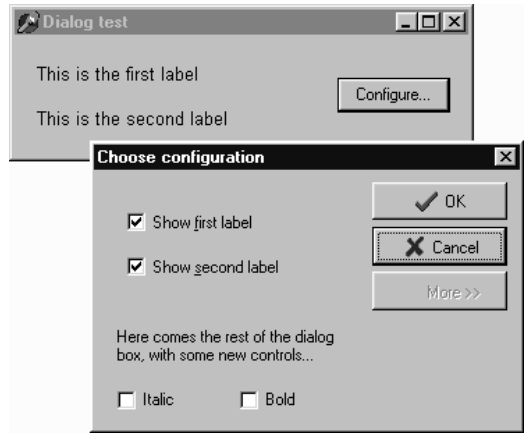


Figure 8.7: The dialog box of the More example after it has been resized. Image from the original book.

About Boxes and Splash Screens

Windows applications usually have an About box, where you can display information, such as the version of the product, a copyright notice, and so on. The simplest

way to build an About box is to use the `MessageDlg` function. With this method, you can show only a limited amount of text and no special graphics.

Therefore, the usual method for creating an About box is to use a simple dialog box, such as the one generated with one of the Delphi default templates. I say *simple* because when you have designed the form with a logo and so on, you seldom need much code. At most, some code might be required to display system information, such as the version of Windows or the amount of free memory, or some user information, such as the registered user name.

note In Chapter 19, we'll see how to create extract the version information from an executable file, which contains this type of Windows resources. This technique can be useful to build an About box that includes the version information.

Building a Custom Hidden Screen

While we build our own About box, we can add a hidden credit screen, which Delphi and many other applications have. You might want to add a hidden credit screen for a number of reasons. If you work in a big company, this might be your way to prove that you worked on that project, which might help you in finding a new job (if the project was successful). At times, a hidden About box can be fun to see, and they sometimes also provide a good occasion for making jokes about your competitors. A more serious reason is that a hidden credit screen can be used to demonstrate who wrote the program, as a sort of legal copyright.

I've written a simple example, showing how you might implement a hidden screen. The dialog box has a Panel component containing two Label components. The panel might contain any number of components to display graphics and text. Some of the strings might even be computed at run time. The only added feature required to show the hidden credits is a PaintBox component covering part of the form.

When the user makes a specific complex action (in this case, right-clicking on the upper label while holding down the Shift key), the panel is hidden and something appears on the screen. A simple solution is to have some text painted on the surface of the form—that is, on its canvas:

```
procedure TAboutBox.Label1MouseDown(Sender: TObject;
  Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if (Button = mbRight) and (ssShift in Shift) then
  begin
    Panel1.Visible := False;
```

```
    PaintBox1.Canvas.Font.Name := 'Arial';
    PaintBox1.Canvas.Font.Size := 20;
    PaintBox1.Canvas.TextOut (40, 50, 'Author: Marco Cantù');
    PaintBox1.Canvas.TextOut (40, 100, 'Version 1.0');
  end;
end;
```

To build a more spectacular hidden screen, we might scroll some text in a `for` loop, as I've done in the final version of the Credits example. Notice that the position of the lines depend on the height of the text, retrieved by calling the `TextHeight` method of the `Canvas` of the PaintBox component:

```
Panel1.Visible := False;
LineH := PaintBox1.Canvas.TextHeight ('0');
for I := 0 to 100 + LineH * 10 do
  with PaintBox1.Canvas do
  begin
    // empty lines are used to delete descendants
    TextOut (40, 100 - I, 'CREDITS example from:');
    TextOut (40, 100 + LineH - I, '"Mastering Delphi"');
    TextOut (40, 100 + LineH * 2 - I, '                          ');
    ...
    // wait 5 milliseconds
    Delay (0, 5);
  end;
Panel1.Visible := True;
```

To avoid a scrolling rate that's too fast, particularly on faster computers, inside the `for` loop I've added a call to a `Delay` procedure, which requires as parameters the seconds and milliseconds you want to wait for. This `Delay` procedure simply checks the current time and then waits in a `while` loop until the required seconds and milliseconds have elapsed:

```
procedure Delay (Seconds, MilliSec: Word);
var
  TimeOut: TDateTime;
begin
  TimeOut := Now + EncodeTime (0,
    Seconds div 60, Seconds mod 60, MilliSec);
  // wait until the TimeOut time
  while Now < TimeOut do
    Application.ProcessMessages;
end;
```

Inside the loop I call the `ProcessMessages` method of the `Application` global object to let Windows generate and dispatch the needed paint messages. This `Delay` procedure is a fairly generic one, so you can use it in other applications quite easily.

note Consider another aspect of the preceding example. We have written some code to draw on the surface of a dialog box. Although it is not very common, dialog boxes can have graphical output and respond to mouse input just like any other form. In fact, a dialog box *is a form*.

Building a Splash Screen

Another typical technique used in applications is to display an initial screen before the main form is shown. This makes the application seem more responsive, because you show something to the user while the program is loading, but it also makes a nice visual effect. Sometimes, this same window is displayed as the application's About box.

For an example in which a splash screen is particularly useful, I've built a program displaying a list box filled with prime numbers. The prime numbers are computed on program start-up, so that they are displayed as soon as the form becomes visible:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 20000 do
    if IsPrime (I) then
      ListBox1.Items.Add (IntToStr (I));
end;
```

This method calls an `IsPrime` function I've added to the program. This function, which you can find in the source code, computes prime numbers in a terribly slow way; but I needed a slow form creation to demonstrate my point. The numbers are added to a list box that covers the full client area of the form and allows multiple columns to be displayed, as you can see in Figure 8.8.

As you can see by running the Splash0 example, the problem with this program is that the initial operation, which takes place in the `FormCreate` method, takes a lot of time. When you start the program, it takes several seconds (on a standard Pentium machine[203]) to display the main form. If your computer is very fast or very slow, you can change the upper limit of the `for` loop of the `FormCreate` method to make the program faster or slower.

203 I know this sounds old, but that's the type of CPU in use at the time this book was originally written.

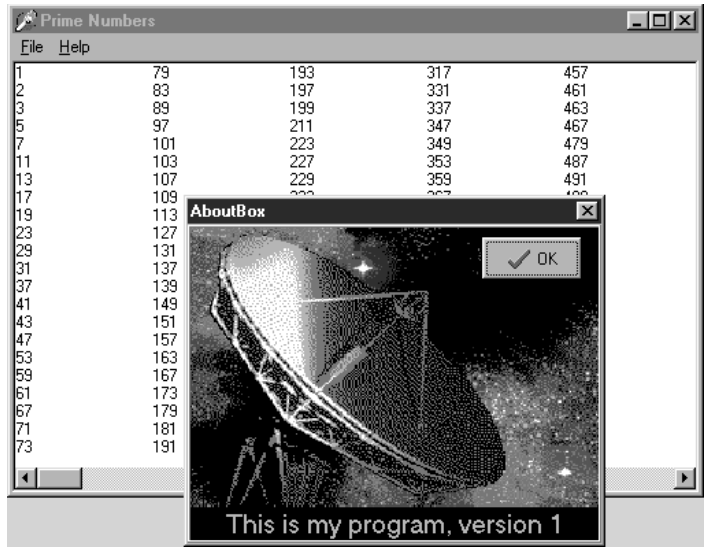


Figure 8.8: The main form of the Splash example, with the About box activated from the menu. Image from the original book.

This program has a simple dialog box with an image component, a simple caption, and a bitmap button, all placed inside a panel taking up the whole surface of the About box. This form is displayed when you select the Help ➢ About menu item. But what we really want is to display this About box while the program starts. You can see this effect by running the Splash1 and Splash2 examples, which show a splash screen using two different techniques.

First of all, I've added a method to the `TAboutBox` class. This method, called `MakeSplash`, changes some properties of the form to make it suitable for a splash form. Basically it removes the border and caption, hides the OK button, makes the border of the panel thick (to replace the border of the form), and then shows the form, repainting it immediately (see Figure 8.9 for the effect):

```
procedure TAboutBox.MakeSplash;
begin
  BorderStyle := bsNone;
  BitBtn1.Visible := False;
  Panel1.BorderWidth := 3;
  Show;
  Update;
end;
```

Figure 8.9: The form of the splash screen of the Splash1 example is slightly different than the original About box (shown in Figure 8.8). Image from the original book.

This method is called after creating the form in the project file of the Splash1 example. This code is executed before creating the other forms (in this case only the main form), and the splash screen is then removed before running the application. These operations take place within a `try-finally` block. Here is the source code of the main block of the project file for the Splash2 example:

```
var
  SplashAbout: TAboutBox;

begin
  Application.Initialize;

  // create and show the splash form
  SplashAbout := TAboutBox.Create (Application);
  try
    SplashAbout.MakeSplash;
    // standard code...
    Application.CreateForm(TForm1, Form1);
    // get rid of the splash form
    SplashAbout.Close;
  finally
    SplashAbout.Free;
  end;

  Application.Run;
end.
```

This approach makes sense only if your application's main form takes a while to create, to execute its start-up code (as in this case), or to open database tables. Notice that the splash screen is the first form created, but because the program doesn't use the `CreateForm` method of the `Application` object, this doesn't become the main form of the application. In this case, in fact, closing the splash screen would terminate the program!

An alternative approach is to keep the splash form on the screen a little longer and use a timer to get rid of it after a while. I've implemented this second technique in the Splash2 example. This example also uses a different approach for creating the splash form: instead of creating it in the project source code, it creates the form at the very beginning of the `FormCreate` method of the main form.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
  SplashAbout: TAboutBox;
begin
  // create and show the splash form
  SplashAbout := TAboutBox.Create (Application);
  SplashAbout.MakeSplash;
  // standard code...
  for I := 1 to 20000 do
    if IsPrime (I) then
      ListBox1.Items.Add (IntToStr (I));
  // get rid of the splash form, after a while
  SplashAbout.Timer1.Enabled := True;
end;
```

note This code works properly regardless of the form's creation order, as indicated by the `OldCreateOrder` property (discussed in Chapter 6).

The timer is enabled just before terminating the method. After its interval has elapsed (in the example, 3 seconds) the `OnTimer` event is activated, and the splash form handles it by closing and destroying itself:

```
procedure TAboutBox.Timer1Timer(Sender: TObject);
begin
  Close;
  Release;
end;
```

note The `Release` method of a form is similar to the `Free` method of objects, only the destruction of the form is delayed until all event handlers have completed execution. Using `Free` inside a form might cause an access violation, as the internal code, which fired the event handler, might refer again to the form object.

There is one more thing to fix. The Main form will be displayed later and in front of the splash form, unless you make this a topmost form. For this reason I've added one line to the `MakeSplash` method of the About box in the Splash2 example:

```
FormStyle := fsStayOnTop;
```

Multiple-Page Forms

When you have a lot of information and controls to display in a dialog box or a form, you can use multiple pages. The metaphor is that of a notebook: Using tabs, a user can select one of the possible pages.

There are two controls you can use to built a multiple-page application in Delphi[204]:

- You can use the Windows 95 PageControl component, which has tabs on one of the sides and multiple pages (similar to panels) covering the rest of its surface. As there is one page per tab, you can simply place components on each page to obtain the proper effect both at design time and at run time.
- You can use the TabControl, which has only the tab portion but offers no pages to host the information. In this case you'll want to use one or more components to mimic the *page change* operation.

A third related component, the TabSheet, represents a single page of the PageControl. This is not a stand-alone component and is not available on the Component palette. You create a TabSheet at design time by using the local menu of the PageControl or at run time by using methods of the same control.

note Delphi still includes the Notebook, TabSet, and TabbedNotebook components introduced in earlier versions. Use these components only if you need to create a 16-bit version of an application. For any other purpose, the PageControl and TabControl components, which encapsulate Win32 common controls, provide a more modern user interface. Actually, in 32-bit versions of Delphi, the TabbedNotebook component was reimplemented using the Win32 PageControl internally, to reduce the code size and update the look.

PageControls and TabSheets

As usual, instead of duplicating the Help system's list of properties and methods of the PageControl component, I've built an example that stretches its capabilities and allows you to change its behavior at run time. The example, called Pages, has a PageControl with three pages. The structure of the PageControl and of the other key components is listed below:

204 This is still true today, although Delphi 12 added a new metaphor for hosting MDI or regular forms within a tab-based UI. The new component is called FormTabsBar and it offers a lot of power while simplifying the coding required in VCL to host forms in tabs.

```
object Form1: TForm1
  BorderIcons = [biSystemMenu, biMinimize]
  BorderStyle = bsSingle
  Caption = 'Pages Test'
  OnCreate = FormCreate
  object PageControl1: TPageControl
    ActivePage = TabSheet1
    Align = alClient
    HotTrack = True
    Images = ImageList1
    MultiLine = True
    object TabSheet1: TTabSheet
      Caption = 'Pages'
      object Label3: TLabel
      object ListBox1: TListBox
    end
    object TabSheet2: TTabSheet
      Caption = 'Tabs Size'
      ImageIndex = 1
      object Label1: TLabel
      // other controls
    end
    object TabSheet3: TTabSheet
      Caption = 'Tabs Text'
      ImageIndex = 2
      object Memo1: TMemo
        Anchors = [akLeft, akTop, akRight, akBottom]
        OnChange = Memo1Change
      end
      object BitBtnChange: TBitBtn
        Anchors = [akTop, akRight]
        Caption = '&Change'
      end
    end
  end
  object BitBtnPrevious: TBitBtn
    Anchors = [akRight, akBottom]
    Caption = '&Previous'
    OnClick = BitBtnPreviousClick
  end
  object BitBtnNext: TBitBtn
    Anchors = [akRight, akBottom]
    Caption = '&Next'
    OnClick = BitBtnNextClick
  end
  object ImageList1: TImageList
    Bitmap = {...}
  end
end
```

Notice that the tabs are connected to the bitmaps provided by an ImageList control and that some controls use the `Anchors` property to remain at a fixed distance from

the right or bottom borders of the form. Even if the form doesn't support resizing (this would have been far too complex to set up with so many controls), the positions can change when the tabs are displayed on multiple lines (simply increase the length of the captions) or on the left side of the form.

Each TabSheet object has its own `Caption`, which is displayed as the sheet's tab. At design time you can use the local menu to create new pages and to move between pages. You can see the local menu of the PageControl component in Figure 8.10, together with the first page. This page holds a list box and a small caption, and it shares two buttons with the other pages.

If you place a component on a page, it is available only in that page. How can you have the same component (in this case, two bitmap buttons) in each of the pages, without duplicating it? Simply place the component on the form, outside of the PageControl (or before aligning it to the client area) and then move it in front of the pages, calling the Bring to Front command of the form's local menu. The two buttons I've placed in each page can be used to move back and forth between the pages and are an alternative to using the tabs. Here is the code associated with one of them:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
  PageControl1.SelectNextPage (True);
end;
```

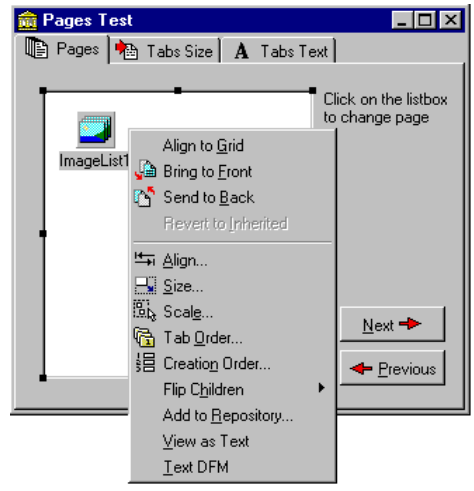


Figure 8.10: The first sheet of the PageControl of the Pages example, with its local menu. Image from the original book.

The other button calls the same procedure, passing `False` as its parameter to select the previous page. Notice that there is no need to check whether we are on the first or last page, because the `SelectNextPage` method considers the last page to be the one before the first and will move you directly between those two pages.

Now we can focus on the first page again. It has a list box, which at run time will hold the names of the tabs. If a user clicks on an item of this list box, the current page changes. This is the third method available to change pages (after the tabs and the Next and Previous buttons). The list box is filled in the `FormCreate` method, which is associated with the `OnCreate` event of the form and copies the caption of each page (the `Page` property stores a list of TabSheet objects):

```
for I := 0 to PageControl1.PageCount - 1 do
  ListBox1.Items.Add (PageControl1.Pages.Caption);
```

When you click on a list item, you can select the corresponding page:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  PageControl1.ActivePage :=
    PageControl1.Pages [ListBox1.ItemIndex];
end;
```

The second page hosts two edit boxes (connected with two UpDown components), two check boxes, and two radio buttons, as you can see in Figure 8.11. The user can input a number (or choose it by clicking on the up and down buttons with the mouse or pressing ↑ or ↓ while the corresponding edit box has the focus), check the boxes and the radio buttons, and then press the Apply button to make the changes:

```
procedure TForm1.BitBtnApplyClick(Sender: TObject);
begin
  // set tab width, height, and lines
  PageControl1.TabWidth := StrToInt (EditWidth.Text);
  PageControl1.TabHeight := StrToInt (EditHeight.Text);
  PageControl1.MultiLine := CheckBoxMultiLine.Checked;
  // show or hide the last tab
  TabSheet3.TabVisible := CheckBoxVisible.Checked;
  // set the tab position
  if RadioButton1.Checked then
    PageControl1.TabPosition := tpTop
  else
    PageControl1.TabPosition := tpLeft;
end;
```

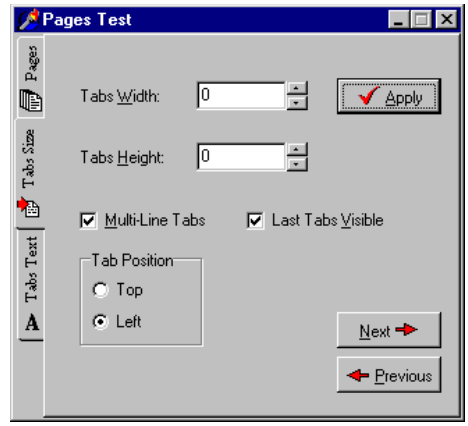


Figure 8.11: The second page of the example can be used to size and position the tabs. Here you can see the tabs on the left of the page control. Image from the original book.

With this code, we can change the width and height of each tab (remember that `0` means the size is computed automatically from the space taken by each string), choose to have either multiple lines of tabs or two small arrows to scroll the tab area, and move them to the left side. The control also allows tabs to be placed on the bottom or on the right; but our program doesn't allow that, because it would make the placement of the other controls quite complex.

You can also hide the last tab on the PageControl, which corresponds to the `TabSheet3` component. If you hide one of the tabs by setting its `TabVisible` property to `False`, you cannot reach that tab by clicking on the Next and Previous buttons, which are based on the `SelectNextPage` method. Instead, you should use the `FindNextPage` function, as shown below in this new version of the Next button's `OnClick` event handler:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
  PageControl1.ActivePage :=
    PageControl1.FindNextPage (
      PageControl1.ActivePage, True, False);
end;
```

The last page has a memo component, again with the names of the pages (added in the `FormCreate` method). You can edit the names of the pages and press the Change button to change the text of the tabs, but only if the number of strings matches the number of tabs:

```
procedure TForm1.BitBtnChangeClick(Sender: TObject);
var
  I: Integer;
begin
```

```
  if Memo1.Lines.Count <> PageControl1.PageCount then
    MessageDlg ('One line per tab, please', mtError, [mbOK], 0)
  else
    for I := 0 to PageControl1.PageCount -1 do
      PageControl1.Pages [I].Caption := Memo1.Lines [I];
  BitBtnChange.Enabled := False;
end;
```

Finally the last button, Add Page, allows you to add a new tab sheet to the page control, although the program doesn't add any components to it. The (empty) tab sheet object is created using the page control as its owner, but it won't work unless you also set the `PageControl` property. Before doing this, however, you should make the new tab sheet visible. Here is the code:

```
procedure TForm1.BitBtnAddClick(Sender: TObject);
var
  strCaption: string;
  NewTabSheet: TTabSheet;
begin
  strCaption := 'New Tab';
  if InputQuery ('New Tab', 'Tab Caption', strCaption) then
  begin
    // add a new empty page to the control
    NewTabSheet := TTabSheet.Create (PageControl1);
    NewTabSheet.Visible := True;
    NewTabSheet.Caption := strCaption;
    NewTabSheet.PageControl := PageControl1;
    PageControl1.ActivePage := NewTabSheet;
    // add it to both lists
    Memo1.Lines.Add (strCaption);
    ListBox1.Items.Add (strCaption);
  end;
end;
```

note Whenever you write a form based on a PageControl, remember that the first page displayed at run time is the page you were in before the code was compiled. This means that if you are working on the third page and then compile and run the program, it will start with that page. A common way to solve this problem is to add a line of code in the `FormCreate` method to set the PageControl or notebook to the first page. This way, the current page at design time doesn't determine the initial page at run time.

Frames and Pages

When you have a dialog box with many pages full of controls, the code underlying the form becomes very complex because all the controls and methods are declared

in a single form. Also, creating all these components (and initializing them) might result in a delay in the display of the dialog box.

The availability of frames in Delphi 5 (see Chapters 1 and 4) can solve both of these issues. First, you can easily divide the code of a single complex form into one frame per page. The form will simply host all of the frames in a PageControl. This certainly helps you to have simpler and more focused units and makes it simpler to reuse a specific page in a different dialog box or application. Reusing a single page of a PageControl without using a frame or an embedded form, in fact, is far from simple.

As an example of this approach I've built the FramePag example, which has some frames placed inside the three pages of a PageControl, as you can see in Figure 8.12. All of the frames are aligned to the client area, using the entire surface of the tab sheet (the page) hosting them[205].

Actually two of the pages have the same frame, but the two instances of the frame have some differences at design time. The frame, called `Frame3` in the example, has a list box that is populated with a text file at start up, has buttons to modify the items in the list and saves them to a file. The filename is placed inside a label, so that you can easily select a file for the frame at design time by changing the `Caption` of the label.

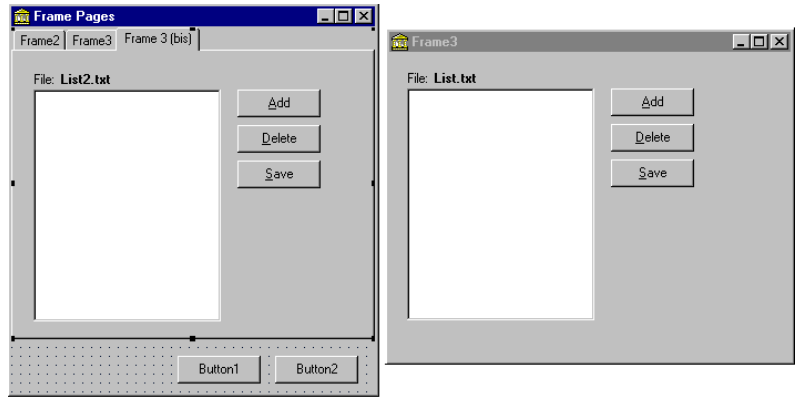


Figure 8.12: Each page of the FramePag example contains a frame, thus separating the code of this complex form into more manageable chunks. Images from the original book.

205 The new component I mentioned earlier, FormTabsBar, offers a similar architecture based on the use of regular forms, rather than frames. The result is similar in the two cases, although I personally tend to prefer using forms for tabbed applications when possible, because forms have a few extra methods and events that can be quite handy..

note Being able to use multiple instances of a frame is one of the reasons this technique was introduced, and customizing the frame at design time is even more important. Because adding properties to a frame and making them available at design time requires some customized and complex code, it is nice to use a component to host these custom values. You have the option of hiding these components (such as the label in our example) if they don't pertain to the user interface.

In the example, we need to load the file when the frame instance is created. Because frames have no `OnCreate` event, our best choice is probably to override the `CreateWnd` method. Writing a custom constructor, in fact, doesn't work as it is executed too early—before the specific label text is available. Here is the frame class code:

```
type
  TFrame3 = class(TFrame)
    ...
  public
    procedure CreateWnd; override;
```

Within the `CreateWnd` method, we simply load the list box content from a file.

Multiple Frames with No Pages

Another approach is to avoid creating all of the pages along with the form hosting them. This can be accomplished by leaving the PageControl empty and creating the frames only when a page is displayed. Actually, when you have frames on multiple pages of a PageControl, the windows for the frames are created only when they are first displayed, as you can find out by placing a breakpoint in the creation code of the last example.

As an even more radical approach, you can get rid of the page controls and use a TabControl. Used this way, the tab has no connected tab sheets (or pages) but can display only one information at a time. For this reason, we'll need to create the current frame and destroy the previous one or simply hide it by setting its `Visible` property to `False` or by calling the `BringToFront` of the new frame. Although this sounds like a lot of work, in a large application this technique can be worth it for the reduced resource and memory usage you can obtain by applying it.

To demonstrate this approach, I've built an example similar to the previous one, this time based on a TabControl and dynamically created frames. The main form, visible at run time in Figure 8.13, has only a TabControl with one page for each frame:

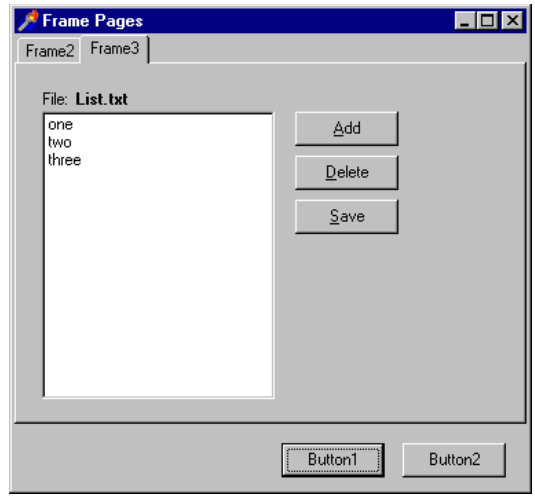


Figure 8.13: The first page of the FrameTab example at run time. The frame inside the tab is created at run time. Image from the original book.

```
object Form1: TForm1
  Caption = 'Frame Pages'
  OnCreate = FormCreate
  object Button1: TButton...
  object Button2: TButton...
  object Tab: TTabControl
    Anchors = [akLeft, akTop, akRight, akBottom]
    Tabs.Strings = (
      'Frame2'
      'Frame3')
    OnChange = TabChange
  end
end
```

I've given each tab a caption corresponding to the name of the frame, because I'm going to use this information to create the new pages. When the form is created, and whenever the user changes the active tab, the program gets the current caption of the tab and passes it to the custom `ShowFrame` method. The code of this method, listed below, checks whether the requested frame already exists (frame names in this example follow the Delphi standard of having a number appended to the class name), and then brings it to the front. If the frame doesn't exist, it uses the frame name to find the related frame class, creates an object of that class, and assigns a few properties to it. The code makes extensive use of class references and dynamic creation techniques (discussed in Chapter 3):

```
type
  TFrameClass = class of TFrame;
```

```
procedure TForm1.ShowFrame(FrameName: string);
var
  Frame: TFrame;
  FrameClass: TFrameClass;
begin
  Frame := FindComponent (FrameName + '1') as TFrame;
  if not Assigned (Frame) then
  begin
    FrameClass := TFrameClass (FindClass ('T' + FrameName));
    Frame := FrameClass.Create (Self);
    Frame.Parent := Tab;
    Frame.Visible := True;
    Frame.Name := FrameName + '1';
  end;
  Frame.BringToFront;
end;
```

To make this code work, you have to remember to add a call to `RegisterClass` in the initialization section of each unit defining a frame.

An Image Viewer with Owner-Draw Tabs

The use of the TabControl and of a dynamic approach, as described in the last example, can also be applied in more general (and simpler) cases. Every time you need multiple pages that all have the same type of content, instead of replicating the controls in each page, you can use a TabControl and change its contents when a new tab is selected.

This is what I'll do in the multiple-page bitmap viewer I'll show in the next example, called TabOnly. The image that appears in the TabControl of this form, aligned to the whole client area, depends on the selection in the tab above it (as you can see in Figure 8.14).

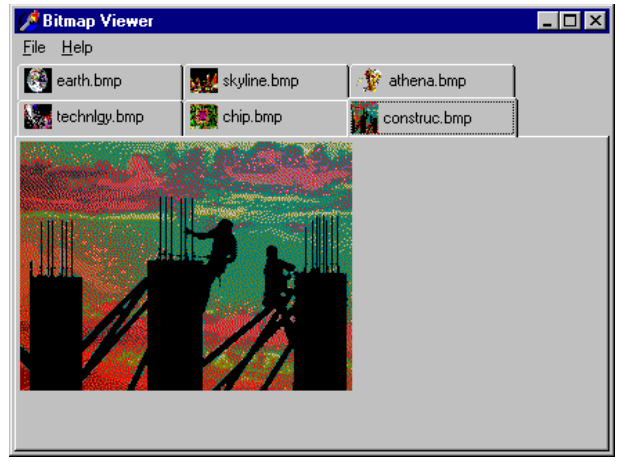


Figure 8.14: The interface of the bitmap viewer in the TabOnly example. Notice the owner-draw tabs. Image from the original book.

At the beginning, the TabControl has only a fake tab describing the situation (*No file selected*). After selecting File ➢ Open, the user can choose a number of files in the File Open dialog box, and the array of strings with the names of the files (the `Files` property of the `OpenDialog1` component) is used as the text for the tabs (the `Tabs` property of `TabControl1`):

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    TabControl1.Tabs := OpenDialog1.Files;
    TabControl1.TabIndex := 0;
    TabControl1Change (TabControl1);
  end;
end;
```

After we display the new tabs, we have to update the image so that it matches the first tab. To accomplish this, the program calls the method connected with the `OnChange` event of the TabControl, which loads the file corresponding to the current tab in the image component:

```
procedure TForm1.TabControl1Change(Sender: TObject);
begin
  Image1.Picture.LoadFromFile (
    TabControl1.Tabs [TabControl1.TabIndex]);
end;
```

The only special feature of the example is that the TabControl has the `OwnerDraw` property set to `True`. This means that the control won't paint the tabs (which will be empty at design time) but will have the application do this, by calling the `OnDrawTab`

event. In its code, the program displays the text vertically centered, using the `DrawText` API function. The text displayed is not the entire file path but only the file-name. Then, if the text is not *None,* the program reads the bitmap the tab refers to and paints a small version of it in the tab itself. To accomplish this, the program uses the `TabBmp` object, which is of type `TBitmap` and is created and destroyed along with the form. The program also uses the `BmpSide` constant to position the bitmap and the text properly:

```
procedure TForm1.TabControl1DrawTab(Control: TCustomTabControl;
  TabIndex: Integer; const Rect: TRect; Active: Boolean);
var
 TabText: string;
 OutRect: TRect;
begin
  TabText := TabControl1.Tabs [TabIndex];
  OutRect := Rect;
  InflateRect (OutRect, -3, -3);
  OutRect.Left := OutRect.Left + BmpSide + 3;
  DrawText (Control.Canvas.Handle,
    PChar (ExtractFileName (TabText)),
    Length (ExtractFileName (TabText)),
    OutRect, dt_Left or dt_SingleLine or dt_VCenter);
  if TabText <> 'None' then
  begin
    TabBmp.LoadFromFile (TabText);
    OutRect.Left := OutRect.Left - BmpSide - 3;
    OutRect.Right := OutRect.Left + BmpSide;
    Control.Canvas.StretchDraw (OutRect, TabBmp);
  end;
end;
```

This example works, unless you select a file that doesn't contain a bitmap. The program will warn the user with a standard exception, ignore the file, and continue its execution.

The User Interface of a Wizard

Just as you can use a TabControl without pages, you can also take the opposite approach and use a PageControl without tabs. What I want to focus on now is the development of the user interface of a wizard. In a wizard, you are directing the user through a sequence of steps, one screen at a time, and at each step you typically want to offer the choice of proceeding to the next step or going back to correct input entered in a previous step. So instead of tabs that can be selected in any order, wizards typically offer Next and Back buttons to navigate. This won't be a complex

example; its purpose is just to give you a few guidelines. The example is called WizardUI.

The starting point is to create a series of pages in a PageControl and set the `TabVisible` property of each TabSheet to `False` (while keeping the `Visible` property set to `True`). Contrary to what happened with past versions, in Delphi 5 you can now hide the tabs also at design time. In this case you'll need to use the shortcut menu of the page control or the combo box of the Object Inspector to move to a another page, instead of the tabs. But why don't you want to see the tabs at design time? So you can place controls on the pages and then place extra controls in front of the pages (as I've done in the example), without seeing their relative positions change at run time. You might also want to remove the useless captions of the tabs, which take up some space in memory and in the resources of the application.

In the first page, I've placed on one side an image and a bevel control and on the other side some text, a check box, and two buttons. Actually, the Next button is inside the page, while the Back button is over it (and is shared by all the pages). You can see this first page at design time in Figure 8.15. The following pages look similar, with a label, check boxes, and buttons on the right side and nothing on the left.

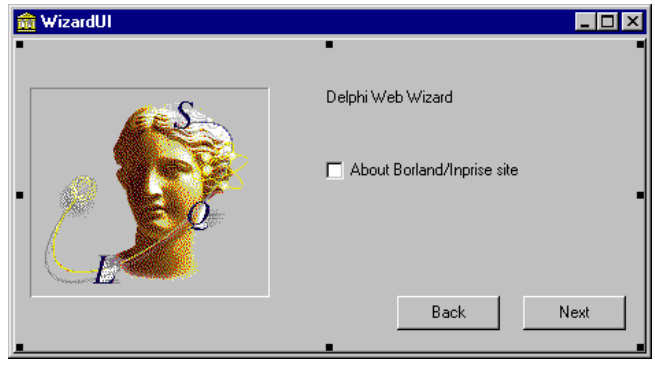


Figure 8.15: The first page of the WizardUI example at design time. Image from the original book.

When you click the Next button on the first page, the program looks at the status of the check box and decides which page is the following one. I could have written the code like this:

```
procedure TForm1.btnNext1Click(Sender: TObject);
begin
  BtnBack.Enabled := True;
  if CheckInprise.Checked then
    PageControl1.ActivePage := TabSheet2
  else
    PageControl1.ActivePage := TabSheet3;
```

```
  // move image and bevel
  Bevel1.Parent := PageControl1.ActivePage;
  Image1.Parent := PageControl1.ActivePage;
end;
```

After enabling the common Back button, the program changes the active page and finally moves the graphical portion to the new page. Because this code has to be repeated for each button, I've placed it in a method after adding a couple of extra features. This is the actual code:

```
procedure TForm1.btnNext1Click(Sender: TObject);
begin
  if CheckInprise.Checked then
    MoveTo (TabSheet2)
  else
    MoveTo (TabSheet3);
end;

procedure TForm1.MoveTo(TabSheet: TTabSheet);
begin
  // add the last page to the list
  BackPages.Add (PageControl1.ActivePage);
  BtnBack.Enabled := True;
  // change page
  PageControl1.ActivePage := TabSheet;
  // move image and bevel
  Bevel1.Parent := PageControl1.ActivePage;
  Image1.Parent := PageControl1.ActivePage;
end;
```

Besides the code I've already explained, the `MoveTo` method adds the last page (the one before the page change) to a list of visited pages, which behaves like a stack. In fact, the `BackPages` object of the `TList` class is created as the program starts and the last page is always added to the end. As the user presses the Back button, which is not dependent on the page, the program extracts the last page from the list, deletes its entry, and moves to that page:

```
procedure TForm1.btnBackClick(Sender: TObject);
var
  LastPage: TTabSheet;
begin
  // get the last page and jump to it
  LastPage := TTabSheet (BackPages [BackPages.Count - 1]);
  PageControl1.ActivePage := LastPage;
  // delete the last page from the list
  BackPages.Delete (BackPages.Count - 1);
  // eventually disable the back button
  BtnBack.Enabled := not (BackPages.Count = 0);
  // move image and bevel
  Bevel1.Parent := PageControl1.ActivePage;
```

```
  Image1.Parent := PageControl1.ActivePage;
end;
```

With this code, the user can move back several pages until the list is empty, at which point we disable the Back button. The complication we need to deal with is that while moving from a particular page, we know which pages are its "next" and "previous," but we don't know which page we came from, because there can be multiple paths to reach a page. Only by keeping track of the movements with a list can we reliably go back.

The rest of the code of the program, which simply shows some Web site addresses, is very simple. The good news is that you can reuse the navigational structure of this example in your own programs and modify only the graphical portion and the content of the pages.

Docking to a PageControl

Another interesting feature of page controls is the specific support for docking. As you dock a new control over a PageControl, a new page is automatically added to host it, as you can easily see in the Delphi environment. To accomplish this, you simply set the PageControl as a dock host and activate docking for the client controls. This works best when you have secondary forms you want to host. Moreover, if you want to be able to move the entire PageControl into a floating window and then dock it back, you'll need a docking panel in the main form.

This is exactly what I've done in the DockPage example, which has a main form with the following settings:

```
object Form1: TForm1
  Caption = 'Docking Pages'
  object Panel1: TPanel
    Align = alLeft
    AutoSize = True
    DockSite = True
    OnMouseDown = Panel1MouseDown
    object PageControl1: TPageControl
      ActivePage = TabSheet1
      Align = alClient
      DockSite = True
      DragKind = dkDock
      object TabSheet1: TTabSheet
        Caption = 'List'
        object ListBox1: TListBox
          Align = alClient
        end
```

```
      end
    end
  end
  object Splitter1: TSplitter
    Cursor = crHSplit
  end
  object Memo1: TMemo
    Align = alClient
  end
end
```

Notice that the Panel has the `UseDockManager` property set to `True` and that the PageControl invariably hosts a page with a list box, as removing all pages apparently causes problems. Now the program has two other forms, with similar settings (although they host different controls):

```
object Form2: TForm2
  Caption = 'Small Editor'
  DragKind = dkDock
  DragMode = dmAutomatic
  object Memo1: TMemo
    Align = alClient
  end
end
```

You can drag these forms onto the page control to add new pages to it, with captions corresponding with the form titles. You can also undock each of these controls and even the entire PageControl. To do this, the program doesn't enable automatic dragging, which would make it impossible to switch pages anymore. Instead, the feature is activated when the user clicks on the area of the PageControl that has no tabs—that is, on the underlying panel:

```
procedure TForm1.Panel1MouseDown(Sender: TObject;
  Button: TmouseButton; Shift: TShiftState; X, Y: Integer);
begin
  PageControl1.BeginDrag (False, 10);
end;
```

You can test this behavior by running the DockPage example, although Figure 8.16 tries to depict it. Notice that when you remove the PageControl from the main form,

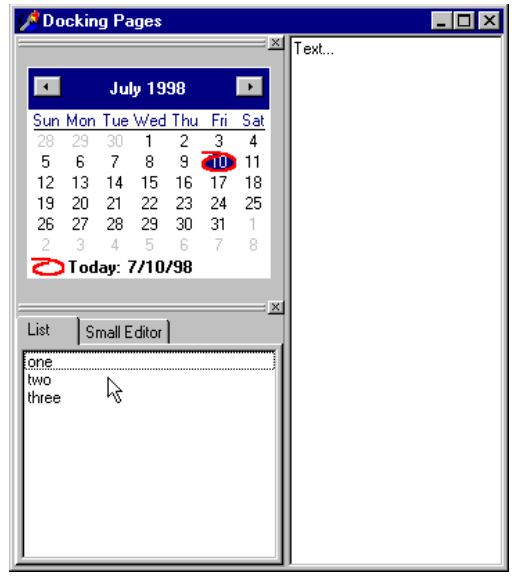


Figure 8.16: The main form of the DockPage example after a form has been docked to the page control on the left. Notice that another form uses part of the area of a hosting panel. Image from the original book.

you can directly dock the other forms to the panel and then split the area with other controls. This is the situation captured by the figure.

Creating MDI Applications

Besides using dialog boxes, or secondary forms, and squeezing components into a form, there is a third approach that used to be common in Windows applications: MDI (*Multiple Document Interface*). An MDI application is made up of a number of forms that appear inside a single main form[206].

If you use Windows Notepad, you can open only one text document, because Notepad isn't an MDI application[207]. But with your favorite word processor, you can

206 The MDI has long been phased out of the modern UI toolkit and Microsoft has in practical terms deprecated this model, as they've neglected fixing bugs in Windows when using MDI on HighDPI screens. Given that MDI has remained fairly popular among Delphi developes, Delphi 12 added renewed support for it VCL styled applications and also HighDPI applications (overcoming some of the platform issues). There was also the addition of the FormTabsBar component to help modernize the UI of MDI applications with the addition of tabs.

207 Microsoft introduced a multi tab UI for Notepad, just recently.

probably open a number of different documents, each in its own child window, because they are MDI applications. All these document windows are usually held by a *frame,* or *application,* window.

In Windows 3 and 3.1, Microsoft stressed the use of MDI. With the advent Windows 95, Microsoft had to acknowledge that many users were not comfortable with this interface. Office 2000 is the first large applications suite that drops the MDI model for the SDI (*Single Document Interface*) model used by Windows Resource Explorer and the entire operating system. MDI isn't dead and can be a useful model at times, but multipage and dockable forms seems to be more popular now.

MDI in Windows: A Technical Overview

This section provides a short overview of MDI, in technical Windows terms. Just forget Delphi for a moment, and I'll try to give you an idea of what MDI really is (not what an MDI application looks like). If you've never built an MDI application and you want a quick start, you might consider skipping this section for now.

The MDI structure gives programmers a number of benefits automatically. For example, Windows handles a list of the child windows in one of the pull-down menus of an MDI application, and there are specific Delphi methods that activate the corresponding MDI functionality, to tile or cascade the child windows. The following is the technical structure of an MDI application in Windows[208]:

- The main window of the application acts as a frame or a container. This window requires a proper menu structure and some specific coding (at least when programming with the API).
- A special window, known as the *MDI client,* covers the whole client area of the frame window, providing some special capabilities. For example, the MDI client handles the list of child windows. Although this might seem strange at first, the MDI client is one of the Windows predefined controls, just like an edit box or a list box. The MDI client window does not have the typical elements of the interface of a window, such as a caption or border, but it is visible. In fact, you can change the standard system color of the MDI work area (called the "Application Background") in the Appearance page of the Display Properties dialog box in Windows.

---

208 This is still the case today. Most of the platform and Delphi MDI features have seen very limited changes over the years, until Delphi 12, as mentioned in an earlier footnote.

- There are a number of child windows, of the same kind or of different kinds. These child windows are not placed in the frame window directly, but each is defined as a child of the MDI client window, which in turn is a child of the frame window. (We might say that the child windows are the "grandchildren" of the frame.)

When you program using the Windows API, some work is usually required to build and maintain this structure, and other coding is needed to handle the menu properly. As you'll see in the next section, these tasks become much easier with Delphi.

Frame and Child Windows in Delphi

Delphi makes the development of MDI applications easy, even without using the MDI Application template available in Delphi (see the Applications page of the File ➢ New dialog box). You only need to build at least two forms, one with the `FormStyle` property set to `fsMDIForm` and the other with the same property set to `fsMDIChild`. That's all, almost. Simply set these two properties in a simple program and run it, and you'll see the two forms nested in the typical MDI style.

Generally, however, the child form is not created at start-up, and you need to provide a way to create one or more child windows. This can be done by adding a menu with a New menu item and writing the following code:

```
procedure TMainForm.New1Click(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  ChildForm := TChildForm.Create (Application);
  ChildForm.Show;
end;
```

In the code fragment above, as well as in the program example I'll discuss shortly, I've named the two forms `MainForm` and `ChildForm`. Another important feature is to add a "Window" pull-down menu and use it as the value of the `WindowMenu` property of the form. This pull-down menu will automatically list all the available child windows. Of course, you can choose any other name for the menu item, but "Window" is the standard.

With these simple operations, you can build a simple MDI application. To make this program work properly, we can add a number to the title of any child window when it is created:

```
procedure TMainForm.New1Click(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  WindowMenu := Window1;
  Inc (Counter);
  ChildForm := TChildForm.Create (Self);
  ChildForm.Caption := ChildForm.Caption + ' ' +
    IntToStr (Counter);
  ChildForm.Show;
end;
```

You can also open a number of child windows, minimize or maximize each of them, close them, and use the Window pull-down menu to navigate among them. If you create more than nine child windows, a More Windows menu item is added to the pull-down menu; when you select this menu item, you'll see a dialog box (provided by Windows, not part of your program) with a complete list of the child windows.

Now suppose that we want to close some of these child windows, to unclutter the client area of our program. Click on the Close box of some of the child windows and they are minimized! What is happening here? Remember that when you close a window, you generally hide it from view. The closed forms in Delphi still exist, although they are not visible. In the case of child windows, simply hiding them won't work, because the MDI Window menu and the list of windows will still list existing child windows, even if they are hidden. For this reason, Delphi simply minimizes the MDI child windows when you try to close them. To solve this problem, we need to delete the child windows when they are closed, setting the `Action` reference parameter of the `OnClose` event to `caFree`.

Building a Complete Window Menu

Our first task is to define a better menu structure for the example. Typically the Window pull-down menu has at least three items, titled Cascade, Tile, and Arrange Icons. To handle the menu commands, we can use some of the predefined methods that are available in forms that have the `fsMDIForm` value for the `FormStyle` property:

- The `Cascade` method cascades the open MDI child windows. The child forms are arranged starting from the upper-left corner of the client area of the frame windows and moving toward the lower-left corner. The windows overlap each other. Iconized child windows are also arranged (see `ArrangeIcons` below).

- The `Tile` method tiles the open MDI child windows. The child forms are arranged so that they do not overlap. The client area of the frame windows is divided into equal portions for the different windows, so that they can all be shown on the screen, no matter how many windows there are. The `Tile` method will also arrange iconized child windows. The default behavior is horizontal tiling, although if you have several child windows, they will be arranged in several columns. This default can be changed by using the `TileMode` property.
- The `TileMode` property determines how the `Tile` procedure should work. The only two choices are `tbHorizontal`, for horizontal tiling, and `tbVertical`, for vertical tiling. Some applications use two different menu commands for the two tiling modes; other applications offer only one Tile menu command but check whether the Shift key is pressed when the user selects it. This actually confuses most users, so you'll probably want to keep your application simple, with one tiling option.
- The `ArrangeIcons` procedure arranges all the iconized child windows, starting from the lower-left corner of the client area of the frame window, and moving to the upper-right corner. Open forms are not moved.

These procedures and properties are useful for handling the Window menu of an MDI application. For example, you can write the following code:

```
procedure TMainForm.Cascade1Click(Sender: TObject);
begin
  Cascade;
end;
```

As a better alternative, you can place an ActionList in the form and add to it a series of predefined MDI actions. The related classes are `TWindowArrange`, `TWindowCascade`, `TWindowClose`, `TWindowTileHorizontal`, `TWindowTileVertical`, and `TWindowMinimizeAll`. The connected menu items will perform the corresponding actions and will be disabled if no child window is available. The MdiDemo example, which we'll look at next, demonstrates the use of the MDI actions, among other things.

There are also some other interesting methods and properties related strictly to MDI in Delphi:

- `ActiveMDIChild` is a run-time and read-only property of the MDI frame form, and it holds the active child window. The user can change this value by selecting a new child window, or the program can change it using the `Next` and `Previous` procedures.

- The `Next` procedure activates the child window that follows the active one in the internal order.
- The `Previous` procedure activates the child window preceding the active one in the internal order.
- The `ClientHandle` property holds the Windows handle of the MDI client window, which covers the client area of the main form.
- The `MDIChildCount` property stores the current number of child windows.
- The `MDIChildren` property is an array of child windows. You can use this and the `MDIChildCount` property to cycle among all of the child windows, for example using a `for` loop. This can be useful for finding a particular child window or to operate on each of them.

Note that the internal order of the child windows is the reverse order of activation. This means that the last child window selected is the active window (the first in the internal list), the second-to-last child window selected is the second, and the first child window selected is the last. This order determines how the windows are arranged on the screen. The first window in the list is the one above all others, while the last window is below all others, and probably hidden away. You can imagine an axis (the *z*-axis) coming out of the screen toward you. The active window has a higher value for the *z*-coordinate and, thus, covers other windows. For this reason, the Windows ordering schema is known as the *z-order*.

The MdiDemo Example

I've built a first example to demonstrate most of the features of a simple MDI application. MdiDemo is actually a full-blown MDI text editor, because each child window hosts a Memo component and can open and save text files. The child form has a `Modified` property used to indicate whether the text of the memo has changed. It is used by the file operations and when the form is closed. The file operations are performed by a couple of extra methods, as you can see in the class declaration of the form:

```
type
 TChildForm = class(TForm)
   Memo1: TMemo;
   procedure FormClose(Sender: TObject; var Action: TCloseAction);
   procedure Memo1Change(Sender: TObject);
   procedure FormCreate(Sender: TObject);
   procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
 private
```

```
    fModified: Boolean;
    procedure SetModified(const Value: Boolean);
  public
    procedure Load (FileName: string);
    procedure Save;
    property Modified: Boolean
      read FModified write SetModified;
  end;
```

note As discussed in Chapter 3, if you want to follow the rules of OOP and provide a good encapsulation, always add properties to a form to export a field instead of making the field public. The Class Completion feature of Delphi 4 makes it so fast to add a property to a form that there are no more excuses not to do it.

The `fModified` flag is set to `True` in the handler of the memo's `OnChange` event, and it is set to `False` in the form's `OnCreate` event handler. It is also set to `False` every time a new file is loaded or saved, as you can see in the code of the two file methods:

```
procedure TChildForm.Load (FileName: string);
begin
  Memo1.Lines.LoadFromFile (FileName);
  Caption := FileName;
  fModified := False;
end;

procedure TChildForm.Save;
begin
  Memo1.Lines.SaveToFile (Caption);
  fModified := False;
end;
```

Notice that the child form uses the caption to store the filename, a shortcut I've adopted instead of adding a second property to the form.

The `fModified` flag is checked when a child form is closed, as you can see in the following listing. Keep in mind that the `OnCloseQuery` method of the child forms is also automatically activated when you close the main form:

```
procedure TChildForm.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  CanClose := not fModified or (MessageDlg ('Close without saving?',
    mtConfirmation, [mbYes, mbNo], 0) = mrYes);
end;
```

As I've already mentioned, the main form of this example is based on an ActionList component. The actions are available through some menu items and a toolbar, as

you can see in Figure 8.17. You can see the details of the ActionList in the source code of the example.

Next, I want to focus on the code of the custom actions. Once more, this example demonstrates that using actions makes it very simple to modify the user interface of the program, without writing any extra code. In fact, there is no code directly tied to the user interface.

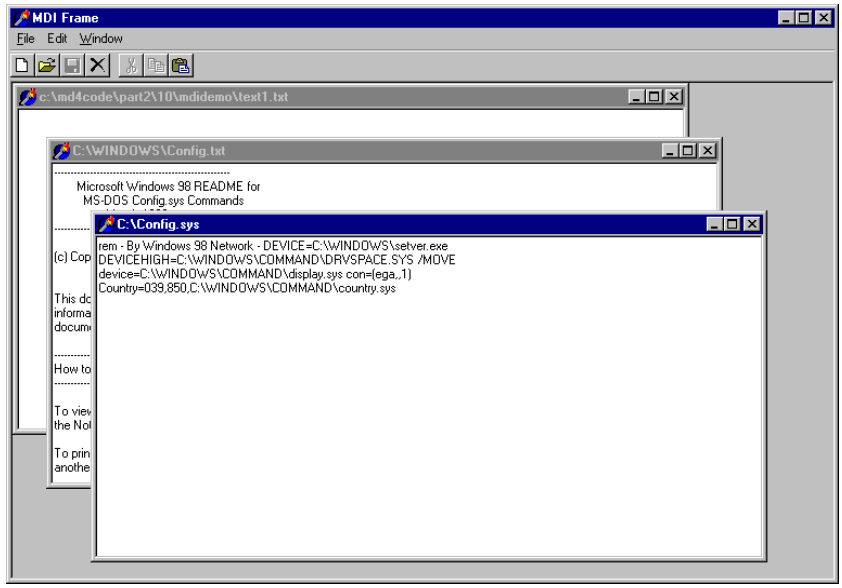


Figure 8.17: The MdiDemo program uses a series of predefined Delphi actions connected to a menu and a toolbar. Image from the original book.

One of the simplest actions is the `ActionFont` object, which has both an `OnExecute` handler, which uses a FontDialog component, and an `OnUpdate` handler, which disables the action (and hence the associated menu item and toolbar button) when there are no child forms:

```
procedure TMainForm.ActionFontExecute(Sender: TObject);
begin
  if FontDialog1.Execute then
    (ActiveMDIChild as TChildForm).Memo1.Font :=
      FontDialog1.Font;
end;

procedure TMainForm.ActionFontUpdate(Sender: TObject);
begin
  ActionFont.Enabled := MDIChildCount > 0;
end;
```

The action named New creates the child form and sets a default filename. The Open action calls the `ActionNewExcecute` method prior to loading the file:

```
procedure TMainForm.ActionNewExecute(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  Inc (Counter);
  ChildForm := TChildForm.Create (Self);
  ChildForm.Caption :=
    LowerCase (ExtractFilePath (Application.Exename)) +
    'text' + IntToStr (Counter) + '.txt';
  ChildForm.Show;
end;

procedure TMainForm.ActionOpenExecute(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    ActionNewExecute (Self);
    (ActiveMDIChild as TChildForm).Load (OpenDialog1.FileName);
  end;
end;
```

The actual file loading is performed by the `Load` method of the form. Likewise, the `Save` method of the child form is used by the Save and Save As actions. Notice the `OnUpdate` handler of the Save action, which enables the action only if the user has changed the text of the memo:

```
procedure TMainForm.ActionSaveAsExecute(Sender: TObject);
begin
  // suggest the current file name
  SaveDialog1.FileName := ActiveMDIChild.Caption;
  if SaveDialog1.Execute then
  begin
    // modify the file name and save
    ActiveMDIChild.Caption := SaveDialog1.FileName;
    (ActiveMDIChild as TChildForm).Save;
  end;
end;

procedure TMainForm.ActionSaveUpdate(Sender: TObject);
begin
  ActionSave.Enabled := (MDIChildCount > 0) and
    (ActiveMDIChild as TChildForm).Modified;
end;

procedure TMainForm.ActionSaveExecute(Sender: TObject);
begin
  (ActiveMDIChild as TChildForm).Save;
end;
```

MDI Applications with Different Child Windows

A common approach in complex MDI applications is to include child windows of different kinds (that is, based on different child forms). We will build a new example, called MdiMulti, to highlight some problems you may encounter with this approach. For this example, we need to build two different types of child forms. The first type will host a circle drawn in the position of the last mouse click, while the second will contain a bouncing square. Another feature I'll add to the main form is a custom background obtained by painting a tiled image in it.

Child Forms and Menus

The first type of child form can display a circle in the position where the user clicked one of the mouse buttons. Figure 8.18 shows an example of the output of the MdiMulti program. The program includes a Circle menu, which allows the user to change the color of the surface of the circle as well as the color and size of its border. What is interesting here is that to program the child form, we do not need to consider the existence of other forms or of the frame window. We simply write the code of the form, and that's all. The only special care required is for the menus of the two forms.

If we prepare a main menu for the child form, it will replace the main menu of the frame window when the child form is activated. An MDI child window, in fact, cannot have a menu of its own. But the fact that a child window can't have any menus should not bother you, because this is the standard behavior of MDI applications. You can use the menu bar of the frame window to display the menus of the child window. Even better, you can merge the menu bar of the frame window and that of the child form. For example, in this program, the menu of the child form can be placed between the frame window's File and Window pull-down menus. You can accomplish this using the following `GroupIndex` values:

- File pull-down menu, main form: 1
- Window pull-down menu, main form: 3

Circle pull-down menu, child form: 2

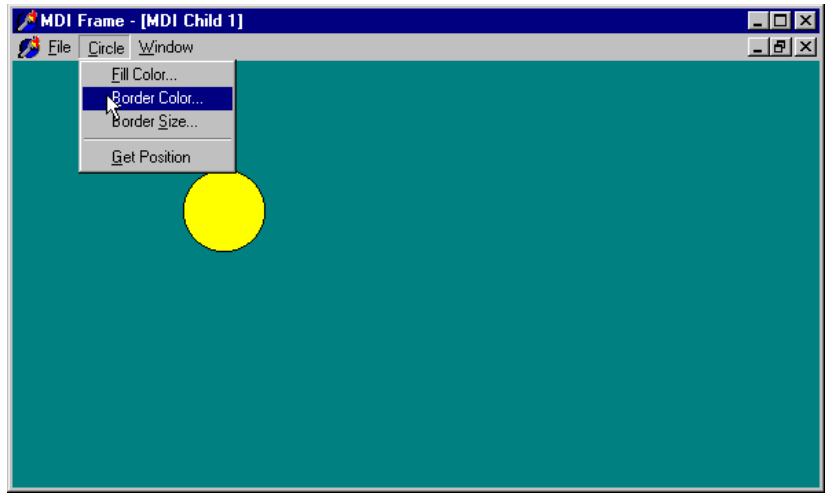


Figure 8.18: The output of the MdiMulti example, with a child window that displays circles. Image from the original book.

Using these settings for the menu group indexes, the menu bar of the frame window will have either two or three pull-down menus. At start-up, the menu bar has two menus. As soon as you create a child window, there are three menus, and when the last child window is closed (destroyed), the Circle pull-down menu disappears. You can see this in Figure 8.18, but you should also spend some time testing this behavior by running the program.

The code of the child window simply draws a shape on its sources. (For a complete discussion of this program, check out the Chapter 22, “Graphics in Delphi”). If you look at the source code, it is interesting to notice how the menu commands of the running program pertains to the two forms, and that in the source code, each form handles its own commands, regardless of the existence of other elements.

The data of the child form, particularly the coordinates of the center of the circle, must be declared using some fields of the form and not other variables declared inside the unit. In fact, we need a specific memory location to store the center of the circle for each child window.

The second type of child form shows a moving image. The square, a Shape component, moves around the client area of the form at fixed time intervals, using a Timer component, and bounces on the edges of the form, changing its direction. This turning process is determined by a fairly complex algorithm, which we don’t have space to examine. The main point of the example, instead, is to show you how menu merging behaves when you have an MDI frame with child forms of different types. (You can study the downloaded source code to see how it works.)

Changing the Main Form

Now we need to integrate the two child forms into an MDI application. The main form must provide a menu command to create a child form of the selected kind and to check the group indexes of the pull-down menus. The File pull-down menu here has two separate New menu items, which are used to create a child window of either kind. The code uses a single child window counter. As an alternative, you could use two different counters for the two kinds of child windows. Again, the Window menu uses the predefined MDI actions.

As soon as a form of this kind is displayed on the screen, its menu bar is automatically merged with the main menu bar. When you select a child form of one of the two kinds, the menu bar changes accordingly. Once all the child windows are closed, the original menu bar of the main form is reset. By using the proper menu group indexes, we let Delphi accomplish everything automatically, as you can see in Figure 8.19.

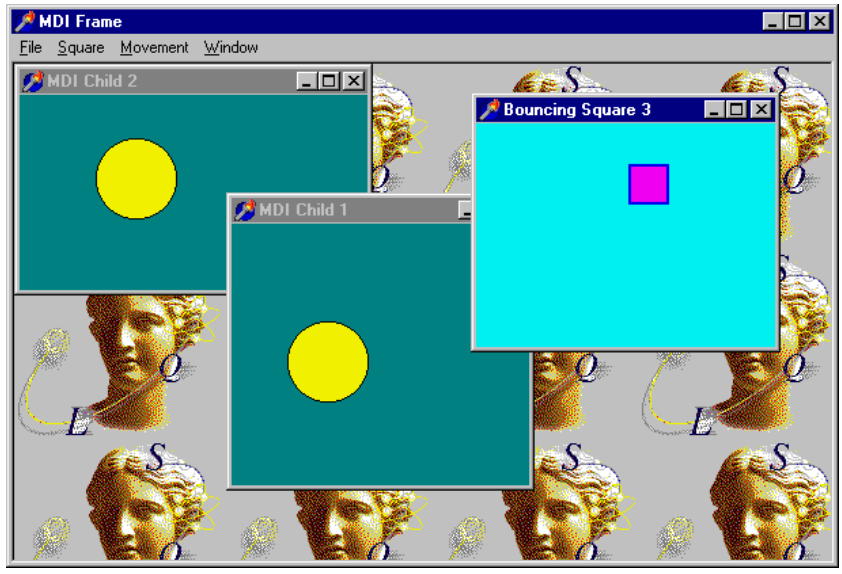


Figure 8.19: The menu bar of the MdiMulti Demo4 application changes automatically to reflect the currently selected child window, as you can see by comparing the menu bar with that of Figure 8.18. Image from the original book.

I've added a few other menu items in the main form. One menu choice is used to close every child window and another shows some statistics about them. The method related to the `Count` command scans the `MDIChildren` array property to count the number of child windows of each kind (using the RTTI operator `is`). Once

these values are computed, they are shown on the screen with the `MessageDlg` function:

```
procedure TMainForm.Count1Click(Sender: TObject);
var
  NBounce, NCircle, I: Integer;
begin
  NBounce := 0;
  NCircle := 0;
  for I := 0 to MDIChildCount - 1 do
    if MDIChildren is TBounceChildForm then
      Inc (NBounce)
    else
      Inc (NCircle);
  MessageDlg (
    Format ('There are %d child forms.'#13 +
      '%d are Circle child windows and ' +
      '%d are Bouncing child windows',
      [MDIChildCount, NCircle, NBounce]),
    mtINFormation, [mbOk], 0);
end;
```

Subclassing the MdiClient Window

Finally, the program includes support for a background-tiled image. The bitmap is taken from an Image component and should be painted on the form in the `wm_EraseBkgnd` Windows message's handler. The problem is that we cannot simply connect the code to the main form, as its surface is covered by a separate window, the MdiClient window described earlier in this chapter.

We have no corresponding Delphi form for this window, so how can we handle its messages? We have to resort to a low-level Windows programming technique known as *subclassing*. (In spite of the name, this has little to do with OOP inheritance). The basic idea is that we can replace the window procedure, which receives all the messages of the window, with a new one we provide. This can be done by calling the `SetWindowLong` API function and providing the memory address of the procedure, the function pointer.

note A window procedure is a function receiving all the messages for a window. Every window must have a window procedure and can have only one. Even Delphi forms have a window procedure; although this is hidden in the system, it calls the `WndProc` virtual function, which you can use. But the VCL has a predefined handling of the messages, which are then forwarded to the message-handling methods of a form after some preprocessing. With all this support, you need to handle window procedures explicitly only when working with non-Delphi windows, as in this case. For a thorough description of this topic you can refer to *Delphi Developer's Handbook* (Sybex, 1998), among other books.[209]

Unless we have some reason to change the default behavior of this system window, we can simply store the original procedure and call it to obtain a default processing. The two function pointers referring to the two procedures (the old and the new one) are stored in two local fields of the form:

```
private
  OldWinProc, NewWinProc: Pointer;
  procedure NewWinProcedure (var Msg: TMessage);
```

The form also has a method we'll use as a new window procedure, with the actual code used to paint on the background of the window. Because this is a method and not a plain window procedure, the program has to call the `MakeObjectInstance` method to add a prefix to the method and let the system use it as if it were a function. All this description is summarized by only two complex statements:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  NewWinProc := MakeObjectInstance (NewWinProcedure);
  OldWinProc := Pointer (SetWindowLong (
    ClientHandle, gwl_WndProc, Cardinal (NewWinProc)));
  OutCanvas := TCanvas.Create;
end;
```

The window procedure we install calls the default one. Then, if the message is `wm_EraseBkgnd` and the image is not empty, we draw it on the screen many times using the `Draw` method of a temporary canvas. This canvas object is created when the program starts (see the code above) and connected to the handle passed as `wParam` parameter by the message. With this approach, we don't have to create a new `TCanvas` object for every background painting operation requested, thus saving a little time in the frequent operation. Here is the code, which produces the output already seen in Figure 8.19:

```
procedure TMainForm.NewWinProcedure (var Msg: TMessage);
var
```

209 That book is now hard to find. Restoring it would be another interesting project.

```
  BmpWidth, BmpHeight: Integer;
  I, J: Integer;
begin
  // default processing first
  Msg.Result := CallWindowProc (OldWinProc,
    ClientHandle, Msg.Msg, Msg.wParam, Msg.lParam);

  // handle background repaint
  if Msg.Msg = wm_EraseBkgnd then
  begin
    BmpWidth := MainForm.Image1.Width;
    BmpHeight := MainForm.Image1.Height;
    if (BmpWidth <> 0) and (BmpHeight <> 0) then
    begin
      OutCanvas.Handle := Msg.wParam;
      for I := 0 to MainForm.ClientWidth div BmpWidth do
        for J := 0 to MainForm.ClientHeight div BmpHeight do
          OutCanvas.Draw (I * BmpWidth,
            J * BmpHeight, MainForm.Image1.Picture.Graphic);
    end;
  end;
end;
```

What's Next?

We have explored different ways to build applications that have several forms or forms with multiple pages. We have seen how you can create a secondary modeless form or a modal dialog box. Besides the basic examples, we have delved into some advanced topics, such as dynamically building a number of forms; creating extensible dialog boxes, using the common dialogs and the Delphi message boxes; building special About boxes, with hidden credits or used as a splash screen; as well as the MDI technique.

There are many things we could do to further explore how to build applications with multiple forms and extend their user interface. I've given equal coverage to various techniques, although I have my preferences: fewer secondary forms, more dialog boxes, MDI only for specific programs, notebooks, and docking whenever possible.

Now we can move forward to a very hot Delphi programming topic: building database applications. This will take the next four chapters, which cover most of the fundamental topics of Delphi database programming. It is possible to write a specific book about this, so the description won't be exhaustive, but you should be able to get a comprehensive overview of this key element of Delphi development.

After these three database chapters, we'll be able to start looking into Delphi behind the scenes and focus on topics such as the construction of Delphi components and ActiveX controls.

Chapter 9: Writing Database Applications

Delphi's support for database applications is one of the key features of the programming environment. Many programmers spend most of their time writing data-access code, which needs to be the most robust portion of a database application. This chapter provides an overview of Delphi's extensive support for database programming. You can create very complex database applications, starting from a blank form or one generated by Delphi's Database Form Wizard[210].

---

210 The Database Form Wizard is no longer available, but also a lot of the specific techniques described in this chapter are not applicable any more. For one, the BDE data access library no longer ships with Delphi, replaced by newer alternatives like FireDAC. Also using paradox tables, while technically possible, has long been deprecated and it no longer recommended (or even suggested). In other words, the content of this chapter and the demos have severe limitations, but some of the key concepts of the core DB.pas unit are still valid today.

What you won't find here is a discussion of the theory of database design. I'm assuming that you already know the fundamentals of database design and have already designed the structure of a database. I won't delve into database-specific problems; my goal is to help you understand how Delphi supports this kind of programming.

We'll begin with an explanation of how data access works in Delphi, and then we'll review the database components that are available in Delphi. I won't discuss the simplest examples and techniques step by step, such as the use of the Database Form Wizard, but instead I will focus on the foundations. Some of the topics covered in this chapter include an in-depth example of the `TField` components, creating new tables with Delphi code, and using graphics. The following chapters will provide information on many other more advanced database programming topics.

Accessing Data with and without the BDE

On a computer, permanent data—including database data—is always stored in files. The two most common approaches are to store a whole database in what appears to the file system as a single file or to store each table, index, and any other elements of the database in separate files, usually on the same directory. Delphi supports both approaches, depending on the database format you are using:

- Paradox and dBASE tables[211] define databases as directories and each table as a separate file (or actually multiple files if you include indexes and other support files).
- Access, InterBase, and most SQL servers use a single file containing the entire database, with all the tables and indexes.[212].

---

211 This is a very old approach, not recommended today. For simple local data storage, a good replacement is to use memory tables (FDMemTable) and store their content to file. The different, though, is that persistent tables can be loaded by one application at a time, while Paradox and dBase accounted for access by multiple apps at the same time, with techniques explained later in this chapter.

212 This is still generally true today.

note The Borland Database Engine (BDE)[213] uses an alias to refer to a database file or directory. You can define new aliases for databases by using the Database Explorer or the Database Engine Configuration utility. It is also possible to define them by writing code in Delphi that calls the `AddStandardAlias` and `AddAlias` methods of the `Session` global object, followed by a call to `SaveConfigFile` to make the alias persistent. The alternative is the low-level `DbiAddAlias` BDE function.

Delphi database applications do not have direct access to the data sources they reference and cannot manipulate database files directly. Instead, they use an available database engine, such as the Borland Database Engine (BDE) or Microsoft's ActiveX Data Objects (ADO)[214].

The BDE has direct access to a number of data sources, including dBASE, Paradox, ASCII, FoxPro, and even Access tables. The BDE can also interface with Borland's SQL Links, a series of drivers allowing access to a number of local and remote SQL servers (available only in Delphi Enterprise). Database servers include Oracle, Sybase, Informix, InterBase, and DB2[215]. If you need access to a different database or data format, the BDE can interface with ODBC drivers, although in this case you might want to use ADO instead. Notice, anyway that the BDE provides advanced features (such as sophisticated caching and heterogeneous joins) that ADO doesn't offer.

ADO is Microsoft's high-level interface[216]. ADO is implemented on Microsoft's data access OLE DB technology, which provides access to relational and non-relational databases as well as e-mail and file systems and custom business objects. Applications built with Delphi 5's ADO components don't require the BDE libraries. Of course, users need to have the ADO/OLE DB run time, which is distributed by Microsoft and is part of the Windows 2000 operating system[217]. ADO will also need to be configured on the user's machine, even if it is already installed. Chapter 12 will more fully cover ADO and related technologies.

---

213 As mentioned, the BDE no longer ships with Delphi, although it has remained available as a separate download for some time. Needless to day the associated Database Explorer and Database Engine Configuration tools are also no longer part of the Delphi installation.

214 Or FireDAC, or IBX, or the now deprecated dbExpress (DBX), or other third party database access libraries.

215 FireDAC offers many more alternatives in terms of direct database support and it also includes an ODBC gateeway.

216 Support for ADO ~(via the dbGo library) is still available in today's Delphi, even if it's not considered a top option as the underlying Microsoft library, while still available, has been neglected in favor of ADO.NET.

217 The library is still part of Windows 11 and also of recent Windows Server editions.

Delphi Enterprise includes native components that access Borland's own InterBase server[218] (available on the Delphi installation CD; see Chapter 11 for more details) and the ClientDataSet component (see Chapter 21), which can be used for local or remote data access. These technologies provide alternatives to the traditional use of the BDE to access a database from Delphi applications. Figure 9.1 shows the alternative data-access techniques available in Delphi 5[219], indicating that all the data-access components inherit from a common base class, `TDataSet`.

If you choose the traditional BDE approach (as most of the examples in this chapter do), you will need to install the BDE along with your applications on your clients' computers. This is not difficult, because Delphi includes the "lite" version of a widely used installation program (InstallShield) that can be used to prepare installation disks for the BDE, along with your own application. The BDE files are required—your Delphi database applications won't work without them—but you can distribute them freely.

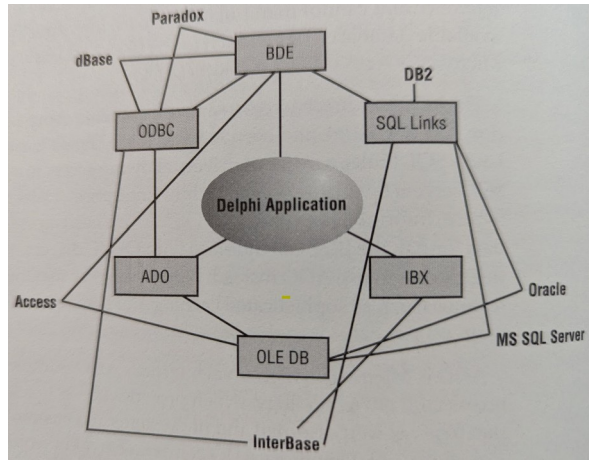


Figure 9.1: The alternative data access technologies available in Delphi 5. Image based on a picture of the original printed book.

218 InterBase Developer edition is an optional installation of todays's Delphi, while the embedded version of the database (IBLite/IBToGo) is installed automatically and is available for desktop and mobile applications. See the InterBase section of Embarcadero web site for more details.

219 This image depicts what was available at the time, very different of what you can do today with FireDAC and other alternatives.

Delphi Database Components

Delphi includes a number of components related to databases. The Data Access page[220] of the Component Palette contains components used to interact with databases in BDE-oriented applications. Most of them are non-visual components, since they encapsulate database connections, tables, queries, and similar elements. Fortunately, Delphi also provides a number of predefined components you can use to view and edit database data. In the Data Controls page, there are visual components used to view and edit the data in a form. These controls are called *data-aware* controls[221].

To access a database in Delphi, you generally need a data source, identified by the DataSource component[222]. The DataSource component, however, does not indicate the data directly; it refers to a DataSet component. This can be a table, the result of a query, the result of a stored procedure, the data fetched from a remote server (using the ClientDataSet component), ADO, InterBase, or some other custom dataset.

As soon as you have placed a dataset component on the form, you can use the DataSet property of the DataSource component to refer to it. For this property, the Object Inspector lists the available datasets of the current form or of other forms and data modules connected with the current one (using the File ➢ Use Unit command).

Tables and Queries

The simplest way to specify data access in Delphi is to use the Table component[223]. A Table object simply refers to a database table. When you use a Table component, you need to indicate the name of the database you want to use in its `DatabaseName`

---

220 The Data Access page still exists, but it hosts general purpose components only, not the BDE ones, which are not in the product any more.

221 The VCL library still offers data-aware controls, but it also include Visual Live Bindings, the primary technique used in FireMonkey to associated UI controls and data.

222 The role of the DataSource component hasn't changed. It still refers to a TDataSet descendant, exactly like in the early days of Delphi. The difference is that newer TDataSet descendants are not available, like FireDAC datasets.

223 With this component missing, a good starting point is now the FDMemTable component, which can be mapped to local data files. The alternative is to use the FDTable component, but it requires the connection to a RDBMS to fetch any data.

property. You can enter an alias or the path of the directory with the table files. The Object Inspector lists the available names, which depend on the aliases installed in the BDE.

You also need to indicate a proper value in the `TableName` property[224]. The Object Inspector lists the available tables of the current database (or directory), so you should generally select the `DatabaseName` property.

A second data set available in Delphi is the Query component[225]. A query is usually more complex than a table, because it requires a SQL language command. However, you can customize a query using SQL more easily than you can customize a table (as long as you know at least the basic elements of SQL, of course). The Query component has a `DatabaseName` property like the Table component, but it does not have a `TableName` property. The table is indicated in the SQL statement, stored in the `SQL` property.

note SQL is a standard language for writing database queries and generally interacting with a database. If you are not fluent in SQL, you can find a description of its basic commands in Chapter 11. Delphi Enterprise includes a tool to create SQL queries, called SQL Builder[226], and is discussed in Chapter 11, as well.

For example, you can write a simple SQL statement like this

```
select * from Country
```

where `Country` is the name of a table, and the star symbol (`*`) indicates that you want to use all of the fields in the table. If you are fluent in SQL, you might use the Query component more often, but the efficiency of a table or a query varies depending on the database you are using. In general, we can say that the Table component tends to be faster on local tables, while the Query component tends to be faster on SQL servers, although this is just a very general rule, and in many cases you might have the opposite effect. I'll cover some efficiency issues in Chapters 10 and 11.

224 By contrast, FDMemTable only needs to refer to a local file.

225 The matching FireDAC component is called FDQuery.

226 This tool is also long gone, but FireDAC designers offer some help in creating queries and specific tools replacing the old query builder..

note The Country table mentioned above refers to the file COUNTRY.DB, which is part of the Delphi's demo database, installed by default in the C:\Program Files\Common Files\Borland Shared\Data directory[227]. This directory is referenced by the DBDEMOS alias, set up by Delphi during the installation. Many of my examples in the following chapters will use tables from this Delphi database. In others, I'll show you how to build new tables, but I'll generally use that demo database, as well.

The third component for data sets is StoredProc, which refers to stored procedures of a SQL server database. You can run these procedures and get the results in the form of a database table. Stored procedures can only be used with SQL servers.

The Status of a Data Set[228]

When you operate on a data set in Delphi, you can work in different states, indicated by a specific `State` property, which can assume several different values:[229]

- `dsBrowse` indicates that the data set is in normal browse mode, used to look at the data and scan the records.
- `dsEdit` indicates that the data set is in edit mode. A data set enters this state when the program calls the `Edit` method or the DataSource has the `AutoEdit` property set to `True`, and the user starts editing a data-aware control, such as a DBGrid or DBEdit. When the changed record is posted, the data set exits the `dsEdit` state.
- `dsInsert` indicates that a new record is being added to the data set. Again, this might happen when calling the Insert method, moving to the last line of a DBGrid, or using the corresponding command of the DBNavigator component.
- `dsInactive` is the state of a closed data set.

227 A similar directory under the demos folder (installed by default under the Windows users public directory, *C:\Users\Public\Embarcadero\xx.0*) includes same of the same old Paradox tables converted to the format required by FireDAC's FDMemTable.

228 The concept of Dataset status is still applicable and very important today, regardless of the data access components used. The core idea remains the same and it's very important to understand for Delphi database access.

229 Additional values compared to this list are dsBlockRead (used for reading data without chancing the current record), dsInternalCalc (similar to dsCalcFields, but for a modified version of the calculated fields logic), and dsOpening (used to indicate the dataset is being opened, but it' isn't ready yet)

- `dsSetKey` indicates that we are preparing a search on the data set. This is the state between a call to the `SetKey` method and a call to the `GotoKey` or `GotoNearest` methods (see the Search example later in this chapter).
- `dsCalcFields` is the state of a data set while a field calculation is taking place; that is, during a call to an `OnCalcFields` event handler. Again, I'll show this in an example.
- `dsNewValue`, `dsOldValue`, and `dsCurValue` are the states of a data set when an update of the cache is in progress.
- `dsFilter` is the state of a data set while setting a filter; that is, during a call to an `OnFilterRecord` event handler.

In simple examples, the transitions between these states are handled automatically, but it is important to understand them because there are many events referring to the state transitions.

---

note We will use a simple state-transition event, the `OnStateChange` event of the DataSource component, in the GridDemo example, the first example of this chapter.

Other Database Related Components

Along with the Table, Query, StoredProc, and DataSource, there are some other components in the Data Access page of the Component palette, the BDE page. I'll cover these components in the next two chapters, but here is a short summary:

- The Database component is used for transaction control, security, and connection control[230]. It is generally used only to connect to remote databases in client/server applications or to avoid the overhead of connecting to the same database in several forms. The Database component is also used to set a local alias used only inside a program. Once this local alias is set to a given path, the Table and Query components of the application can refer to the local database alias. This is much better than replicating the hard-coded path in each DataSet component of the program.

---

230 In FireDAC, this is replaced by a combination of the FDConnection and FDTransaction components.

- The Session component[231] provides global control over database connections for an application, including a list of existing databases and aliases and an event to customize database login.
- The BatchMove[232] component is used to perform batch operations, such as copying, appending, updating, or deleting values, on one or more databases.
- The UpdateSQL[233] component allows you to write SQL statements to perform various update operations on the data set, when using a read-only query (that is, when working with a complex query). This component is used as the value of the UpdateObject property of tables or queries.

Delphi Data-Aware Controls

We have seen how it is possible to connect a data source to a database, using either a table or query, but we still do not know how to view the data. For this purpose, Delphi provides many components that resemble the usual Windows controls but are data-aware. For example, the DBEdit component is similar to the Edit component, and the DBCheckBox component corresponds to the CheckBox component. You can find all of these components in the Data Controls page[234] of the Delphi Component Palette:

- DBGrid is a grid capable of displaying a whole table at once. It allows scrolling and navigation, and you can edit the grid's contents. It is an extension of the other Delphi grid controls.
- DBNavigator is a collection of buttons used to navigate and perform actions on the database. The buttons perform basic actions, so you can easily replace them with your own toolbar.
- DBText displays the contents of a field that cannot be modified. It is a data-aware Label graphical control.

---

231 The concept of Session was specific to the BDE. It has no match in other data access libraries.

232 FireDAC offers an extremely sophisticated "*batch move*" subsystem, based on FDBatchMove but also many other specific components for reading and writing different data formats, from database tables to XML, from JSON to CVS, from text files to other formats.

233 The same concept exists in FireDAC and other libraries, although FDUpdateSQL is only used in very complex scenarios, as the core components like FDQuery offer automatic updates in many cases.

234 These data-aware controls still exists aand are still frequently used. The only exception is the DBCltrGrid, which is no longer very commonly used even if it's still available.

- DBEdit lets the user edit a field (change the current value) using an Edit control.
- DBMemo lets the user see and modify a large text field, eventually stored in a memo or BLOB (Binary Large OBject) field. It resembles the Memo component.
- DBImage is an extension of an Image component that shows a picture stored in a BLOB field.
- DBListBox and DBComboBox let the user select a single value from a specified set. If this set is extracted from another database table or is the result of another query, you should use the DBLookupListBox or DbLookupComboBox components instead.
- DBCheckBox can be used to show and toggle an option, corresponding to a Boolean field, and extends the CheckBox component.
- DBRadioGroup provides a series of choices, with a number of exclusive selection radio buttons, such as the RadioGroup control.
- DBRichEdit is a component that lets the user edit a formatted text file; it is based on a Windows 95 RichEdit control.
- DBCtrlGrid is a multi-record grid, which can host a number of other data-aware controls. These controls are duplicated for each record of the data set.
- DBChart[235] is a data-aware business graphic component or the data-aware version of the Chart component.

All of these components are connected to a data source using the corresponding property, `DataSource`. Some of them relate to the entire data set, such as the DBGrid and DBNavigator components, while the others refer to a specific field of the data source, as indicated by the `DataField` property. Once you select the `DataSource` property, the `DataField` property will have a list of values available in the drop-down combo box of the Object Inspector.

235 This is tied to the Steema TeeChart add-on available in Delphi.

Customizing a Database Grid

Our first database example, called GridDemo, uses the COUNTRY.DB[236] table from the DBDEMOS database, which lists New World countries, along with each one's capital and population. Simply place a Table, a DataSource, and a DBGrid component on a form and connect them. If you set the `Active` property of the table to `True`, the data will appear in the form at design time. (This technique is usually called *live-data* design[237].) When a grid displays live data, you can even use its scroll bars to navigate through the records.

At this point, you can already run the program and even edit the data of the database table, making permanent changes. This is possible because the DBGrid component's `Options` property includes the flag `dgEditing` and the `ReadOnly` property is set to `False`. This program also allows you to insert a new record in a given position by pressing the Insert key, to append a new record at the end by going to the last record and pressing ↓, and to delete the current record by pressing Ctrl+Del.

note Try using this program for a while, testing how it works when you toggle the various flags of the `Options` property of the grid on and off. These flags determine the behavior of the grid, which can vary a lot. You can also see the description of the various options in Delphi's help file.

Besides the `Options` property, you can customize the DBGrid component with the easy-to-use yet very powerful `Columns` property. This property is a collection, so you can choose one of the items in the list and then set its property in the Object Inspector, as you can see in Figure 9.2.

You can easily choose the fields of the table you want to see in the grid as columns and then set a number of column properties (color, font, width, alignment, and so on) for each field and title properties, such as the caption, font, and colors. This allows you to customize a grid easily, in a number of ways. Some of the more advanced properties, such as `ButtonStyle` and `DropDownRows`, can be used to provide custom editors for the cells of a grid or a drop-down list.

236 This table remains available in multiple format, including FireDAC's FDMemTable native format (with .fds extension).

237 Nowadays, the feature is generally indicated as "*live-data at design time*" and, after all of these years, Delphi remains one of the few dev tools offering this very handy feature.

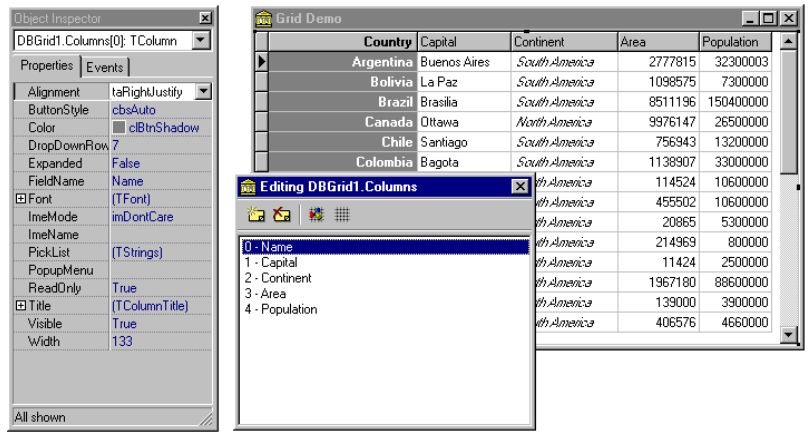


Figure 9.2: You can edit the properties of the Columns of a DBGrid by selecting one of the columns in the collection editor and using the Object Inspector. Image from the original book.

In the GridDemo example, I've changed the caption of the first column and the font of the first and third. I've also chosen a dark gray background and a white font color for the first column. I've also entered the names of a few continents in the `PickList` string list of the Continent field. You can see the result in Figure 9.3.

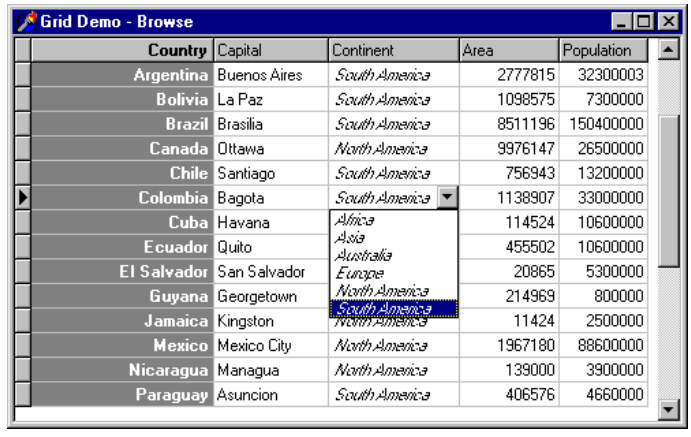


Figure 9.3: The DBGrid of the GridDemo example has a few customized Columns, including a PickList for the continents. Image from the original book.

note Notice that once you have defined the `Columns` property of the DBGrid, you can size the columns at design time simply by dragging the lines separating them. The same capability is optionally available at run time, and it can be set along with many others using the `Options` property of the grid.

To summarize the features of this simple example, here is an extract of the form description file:

```
object Form1: TForm1
  ActiveControl = DBGrid1
  Caption = 'Grid Demo'
  object DBGrid1: TDBGrid
    Align = alClient
    DataSource = DataSource1
    Columns = <
      item
        Alignment = taRightJustify
        Color = clBtnShadow
        FieldName = 'Name'
        Font.Style = [fsBold]
        ReadOnly = True
        Title.Alignment = taRightJustify
        Title.Caption = 'Country'
        Title.Font.Style = [fsBold]
      end
      item
        FieldName = 'Capital'
      end
      item
        Expanded = False
        FieldName = 'Continent'
        Font.Style = [fsItalic]
        PickList.Strings = (
          'Africa'
          'Asia'
          'Australia'
          'Europe'
          'North America'
          'South America')
      end
      item
        FieldName = 'Area'
      end
      item
        FieldName = 'Population'
      end>
  end
  object Table1: TTable
    Active = True
    DatabaseName = 'DBDEMOS'
    TableName = 'COUNTRY.DB'
  end
  object DataSource1: TDataSource
    DataSet = Table1
    OnStateChange = DataSource1StateChange
  end
end
```

The Table State

There are many more things you can do to customize grids, and we'll explore some of them in the next chapter, where we will also discuss how to add graphics to a grid. For the moment, I want to add an extra feature (and some code) to the example. If you look at the caption of the form in Figure 9.3, you'll notice something new: the title of the form indicates the status of the Table component. How do we get this information? Simply by handling the `OnStateChange` event of the DataSource component. In this event handler, the DemoGrid example merely outputs the current status, determined by using a simple `case` statement:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
var
  Title: string;
begin
  case Table1.State of
    dsBrowse: Title := 'Browse';
    dsEdit: Title := 'Edit';
    dsInsert: Title := 'Insert';
  else
    Title := 'Other state';
  end;
  Caption := 'Grid Demo - ' + Title;
end;
```

The code considers only the three states the Table component[238] of this program can have as the user interacts with the corresponding DBGrid.

Field-Oriented Data-Aware Controls

The GridDemo example works well, but we want to try using other controls, such as edit boxes, and we want to see specific information rather than all the data in our database. Before we really look at the core of the VCL database structure, by examining the `TField` components, I want to cover the usage of some of the data-aware controls you can use to see and edit the value of a database field. The starting point is the use of edit boxes.

---

238 Given the State property is part of the TDataSet base class, the same logic and code would work for any TDataSet descendant, including the FDMemTable component.

Using DBEdit Controls

The next example, called EditDemo, uses some DBEdit components and some labels, along with the table and the data source. We also need to add a brand-new component, the DBNavigator. Figure 9.4 shows the form of the EditDemo example at design time (with live data).

Again, we need to connect the three data-aware controls to the data source by setting their `DataSource` property, and we must also indicate a specific field for each of the three edit boxes in their `DataField` property (Name, Capital, and Continent are the fields for this example). If you have already connected the data source to the table and the edit boxes to the data source, you can simply select a field in the list displayed by the Object Inspector for the `DataField` property. When this connection is made, if the `Active` property of the Table is set to `True` the values of the first record's fields appear automatically in the edit boxes (see Figure 9.4).

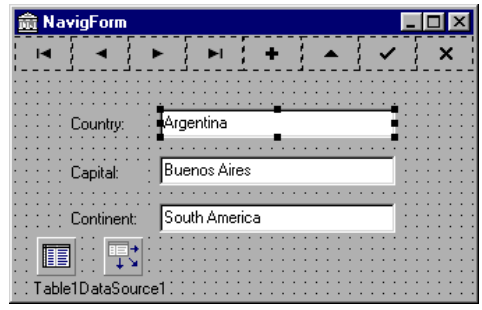


Figure 9.4: The three DBEdit and the DBNavigator components of the EditDemo example, with live data. Image from the original book.

Another step we can take is to disable some of the buttons of the DBNavigator control, by removing some of the elements of the `VisibleButtons` set.

note If you turn on its `ShowHint` property, the navigator will show a different fly-by hint for every button. You can provide a customized description of each of them, using the `Hints` string list. The strings you insert are used for the buttons in order: the first string is used for the first button, the second for the second, and so on. If some buttons are not visible, you can provide an empty string as a placeholder.

In the EditDemo program, I've used only some of the buttons, disabling the delete and refresh operations. I've also aligned the navigator to the top of the form and set its `Flat` property to `True` to activate the flat button style. You can run it to test whether it works properly, and look at the caption again: I've copied to this program

the `OnStateChange` event handler of the GridDemo program's DataSource component.

Notice that when the program is running at the beginning or when you jump to the first or last record of the table, two of the navigator's buttons will be disabled automatically. However, if you move step-by-step to the first or last record, the buttons are disabled only when you try to move beyond those records. The navigator (or the dataset, to be more precise) only realizes at this point that there are no more records in that direction. Other buttons are automatically enabled and disabled when you enter or exit the edit state.

Creating a Database Table

Before we can move to some other data-aware controls, we need to do an extra step. In the first two examples of this chapter, I've used existing tables of the DBDEMOS database, but for the following ones I need to create a table with specific types of fields. For this reason, I'll introduce here a topic we'll return to later on: the creation of new database tables.

Starting with version 4, Delphi allows you to set the definition of the fields of a table —its internal structure—at design time, using the collection editor of the `FieldDefs` property. You can see the settings I've used for the DbAware example in Figure 9.5.

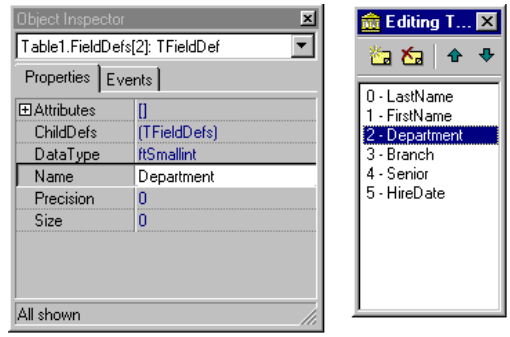


Figure 9.5: The editor of the field definitions collection. Images from the original book.

Having defined the fields, you can now right-click the table component and select the Create Table command[239]. This creates the new table at design time. In this specific example, there is no need to do this, since the program creates the table when it starts, unless the table already exists:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if not Table1.Exists then
    Table1.CreateTable;
  Table1.Open;
end;
```

To make this code work, the Table component must save the definition of the fields in the DFM file along with the other properties. This is done only if you set the `StoreDefs` property of the table to `True`. In Table 9.1, you can see the table field definitions, and the following listing shows the initial portion of the corresponding definitions in the DFM file.

Table 9.1: The Fields of the Workers Database Table

Name	DataType	Size
LastName	ftString	20
FirstName	ftString	20
Department	ftSmallint	
Branch	ftString	20
Senior	ftBoolean	
HireDate	ftDate	

```
object Table1: TTable
  FieldDefs = <
    item
      Name = 'LastName'
      DataType = ftString
```

239 A very similar mechanism is available for the FDMemTable component, which offers design time commands for creating a table or a `CreateDataset` method. The same approach of using `FieldDefs` described in the following paragraphs applies to FDMemTable and other datasets.

```
      Size = 20
    end
    item
      Name = 'Department'
      DataType = ftSmallint
    end
```

This new database table, called Workers, is intended to store some data about the employees of a company. (Note that calling it "Employee" might have caused a name conflict or confusion with one of the predefined tables.)

note The effect of the `StoreDefs` property is more complex than it seems at first. If you right-click the form, you'll notice that its local menu offers an Update Table Definition option, along with the expected Delete Table and Rename Table. That is, you can store the field definitions locally, but if the structure of the physical table changes, you should then update this definition, as well. In previous versions of Delphi, the field definitions were invariably loaded from the database table at run time; now you can preload them, speeding up the table opening. However, if the local and the actual table definitions do not match, you can get in trouble.

As we've seen, the DbAware example creates the table at start-up, unless it was already created. The program then opens up the table. To avoid having you type in data to start using the program, I've added to the program a simple `AddRandomData` method:

```
const
  FirstNames : array [1..10] of string =
    ('John', 'Paul', 'Mark', 'Joseph', 'Bill',
    'Peter', 'Tim', 'Ralph', 'Bob', 'Gary');
  LastNames : array [1..10] of string =
    ('Ford', 'Osborse', 'White', 'MacDonald', 'Lee',
    'Young', 'Parker', 'Reed', 'Gates', 'Green');
  NoDept = 3;
  NoBranch = 30;
  NewRecords = 10;

procedure TDbaForm.AddRandomData;
var
  I: Integer;
begin
  Randomize;
  for I := 1 to NewRecords do
    Table1.InsertRecord ([
      LastNames [Random (High (LastNames)) + 1],
      FirstNames [Random (High (FirstNames)) + 1],
      Random (NoDept) + 1,
      DbComboBox1.Items [Random (NoBranch)],
      Boolean (Random (2)),
      Date - Random (1000)]);
  ShowMessage (IntToStr (NewRecords) + ' added');
```

```
end;
```

`AddRandomData` calls the `InsertRecord` method of the table, which adds new data in a direct way—without setting the table in insert mode, setting the value of the fields, and then posting the data. In other examples, we'll see alternative approaches for adding data to a database table. Notice also that for the "branches" field I've used the list of values available in the associated data-aware combo box.

Listing Alternative Values

Now that I've created the table I can use it for creating a simple demo application of some of the other data-aware controls available in Delphi. For example, we can connect the Boolean field, Senior, with a DBCheckBox control. This allows a user to change the status of the field by clicking the control and setting or removing the check mark.

While this is quite trivial, using the components that list alternative values requires a little extra effort. There are basically three components with this capability: the DBListBox, the DBComboBox, and the DBRadioGroup. In general, all the three components provide a selection, which saves the user some typing and reduces the chance of input errors. If the three components seem similar, providing a list of strings in the `Items` property, they do have some differences:

- The DBListBox component allows selection of predefined items ("closed selection"), but not text input, and can be used to list many elements. Generally it's best to show only about six or seven items, to avoid using up too much space on the screen.
- The DBComboBox component can be used both for closed selection and for user input. It also uses a smaller area of the form because the drop-down list is usually displayed only on request.
- The DBRadioGroup component allows only a closed selection, should be used only for a limited number of alternatives, and allows a mapping of the display `values` to different internal values, through the Values string list.

In the DbAware example I've used the combo box for the selection of a country and a radio group for the selection for the department. This is actually saved in the database with a code, so I've mapped it as follows:

```
object DBRadioGroup1: TDBRadioGroup
  Caption = 'Department'
  DataField = 'Department'
```

```
  DataSource = DataSource1
  Items.Strings = (
    'Sales'
    'Accounting'
    'Production'
    'Management')
  Values.Strings = (
    '1'
    '2'
    '3'
    '4')
end
```

You can see an example of this program in Figure 9.6. Notice that the program is based on a page control: moving to the second page, you can see the database data inside a DBGrid. This should help you understand the mapping done by the RadioGroup control. The other element is that the main page doesn't allow you to edit the hire date, which is displayed in a read-only DBText control. We'll see how to handle dates in later examples.

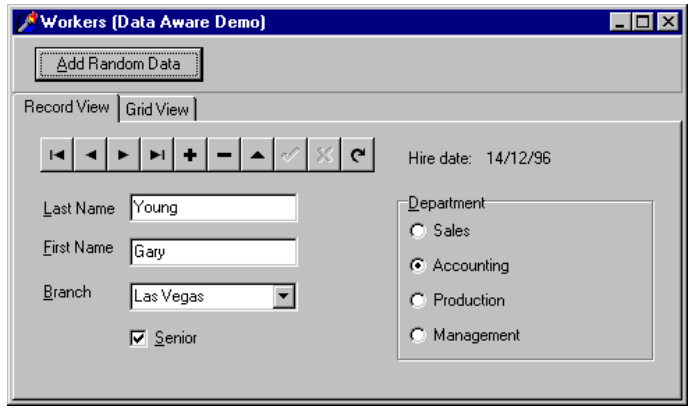


Figure 9.6: The output of the DbAware demo, which uses check box, combo box, and radio group data-aware controls. Image from the original book.

Accessing the Data Fields

Before we try to build more attractive and complex application examples, there are a few more technical elements we should explore. Up to now, we have included all of the fields in the source database tables. Suppose that we want to remove a field or add a new one, such as calculated fields? In trying to solve these problems, we face a

more general question: How do we access the values—the fields—of the current record from a program? How can we change them without direct editing by the user?

The answer to all of these questions lies in the concept of *field*. Field components (instances of class `TField` or of one of its subclasses) are non-visual components that are fundamental for every Delphi database application. Data-aware controls are directly connected to these Field objects, which correspond to database fields.

In the examples we have built up to now, Delphi automatically created the `TField` classes at run time[240]. This happens each time the program opens a data set component. These fields are stored in the `Fields` array property of tables and queries, which is an array of fields. We can access these values in our program by number (accessing the array directly) or by name (using the `FieldByName` method or the array notation):

```
Table1.Fields[0].AsString
Table1.FieldByName('LastName').AsString
Table1 ['LastName'].AsString
```

As an alternative, the field components can be created at design time, using the Fields editor. In this case, you can also set a number of properties for these fields at design time. These properties affect the behavior of the data-aware controls using them, both for visualization and for editing. When you define new fields at design time, they are listed in the Object Inspector, just like any other component.

note Although the Fields editor is similar to the editors of the collections used by Delphi, fields are not part of a collection. They are components created at design time, listed in its published section of the form class, and available in the drop-down combo box at the top of the Object Inspector.

To open the Fields editor for a table, select a Table object, activate its local menu with a right-click, and choose the Fields Editor command. Double-clicking the table component produces the same effect. An empty Fields editor appears. Now you have to activate the local menu of this editor, to access its capabilities. The simplest operation you can do is to select the Add command, which allows you to add any other fields in the database table to the list of fields. Figure 9.7 shows the Add Fields dialog box, which lists all the fields that are available in a table. These are the database table fields that are not already present in the list of fields in the editor.

The Define command of the Fields editor, instead, lets you define a new calculated field, a lookup field, or a field with a modified type. In this dialog box, you can enter

240 The concept of `TField` and the role of these objects in Delphi database application hasn't changed at all, even if some new features have been added over the years.

a descriptive field name, which might include blank spaces. Delphi generates an internal name—the name of the field component—that you can further customize. Next, select a data type for the field. If this is a calculated field or a lookup field, and not just a copy of a field redefined to use a new data type, simply check the proper radio button. We'll see how to define a calculated field in the section "Adding a Calculated Field" and a lookup field in the next chapter.

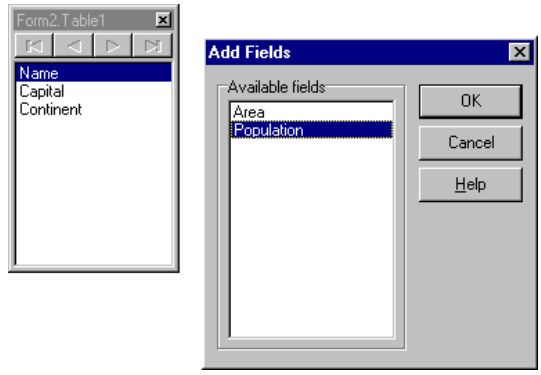


Figure 9.7: The Fields editor with the Add Fields dialog box. Images from the original book.

note A `TField` component has both a `Name` property and a `FieldName` property. The `Name` property is the usual component name. The `FieldName` property is either the name of the column in the database table or the name you define for the calculated field. It can be more descriptive than the `Name`, and it allows blank spaces. The `FieldName` property of the `TField` component is copied to the `DisplayLabel` property by default, but this field name can be changed to any suitable text. It is used, among other things, to search a field in the `FieldByName` method of the `TDataSet` class and when using the array notation.

All of the fields that you add or define are included in the Fields editor and can be used by data-aware controls or displayed in a database grid. If a field of the original database table is not in this list, it won't be accessible. When you use the Fields editor, Delphi adds the declaration of the available fields to the class of the form, as new components (much as the Menu Designer adds `TMenuItem` components to the form). The components of the `TField` class, or more specifically its subclasses, are fields of the form, and you can refer to these components directly in the code of your program to change their properties at run time or to get or set their value, as in the expression:

```
Table1LastName.AsString
```

In the Fields editor, you can also drag the fields to a different position to change their order. Proper field ordering is particularly important when you define a grid, which arranges its columns using this order.

note An even better feature of the Fields editor is that you can drag fields from this editor to the surface of a form and have Delphi automatically create a corresponding data-aware control (such as a DBEdit, a DBMemo, or a DBImage). The type of control created depends on the data type of the field and on eventual definitions in the Data Dictionary (as discussed in the next chapter). This is a very fast way to generate custom forms, and I suggest you try it out if you've never used it before. This is my preferred way to build database-related forms, much better than using the Database Form Wizard.

The Hierarchy of Field Classes

Before we look at an example, let's go over the use of the `TField` class. The importance of this component should not be underestimated. Although it is often used behind the scenes, its role in database applications is fundamental. As I already mentioned, even if you do not define specific objects of this kind, you can always access the `fields` of a table or a query using their `Fields` array property, the `FieldValues` indexed property, or the `FieldByName` method. Both the `Fields` property and the `FieldByName` function return an object of type `TField`, so you sometimes have to use the `as` operator to downcast their result to its actual type (like `TFloatField` or `TDateField`) before accessing specific properties of these subclasses.

The FieldAcc example is a simple extension of a form generated by the Database Form Wizard. I've added to it three speed buttons in the toolbar panel, accessing various Field properties at run time. The first button changes the formatting of the population column of the grid. To do this, we have to access the `DisplayFormat` property, a specific property of the `TFloatField` class. For this reason we have to write:

```
procedure TForm2.SpeedButton1Click(Sender: TObject);
begin
  (Table1.FieldByName ('Population') as
    TFloatField).DisplayFormat := '###,###,###';
end;
```

When you set field properties related to data input or output, the change applies to every record in the table. When you set properties related to the `value` of the field, instead, you always refer to the current record only. For example, we can output the population of the current country in a message box by writing:

```
procedure TForm2.SpeedButton2Click(Sender: TObject);
begin
  ShowMessage (string (Table1 ['Name']) +
    ': ' + string (Table1 ['Population']));
end;
```

When you access the value of a field, you can use a series of *As* properties to handle the current field value using a specific data type (if this is available, otherwise an exception is raised):

```
AsBoolean: Boolean;
AsDateTime: TDateTime;
AsFloat: Double;
AsInteger: LongInt[241];
AsString: string;
AsVariant: Variant;
```

These properties can be used to read or change the value of the field. Changing the value of a field is possible only if the DataSet is in edit mode. As an alternative to the *As* properties indicated above, you can access the value of a field by using its `Value` property, which is defined as a `Variant`.

Most of the other properties of the `TField` component, such as `Alignment`, `DisplayLabel`, `DisplayWidth`, and `Visible`, reflect elements of the field's user interface and are used by the various data-aware controls, particularly DBGrid. In the FieldAcc example, clicking the third speed button changes the `Alignment` of every field:

```
procedure TForm2.SpeedButton3Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to Table1.FieldCount - 1 do
    Table1.Fields[I].Alignment := taCenter;
end;
```

This affects the output of the DBGrid, and of the DBEdit control I've added to the toolbar, which shows the name of the country. You can see this effect, along with the new display format, in Figure 9.8.

---

241 The type of `AsInteger` is now Integer.

Figure 9.8: The output of the FieldAcc example after the Center and Format buttons have been pressed. Image from the original book.

There are several field class types in the VCL. Delphi automatically uses one of them depending on the data definition in the database, when you open a table at run time or when you use the Fields editor at design time. Table 9.2 shows the complete list of subclasses of the `TField` class[242].

Table 9.2: The Subclasses of TField (the field types in bold are new to Delphi 5 and relate with ADO support)

Subclass	Base Class	Definition
`TADTField`	`TObjectField`	An ADT (Abstract Data Type) field, corresponding to an object field in an object relational database.
`TAggregateField`	`TField`	An aggregate field represents a maintained aggregate. It is used in the ClientDataSet component and discussed in Chapter 21.
`TArrayField`	`TObjectField`	An array of objects in an object relational database.
`TAutoIncField`	`TIntegerField`	Whole positive number connected with a Paradox auto-increment field of a table, a special field automatically assigned a different value for each record. Note that Paradox AutoInc fields do not always work perfectly, as discussed in the next chapter.

242 There have been only a few additions to the list of `TField` descendant data types, including `TSQLTimeStampField` and `TFMTBCDField`.

`TBCDField`	`TNumericField`	Real numbers, with a fixed number of digits after the decimal point.
`TBinaryField`	`TField`	Generally not used directly. This is the base class of the next two classes.
`TBlobField`	`TField`	Binary data and no size limit (BLOB stands for Binary Large OBject). The theoretical maximum limit is 2GB.
`TBooleanField`	`TField`	Boolean value.
`TBytesField`	`TBinaryField`	Arbitrary data with a large (up to 64K characters) but fixed size.
`TCurrencyField`	`TFloatField`	Currency values, with the same range as the new `Real` data type.
`TDataSetField`	`TObjectField`	An object corresponding to a separate table in an object relational database.
`TDateField`	`TDateTimeField`	Date value.
`TDateTimeField`	`TField`	Date and time value.
`TFloatField`	`TNumericField`	Floating-point numbers (8 byte).
`TGraphicField`	`TBlobField`	Graphic of arbitrary length.
`TGuidField`	`TStringField`	A field representing a COM Globally Unique Identifier, part of the ADO support.
`TIDispatchField`	`TInterfaceField`	A field representing pointers to `IDispatch` COM interfaces, part of the ADO support.
`TIntegerField`	`TNumericField`	Whole numbers in the range of long integers (32 bits).
`TInterfaceField`	`TField`	Generally not used directly. This is the base class of fields that contain pointers to interfaces (`IUnknown`) as data.
`TLargeIntField`	`TIntegerField`	Very large integers (64 bit).

TMemoField	TBlobField	Text of arbitrary length.
TNumericField	TField	Generally not used directly. This is the base class of all the numeric field classes.
TObjectField	TField	Generally not used directly. The base class for the fields providing support for object relational databases.
TReferenceField	TObjectField	A pointer to an object in an object relational database.
TSmallIntField	TIntegerField	Whole numbers in the range of integers (16 bits).
TStringField	TField	Text data of a fixed length (up to 8192 bytes).
TTimeField	TDateTimeField	Time value.
TVarBytesField	TBytesField	Arbitrary data, up to 64K characters. Very similar to the TBytes-Field base class.
TVariantField	TField	A field representing a variant data type, part of the ADO support.
TWideStringField	TStringField	A field representing a Unicode (16-bit per character) string.
TWordField	TIntegerField	Whole positive numbers in the range of words or unsigned integers (16 bits).

The availability of any particular field type, and the correspondence with the data definition, depends on the database in use. For example, InterBase doesn't support BCD, so you'll never get a BCDField for a table on the InterBase server. This is particularly true for the new field types that provide support for object relational databases.

Adding a Calculated Field

Now that you've been introduced to TField objects and seen an example of their run-time use, it is time to build a simple example based on the declaration of field objects at design time using the Fields editor. We can start again from the first

example we've built, GridDemo, and add a calculated field. The `COUNTRY.DB` database table we are accessing has both the population and the area of each country, so we can use this data to compute the population density.

To build the new example, named Calc, select the Table component in the form, and open the Fields editor (using the form's SpeedMenu). In this editor, choose the Add command, and select some of the fields. (I've decided to include them all.) Now select the Define command, and enter a proper name and data type (`TFloatField`) for the new calculated field, as you can see in Figure 9.9[243].

note It is obvious that as you create some field components at design time using the Fields editor, the fields you skip won't get a corresponding object. What might not be obvious is that the fields you skip will not be available even at run time, with `Fields` or `FieldByName`. When a program opens a table at run time, if there are no design-time field components, Delphi creates field objects corresponding to the table definition. If there are some design-time fields, however, Delphi uses those fields without adding any extra ones.

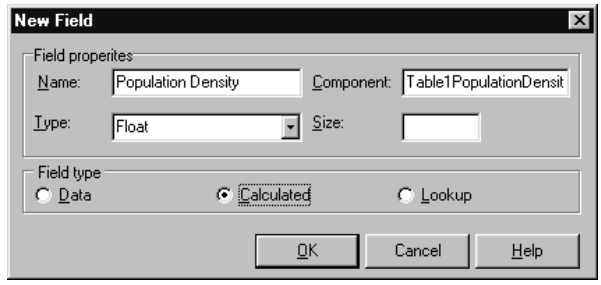


Figure 9.9: The definition of a calculated field in the Calc example. Image from the original book.

Of course, we also need to provide a way to calculate the new field. This is accomplished in the `OnCalcFields` event of the Table component, which has the following code (at least in a first version):

```
procedure TForm2.Table1CalcFields(DataSet: TDataSet);
begin
```

243 There is now a second flavor of calculated fields, called internally calculated fields and available for component with memory storage (like FDMemTable and ClientDataSet). The difference is that in this second case the calculated value is kept in memory and used for display. Rather than calculating the value each time the record becomes active, the calculation is performed only the first time or when one of the other fields of the same record changes.

```
  Table1PopulationDensity.Value :=
    Table1Population.Value / Table1Area.Value;
end;
```

Everything fine? Not at all! If you enter a new record and do not set the value of the population and area, or if you accidentally set the area to zero, the division will raise an exception, making it quite problematic to continue using the program. As an alternative, we could have handled every exception of the division expression and simply set the resulting value to zero:

```
  try
    Table1PopulationDensity.Value :=
      Table1Population.Value / Table1Area.Value;
  except
    on Exception do
      Table1PopulationDensity.Value := 0;
  end;
```

However, we can do even better. We can check if the value of the area is defined—if it is not null—and if it is not zero. It is better to avoid using exceptions when you can anticipate the possible error conditions:

```
  if not Table1Area.IsNull and
      (Table1Area.Value <> 0) then
    Table1PopulationDensity.Value :=
      Table1Population.Value / Table1Area.Value
  else
      Table1PopulationDensity.Value := 0;
```

The code of the `Table1CalcFields` method above (in each of the three versions) accesses some fields directly. This is possible because I used the Fields editor, and it automatically created the corresponding field declarations, as you can see in this excerpt of the interface declaration of the form:

```
type
  TCalcForm = class(TForm)
    Table1: TTable;
    Table1PopulationDensity: TFloatField;
    Table1Area: TFloatField;
    Table1Population: TFloatField;
    Table1Name: TStringField;
    Table1Capital: TStringField;
    Table1Continent: TStringField;
    procedure Table1CalcFields(DataSet: TDataset);
    ...
```

Each time you add or remove fields in the Fields editor, you can see the effect of your action immediately in the grid present in the form. Of course, you won't see the

values of a calculated field at design time; they are available only at run time, because they result from the execution of compiled Pascal code.

Since we have defined some components for the fields, we can use them to customize some of the visual elements of the grid. For example, to set a display format that adds a comma to separate thousands, we can use the Object Inspector to change the `DisplayFormat` property of some field components to *"###,###,###"*. This change has an immediate effect on the grid at design time.

note The display format I've just mentioned (and used in the previous example) uses the Windows International Settings to format the output. When Delphi translates the numeric value of this field to text, the comma in the format string is replaced by the proper `ThousandSeparator` character. For this reason, the output of the program will automatically adapt itself to different International Settings. On computers that have the Italian configuration, for example, the comma is replaced by a period.

After working on the table components and the fields, I've customized the DBGrid using its `Columns` property editor. I've set the Population Density column to read-only and set its `ButtonStyle` property to `cbsEllipsis`, to provide a custom editor. When you set this value, a small button with an ellipsis is displayed when the user tries to edit the grid cell. Pressing the button invokes the `OnEditButtonClick` event of the DBGrid:

```
procedure TCalcForm.DBGrid1EditButtonClick(Sender: TObject);
begin
  MessageDlg (Format (
     'The population density (%.2n)'#13 +
     'is the Population (%.0n)'#13 +
     'divided by the Area (%.0n).'#13#13 +
     'Edit these two fields to change it.',
     [Table1PopulationDensity.AsFloat,
     Table1Population.AsFloat,
     Table1Area.AsFloat]),
     mtInformation, [mbOK], 0);
end;
```

Actually, I haven't provided a real editor, but rather a message describing the situation, as you can see in Figure 9.10, which shows the values of the calculated fields. To create an editor, you might build a secondary form to handle special data entries.

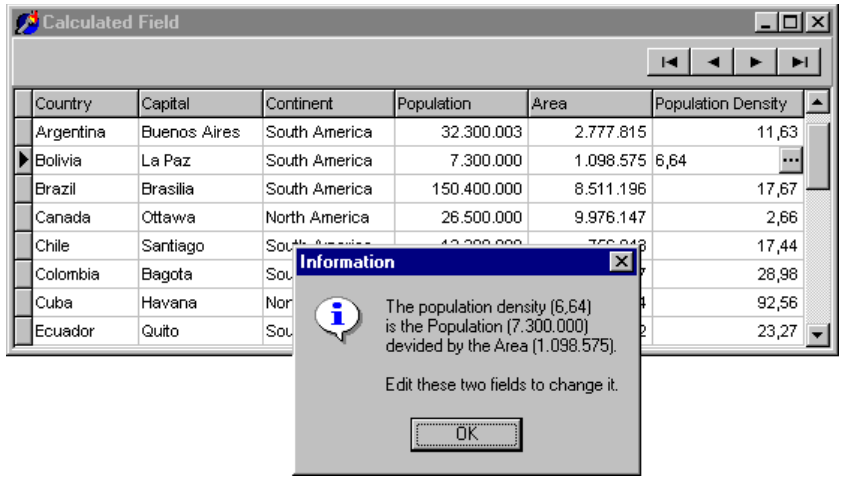


Figure 9.10: The output of the Calc example. Notice the Population Density calculated column, the ellipsis button, and the message displayed when you select it. Image from the original book.

Searching and Adding the Fields of a Table

`TField` components can be used to access data and manipulate a table at run time. We have seen only a limited example of direct data access; in the previous example, we used the value of two fields to calculate a third one. Now we will build some simple examples that will allow us to use the fields to search elements in a table, operate on the values, and access information about the tables of a database. There are many more possible uses of field components, but this should give you an idea of what can be done.

Looking for Records in a Table

For this example we need a new form, this time connected to EMPLOYEE.DB, another of the sample Delphi tables. To prepare the form, you can use the Database

Form Wizard or drag the fields from the Fields editor, an operation that will automatically add the corresponding labels[244].

note If you place the data-aware edit boxes inside a scroll box aligned to the client area, you can freely resize the form without any problems. When the form becomes too small, scroll bars will appear automatically in the area holding the edit boxes.

Instead of the default Delphi navigator component, we can add a standard Toolbar control and connect the buttons to some of the predefined dataset actions available in the ActionList component. I've simply added an ImageList to the form and connected it to the ActionList, to let the image list receive the images for the standard actions. Then I've added to the ActionList the predefined standard actions `TDataSetFirst`, `TDataSetLast`, `TDataSetNext`, and `TDataSetPrior`, plus two normal actions to host the custom search code.

Now you can simply connect the buttons of the toolbar with the corresponding actions and add to the toolbar an edit box where the user can enter the name to search for, as you can see in Figure 9.11. The buttons will carry out the proper action when pressed, and they will be disabled when the data set is at its beginning or end.

The searching capabilities are activated by the two buttons connected with custom actions. The first button is connected with the `ActionGoto`, used for an exact match, and the second with `ActionGoNear` for a nearest search. In both cases, we want to compare the text in the edit box with the LastName fields of the Employee table.

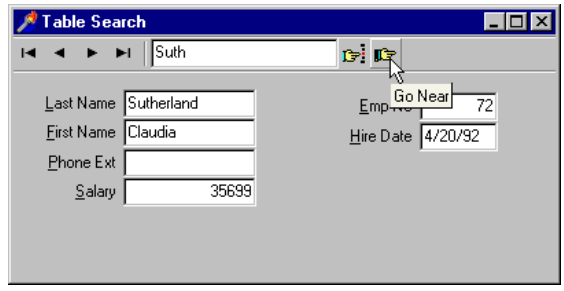


Figure 9.11: An example of a best-match search using the Search application. Image from the original book.

244 In terms of the database form wizards, this has never been ported to recent data access libraries and is not available today. Notice, instead, that the ability to drag fields from the field editor to the form to display a matching UI control is still available today. This is a very handy and fast way to build a UI, but it's little known and rarely used by Delphi developers. I'm really not sure why, considering it can be configured (in code) including the UI controls mapping.

The Table component has methods to accomplish this look up, such as `GotoKey`, `FindKey`, `GotoNearest`, `FindNearest`, and `Locate`. The `Locate` method uses the optimal access: if an index is available it uses the index for a faster search; otherwise, it does a plain sequential search. To use the first group of search methods, you need to set the `IndexFieldNames` property of the Table component to the proper value. (In this case, you can directly select the string *LastName;FirstName* in the drop-down list.)

The Find Methods

When the index is properly set, we can make the actual search. The simplest approach is to use the `FindNearest` method for the approximate search and the `FindKey` method to look for an exact match:

```
// goto
Table1.FindNearest ([EditName.Text]);

// go near
if not Table1.FindKey ([EditName.Text]) then
  MessageDlg ('Name not found', mtError, [mbOk], 0);
```

Both Find methods use as parameters an array of constants. Each array element corresponds to an indexed field. In our case, we pass only the value for the first field of the index, so the other fields will not be considered.

The Goto Methods

The `FindNearest` and `FindKey` methods are easy to use. To better understand how they work, though, we can look at the usage of the `GotoNearest` and `GotoKey` methods. These last two methods, in fact, map very closely to the actual low-level BDE calls. The simpler of the two is the best-guess search of the `GotoNearest` speed button:

```
// go near
Table1.SetKey;
Table1 ['LastName'] := EditName.Text;
Table1.GotoNearest;
```

As you can see in this code, each search on a table is done in three steps: start up the search state of the table, set a target value for each lookup field, and start the lookup process, by moving the current record to the requested position.

The code used to call the other search method, using an exact-match algorithm, is similar. The differences are in two statements:

```
// go to
Table1.SetKey;
Table1 ['LastName'] := EditName.Text;
Table1.KeyFieldCount := 1;
if not Table1.GotoKey then
  MessageDlg ('Name not found', mtError, [mbOK], 0);
```

As I've mentioned before, this code requires a proper index for the table. Notice the value set for the `KeyFieldCount` property, which indicates that I want to use just the first of the two fields that contribute to the index. The second difference is that the `GotoNearest` procedure always succeeds, moving the cursor to the closest match (a closest match always exists, even if it is not very close). On the other hand, the `GotoKey` method fails if no exact match is available, and you can check the return value of this function, and eventually warn the user of the error.

`FindKey` performs exactly the same steps as the `GotoKey` version of the above code. `FindKey` and `GotoKey` provide equivalent functionality, except that the former is easier to use and the latter provides for better error handling.

The Locate Method

If the table doesn't have an index on the field you are searching for (at least for local tables), you cannot use the two techniques above. A third, more general, technique is to use the `Locate` method. This approach is very handy in any case, because if there is an index on the field you are searching, `Locate` automatically uses it; otherwise it does a plain (and slower) search.

Using `Locate` is quite simple: Just provide a first string with the fields you want to search and a variant with the value or values you are searching for. To search for multiple fields, you need an array of values. (You can create one with the `VarArrayCreate` call.) Here is an example of its use, extracted again from the Search program:

```
// goto
if not Table1.Locate ('LastName', EditName.Text, []) then
  MessageDlg ('Name not found', mtError, [mbOk], 0);
```

The Total of a Table Column

So far in our examples, the user can view the current contents of a database table and manually edit the data or insert new records. Now we will see how we can change some data in the table through the program code. The idea behind this example is quite simple. The Employee table we have been using has a Salary field.

A manager of the company could indeed browse through the table and change the salary of a single employee. But what will be the total salary expense for the company? And what if the manager wants to give a 10 percent salary increase (or decrease) to everyone?

These are the two aims of the Total example, which is an extension of the previous program. The toolbar of this new example has two more buttons (and two related actions) and a SpinEdit component. There are few other minor changes from the previous example. I opened the Fields Editor of the table and removed the `Table1Salary` field, which was defined as a `TFloatField`. Then I selected the New Field command and added the same field, with the same name, but using the `TCurrencyField` data type. This is not a calculated field; it's simply a field converted into a new (but equivalent) data type. Using this new field type the program will default to a new output format, suitable for currency values.

Now we can turn our attention to the code of this new program. First, let's look at the code of the total action. This action lets you calculate the sum of the salaries of all the employees, then edit some of the values, and compute a new total. Basically, we need to scan the table, reading the value of the `Table1Salary` field for each record:

```
begin
  Table1.First;
  while not Table1.EOF do
  begin
    Total := Total + Table1Salary.Value;
    Table1.Next;
  end;
end
```

This code works, as you can see from the output in Figure 9.12, but it has a number of problems. One problem is that the record pointer is moved to the last record, so the previous position in the table is lost. To avoid this problem, we need to store the current position of the record pointer in the table and restore it at the end. This can be accomplished using a *table bookmark,* a special variable storing the position of a record in a database table. The traditional approach is to declare a variable of the `TBookmark` data type, and initialize it while getting the current position from the table:

```
var
  Bookmark: TBookmark;
begin
  Bookmark := Table1.GetBookmark;
```

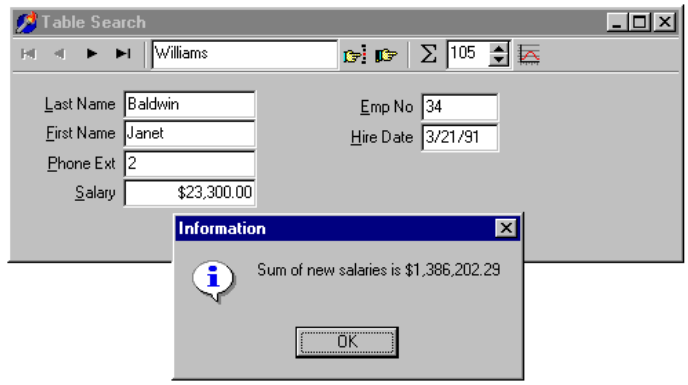


Figure 9.12: The output of the Total program, showing the total salaries of the employees. Image from the original book.

At the end of the `ActionTotalExecute` method, we can restore the position and delete the bookmark with the following two statements:

```
Table1.GotoBookmark (Bookmark);
Table1.FreeBookmark (Bookmark);
```

As a better alternative, we can use the `Bookmark` property of the `TDataset` class, which refers to a bookmark that is disposed of automatically. (This is technically implemented as an *opaque string*, a structure subject to string lifetime management, but it is not a string, so you're not supposed to look at what's inside it.) This is how you can modify the code above:

```
var
  Bookmark: TBookmarkStr;
begin
  Bookmark := Table1.Bookmark;
  ...
  Table1.Bookmark := Bookmark;
```

Another side effect of the program is that, although we will restore the record pointer to the initial position, we might see the records scrolling while the routine browses through the data. This can be avoided by disabling the controls connected with the table during browsing. The table has a `DisableControls` method we can call before the `while` loop starts and an `EnableControls` method we can call at the end, after the record pointer is restored.

note Disabling the data-aware controls connected with a table during long operations not only improves the user interface (since the output is not changing constantly), it also speeds up the program considerably. In fact, the time spent to update the user interface is much greater than the time spent performing the calculations. To test this, try commenting out the `DisableControls` and `EnableControls` methods of the Total example, and see the speed difference.

Finally, we face some dangers from errors in reading the table data, particularly if the program were reading the data from a server using a network. If any problem occurs while retrieving the data, an exception takes place, the controls remain disabled, and the program cannot resume its normal behavior. So we should use a `try-finally` block. Actually, if you want to make the program 100 percent error-proof you should use two nested `try-finally` blocks. Including this change and the two discussed above, here is the resulting code:

```
procedure TSearchForm.ActionTotalExecute(Sender: TObject);
var
  Bookmark: TBookmarkStr;
  Total: Real;
begin
  Bookmark := Table1.Bookmark;
  try
    Table1.DisableControls;
    Total := 0;
    try
      Table1.First;
      while not Table1.EOF do
      begin
        Total := Total + Table1Salary.Value;
        Table1.Next;
      end;
    finally
      Table1.EnableControls;
  end
  finally
    Table1.Bookmark := Bookmark;
  end;
  MessageDlg ('Sum of new salaries is ' +
    Format ('%m', [Total]), mtInformation, [mbOK], 0);
end;
```

I've written this code to show you an example of a loop to browse the contents of a table, but keep in mind that there is an alternative approach based on the use of a SQL query returning the sum of the values of a field.

note When you use a SQL server, the speed advantage of a SQL call to compute the total can be very large, since you don't need to move all the data of each field from the server to the client computer. The server sends the client only the final result.

Editing a Table Column

The code of the increase action is similar to the one we have just seen. The `ActionIncreaseExecute` method also scans the table, computing the total of the salaries, as the previous method did. Although it has just two more statements, there is a key difference. When you increase the salary, you actually change the data in the table. The two key statements are within the `while` loop:

```
while not Table1.EOF do
begin
  Table1.Edit;
  Table1Salary.Value := Round (Table1Salary.Value *
    SpinEdit1.Value) / 100;
  Total := Total + Table1Salary.Value;
  Table1.Next;
end;
```

The first statement brings the table into edit mode, so that changes to the fields will have an immediate effect. The second statement computes the new salary by multiplying the old one by the value of the SpinEdit component (by default, 105) and dividing it by 100. That's a 5 percent increase, although the values are rounded to the nearest dollar. With this program, you can change salaries by any amount—even double the salary of each employee—with the click of a button.

note Notice that the table enters the edit mode every time the `while` loop is executed. This is because in a dataset, edit operations can take place only one record at a time. You must finish the edit operation, calling `Post` or moving to a different record as in the code above. At that time, if you want to change another record, you have to enter edit mode once more.

Database Application with Standard Controls

Although it is generally faster to write Delphi applications based on data-aware controls, this is certainly not required. When you need to have very precise control over

the user interface of a database application, you might want to customize the transfer of the data from the field objects to the visual controls. My personal view is that this is necessary only in very specific cases, as you can customize the data-aware controls extensively by setting the properties and handling the events of the field objects. However, trying to work without the data-aware controls should help you understand the default behavior of Delphi, and it will help me introduce some of the database-related events (discussed in the sections "Database Events" and "Field Events").

The development of an application not based on data-aware controls can follow two different approaches. You can mimic the standard Delphi behavior in code, possibly departing from it in specific cases, or you can go for a much more customized approach. I'll demonstrate the first technique in the NonAware example and the latter in the SendToDb example.

Mimicking Delphi Data-Aware Controls

If you want to build an application that doesn't use data-aware controls but behaves like a standard Delphi application, you can simply write event handlers for the operations that would be performed automatically by data-aware controls[245]. Basically you need to place the data set in edit mode as the user changes the content of the visual controls, and update the field objects of the data set as the user exits from the controls, moving the focus to another element.

note This approach can be handy for integrating a control that's not data-aware, such as a DateTimePicker component, into a standard application.

The other element of the NonAware example is another list of buttons corresponding to some of those in the DBNavigator control. The five buttons are connected to five methods of the table component: `Next`, `Previous`, `Insert`, `Cancel`, and `Delete`. This is a summary of the Delphi form file:

```
object Form1: TForm1
  Caption = 'Non Aware'
  // 5 labels omitted
  object EditName: TEdit
    Text = 'EditName'
    OnExit = EditNameExit
```

245 The additional alternative available today is the use of Live Bindings to associated database fields and regular (non data-aware) Ui controls.

```
    OnKeyPress = EditKeyPress
  end
  object EditCapital: TEdit...
  object EditPopulation: TEdit...
  object EditArea: TEdit...
  object ComboContinent: TComboBox
    Items.Strings = (
      'South America'
      'North America'
      'Europe'
      'Asia'
      'Africa')
    Text = 'ComboContinent'
    OnDropDown = ComboContinentDropDown
    OnExit = ComboContinentExit
    OnKeyPress = EditKeyPress
  end
  // 5 buttons omitted
  object StatusBar1: TStatusBar
    SimplePanel = True
  end
  object DataSource1: TDataSource
    DataSet = Table1
    OnStateChange = DataSource1StateChange
    OnDataChange = DataSource1DataChange
  end
  object Table1: TTable
    Active = True
    AfterInsert = Table1AfterInsert
    BeforePost = Table1BeforePost
    DatabaseName = 'DBDEMOS'
    TableName = 'COUNTRY.DB'
    // 5 field objects omitted
  end
end
```

As you can see in the listing above, the program has several event handlers we've not used for past applications using data-aware controls. First of all, we have to show the data of the current record in the visual controls (as in Figure 9.13), by handling the `OnDataChange` event of the `DataSource1` component:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field:
TField);
begin
  EditName.Text := Table1Name.AsString;
  EditCapital.Text := Table1Capital.AsString;
  ComboContinent.Text := Table1Continent.AsString;
  EditArea.Text := Table1Area.AsString;
  EditPopulation.Text := Table1Population.AsString;
end;
```

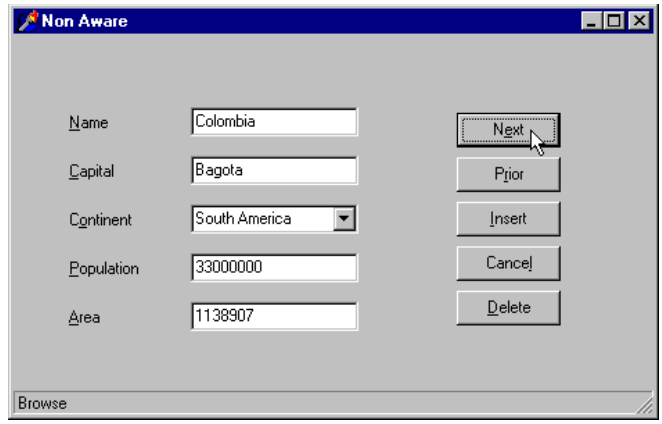


Figure 9.13: The output of the NonAware example in Browse mode. The program manually fetches the data every time the current record changes. Image from the original book.

The handler of the `OnStateChange` event of the control, instead, uses some code we've already seen in the GridDemo example. This time, the status of the table is displayed in a status bar control. As the user starts typing in one of the edit boxes or drops down the combo box list, the program sets the table in edit mode:

```
procedure TForm1.EditKeyPress(Sender: TObject; var Key: Char);
begin
  if not (Table1.State in [dsEdit, dsInsert]) then
    Table1.Edit;
end;
```

This method is connected with the `OnKeyPress` event of the five components and is similar to the `OnDropDown` event handler of the combo box. As the user leaves one of the visual controls, the handler of the `OnExit` event copies the data to the corresponding field, as in this case:

```
procedure TForm1.EditCapitalExit(Sender: TObject);
begin
  if (Table1.State = dsEdit) or (Table1.State = dsInsert) then
    Table1Capital.AsString := EditCapital.Text;
end;
```

The operation takes place only if the table is in Edit mode; that is, only if the user has typed in this or another control. This is not really ideal, because extra operations are done even if the text of the edit box didn't change, but the extra steps happen fast enough not to be a concern. For the first edit box, we check the text before copying it, raising an exception if the edit box is empty:

```
procedure TForm1.EditNameExit(Sender: TObject);
begin
```

```
  if (Table1.State = dsEdit) or (Table1.State = dsInsert) then
    if EditName.Text <> '' then
      Table1Name.AsString := EditName.Text
    else
    begin
      EditName.SetFocus;
      raise Exception.Create ('Undefined Country');
    end;
end;
```

An alternative approach for testing the value of a field is to handle the `BeforePost` event of the data set (with the effect shown in Figure 9.14). Keep in mind that in this example the posting operation is not handled by a specific button but takes place as soon as a user moves to a new record or inserts a new one:

```
procedure TForm1.Table1BeforePost(DataSet: TDataSet);
begin
  if Table1Area.Value < 100 then
    raise Exception.Create ('Area too small');
end;
```

In each of these cases, an alternative to raising an exception is to set a default value. However, if a field has a default value it is better to set it up front, so that a user can see which value will be sent to the database. To accomplish this, you can handle the `AfterInsert` event of a data set, which is fired immediately after a new record has been created (we could have used the `OnNewRecord` event, as well):

```
procedure TForm1.Table1AfterInsert(DataSet: TDataSet);
begin
  Table1Continent.Value := 'Asia';
end;
```

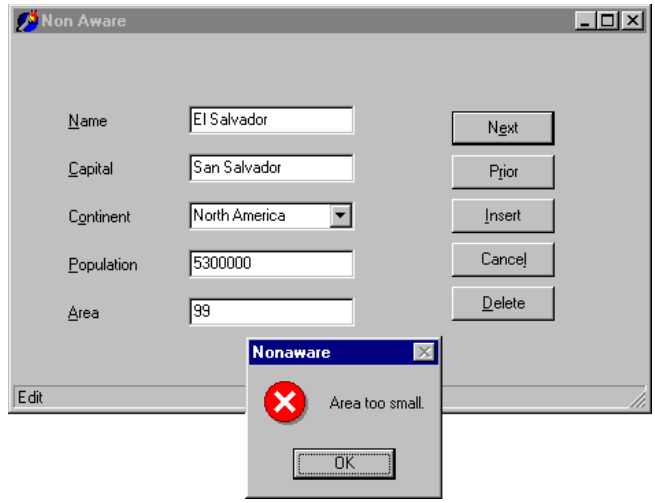


Figure 9.14: The error message displayed when the value of the area is too small. Image from the original book.

Sending Requests to the Database

You can further customize the user interface of your application if you decide not to handle the same sequence of editing operations as in standard Delphi data-aware controls. This allows you complete freedom, although there might be some side effects (such as limited ability to handle concurrency, which is something I'll discuss in the next chapter).

For this new example, I've replaced the first edit box with another combo box, and replaced all the buttons related to table operations (which corresponded to DBNavigator buttons) with two custom ones, used to get the data from the database and send an update to it. To underline the difference of this example, I've even removed the DataSource component.

The `GetData` method, connected with the corresponding button, simply gets the fields corresponding to the record indicated in the first combo box:

```
procedure TForm1.GetData;
begin
  Table1.FindNearest ([ComboName.Text]);
  ComboName.Text := Table1Name.AsString;
  EditCapital.Text := Table1Capital.AsString;
  ComboContinent.Text := Table1Continent.AsString;
  EditArea.Text := Table1Area.AsString;
```

```
  EditPopulation.Text := Table1Population.AsString;
end;
```

This method is called whenever the user presses the button, selects an item of the combo box, or presses the Enter key while in the combo box:

```
procedure TForm1.ComboNameClick(Sender: TObject);
begin
  GetData;
end;

procedure TForm1.ComboNameKeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then
    GetData;
end;
```

To make this example work smoothly, at start-up the combo box is filled with all the names of the countries of the table:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // fill the list of names
  Table1.Open;
  while not Table1.Eof do
  begin
    ComboName.Items.Add (Table1Name.AsString);
    Table1.Next;
  end;
end;
```

With this approach, the combo box becomes a sort of selector of the record, as you can see in Figure 9.15. Notice that thanks to this selection, the program doesn't need navigational buttons.

Finally, the user can change the values of the controls and press the Send button. The code to be executed depends on whether the operation is an update or an insert. We can determine this by looking at the name (although with this code, a wrong name cannot be modified any more):

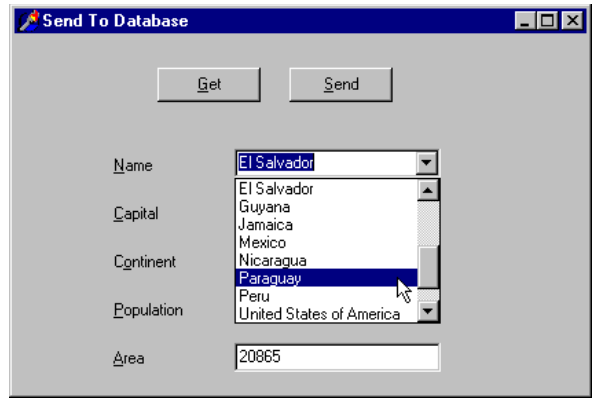


Figure 9.15: In the SendToDb example, you can select the record you want to see in a combo box. Image from the original book.

```
procedure TForm1.SendData;
begin
  // raise an exception if there is no name
  if ComboName.Text = '' then
    raise Exception.Create ('Insert the name');

  // check if the record is already in the table
  if Table1.FindKey ([ComboName.Text]) then
  begin
    // modify found record
    Table1.Edit;
    Table1Capital.AsString := EditCapital.Text;
    Table1Continent.AsString := ComboContinent.Text;
    Table1Area.AsString := EditArea.Text;
    Table1Population.AsString := EditPopulation.Text;
    Table1.Post;
  end
  else
  begin
    // insert new record
    Table1.InsertRecord ([ComboName.Text,
      EditCapital.Text, ComboContinent.Text,
      EditArea.Text, EditPopulation.Text]);
    // add to list
    ComboName.Items.Add (ComboName.Text)
  end;
```

Before sending the data to the table, you can do any sort of validation test on the values. In this case, it doesn't make much sense to handle the events of the database components, because we have full control on when the update or insert operation is done.

Database Events

To further illustrate how you can use the events of a database application, I've written a simple program that logs all the events being fired. This program handles all of the events of a table and a data source component (although some of these events won't actually be executed, unless you add some extra code, as described later). For each event, I simply send its description to a list box, with the effect you can see in Figure 9.16.

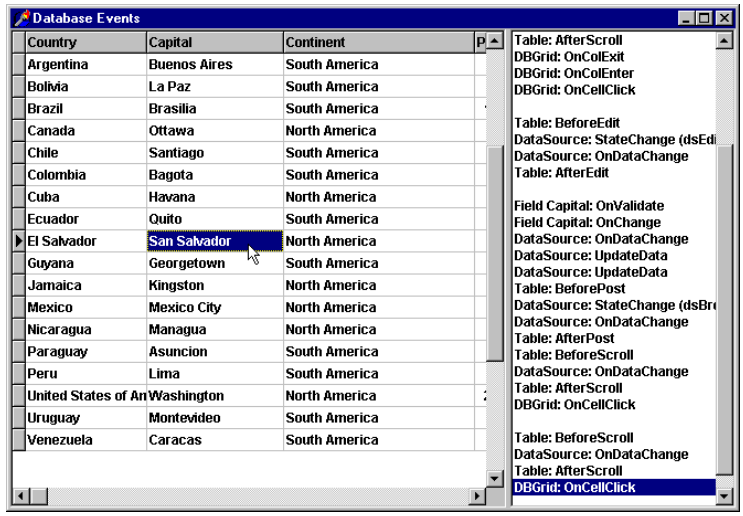


Figure 9.16: The output of the DbEvts program, which logs all the events related to database components. Image from the original book.

Most of the event handlers simply display the name of the component and that of the event, as in

```
procedure TForm1.Table1AfterEdit(DataSet: TDataset);
begin
  AddToList ('Table: AfterEdit');
end;
```

The field events are slightly more complex, but they use a single handler for the various field components:

```
procedure TForm1.FieldChange(Sender: TField);
begin
  AddToList ('Field ' + Sender.FieldName + ': OnChange');
end;
```

The form's `AddToList` method adds a new item to the list box and selects it, automatically scrolling the list if required:

```
procedure TForm1.AddToList(Str: string);
begin
  // add item and select it
  Listbox1.ItemIndex := Listbox1.Items.Add (Str);
end;
```

Finally, the program has a pop-up menu connected to the list box to clear the list or save the items to a file. The menu also has a command you can use to add a blank line, thus separating blocks of events. This operation is also done automatically by a timer, which adds a blank line to the list box unless the last item is already an empty string. This makes the output more readable, as you can see in Figure 9.16.

It is very important to study the output of this program as well as its code. You can try doing all the various operations on the table using the DBGrid, such as inserting, editing, and deleting records, and see the corresponding effect in terms of events fired by the VCL components. To see even more events, you can set the `Filtered` property of the table to `True`, define a calculated field, try to cause errors (for example, by duplicating the value of the name field), add a check box to open or close the table, and so forth.

Field Events

The DbEvts program shows the calls to the `OnChange` and `OnValidate` events of the field objects. Two other events, `OnSetText` and `OnGetText`, are not shown, because the handlers of these events are not simply called to indicate that an operation occurred. On the contrary, their event handler must perform the operation of getting data from or setting it to the corresponding field objects.

These two events are quite special, and their use is not as simple as it might seem at first sight. For this reason, they require a separate example, named FldText. This is only a slight revision of the DbAware example described earlier in this chapter, replacing the DBRadioGroup control with a DBListbox control. The problem is that a DBListBox control directly connects with a string field, while I want to connect it with an integer field, with each value indicating an option. Of course, I don't want a user to see or select a number, so I have to map the numbers stored in the database to the strings visible on the screen. In the earlier example, the DBRadioGroup control provided that mapping. Now I have to use an alternative approach.

In the FldText example, the Department field has two handlers for the `OnGetText` and `OnSetText` events. In the `OnGetText` event handler you can extract the numeric value of the `Sender` field and set the value of the `Text` reference parameter:

```
procedure TDbaForm.Table1DepartmentGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  case Sender.AsInteger of
    1: Text := 'Sales';
    2: Text := 'Accounting';
    3: Text := 'Production';
    4: Text := 'Management';
  else
    Text := '[Error]';
  end;
end;
```

note In the code of the `OnGetText` event handler you cannot refer to the text of the field, for example, using the `DisplayText` property or the `GetData` method, since they would call the `OnGetText` event, in an infinite recursion.

In the `OnSetText` event handler you can examine the string and decide the value of the field, according to the conversion rule, in this case a simple mapping of values done with an `if-then-else` statement:

```
procedure TDbaForm.Table1DepartmentSetText(Sender: TField;
  const Text: String);
begin
  if Text = 'Sales' then
    Sender.Value := 1
  else if Text = 'Accounting' then
    Sender.Value := 2
  else if Text = 'Production' then
    Sender.Value := 3
  else if Text = 'Management' then
    Sender.Value := 4
  else
    raise Exception.Create (
      'Error in Department field conversion');
end;
```

The effect is that not only is the value visible in the DBListBox (as you can see in Figure 9.17), it also shows up in the DBGrid. By contrast, in the DbAware example, the grid displayed the numeric value.

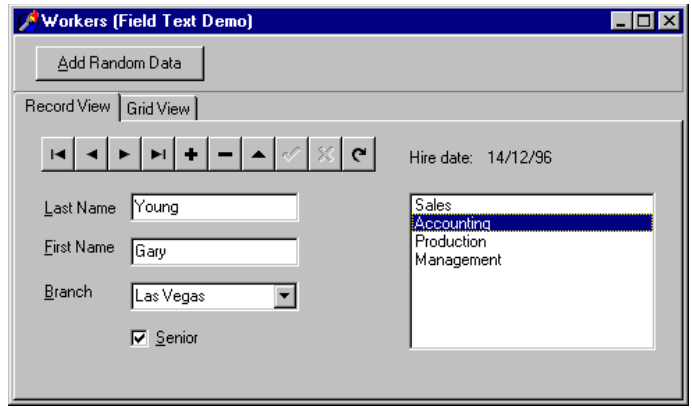


Figure 9.17: The output of the FldText example, which demonstrates the use of the OnGetText and OnSetText events of the field objects. Image from the original book.

Editing Dates with a Calendar

As a final example of the use of non–data-aware controls, the DbDates application shows how to use a MonthCalendar component to handle dates with a nice graphical component instead of a plain edit box. This example is based on the Events table from the DBDemos database, which lists Olympic events.

This example uses (for the first time) a DBImage control, with the following settings (whose effect is illustrated in Figure 9.18):

```
object DBImage1: TDBImage
  DataField = 'Event_Photo'
  DataSource = DataSource1
  Stretch = True
end
```

note Graphic, memo, and BLOB fields in Delphi are handled exactly like other fields. Just connect the proper editor or viewer, and most of the work is done behind the scenes by the system.

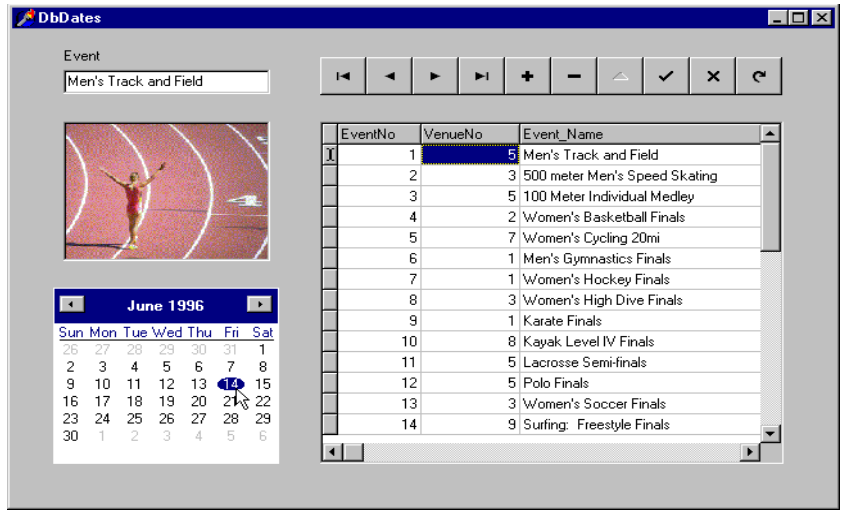


Figure 9.18: The selection of a date with the monthly calendar. Image from the original book.

Although the DBImage control works with no extra effort on our part, we must connect the MonthCalendar control with the corresponding field by handling two events of the DataSource control:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field:
TField);
begin
  MonthCalendar1.Date := Table1Event_Date.Value;
end;

procedure TForm1.DataSource1UpdateData(Sender: TObject);
begin
  Table1Event_Date.Value := MonthCalendar1.Date;
end;
```

Besides copying the data back and forth, with the code listed above, the program must also put the table into edit mode as the user clicks the calendar control. The most obvious approach is to write a handler for the `OnClick` event of the control:

```
procedure TForm1.MonthCalendar1Click(Sender: TObject);
begin
  Table1.Edit;
end;
```

However, this code doesn't work properly. As you set the table in edit mode, the `OnDataChange` event is executed once more, resetting the selection in the calendar. The overall effect is that the user's first click doesn't change the selection. To avoid this problem we can set a flag in the `OnClick` event handler and test it in the

`OnDataChange` event handler, or we can temporarily disconnect the second event handler. In the following code, I've taken the second approach:

```
procedure TForm1.MonthCalendar1Click(Sender: TObject);
begin
  // disconnect handler
  DataSource1.OnDataChange := nil;
  // set table in edit mode
  Table1.Edit;
  // reconnect handler
  DataSource1.OnDataChange := DataSource1DataChange;
end;
```

Exploring the Tables of a Database

In our examples so far, we have always accessed a database table by setting its name at design time. But what if you do not know which table your program will be connected to? At first, you might think that if you do not know the details of the database at design time, you won't be able to create forms and operate on the table. This is not true. Setting everything at design time is certainly easier. Changing almost anything at run time requires you to write more code. This is what I've done in the next example, called Tables, which demonstrates how to access the list of databases available to the Borland Database Engine[246], how to access the list of the tables for each database, and how to select which fields to view from a specific table.

Choosing a Database and a Table at Run Time

For the Tables example, I've prepared a form with a combo box you can use to select a database and a list box you can use to select a table of that database. The form also hosts a DBGrid, which can be connected with the selected database table. You can see the output of this program in Figure 9.19.

When the program starts, it fills the combo box, fills the list box (forcing the selection of the first item of the combo box), and then shows a table in the DBGrid (simulating the selection of the first item of the list box):

246 Something similar could be done by picking a stored FireDAC table file to use with the FDMemTable component.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Session.GetDatabaseNames (ComboBox1.Items);
  // force an initial list in the listbox
  ComboBox1.Text := 'DBDEMOS';
  ComboBox1Change (Self);
  // force an initial selection in the DBGrid
  ListBox1.ItemIndex := 0;
  ListBox1Click (Self);
end;
```

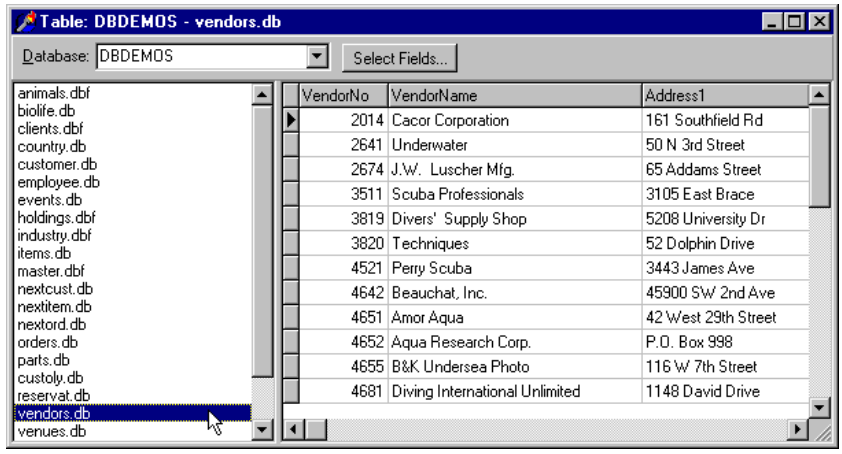


Figure 9.19: The output of the Tables program, which shows the data of a table selected at run time. Image from the original book.

The key element is the call to the `GetDatabaseNames` procedure of the `Session` global object[247]. An object of class `TSession` is automatically defined and initialized by each Delphi database application (even if you don't define one), and to access its methods, you only need to refer to the DBTables unit in the `uses` statement. When the combo box is filled, the program immediately selects one of the databases and then triggers the `ComboBox1Change` event handler, which uses another method of the `TSession` class, `GetTableNames`. This method has five parameters: the name of a database, a filter string, two Boolean values indicating whether to include the table file extensions (for local tables only) and whether to include system tables in the list (for SQL databases only), and the `TStringList` that will be filled with the names of the tables. Here is the code the program executes when the user selects an item in the combo box:

```
procedure TMainForm.ComboBox1Change(Sender: TObject);
```

247 This isn't available any more, as it was specific to the BDE.

```
begin
  Session.GetTableNames (ComboBox1.Text, '',
    True, False, ListBox1.Items);
end;
```

In the `FormCreate` method, a further step is automatically executed at start-up; the program fills the DBGrid as if a list box item had been selected:

```
procedure TMainForm.ListBox1Click(Sender: TObject);
begin
  Table1.Close;
  Table1.DatabaseName := ComboBox1.Text;
  Table1.Tablename := Listbox1.Items [Listbox1.ItemIndex];
  Table1.Open;
  Caption := Format ('Table: %s - %s',
    [Table1.DatabaseName, Table1.Tablename]);
end;
```

Viewing Multiple Tables

The program allows a user to see the content of any table. As a further extension, when the user double-clicks the list box, the program displays the grid in a separate form. This allows the user to open multiple modeless forms and see different tables at once, as you can see in Figure 9.20.

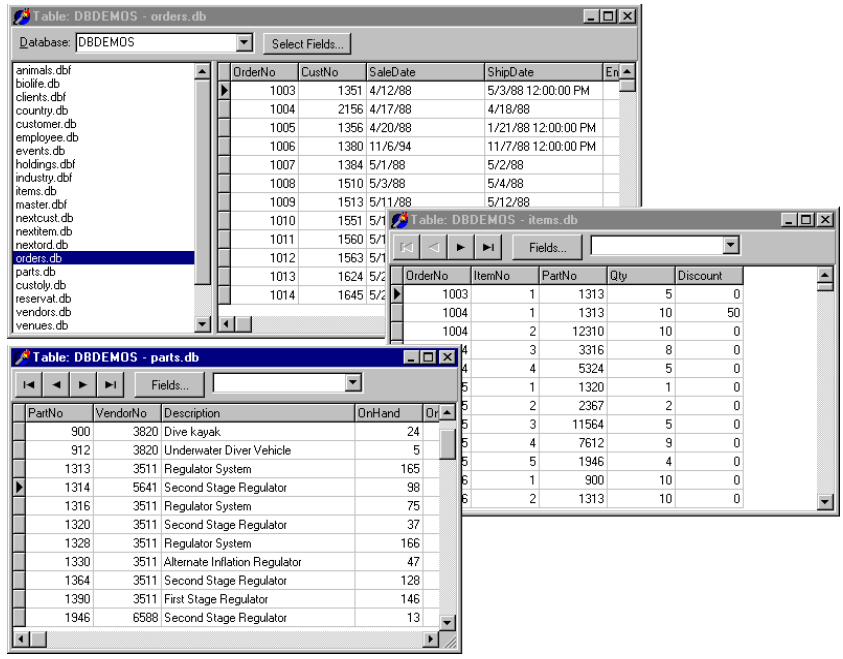


Figure 9.20: The Tables program can be used to open two or more grid-based table viewers. Images from the original book.

When the user double-clicks the list box in the main form, the code creates a `TGridForm` object, connects the `Table1` component of this form to the proper database and table, and shows the form:

```
procedure TMainForm.ListBox1DblClick(Sender: TObject);
var
  GridForm: TGridForm;
begin
  GridForm := TGridForm.Create (Self);
  {connect the table component to the selected
  table and activate it}
  GridForm.Table1.DatabaseName := ComboBox1.Text;
  GridForm.Table1.TableName :=
    Listbox1.Items [Listbox1.ItemIndex];
  try
    GridForm.Table1.Open;
    GridForm.Show;
  except
    GridForm.Close;
  end;
end;
```

note Notice that the code above simply creates the form and never destroys it. It is the responsibility of the form to delete itself in its `OnClose` event handler by setting the `Action` reference parameter to `caFree`.

When the secondary form is created, the program fills a combo box with the names of the fields of the table. However, this code can't go in the `OnCreate` event of the form, because the form is created before its `Table1` component is properly set up. Instead of adding a custom method and calling it, I've used the `OnShow` event handler, which also sets the caption of the form using the name of the table and the database:

```
procedure TGridForm.FormShow(Sender: TObject);
var
  I: Integer;
begin
  Caption := Format ('Table: %s - %s',
    [Table1.DatabaseName, Table1.TableName]);

  // fill the combo box with the names of the fields
  ComboBox1.Items.Clear;
  for I := 0 to Table1.FieldCount - 1 do
    ComboBox1.Items.Add (Table1.Fields[I].FieldName);
end;
```

note A possible extension to this program would be to generate a form based on data-aware controls, chosen depending on the type of field. You can find a Database Form Wizard capable of generating similar forms on my Web site, `www.marcocantu.com`.

What is the purpose of this combo box? Each time a user selects an element, the corresponding field is either shown or hidden, depending on its current state:

```
procedure TGridForm.ComboBox1Change(Sender: TObject);
begin
  // toggle the visibility of the field
  Table1.FieldByName (ComboBox1.Text).Visible :=
    not Table1.FieldByName (ComboBox1.Text).Visible;
end;
```

Notice the use of the `FieldByName` method to retrieve the field using the current selection of the combo box and the use of the `Visible` property. Once a field becomes invisible, it is immediately removed from the grid associated with the table. Therefore, by simply setting this property, we change the grid automatically.

The combo box I've placed in the toolbar of the GridForm works, but if you need to select several fields in a big table, it is slow and error-prone. As an alternative, I've

created a field-editor form, which is used both by the main and by the secondary form. This is the third form of the Tables example, named FieldsForm.

This form is displayed as a modal dialog box, so we can use a single global object every time. The new form has no code of its own. When the form is activated, its multiple-selection list box is filled with the names of the fields of the table. At the same time, the code selects the list box items corresponding to visible fields, as you can see in Figure 9.21.

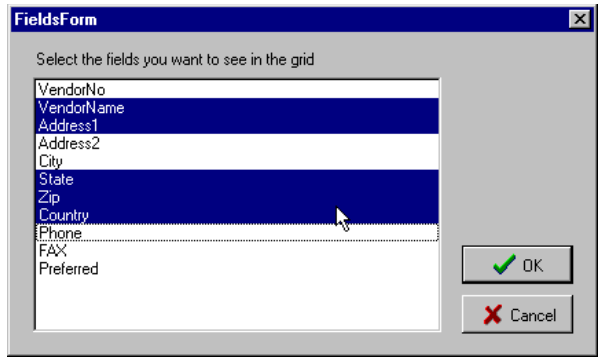


Figure 9.21: The list box can be used to select the table fields to show in the grid. Image from the original book.

The user can toggle the selection of each item in this list box while the modal form is active. When it is closed, the other form retrieves the values of the selected items and sets the Visible property of the fields accordingly. Here is the complete code of this method:

```
procedure TGridForm.SpeedButton1Click(Sender: TObject);
var
  I: Integer;
begin
  FieldsForm.FieldsList.Clear;
  for I := 0 to Table1.FieldCount - 1 do
  begin
    FieldsForm.FieldsList.Items.Add (
      Table1.Fields [I].FieldName);
    if Table1.Fields [I].Visible then
      FieldsForm.FieldsList.Selected [I] := True;
  end;
  if FieldsForm.ShowModal = mrOK then
     for I := 0 to Table1.FieldCount - 1 do
       Table1.Fields.Visible [I] :=
         FieldsForm.FieldsList.Selected [I];
  FieldsForm.FieldsList.Clear;
end;
```

This code ends the description of this example. We have seen that you can write database applications that do most of the work at run time, although this approach is slightly more complex.

A Multi-Record Grid

So far we have seen that you can either use a grid to display a number of records of a database table or build a form with specific data-aware controls for the various fields, accessing the records one by one. There is a third alternative: use a multi-record object (a DBCtrlGrid[248]), which allows you to place many data-aware controls in a small area of a form and automatically duplicate these controls for a number of records.

Here is what we can do to build the Multi1 example. Create a new blank form, place a Table component and a DataSource component in it, and connect them to the COUNTRY.DB table. Now place a DBCtrlGrid on the form, set its size and the number of rows and columns, and place two edit components connected with the Name and Capital fields of the table. To place these DBEdit components, you can also open the Fields editor and drag the two fields to the control grid. At design time, you simply work on the active portion of the grid (see Figure 9.22, on the right), and at run time, you can see these controls replicated a number of times (see Figure 9.22, on the left).

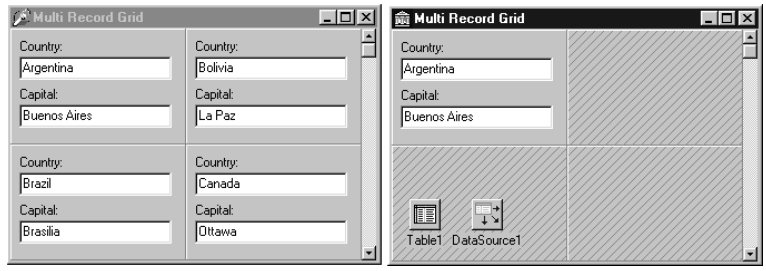


Figure 9.22: The DBCtrlGrid of the Multi1 example at design time (on the right) and at run time (on the left). Images from the original book.

Here are the most important properties of the DBCtrlGrid object and the other components of this example:

248 As mentioned early in this chapter, this component still exists but it's not commonly used.

```
object Form1: TForm1
  Caption = 'Multi Record Grid'
  object DBCtrlGrid1: TDBCtrlGrid
    ColCount = 2
    DataSource = DataSource1
    RowCount = 2
    object DBEdit1: TDBEdit
      DataField = 'Name'
      DataSource = DataSource1
    end
    object DBEdit2: TDBEdit...
  end
  object Table1: TTable
    Active = True
    DatabaseName = 'DBDEMOS'
    TableName = 'COUNTRY.DB'
  end
  object DataSource1: TDataSource
    DataSet = Table1
  end
end
```

Actually, you can simply set the number of columns and rows. Then each time you resize the control, the width and height of each panel are set accordingly. What is not available is a way to align the grid automatically to the client area of the form.

Moving Control Grid Panels

To improve the last example, we might resize the grid using the `FormResize` method. We could simply write the following code (in the Multi2 example):

```
procedure TForm1.FormResize(Sender: TObject);
begin
  DBCtrlGrid1.Height := ClientHeight - Panel1.Height;
  DBCtrlGrid1.Width := ClientWidth;
end;
```

This works, but it is not what I want. I'd like to increase the number of panels, not enlarge them. To accomplish this, we can define a minimum height for the panels and compute how many panels can fit in the available area each time the form is resized. For example, in Multi2, I've added one more statement to the `FormResize` method above, which now becomes

```
procedure TForm1.FormResize(Sender: TObject);
begin
  DBCtrlGrid1.RowCount :=
    (ClientHeight - Panel1.Height) div 100;
```

```
  DBCtrlGrid1.Height := ClientHeight - Panel1.Height;
  DBCtrlGrid1.Width := ClientWidth;
end;
```

Instead of doing the same for the columns of the control grid component, I've added a TrackBar component to a panel. When the position of the trackbar changes (the range is from 2 to 10), the program sets the number of columns of the control grid and resizes it. In fact, if you simply set the number of columns, they'll have the same width as before. Here is the code of the trackbar's `OnChange` event handler:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  LabelCols.Caption := Format (
    '%d Columns', [TrackBar1.Position]);
  DBCtrlGrid1.ColCount := TrackBar1.Position;
  DBCtrlGrid1.Width := ClientWidth;
end;
```

This code and the `FormResize` method above allow you to change the configuration of the control grid at run time in a number of ways. You can see an example of a crammed version of the form in Figure 9.23.

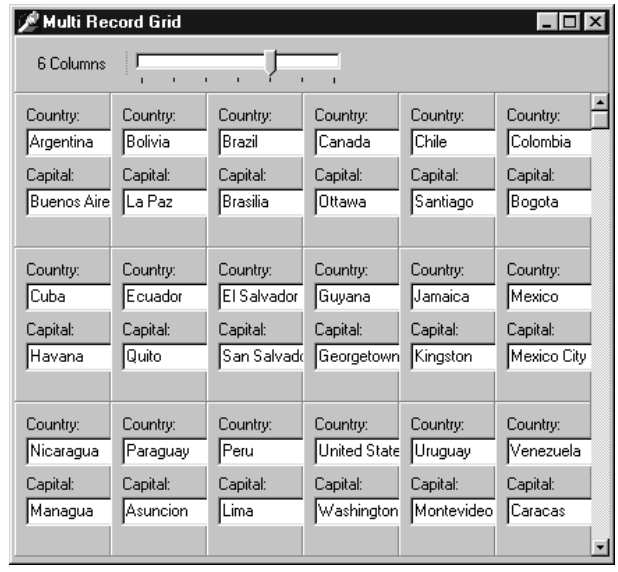


Figure 9.23: The output of the Multi2 example, with an excessive number of columns. Image from the original book.

Database Charts

Another interesting component you can use in database applications is the data-aware version of the TeeChart control built by David Berneda and available in the Professional and Enterprise versions of Delphi[249]. This component is very easy to use, particularly if your version of Delphi includes the corresponding TeeChart Wizard (found in the Business page of the File ➢ New dialog box).

To demonstrate the use of the DBChart control, I've added this component to the GridDemo example. The new application, called ChartDB, shows a DBGrid in the upper portion and a pie chart with the surface of each country at the bottom, as you can see in Figure 9.24.

The program has almost no code, as all the settings can be done using the specific component editor, which has a number of options but is quite easy to use. Here are some of the key properties of the component, taken from the form description:

```
object DBChart1: TDBChart
  Legend.Visible = False
  Align = alClient
  object Series1: TPieSeries
    Marks.ArrowLength = 8
    Marks.Visible = True
    DataSource = Table1
    XLabelsSource = 'Name'
    ExplodeBiggest = 3
    OtherSlice.Style = poBelowPercent
    OtherSlice.Text = 'Others'
    OtherSlice.Value = 2
    PieValues.ValueSource = 'Area'
  end
end
```

249 The light version of the component is still available in Delphi as an extra installer opotion.

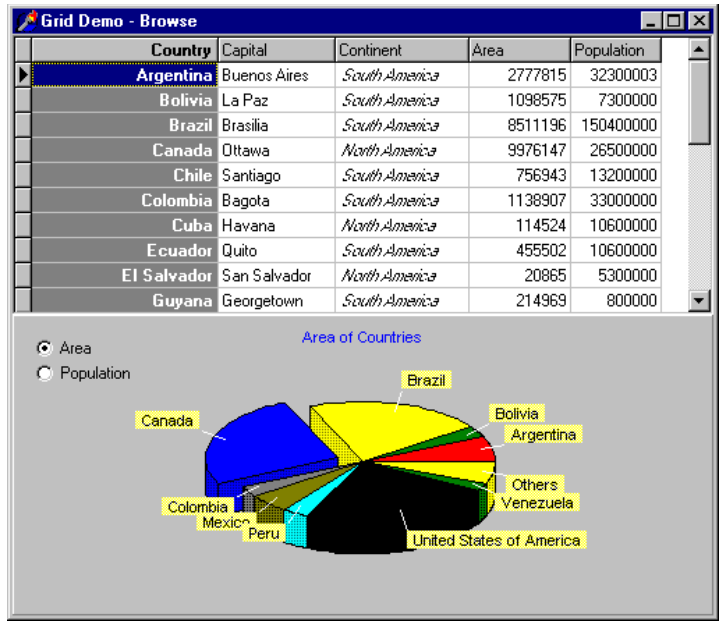


Figure 9.24: The output of the ChartDb example, which is based on the TDbChart control. Image from the original book.

note To understand these properties and the structure of the charts and the series, you can refer to the examples of the Chart component in the Chapter 22 “Graphics in Delphi”. This same chapter shows also how to dynamically export from a Web server application the graph produced by the DBChart, after converting it to a JPEG image.

What I’ve done was to show the area field as the data source for the pie chart (the `PieValues.ValueSource` property of the series), use the name field for the labels (the `XLabelsSource` property of the series), and condense all the countries with a value below 2 percent in a single section indicated as ‘Others’ (the `OtherSlide` sub-properties).

As a minor addition to the code, I’ve added two radio buttons you can use to toggle between the area and the population. The code of the two radio buttons simply sets the source of the series, after casting it to the proper series type, as in:

```
procedure TForm1.RadioPopulationClick(Sender: TObject);
begin
  DBChart1.Title.Text [0] := 'Population of Countries';
  (DBChart1.Series [0] as TPieSeries).
    PieValues.ValueSource := 'Population';
end;
```

What's Next?

In this chapter, we have seen a number of examples of database access from Delphi programs. I have covered the basic data-aware components as well as the development of database applications based on standard controls. We've explored the internal architecture of the field objects, created brand-new database tables at design time and at run time, and worked though many examples.

In particular, besides looking at the use of the data-aware controls, we've also used a couple of different manual approaches. You might wonder when the harder and lower-level approach might make sense. The short answer is to use the data-aware controls unless you need to do something unusual that conflicts with the default behavior of the data-aware controls. A typical example is the use of particular techniques for concurrency in multi-user applications, as we'll see in the next two chapters.

Is this all there is to say about Delphi database programming? Not at all. Delphi database support is very extensive and complete. The purpose of this chapter has been to give you an idea of what you can do, concentrating on the use of the Table component for database access. In the next chapter, we'll focus on the Query component, on working with multiple database tables (with joins and with master-detail and lookup structures), and on many other advanced features. We'll also cover the use of the new Data Module Designer in Delphi 5.

Chapter 10: Advanced Database Access

In the previous chapter we saw how Delphi makes it easy to create database applications. All of the sample programs were based on a single table and used the Table component to access it. Moreover, all the code related to the user interface was mixed with the code related to database access; using data modules, we'll be able to keep the two functional areas separate.

These are just few of the topics explored in this chapter, which is devoted to slightly more advanced database techniques[250]: the Data Dictionary, BDE calls, table joins through SQL, master/detail connections, and lookup fields. Then, in Chapters 11 and 12, we'll move on to client/server programming and ADO components.

---

250 Some of these are not applicable any more in today's Delphi.

First, however, we start this chapter by discussing one of the most important innovations Delphi 5 provides to database programmers, the data module Designer. This tool allows a new visual approach to structuring database applications, significantly extending Delphi's traditional data modules.

The Delphi 5 Data Module Designer[251]

In the previous chapter we placed both the data-access controls and the data-aware controls in forms. This is handy for a simple program, but having the user interface and the data access and data model in a single (often large) unit is far from a good idea. Delphi has since version 2 used the idea of a *data module*, a container of non-visual components.

At design time a data module is similar to a form, but at run time it exists only in memory. The `TDataModule` class derives directly from `TComponent`, so it is completely unrelated to the Windows concept of a window. And unlike a form, a data module has just a few properties and events. For this reason, it's useful to think of data modules as components and method containers in memory.

However, data modules are similar to forms in many respects. Like a form, a data module is related to a specific Object Pascal unit for the definition of its class and to a form definition (DFM) file that list the components included in the module and their properties. Here is some code from the DFM file of a data module:

```
object DataModule2: TDataModule2
  Height = 159
  Width = 196
  object Table1: TTable...
  object DataSource1: TDataSource...
end
```

Also, the structure of the Delphi unit for a data module is very similar to that of a form. The key difference is in the parent class:

```
type
  TDataModule2 = class (TDataModule)
```

---

251 The "Designer" portion of the data module has long been removed form the Delphi IDE. The data module itself, though, it still a foundation of Delphi development. This is an area with very significant changes and I'll highlight them in notes, as usual.

Another thing forms and data modules have in common is that they can both be created either when the application starts or later. In fact, data modules are even listed in the Forms page of the Project Options.

There are several reasons to use data modules. The simplest one is to share data-access components among multiple forms, as I'll demonstrate in the TwoViews example. This technique works in conjunction with visual form linking, the ability to access components of another form or data module at design time (with the File ➢ Use Unit command). The second reason is to separate the data from the user interface, improving the structure of an application. In fact, Delphi allows you to further extend this model to a full three-tier system, using the MIDAS[252] technology (which will be briefly introduced in Chapter 20).

In Delphi 5, the Data Module Designer adds even more reasons to use data modules. You can select components and connect them in an easier way, using the Tree View, and you can see the overall design of a database application (or part of it) and even connect properties and components graphically, using the Data Diagram view[253]. The Data Module Designer is an extension of the data module. Every time you create a new data module (by using File ➢ New and selecting Data Module in the resulting New Items dialog box) or open an existing one, you will see the new designer.

The left side of the Data Module Designer hosts a tree of the components on the container, organized in a logical hierarchy. The right side has two pages, which show the Components view or the Data Diagram, depending on the selected tab. The Components view corresponds to the original data modules in earlier versions of Delphi; initially, it is an empty white window, where you can add components (but not controls).

The Tree View

The Tree view of the Data Module Designer starts with a single node, the data module itself[254]. You can select a component from the palette (let's say a Table), move the mouse over the tree, and drop it. The designer will start organizing the information logically, adding two extra nodes to the tree: a BDE session and a database alias, as you can see in Figure 10.1. The session is simply the default session, which

252 This is the technology later renamed DataSnap and it still exists today.

253 The Data Diagram is the feature that's no longer available.

254 This feature has been folded (or better, expanded) into today's Structure View, which offers a similar view of the logical relationship among data components.

is a Delphi global object rather than a specific component. For some purposes you might replace it with a `TSession` component having specific properties[255]. The second node (under the session) is an alias. You cannot edit it directly but you can change it by assigning a value to the `Database` property of the table.

These two nodes correspond to "fake" or "dummy" components, and for this reason their icons are grayed. Technically, gray icons are used for components that do not have design-time persistence. They are real components (at design time and at run time), but because they are default objects constructed at run time and have no persistent data that can be edited at design time, the Data Module Designer does not allow you to edit their properties.

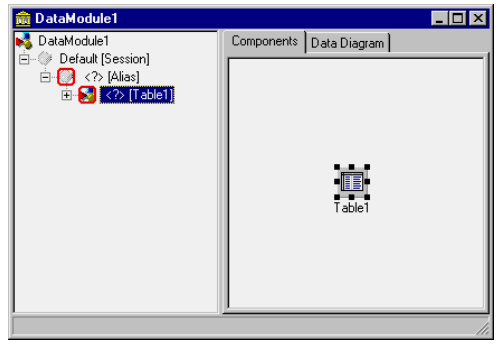


Figure 10.1: The Data Module Designer of a new data module after you add a table component to it. Image from the original book.

There are many operations you can do within Tree view. For example, after setting the alias you can drag another table component under it to hook it up directly with the database. In a similar way, you can drop a data source component below a table to connect the two. You can also perform these dragging operations with components that are already in the Tree view—for example, to change the data set a data source refers to.

Note that this is a major aspect of the new designer—it saves you from the tedium of having to manually "wire up" the Session, Database, Table, and DataSource components using properties. In fact, the relationship you see in the tree generally relates a component to its "parent context," which used to be set up by assigning a property. Now you can determine a parent context or a container relationship by dragging items in the tree. For example, Session components are the context in which one or more Database components operate, Tables and Queries operate within the context of a Database, and Field objects live inside Tables and Queries.

255 As mentioned in the previous chapter, this `TSession` component is specific to the BDE library and no matching concept exists today.

Right-clicking any element of the Tree view displays a shortcut menu similar to the component menu you get when the component is in a form (and in both cases case the shortcut menu may include items related to the custom component editors). You can even delete items from the tree; if an item has sub-items, Delphi will prompt for confirmation before destroying them.

The Data Diagram View[256]

If the Tree view provides a few extra features compared to the traditional data module, what's completely new in Delphi 5 is the Data Diagram view. This view shows dependencies among components, including master/detail relationships, lookup connections, linked properties, and generic relationships. You can even add your comments in text blocks linked to specific components. Note that even though it is part of a data module, this view is not limited to database-related components; it works with any non-visual component (menus, actions, and so on).

The Data Diagram view is not built automatically. You have to drag components from the Tree view to the diagram, although it displays the connections directly if you have already set them up. What's nice is that you can create connections and set properties by simply drawing arrows among the components. For example, after moving a table and a data source to the Data Diagram view, you can select the Property connector icon, click the data source, and drag the mouse cursor over the table. When you release it the designer will set up the property relationship, as you can see in Figure 10.2. The arrow will automatically show the name of the property used to hook the two components, in this case `Dataset`. As you can see, setting properties is *directional*: if you drag the property relationship line from the table to the data source you end up using the data source for the `MasterSource` property of the table, hooking up the two components in the opposite way.

That's not all. You can also drag specific fields to the view, and they will be connected to the table with a child relationship, marked by a white arrow. You can even move an ActionList to the tree, and drag it to the diagram. The actions will be shown as child items, and if they relate to a dataset they may be connected to the data source. To produce Figure 10.3, I've taken these steps, added a comment, added some text to it by double clicking it, and hooked the comment to a component with a custom relationship. Besides database components, you can use any non-visual component in the Data Module Designer, including the ActionList, menus, Internet

256 As mentioned already, this view doesn't exist any more. It was dropped when moving to the *new* IDE architecture with Delphi 2007. There is now a totally different concept, the Live Binding designer, which has a different role: associating data with UI controls.

producer and dispatcher components, MIDAS connections, decision cube components, and even application servers.

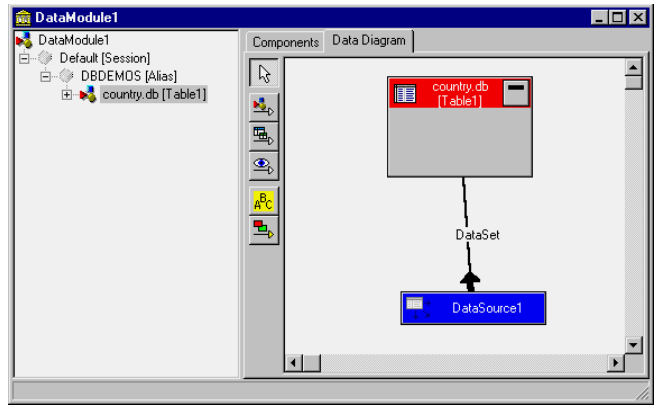


Figure 10.2: A property relationship between two components in the Data Diagram view. Image from the original book, as this feature no longer exists.

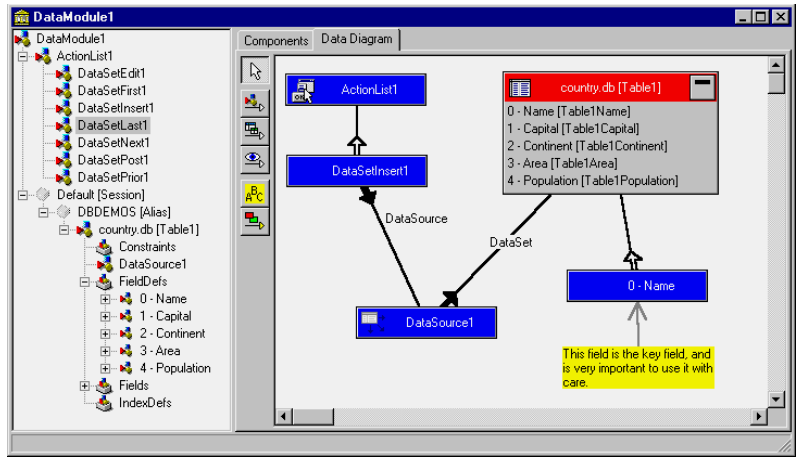


Figure 10.3: A diagram showing complex relationships among database and non-database components (such as actions). Image from the original book.

Although you can use the Data Diagram view, usually in a large window, to set up relationships, its main role is to document your design. For this reason it is important to be able to print the content of this view, but it is nice also to be able to print the Tree view, particularly when it is too long to fit on the screen. To make more tables fit in the diagram you can always toggle the list of fields inside them by clicking the minimize/maximize icon in the top-right corner of the box.

The information in the Data Diagram view is saved in a separate design-time information (DTI) file, not as part of the DFM file. DTI files have a similar structure to INI files, and are obviously useless at run time (it makes no sense to include them in the compilation of the executable file).

A Data Module for Multiple Views

As I mentioned earlier, one of the traditional uses of a data module is to provide different views of the same data and to keep the views in synch. This is what I've done in the TwoViews example. Later on, I'll extend this example by adding data rules and filtering capabilities to the program. In the TwoViews example, I've created two forms and a data module[257]. The data module includes a table related to the CUSTOMER.DB file of the DBDEMOS database and a data source. I've also created `TField` components for each of the fields of the table. You can see the resulting data module in Figure 10.4 (which shows the final structure of the data module, including a field and the related index).

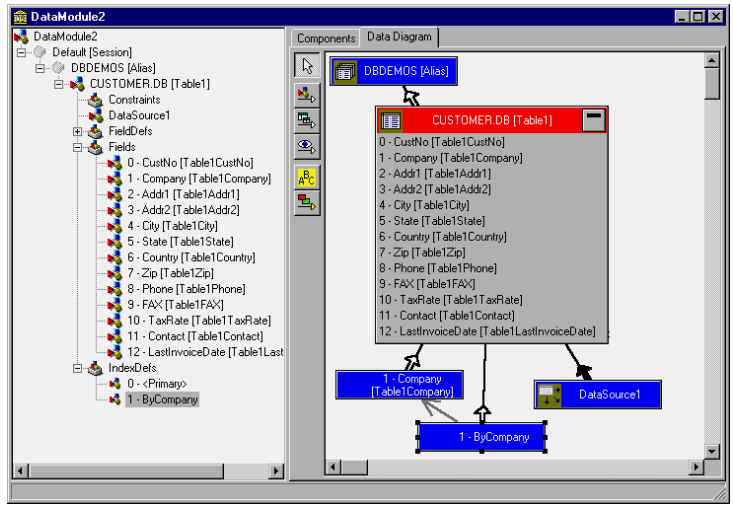


Figure 10.4: The data module of the TwoViews example. Image from the original book.

257 The core concepts are still fully applicable, if you ignore the visual representation of the components relationships.

I've also built a toolbar for the main form of the program, using a panel aligned to the top, a speed button, and a DBNavigator. The speed button has an appropriate icon and is used to show the secondary form. The rest of the main form is filled with a DBGrid control. After connecting the data module to the form, with the File ➢ Use Unit menu command, you can set the `DataSource` property of both the DBNavigator and the DBGrid to `DataModule2.DataSource1`.

note Before you use another unit in a form, you should properly name the unit to which you want to refer. In fact, if you use a unit (for example, Unit2) and then rename it when you first save the file, the connection will be lost, and you'll need to replace all the references to the renamed unit manually. This happens even if Delphi built those `uses` statements automatically or with the File ➢ Use Unit command.

The second view is based on a form with many DBEdit components, one for each field of the database table except the last. Instead of placing a number of DBEdit components and connecting each of them, you can open the Fields editor of the Table component, add all the fields, select all of them except the last, and then drag the selected fields to the secondary form. With this simple operation, you have Delphi arrange all the proper DBEdit and Label components on the form at once. I've actually set the `Visible` property of the secondary form to `True`, so that it becomes immediately visible when the program starts, as you can see in Figure 10.5.

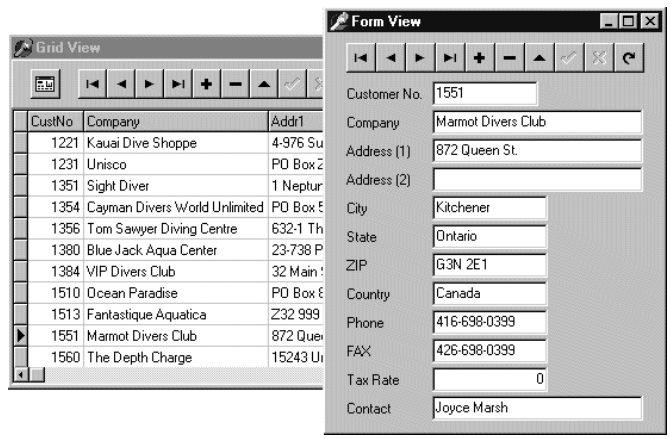


Figure 10.5: The TwoViews program at run time, with the two synchronized forms referring to the same record. Image from the original book.

If you display both forms, they are kept in sync. Using either form's navigator affects both forms. In fact, the navigator is connected to neither of them: it is connected to the data source in the data module, and the visual components of both forms are affected by any change in the common data access components. Edit one

form, and the other will be updated as soon as you accept the changes. Add a new record, and the action will take place on both forms.

Note that you can also navigate through the records at design time. While you scroll the grid, the data in the secondary form will change, allowing you to size the DBEdit components if any field of the current record is too long to fit in the available space, for example.

Setting Field Properties and Initial Values

Using a data module to keep two forms in synch can be handy and is quite simple. We want to add to the program some more capabilities related to the data itself, not to the specific viewer. For example, in the data module we can edit the properties of the fields, using a special value for the `EditMask` properties of the `Table1Phone` and `Table1FAX` field components. This customization will affect the output and the editing of these fields in both forms at the same time.

To accomplish something a little more complex, we can introduce a rule in the table or at least a suggestion to the users. We want to automatically provide a new unique value for the customer number and make it the current highest value plus one for this field. I've accomplished this by adding some code to the data module.

Basically, we want to set the proper value of the `Table1CustNo` field each time the user adds a new record to the table. To accomplish this we can handle the `OnNewRecord` event of the table as follows:

```
procedure TDataModule2.Table1NewRecord(DataSet: TDataSet);
begin
  Table1CustNo.Value := Max + 1;
end;
```

How do we compute the `Max` value? We can simply browse the table, as we did in the last chapter, and check for the highest value of the CustNo field. However, we cannot do this in the event handler above, because this will put the table back in `tsBrowse` mode from the `tsInsert` mode. An alternative is to recompute the highest value each time the user inserts a new record, using the `BeforeInsert` event:

```
procedure TDataModule2.Table1BeforeInsert(DataSet: TDataSet);
begin
  ComputeMax;
end;
```

This `ComputeMax` procedure might scan the table looking for the maximum value, with code like this:

```
Max := 0;
try
  Table1.First;
  while not Table1.EOF do
  begin
    if Table1CustNo.AsInteger > Max then
      Max := Table1CustNo.AsInteger;
    Table1.Next;
  end;
```

An alternative is to add a second `TTable` component, indexed on the CustNo field. The `ComputeMax` code would then simply look at the end of the table for the highest value of CustNo:

```
procedure TDataModule2.ComputeMax;
begin
  Table2.Last;
  Max := Table2CustNo.AsInteger;
end;
```

By adding some methods to the data module, we move toward the structure of a three-tier application. This code, in fact, is completely independent from the user interface (the two views). The code of this example is very simple, but it is meant to highlight this important idea.

note An alternative to the code above, which computes the maximum identifier used by a table, is to use an auto-increment field, at least if you are using a Paradox table[258]. This would be better in a multi-user environment or a multi-threaded program, where the approach just shown might lead to problems in case of two concurrent requests.

Standard Table Filtering

Now we want to add to the application the capability of filtering the records in both views (again using the data module). The simplest filtering capability in Delphi tables is to set a range of values for an indexed field. For example, I've ordered the TwoViews program's table using the *ByCompany* secondary index (just select this value for the `IndexName` property). Then I've chosen all records between two values supplied by the user, by writing the following:

```
Table1.SetRange (['Abacus'], ['Custom']);
```

258 Relational databases often include a mechanism like sequences and generators to obtain a similar result to auto increment fields. This is a more complex topic than I can cover here.

As an alternative you can set key values as in the `GotoKey` method, calling the `SetRangeStart`, `SetRangeEnd`, and `ApplyRange` methods in sequence. Usually it is much simpler to call `SetRange` and pass it two arrays of values, with the same number of items (and the same order) as the fields in the current index. When you want to stop applying the range, simply call the `CancelRange` method.

Actually, in the TwoViews program I didn't indicate a fixed range of values, as suggested above; instead, I added to the data module the `ChooseRange` method, which is going to be called by the two views. As the data module has no visual interface by itself, it uses a dialog box to ask the user for the initial and final values of the range. I could have used the toolbar of the main form, instead of a dialog box, but I wanted to relate the code used to set the range to the data module itself, not to a specific form used to view the data. Another approach is to dock the modeless dialog to the side of the main form, as shown in Figure 10.6. You can see in the figure that the secondary form has more components, which we'll use later to customize table filtering. Also notice the effect of the range on the table (and the grid) content.

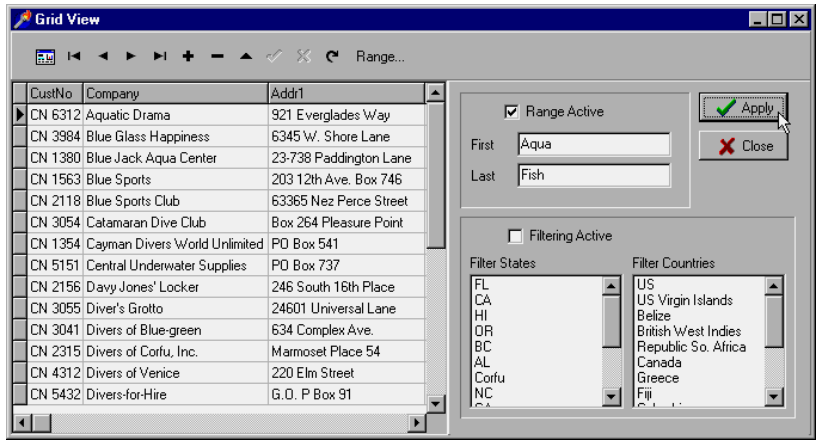


Figure 10.6: The secondary form used to set a range and a filter on the table can be docked to a panel on the side of the main form. Image from the original book.

The `ChooseRange` method simply displays the `FormRange` form (as a modeless form). The form has an Apply button you can use to activate the new settings in the data module:

```
procedure TFormRange.BitBtn1Click(Sender: TObject);
begin
  with DataModule2.Table1 do
  begin
    if CheckBoxRange.Checked then
      SetRange ([Edit1.Text], [Edit2.Text])
    else
```

```
      CancelRange;
  end;
end;
```

Custom Table Filtering

Besides giving the Table component a range of values to work on, we can set a custom filtering algorithm. Simply set the `Filtering` property of the Table component to `True`, and for each record the `OnFilterRecord` event will be called. In the method connected to this event, we can set a custom filter. Here is an example:

```
procedure TDataModule2.Table1FilterRecord(
  DataSet: TDataSet; var Accept: Boolean);
begin
  if (Table1Country.Value = 'US') or
      (Table1Country.Value = 'US Virgin Islands') or
      (Table1State.Value = 'Jamaica') then
    Accept := True
  else
    Accept := False;
end;
```

Again, connecting this filtering rule to the data module will affect each of the two views. Besides writing a fixed rule, as in the case above, we can allow the user to build his or her own rule, with the components added to the lower portion of the range dialog box; namely, another check box and two list boxes, as shown in Figure 10.6. The lists are filled with the names of the countries and the states when the form is created:

```
procedure TFormRange.FormCreate(Sender: TObject);
begin
  with DataModule2 do
  begin
    Table1.First;
    while not Table1.EOF do
    begin
      // add unique values
      if not Table1Country.IsNull and
        (ListBoxCountries.Items.IndexOf (
          Table1Country.AsString) < 0) then
        ListBoxCountries.Items.Add (Table1Country.AsString);
      if not Table1State.IsNull and
        (ListBoxStates.Items.IndexOf (
          Table1State.AsString) < 0) then
        ListBoxStates.Items.Add (Table1State.AsString);
      Table1.Next;
    end;
```

```
    // reset the table
    Table1.First;
  end;
end;
```

This code checks to see whether the value of the current record is null or is already present in the list box. If it is not, the value is added to the proper list box. These two lists should be updated each time a new record is added to the database table, or whenever an existing record changes. I've omitted this capability, but it should be quite simple for you to implement it by handling the `AfterPost` event of the table and writing two lines of code, similar to the body of the `while` loop above, referring to the new or updated record.

Once the program has filled the list boxes, they are displayed along with the range options. The Apply button also sets the filters and refreshes the table:

```
with DataModule2.Table1 do
  begin
    ...
    Filtered := CheckBoxFiltering.Checked;
    Refresh;
```

The `Refresh` call is necessary because if the rules change when table filtering is already active, Delphi will not automatically recompute the current active records. The most important piece of the filtering code is the handler of the `OnFilterRecord` events, which checks if the country or state of the current record is one of the selected items of the two list boxes (which allow multiple selection). Here is the code:

```
procedure TDataModule2.Table1FilterRecord(
  DataSet: TDataSet; var Accept: Boolean);
begin
  {if the item corresponding to the country in the
  listbox is active, then view the record}
  with FormRange.ListBoxCountries do
    Accept := Selected [Items.IndexOf (Table1Country.AsString)];
  with FormRange.ListBoxStates do
    if Selected [Items.IndexOf (Table1State.AsString)] then
      Accept := True;
end;
```

Notice that in the second `if` statement, the value of `Accept` should be added to the previous one with an `or` statement. Actually, we can simply set it to `True` regardless of the previous value (since an `or` with `True` always returns `True`) or let it maintain its current value (since an `or` with `False` keeps the existing value).

An MDI Application with Independent Views

The TwoViews example is an SDI application, with two separate forms floating on the screen. This is not always the best user interface; as discussed in Chapter 8, an alternative is to use the MDI approach. Another limitation of the program so far is that the two views always share the same current record. It would be nice to build an application in which each view could have a different active record. And so it would also be reasonable to create multiple record and grid views. This is what I've done in the following example, called MdiView.

Since we need to have forms that each show a different current record, you might be tempted to remove the data module altogether and place the database-related components in the forms. That might work in this simple example, but in general it is better to keep the logical separation and the designer provided by the data modules. The alternative solution to the problem, in fact, is to keep the data modules in the program and simply create a new copy for each form to connect.

note If you place a `TDatabase` component in a data module, you cannot create multiple instances of the data module unless you set the `HandleShared` property to `True`. If you fail to do so, the program generates the exception "Name not unique in this context," as the two database components in the same database session cannot have the same name.

The main form of the MdiView application has the `fsMdiFrame` value for the `FormStyle` property. It is the only form created at startup, so that it is displayed empty. The child forms are created using the commands of the File menu and they create the data modules automatically. For this reason I've not only removed the child forms and the data module from the list of the automatically created forms in the project options, I've even removed the global variables referring to these objects.

note Again, removing the global variables for forms that will have multiple instances is generally a good idea, so that you don't confuse one of the instances of the form (referenced by the global variable) with the class itself.

The code for creating child forms is quite simple. There is no need to keep track of the forms created, as the Windows MDI support does this automatically for us:

```
procedure TFrameForm.NewRecordView1Click(Sender: TObject);
begin
  with TRecordForm.Create (Application) do
    Show;
end;
```

As the view is created, it generates a data module and hooks it to a local `DM` field, declared inside each form. Then the program connects all the data aware controls to the data source of the newly created data module:

```
procedure TRecordForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  DM := TCustomerDM.Create (self);
  // connect the navigator
  DBNavigator1.DataSource := DM.DataSource1;
  // connect all DBEdit controls
  for I := 0 to ControlCount - 1 do
    if Controls [i] is TDBEdit then
      TDBEdit (Controls [I]).DataSource :=
        DM.DataSource1;
end;
```

The only other code for the two child forms is the closing code, which destroys the form, setting the `Action` parameter of the `OnClose` event handler to `caFree`.

The data module has almost no code, as I've removed the calculation of the maximum ID used in this chapter's earlier examples. The only operation done by the data module is changing the title of the connected form to reflect the current record. This is useful, as the list of MDI child windows becomes meaningless if all the windows of the same type have the same title. To accomplish this I've used a trick—storing the `Hint` property of each form (which is not used) as the first part of the caption, followed by the Company field of the current record:

```
procedure TCustomerDM.DataSource1DataChange(
  Sender: TObject; Field: TField);
begin
  (Owner as TForm).Caption :=
    (Owner as TForm).Hint + ' - ' + Table1Company.AsString;
end;
```

You can see an example of the output of this program in Figure 10.7, with three child forms open and few more minimized. Notice that the different captions we've set for each form are reflected in the Windows menu; this is handled automatically by the MDI support.

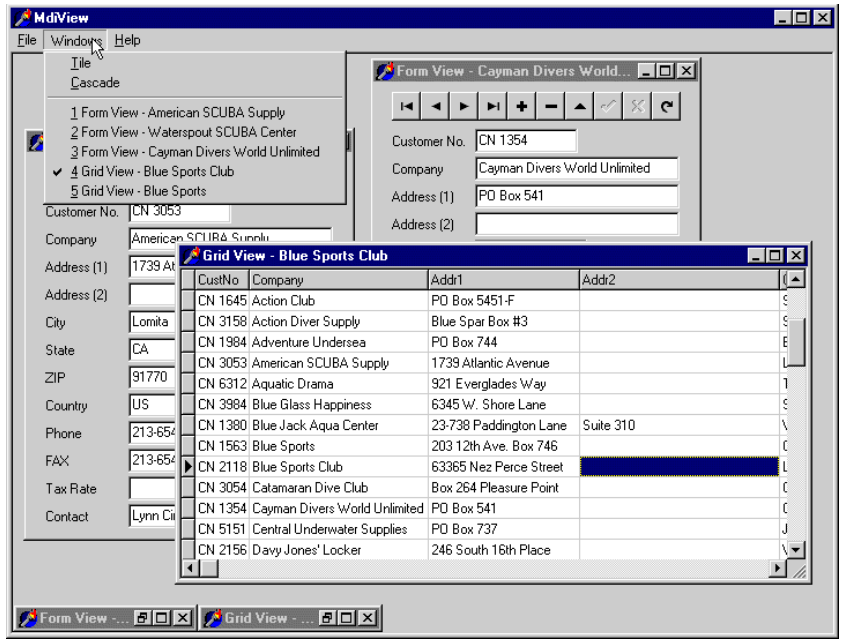


Figure 10.7: The output of the MdiView program, which connects a different data module object to each view. Image from the original book.

Using a Query

All the database examples up to this point used a Table component. The next example accesses the data using a Query component, instead, and is called DynQuery—or dynamic query. I've connected the query to the usual `DBDEMO` database alias and entered the text of a simple SQL statement:

```
select * from Country
```

Simply activate the Query component and the values of the fields of the first record should appear in the edit boxes as usual. Of course, this happens only if the SQL statement you have inserted is correct. Otherwise, Delphi will issue an error message, and the query won't be activated. If you want to change the current SQL statement of a query at run time, you need to set the `Active` property to `False` first. After changing the text of the query you can then activate it again.

note In the Enterprise edition of Delphi, you can build queries using a graphical query builder called SQL Builder[259]. This is a two-way tool, which can interpret queries you've written in text and show them graphically. You'll see examples of the use of the SQL Builder in Chapter 11.

Of course, this example won't be particularly interesting. Why use the Query component instead of the Table component if all we want is to select an entire table? We can take advantage of the Query component by adding some radio buttons to select different queries at run time. I decided to add four different options.

The first radio button is used to select the default SQL statement, and it is checked at startup. The second and third buttons can be used to choose only the records that have a specific value for their Continent field, adding a `where` clause to the SQL statement. The last radio button allows a user to enter the text of the `where` SQL statement directly in an edit box.

Letting a user type in a statement is dangerous, since entering the wrong text can cause an error, but Delphi's exception-handling support can help us to withstand this risk. Here is the code associated with the first radio button:

```
procedure TNavigForm.RadioButton1Click(Sender: TObject);
begin
  Query1.Close;
  Query1.Sql.Clear;
  Query1.Sql.Add ('select * from Country');
  Query1.Open;
end;
```

Notice that the `SQL` property is not a string, but a list of strings. This can be used to build very long queries (the text limit for an array of strings is high) and to define different portions of the query in different places of the code and merge them. The second and third radio buttons share the same code, which uses their `Caption` property to build the text of the SQL statement:

```
Query1.Sql.Clear;
Query1.Sql.Add ('select * from Country');
Query1.Sql.Add ('where Continent = "' +
  (Sender as TRadioButton).Caption + '"');
```

note In the SQL statements above I've used a double quotation mark for strings. It is also possible to use single quotation marks. However, to have a single quotation mark inside a Pascal string, you have to use two consecutive single quotation marks. For this reason, in the code above, we would have had triple and even quadruple quotation marks, certainly a confusing situation.

259 Also this tool, SQL Builder, doesn't exist any more.

For the last radio button, we only need to merge the default statement with the text of the edit box, handling a possible exception:

```
procedure TQueryForm.RadioButton4Click(Sender: TObject);
begin
  Query1.Close;
  if (Edit1.Text <> '') then
  begin
    Query1.Sql.Clear;
    Query1.Sql.Add ('select * from Country');
    Query1.Sql.Add ('where ' + Edit1.Text);
  end;
  try
    Query1.Open;
  except
    on EDatabaseError do
      ShowMessage ('Invalid condition:'#13 + Edit1.Text);
  end;
end;
```

This code is executed any time the edit box is not empty. It includes a test to make sure that the text is a correct SQL statement and displays a custom error message if it is not. To improve the program slightly, the last radio button is automatically disabled each time the edit box has no text. This check takes place in the `OnChange` event of the Edit component.

When the user presses Enter while in the edit box, the new condition is automatically activated, either by checking the radio button (to visually indicate the selection and make it trigger the event handler) or by calling the handler (because selecting a button that's already active doesn't fire its event handler):

```
procedure TQueryForm.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then
  begin
    if RadioButton4.Checked then
      RadioButton4Click (self)
    else
      RadioButton4.Checked := True;
    Key := #0;
  end;
end;
```

note The `Edit1KeyPress` method sets the Enter key to a null key, to avoid the beep produced by default when you press the Enter key in an edit box.

When you run this program, you can choose any of the four buttons and immediately see the effect on the list of records in the DBGrid. Figure 10.8 shows two

examples of the use of the edit box for customizing the SQL query at run time. One example shows a single country of the database selected, and the other shows the result of selecting a population range.

note By default, you cannot edit the result of a query. In order to do that, you must first set the `RequestLive` property of the Query to `True`. This makes the data fully editable only when some given conditions determined by the BDE are met. Simple queries referring to a single database table can generally be "live," while complex queries joining several tables generally cannot. The `TUpdateSQL` component and the cached updates technology allow you to make complex queries live, as we'll see in the next chapter.

As an alternative to the use of `where` SQL statements, you can use a table and set a range of records you want to consider or use the `Filtered` property, as demonstrated in the last example. Filters are also available for queries, but the natural way to filter a query is to use an SQL statement, so that the database engine or the SQL server elaborates the query instead of our program. This method is particularly appropriate if the database engine or the SQL server elaborating the request is on a different computer than the one originating the query, because it will split the effort between the two machines and often reduce network traffic, as we'll see in the next chapter.

A Query with Parameters

All of the queries in the DynQuery example were very similar. Instead of building a new query each time, we can write a query with a parameter and simply change the value of the parameter. If we decide to choose the countries of a continent, for example, we can write the following statement:

```
select * from Country where Continent = :Continent
```

In this SQL clause, `:Continent` is a parameter. We can set its data type and startup value, using the editor of the `Params` property collection of the Query component. When the Parameters collection editor is open, as shown in Figure 10.9, you see a list of the parameters defined in the SQL statement and set the data type and the initial value of these parameters.

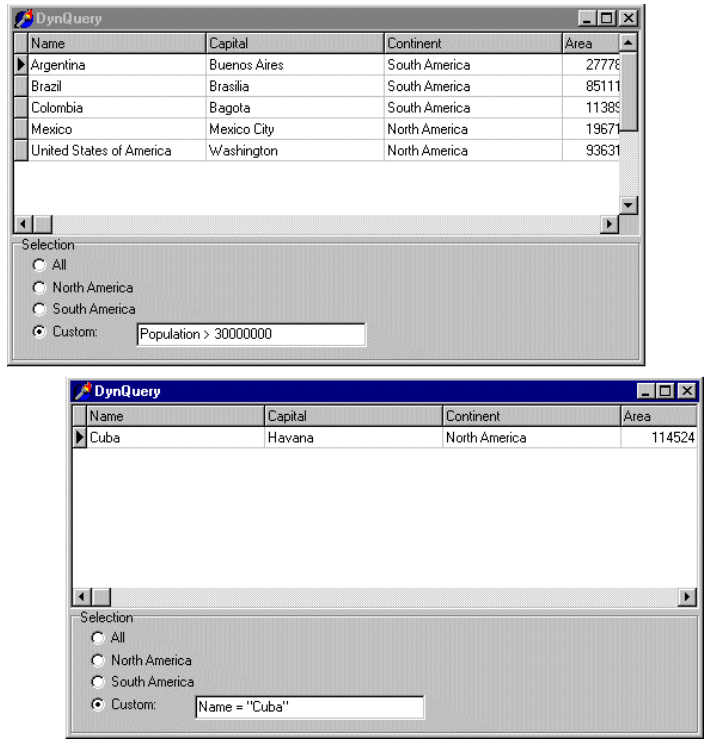


Figure 10.8: Two copies of the DynQuery program, each using a different custom SQL Where clause. Image from the original book.

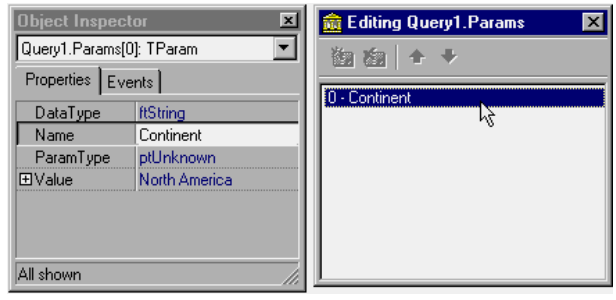


Figure 10.9: Editing the collection of parameters of a Query component. Image from the original book.

The form displayed by this new program, ParQuery, uses a list box instead of the radio button for the selection. Instead of preparing the items of the list box at design time, we extract the different available continents from the database as the program starts. This is accomplished using a second query component, with this SQL statement:

```
select distinct Continent from Country
```

After activating this query, the program scans its result set, extracting all the values and adding them to the list box:

```
procedure TQueryForm.FormCreate(Sender: TObject);
begin
  // get the list of continents
  Query2.Open;
  while not Query2.EOF do
  begin
    ListBox1.Items.Add (Query2.Fields [0].AsString);
    Query2.Next;
  end;
  ListBox1.ItemIndex := 0;

  // open the first query
  Query1.Params[0].Value := ListBox1.Items [0];
  Query1.Open;
end;
```

Before opening the query, the program selects as its parameter the first item of the list box, which is also activated by setting the `ItemIndex` property to 0. When the list box is selected, the program closes the query and changes the parameter:

```
procedure TQueryForm.ListBox1Click(Sender: TObject);
begin
  Query1.Close;
  Query1.Params[0].Value :=
    ListBox1.Items [Listbox1.ItemIndex];
  Query1.Open;
end;
```

This displays the countries of the selected continent in the list box, as you can see in Figure 10.10. The final refinement is that when the user enters a record with a new continent, it is added automatically to the list box. Instead of refreshing the entire list, with the same code executed in the `FormCreate` method, we can do this by handling the `BeforePost` event and adding the continent to the list if it is not already there:

```
procedure TQueryForm.Query1BeforePost(DataSet: TDataSet);
var
  StrNewCont: string;
begin
  // add the continent, if not already in the list
  StrNewCont := Query1.FieldByName ('Continent').AsString;
  if ListBox1.Items.IndexOf (StrNewCont) < 0 then
    ListBox1.Items.Add (StrNewCont);
end;
```

Figure 10.10: The ParQuery example at run time. Image from the original book.

We can add a little extra code to this program to take advantage of a specific feature of parameterized queries. To react faster to a change in the parameters, these queries can be optimized, or *prepared*. Simply call the `Prepare` method before the program first opens the query (after setting the `Active` property of the Query component to `False` at design time) and call `Unprepare` once the query won't be used anymore:

```
procedure TQueryForm.FormCreate(Sender: TObject);
begin
  ...
  // prepare and open the first query
  Query1.Prepare;
  Query1.Params[0].Value := ListBox1.Items [0];
  Query1.Open;
end;

procedure TQueryForm.FormDestroy(Sender: TObject);
begin
  Query1.Close;
  Query1.Unprepare;
end;
```

note Prepared parameterized queries are very important when you work on big tables and a complex query. In fact, to optimize such a query, many databases create temporary indexes. Instead of creating an index each time you open it, a prepared query can set up this optimization only once at the beginning, saving a lot of time when a parameter changes.

Using Multiple Tables

In the database programs we've written so far, we've invariably used only one database table. In the real world, however, programs usually access multiple tables. There might be a main table with the names of the customers and a secondary table with the orders the customers have made. There can be a table with company sites and another with the employees, with a numeric ID for the site where each employee works. There are countless examples of relationships among database tables.

The important thing to focus on in this short introduction to the topic is that you can use several approaches in Delphi to connect different tables:

- The master/detail relationship between tables or queries allows you to select in the secondary data set only the records related to the current element of the master data set. For example, you can select a customer in the main table and see all the orders made by that customer in the secondary table.
- A lookup field in one table (or query) displays the value of another field in a corresponding record of a related table. For example, in a Purchase Orders table, the ID of the customer who placed a given order might be a lookup field displaying the customer's name from the Customers table.
- A join specified inside a SQL query can define many other types of relationships among tables.

Because a join in a SQL query is executed on the server, while the master/detail and lookup relationships are computed on the client, SQL queries make more sense in a client/server environment, and the master/detail and lookup connections can provide the best performance when accessing local tables. These last two approaches, however, improve the user interface by far, and for this reason can be used also in a client/server architecture.

Master/Detail with Tables

Delphi provides several simple ways to create a master/detail structure. Perhaps the simplest is to use the Database Form Wizard, selecting a master/detail form in the first page. (As noted in Chapter 9, the Database Form Wizard is actually so easy to use that it doesn't need to be demonstrated in this book.) Almost as easy is a new approach we can use in Delphi 5 to accomplish the same task—using the Data Dia-

gram view of a data module. We'll do that in the next example, the MastDet program.

Since we're using the sample tables available in Delphi, there are not many choices for building a master/detail form. Our example will use the Customer and Order tables, which are used also by some Delphi sample programs. Simply place two table components in a data module, connect them with the `DBDEMOS` alias, and connect them with the two tables. Now you can drag the tables to the Diagram view, select the Master Detail connection, and drag it from the Customer table to the Orders table. The Data Module Designer will display the Field Link Designer dialog box[260], shown in Figure 10.11, where you define how to connect the tables.

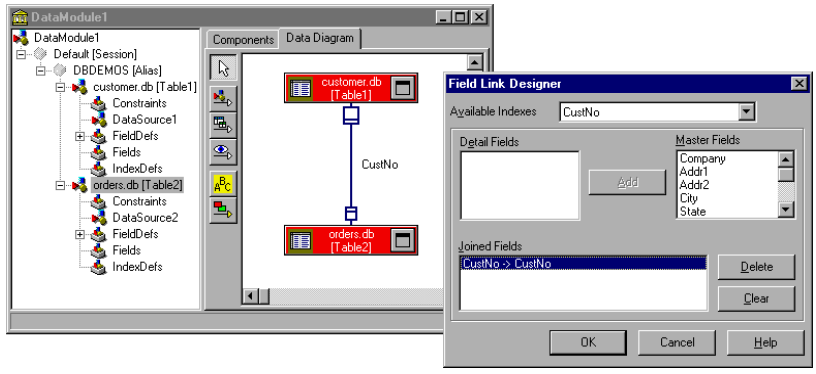


Figure 10.11: The Field Link Designer dialog box of the Data Module Designer is activated when you build a master/detail relationship. Once you've defined such a relationship, you can modify it at any time by choosing Edit in the Data Module Designer. Image from the original book.

In Figure 10.12 you can see an example of the main form of the MastDet program at run time. I've placed data-aware controls related to the master table in the upper portion, and I've placed a grid connected with the detail table in the lower portion of the form. This way, for every master record, you immediately see the list of the connected detail record, in this case all the orders done by the current client. Each time you select a new customer, the grid below displays only the orders pertaining to that customer.

260 This doesn't exist any more, of course, given the entire designer is unavailable. FireDAC and other components offer properties to define master details relationship and even specific property editors to help you set up the connection.

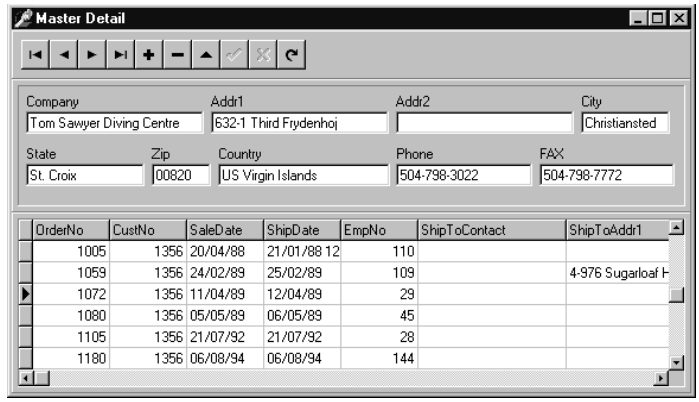


Figure 10.12: The MastDet example at run time. Image from the original book.

How does this program work? The answer is very simple. If you open the data module and look at the properties of the two Table components in the Object Inspector, you can see the following values:

```
object Table1: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'customer.db'
end
object Table2: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'orders.db'
  IndexFieldNames = 'CustNo'
  MasterFields = 'CustNo'
  MasterSource = DataSource1
end
```

The second table has a master source (the data source connected to the first table), and it relates to a specific field, which provides the cross-reference.

A Master/Detail Structure with Queries

The previous example used two tables to build a master/detail form. As an alternative, you can define this type of join using a SQL statement.

For this example, I've joined the ORDERS.DB table with ITEMS.DB, which describes each item in each order. The two tables can be joined using the OrderNo field. When you generate the code, the program, named Orders, behaves exactly like the previous one. This time, however, the trick is in the SQL statements of the second query object:

```
select
  items.”OrderNo”,
  items.”ItemNo”,
  items.”PartNo”,
  items.”Qty”
from
  items
where
  “items”.”OrderNo” = :”OrderNo”
```

As you can see, this SQL statement uses a parameter, `OrderNo`. This parameter is connected directly to the first query, because the `DataSource` property of `Query2` is set to `DataSource1`, which is connected to `Query1`. In other words, the second query is considered to be a data control connected to the first data source. Each time the current record in the first data source changes, the `Query2` component is updated, just like any other component connected to `DataSource1`. The field used for the connection, in this case, is the field having the same name as the query parameter.

Using a Lookup Combo Box

If you build a standard form for Orders, you need to work with the customer number of the order, which is not the most natural way—most users will prefer to work with customer names. However, in the database, the names of the customers are stored in a different table, to avoid duplicating the customer data for each order by the same customer. To get around working with customer numbers, I placed a new component in the form: a DBLookupComboBox control. This component can be connected to two data sources at the same time, one source containing the actual data and a second one containing the display data. Basically, we want to connect it with the CustNo value of `DataSource1`, the master query, but let it show the information extracted from another table, CUSTOMER.DB.

To accomplish this, I removed the standard DBEdit component connected to the customer number and replaced it with a DBLookupComboBox component and a DBText component. DBText is a sort of label, or text that can’t be edited. Then I added a new data source (`DataSource3`) connected to a table (`Table1`), which relates to the CUSTOMER.DB file. For the program to work, you need to set several properties of the `DBLookupComboBox1` component. Here is a list of the relevant values:

```
object DBLookupComboBox1: TDBLookupComboBox
  DataField = 'CustNo'
  DataSource = DataSource1
  KeyField = 'CustNo'
  ListField = 'Company'
```

```
  ListSource = DataSource3
end
```

The first two properties determine the main connection, as usual. The other three properties determine the secondary source (`ListSource`), the field used for the join (`KeyField`), and the information to display (`ListField`). Besides entering the name of a single field, you can provide multiple fields. Only the first field is displayed as combo box text, but if you set a large value for the `DropDownWidth` property, the pull-down list of the combo box will include multiple columns of data. You can see this output in Figure 10.13.

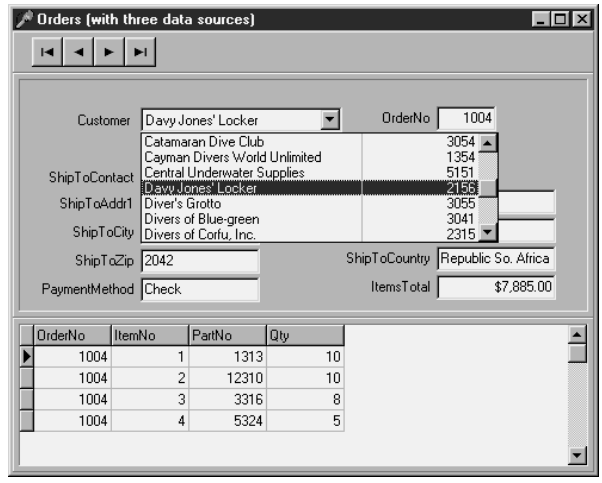


Figure 10.13: The output of the Orders example, with the DBLookupComboBox showing multiple fields in its drop-down list. Image from the original book.

note If you set the index of the table connected with the DBLookupComboBox to the Company field, the drop-down list will show the companies in alphabetical order instead of customer-number order. This is what I've done in the example.

What about the code of this program? Well, there is none. Everything works just by setting the correct properties. The three joined data sources do not need custom code. This demonstrates that using master/detail and lookup connections can be very fast to set up and very efficient. The only real drawback is that these techniques, particularly the lookup, cannot be used when the number of records becomes too large, particularly in a networked or client/server environment. Moving hundreds of thousands of records just to make a nice-looking lookup combo box probably won't be very effective.

A Lookup in a Grid

As an alternative to placing a DBLookupComboBox component in a form, we can also add a drop-down lookup list in a DBGrid component. To add a fixed selection to a DBGrid we can simply edit the `PickList` subproperty of the `Columns` property. To customize the grid with a live lookup, we have to define a lookup field using the Fields editor.

As an example, I've taken the MastDet program discussed earlier and turned it into the MastDet2 version. In the original program, the grid displayed the code number of the employee who took the order. Why not show the employee name, instead, and let the user choose it from a drop-down list of employees?

To accomplish this, I added to the data module two components: a Table and a DataSource, referring to the `EMPLOYEE.DB` database table. Then I opened the Fields editor for the `ORDERS` table and added all the fields. I selected the `EmpNo` field and set its `Visible` property to `False`, to remove it from the grid (we cannot remove it altogether, because it is used to build the cross-reference with the corresponding field of the Employee table).

Now it is time to define the lookup field. If you've followed the preceding steps, you can go to the Data Diagram tab and drag a lookup relationship from the `ORDERS` table to the `EMPLOYEE` table, connecting the two in the resulting dialog box (see Figure 10.14). You can activate a similar dialog box by using the New Field command of the Fields editor.

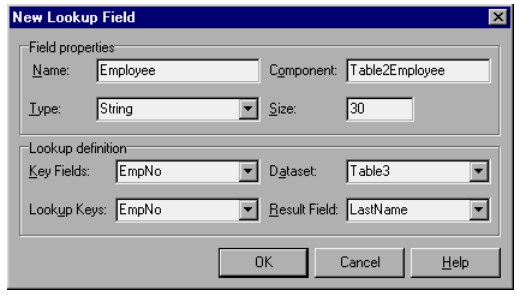


Figure 10.14: The New Lookup Field dialog box is activated by dragging a lookup link between two data sets of the Data Diagram. Image from the original book.

The values you specify in the New Lookup Field dialog box will affect the properties of a new `TField` added to the table, as demonstrated by the DFM description of the field corresponding to the settings shown in Figure 10.14:

```
object Table2Employee: TStringField
  FieldKind = fkLookup
```

```
  FieldName = 'Employee'
  LookupDataSet = Table3
  LookupKeyFields = 'EmpNo'
  LookupResultField = 'LastName'
  KeyFields = 'EmpNo'
  FixedChar = False
  Size = 30
  Lookup = True
end
```

This is all that is needed to make the drop-down list work (see Figure 10.15) and to view the value of the cross-references field at design time, too. Notice that there is no need to customize the `Columns` property of the grid, because the drop-down button and the value of seven rows are taken by default. This doesn't mean you cannot use this property to further customize these and other visual elements of the grid.

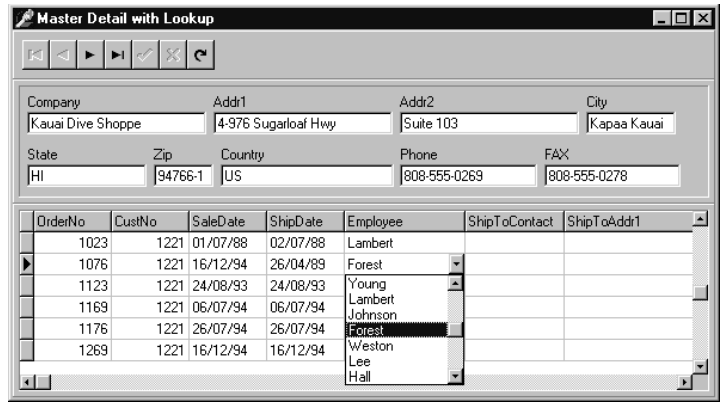


Figure 10.15: The output of the MastDet2 example, with the drop-down list inside the grid displaying values taken from another database table. Image from the original book.

The new master/detail relationship will be clearly visible in the Data Diagram view of the Data Module Designer[261]. If you add to this view the lookup field, its layout will be even more detailed, as you can see in Figure 10.16.

261 Again, nothing like this exists today in the Delphi IDE.

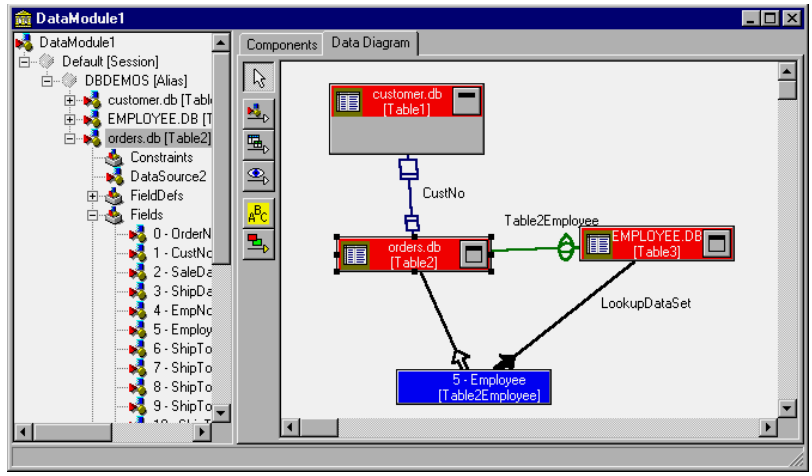


Figure 10.16: The master/detail relationship in the Data Diagram view. Notice the connections of the lookup field. Image from the original book.

Advanced Use of the DBGrid Control

We've used the DBGrid control in many examples in this chapter and the previous one, simply because it provides a very handy way of showing information about multiple fields and multiple records at a time. Unlike most other data-aware controls, which are quite simple to use, the DBGrid control has many options and is more powerful than you might think.

The next few sections explore some of the advanced operations you can do using a DBGrid control. A first example shows how to draw in a grid, a second one shows how to clone the behavior of a check box for a Boolean selection inside a grid, and a final example shows how to use the multiple-selection feature of the grid.

Painting a DBGrid

There are many reasons you might want to customize the output of a grid. A good example is to highlight specific fields or records. Another is to provide some form of output for fields that usually don't show up in the grid, such as BLOB, graphic, and memo fields.

To thoroughly customize the drawing of a DBGrid control, you have to set its `DefaultDrawing` property to `False` and handle its `OnDrawColumnCell` event. In fact, if you leave the value of `DefaultDrawing` set to `True`, the grid will display the default output before the method is called. This way, all you can do is add something to the default output of the grid, unless you decide to draw over it, which will take extra time and cause flickering.

The alternative approach is to call the `DefaultDrawColumnCell` method of the grid, perhaps after changing the current font or restricting the output rectangle. In this last case you can provide an extra drawing in a cell and let the grid fill the remaining area with the standard output. This is what I've done in the DrawData program.

The DBGrid control in this example, which is connected to the commonly used `BIOLIFE` table[262] of the DBDEMOS database, has the following properties:

```
object DBGrid1: TDBGrid
  Align = alClient
  DataSource = DataSource1
  DefaultDrawing = False
  Font.Height = -16
  Font.Name = 'MS Sans Serif'
  Font.Style = [fsBold]
  TitleFont.Height = -11
  TitleFont.Name = 'MS Sans Serif'
  TitleFont.Style = []
  OnDrawColumnCell = DBGrid1DrawColumnCell
end
```

The `OnDrawColumnCell` event handler is called once for every cell of the grid and has several parameters, including the rectangle corresponding to the cell, the index of the column we have to draw, the column itself (with the field, its alignment, and other subproperties), and the status of the cell. How can we set the color of specific cells to red? We can simply change it in the special cases:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
  // red font color if length > 100
  if (Column.Field = Table1Lengthcm) and
      (Table1Lengthcm.AsInteger > 100) then
    DBGrid1.Canvas.Font.Color := clRed;

  // default drawing
  DBGrid1.DefaultDrawDataCell (Rect, Column.Field, State);
end;
```

---

262 The table still exists today in multiple format, including *.FDS* for FireDAC FDMemTable.

The next step is to draw the memo and the graphic fields. For the memo we can simply implement the memo field's `OnGetText` and `OnSetText` events. In fact, the grid will even allow editing on a memo field if its `OnSetText` event is not `nil`. Here is the code of the two event handlers. I've used `Trim` to remove trailing nonprinting characters, which make the text appear to be empty when editing:

```
procedure TForm1.Table1NotesGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  Text := Trim (Sender.AsString);
end;

procedure TForm1.Table1NotesSetText(Sender: TField;
  const Text: String);
begin
  Sender.AsString := Text;
end;
```

For the image, the simplest approach is to create a temporary `TBitmap` object, assign the graphics field to it, and paint the bitmap to the `Canvas` of the grid. As an alternative, I've removed the graphic field from the grid, by setting its `Visible` property to `False`, and added the image to the fish name, with the following extra code in the `OnDrawColumnCell` event handler:

```
var
  Bmp: TBitmap;
  OutRect: TRect;
  BmpWidth: Integer;
begin
  // default output rectangle
  OutRect := Rect;

  if Column.Field = Table1Common_Name then
  begin
    // draw the image
    Bmp := TBitmap.Create;
    try
      Bmp.Assign (Table1Graphic);
      BmpWidth := (Rect.Bottom - Rect.Top) * 2;
      OutRect.Right := Rect.Left + BmpWidth;
      DBGrid1.Canvas.StretchDraw (OutRect, Bmp);
    finally
      Bmp.Free;
    end;
    // reset output rectangle, leaving space for the graphic
    OutRect := Rect;
    OutRect.Left := OutRect.Left + BmpWidth;
  end;

  // red font color if length > 100 (omitted – see above)
```

```
// default drawing
DBGrid1.DefaultDrawDataCell (OutRect, Column.Field, State);
```

As you can see in the code above, the program shows the image in a small rectangle on the left of the grid cell and then changes the output rectangle to the remaining area before activating the default drawing. You can see the effect in Figure 10.17.

Note In the example I've used a large font to increase the height of each row of the cell. It would be nice to be able to customize the row heights, but that is not very simple. *Delphi Developer's Handbook* (Sybex, 1998) describes the development of an extended DBGrid component with variable-height rows, and it lists the component's source code[263]. You can download the compiled version of this component on my Web site.

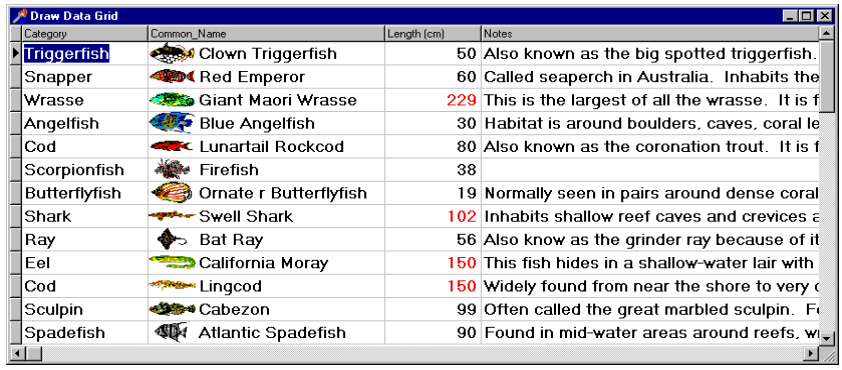


Figure 10.17: The DrawData program displays a grid that includes the text of a memo field and the ubiquitous Borland fishes. Image from the original book.

A Check Box Cell

Another common extension of the DBGrid control, found in many third-party components, is the use of check boxes to select the status of Boolean field values. A simple way to do this is to place a DBCheck box control in front of the grid when the user selects the corresponding item. I've done this in an extension of the FldText example from the last chapter.

The form displayed by the new program, called CheckDbg, contains only the grid and the check box and is based on a custom database table you can fill with data

263 As mentioned, that book is hard to find today.

using the DBAware example from the last chapter. This is a summary of the textual description of the form:

```
object DbaForm: TDbaForm
OnCreate = FormCreate
  object DBGrid1: TDBGrid
    Align = alClient
    DataSource = DataSource1
    OnColEnter = DBGrid1ColEnter
    OnDrawColumnCell = DBGrid1DrawColumnCell
    OnKeyPress = DBGrid1KeyPress
  end
  object DBCheckBox1: TDBCheckBox
    Caption = 'Senior'
    DataField = 'Senior'
    DataSource = DataSource1
    ValueChecked = 'True'
    ValueUnchecked = 'False'
    Visible = False
  end
  object Table1: TTable
    DatabaseName = 'DBDEMOS'
    TableName = 'Workers'
  end
end
```

Notice that the check box is initially hidden and that the program handles several events of the DBGrid control. The first is the `OnDrawColumnCell` event, which is not used to customize the drawing (the `DefaultDrawing` property is set to `True`), but only to compute the position of the check box when a cell of the corresponding field is selected:

```
procedure TDbaForm.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
  if (gdFocused in State) and
    (Column.Field = Table1Senior) then
  begin
    DBCheckBox1.SetBounds (
      Rect.Left + DBGrid1.Left + 1,
      Rect.Top + DBGrid1.Top + 1,
      Rect.Right - Rect.Left,
      Rect.Bottom - Rect.Top);
  end;
end;
```

The check box itself is displayed or hidden as the user enters or exits the corresponding column, by the handler of the `OnColEnter` event. Note that we cannot refer to the column by position, since a user can move the columns:

```
procedure TDbaForm.DBGrid1ColEnter(Sender: TObject);
begin
  if DBGrid1.Columns [DBGrid1.SelectedIndex].
      Field = Table1Senior then
    DBCheckBox1.Visible := True
  else
    DBCheckBox1.Visible := False;
end;
```

Finally, as an extra extension, when the check box is visible (that is, when the user has activated the corresponding field) the program intercepts the keyboard input in the grid, toggling the selection of the check box instead of accepting the input:

```
procedure TDbaForm.DBGrid1KeyPress(Sender: TObject; var Key: Char);
begin
  if DBCheckBox1.Visible and (Ord (Key) > 31) then
  begin
    Key := #0;
    Table1.Edit;
    DBCheckBox1.Checked := not
      DBCheckBox1.Checked;
    DBCheckBox1.Field.AsBoolean :=
      DBCheckBox1.Checked;
  end;
end;
```

To make this work we must not only toggle the status of the check box, but also go into edit mode and update the data of the field. You can see an example of the output of this program in Figure 10.18.

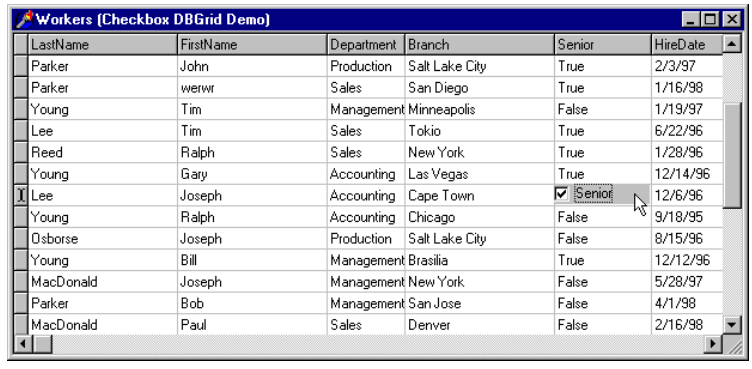


Figure 10.18: The grid of the CheckDbg example uses a check box for selecting the value of a Boolean field. Image from the original book.

A Grid Allowing Multiple Selection

The third and last example of customizing the DBGrid control relates to multiple selection. You can set up the DBGrid so that a user can select multiple rows (that is, multiple records). This is very easy, since all you have to do is toggle the `dgMultiSelect` element of the `Options` property of the grid. Once you've selected this option, a user can keep the Ctrl key pressed and click with the mouse to select multiple rows of the grid, with the effect you can see in Figure 10.19.

Since the database table can have only one active record, what information is stored in the grid for the selected items? The grid simply keeps a list of bookmarks to the selected records. This list is available in the `SelectedRows` property, which is of type `TBookmarkList`. Besides accessing the number of objects in the list with the `Count` property, you can get to each bookmark with the `Items` property, which is the default array property. Each item of the list is on a `TBookmarkStr` type, which represents a bookmark pointer you can assign to the `Bookmark` property of the table.

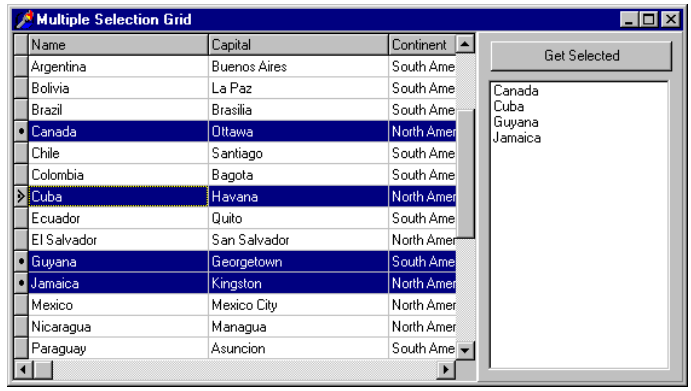


Figure 10.19: The MltGrid example has a DBGrid control that allows the selection of multiple rows. Image from the original book.

note The `TBookmarkStr` is a string type for convenience, but its data should be considered "opaque" and volatile. You shouldn't rely on any particular structure to the data you may find if you peek at a bookmark's value, and you shouldn't hold on to the data too long or store it in a separate file. Bookmark data will vary with database driver and index configuration, and it may be rendered unusable when rows are added to or deleted from the dataset (by you or by other users of the database).

To summarize the steps, here is the code of the MltGrid example, activated by pressing the button to move the Name field of the selected records to the list box:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  BookmarkList: TBookmarkList;
  Bookmark: TBookmarkStr;
begin
  // store the current position
  Bookmark := Table1.Bookmark;
  try
    // empty the list box
    ListBox1.Items.Clear;
    // get the selected rows of the grid
    BookmarkList := DbGrid1.SelectedRows;
    for I := 0 to BookmarkList.Count - 1 do
    begin
      // for each, move the table to that record
      Table1.Bookmark := BookmarkList[I];
      // add the name field to the listbox
      ListBox1.Items.Add (Table1.FieldByName (
        'Name').AsString);
    end;
  finally
    // go back to the initial record
    Table1.Bookmark := Bookmark;
  end;
end;
```

The Data Dictionary[264]

It is very common to use fields with a similar layout (for example, the same display mask) throughout a single application or in different applications. If you use integer numbers, decimal numbers, percent values, phone and fax numbers (possibly the same number with different extensions), and other standard fields, it is extremely tedious to set each one of them from scratch. For this reason, Delphi includes a Data Dictionary. This is a sort of database that stores the properties of fields. You can define the properties of these standard fields using the Dictionary, or you can simply copy them from existing fields (of course, you can also use the existing entries in the Data Dictionary without further work).

---

264 This is another database related feature that was dropped quite some time ago and naver made a migration form the BDE to modern data access libraries. The concept was quite interesting, though.

In a client/server environment (with several Delphi programmers working on a project), the Data Dictionary can reside on a remote server for additional sharing of information.

note The default Data Dictionary is implemented using a Paradox table, but you can define a new one based on a SQL server table.

The Data Dictionary and the Fields Editor

Most of the operations involving the Data Dictionary take place in the Fields editor of a table or query component[265]. The local menu of the Fields editor, in fact, has five commands related to the use of the Data Dictionary (see Figure 10.20).

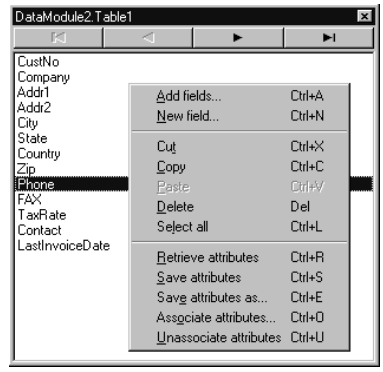


Figure 10.20: The local menu of the Fields editor, with the menu commands used to interact with the Data Dictionary. Image from the original book.

Operation	Meaning
Associate Attributes	This command is used to associate an attribute set with a given field. In practice, you can select one of the attribute sets from the Dictionary to use with the current field. When you associate a field with an attribute set from the Data Dictionary, the attributes will be copied to the properties of the field.

265 These options have been removed fomr the Fields Editor, when the Data Dictionary was removed form the IDE.

Unassociate Attributes	This command is the reverse of the Associate Attributes operation. It breaks the association between the field and the attribute set.
Retrieve Attributes	This command is used to get the current values from the related attribute set. It can be used only while the field is associated with an attribute set. You can think of this as a loading command.
Save Attributes	This command is the reverse of the Retrieve Attributes operation. It copies the values of the properties of the current field to the associated attribute set. If no attribute set is associated, it will prompt you for a new name, as does the Save Attributes As command below.
Save Attributes As	This command is used to associate the field with a new attribute set, for which you have to provide a name in the dialog box that appears.

These commands should be quite intuitive, and I suggest that you work with the Data Dictionary to learn how to use them. You'll get used to the Dictionary quickly. Defining an association between an attribute type and one or more fields of the tables of a database forces Delphi to use the proper attributes every time you use a table with one of those fields in an application. For example, all the "Phone" fields (such as phone number and fax number) of a table can be associated with a specific attribute set, so that every time you use that table in a program, Delphi will automatically set the proper input mask of the field object, as well as other attributes. This also works if the field is retrieved as part of a query, but only if you create the field objects at design time.

What's in an Attribute Set?

In the previous section, I used the term *attribute set* several times. An attribute set is an entry (a *record*) of the Data Dictionary[266]. An attribute set refers to several properties of a `TField` object, but also includes other general properties. Many of the properties of the attribute set correspond to properties of the various `TField` sub-classes; these should be quite simple to understand. Here is an alphabetical list of these properties:

---

266 Again, not applicable today.

```
Alignment       EditMask
BlobType        MaxValue
Currency        MinValue
DisplayFormat   Precision
DisplayLabel    ReadOnly
DisplayValues   Required
DisplayWidth    Transliterate
EditFormat      Visible
```

Of course, if you set a property as `Precision` (a value used only for floating-point numbers) and then associate the attribute set with a `TStringField`, the value will be ignored.

A few other values of the attribute set define the general behavior of a field and are used to determine how to create a new field object for a given field of a table or query:

Attribute	Usage
`TField` Class	Indicates the type of field (the `TField` subclass) to create when a field is added to a dataset.
`TControl` Class	Determines the type of data-aware component Delphi will create when you drag a field from the Fields editor to a form. If no value is provided, Delphi will use a standard approach, which depends on the type of the field object.
Based On	This is a value you are asked to provide when you save an attribute set with a new name. It indicates the attribute set upon which the current one is based. This means that if you make a change to an attribute of the original set and this attribute is not overridden by the current set, the change will affect the current set, too. The analogy that comes to mind to explain this is that of styles in word processing programs, which can be based on other similar styles. Of course, this resembles a sort of inheritance, too.

Exploring the Data Dictionary[267]

You can easily define a new attribute set by saving the attributes of a current field from the Fields editor, as I've mentioned before. However, you can also define new attribute sets, or simply view them using the SQL Explorer (called Database

267 You might want to skip this section as this feature has long been dropped.

Explorer in the Professional version of Delphi). Just open this tool, select the Dictionary tab above the left pane, and you'll see the Default Data Dictionary (DefaultDD), based on the BDESDD Paradox table. If you've added new Data Dictionaries, you'll see them all. Under the Dictionary entry, you'll find two subtrees, Databases and Attribute Sets, as you can see in Figure 10.21.

Under Attribute Sets, you'll find a list of these sets, each one with the values of its properties. There are also lists of database tables using each set and of the other attribute sets based on the sets. You can also use the SQL Explorer to create new attribute sets or modify existing sets.

The associations between fields of the database tables and the attribute sets can also be seen and modified by exploring the second subtree, Databases. Once you have selected a field of a table, a combo box will allow you to associate it with one of the available attribute sets in the Data Dictionary.

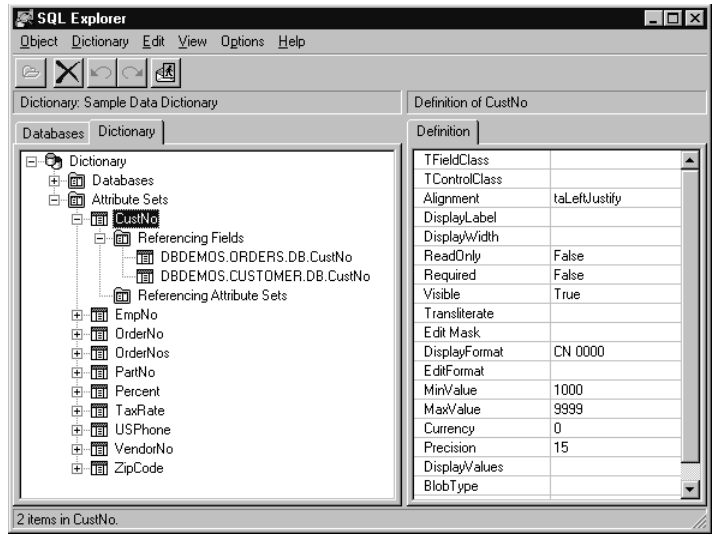


Figure 10.21: The Data Dictionary seen in the SQL Explorer. Image from the original book.

On the whole, when you are starting a new project and you have planned its database tables, I suggest that you start setting up some attribute sets and their associations before starting to work in Delphi. If your plan is not so well defined, though, you should simply use the Fields editor to build up your Data Dictionary along with your tables and then use the Explorer to revise the current situation and document your changes.

Handling Database Errors[268]

Another important element of database programming is handling database errors in custom ways. Of course, you can let Delphi show an exception message each time a database error occurs, but you might want to try to correct the errors or simply show more details. There are basically three approaches you can use to handle database-related errors:

- You can wrap a `try-except` block around risky database operations, such as a call to the `Open` method of a Query or to the `Post` method of a dataset. This is not possible when the operation is generated by the interaction with a data-aware control.
- You can install a handler for the `OnException` event of the global `Application` object or use the ApplicationEvents component, as described in the next example.
- You can handle specific events of the datasets related to errors, as `OnPostError`, `OnEditError`, `OnDeleteError`, and `OnUpdateError`. These events will be discussed later on in the example.

While most of the exception classes in Delphi simply deliver an error message, with database exceptions you see a list of errors, showing local BDE error codes and also the native error codes of the SQL server you are connected with. Besides the `Message` property, the `EDBEngineError` class[269] has two more properties, `ErrorCount` and `Errors`. This last property is a list of errors:

```
property Errors[Index: Integer]: TDBError;
```

Each item within this list is an object of the class `TDBError`, which has the following properties:

```
type
  TDBError = class
    ...
  public
    property Category: Byte read GetCategory;
    property ErrorCode: DBIResult read FErrorCode;
    property SubCode: Byte read GetSubCode;
    property Message: string read FMessage;
```

---

268 This core ideas and error handling alternatives are still relevant today, even if some of the specific details are tied to the BDE and it's error management classes.

269 This is a BDE specific class, other data access libraries offer similar implementations, though.

```
    property NativeError: Longint read FNativeError;
  end;
```

I've used this information to build a simple database program showing the details of the errors in a memo component. To handle all of the errors, the DBError example installs a handler for the `OnException` event of an ApplicationEvents component. The event handler simply calls a specific method used to show the details of the database error, in case it is an `EDBEngineError`:

```
procedure TForm1.ApplicationEvents1Exception (Sender: TObject; E:
Exception);
begin
  Beep;
  if E is EDBEngineError then
    ShowError (EDBEngineError (E))
  else
    ShowMessage (E.Message);
end;
```

I decided to separate the code used to show the error to make it easier for you to copy this code and use it in different contexts. Here is the code of the `ShowError` method, which outputs all of the available information to the `Memo1` component that I've added to the form:

```
procedure TForm1.ShowError(E: EDBEngineError);
var
  I: Integer;
begin
  Memo1.Lines.Add('');
  Memo1.Lines.Add('Error: ' + (E.Message));
  Memo1.Lines.Add('Number of errors: ' +
    IntToStr(E.ErrorCount));
  // iterate through the Errors records
  for I := 0 to E.ErrorCount - 1 do
  begin
    Memo1.Lines.Add('Message: ' +
      E.Errors[I].Message);
    Memo1.Lines.Add('   Category: ' +
      IntToStr(E.Errors[I].Category));
     Memo1.Lines.Add('   Error Code: ' +
      IntToStr(E.Errors[I].ErrorCode));
    Memo1.Lines.Add('   SubCode: ' +
      IntToStr(E.Errors[I].SubCode));
    Memo1.Lines.Add('   Native Error: ' +
      IntToStr(E.Errors[I].NativeError));
    Memo1.Lines.Add('');
  end;
end;
```

Besides this error-handling code, the program has a table and a query, along with the error-related event handlers. As already mentioned, you can install an event handler related to specific errors of a dataset. The three events `OnPostError`, `OnDeleteError`, and `OnEditError` have the same structure. Their handlers receive as parameters the dataset, the error itself, and an action you can request from the system; this can be set to `daFail`, `daAbort`, or `daRetry`:

```
procedure TForm1.Table1PostError(DataSet: TDataSet; E: EDatabaseError;
  var Action: TDataAction);
begin
  Memo1.Lines.Add (' -> Post Error: ' + E.Message);
end;
```

If you don't specify an action, as in the code above, the default `daFail` is used, and the exception reaches the global handler. Using `daAbort`, instead, stops the exception and can be used if your event handler already displays a message. Finally, if you have a way to determine the cause of the error and fix it, you can use the `daRetry` action.

note The fourth error event, `OnUpdateError`, has a different structure and is used along with cached updates as the information is sent back from the local cache to the database. This handler is important for handling update conflicts among different users as described in the next example.

The example has also a DBGrid connected with the table. You can use the DBGrid to perform some illegal operations, such as adding a new record with the same key as an existing one or trying to execute illegal SQL queries. Pressing the four buttons on the left of the memo generate errors, as you can see in Figure 10.22.

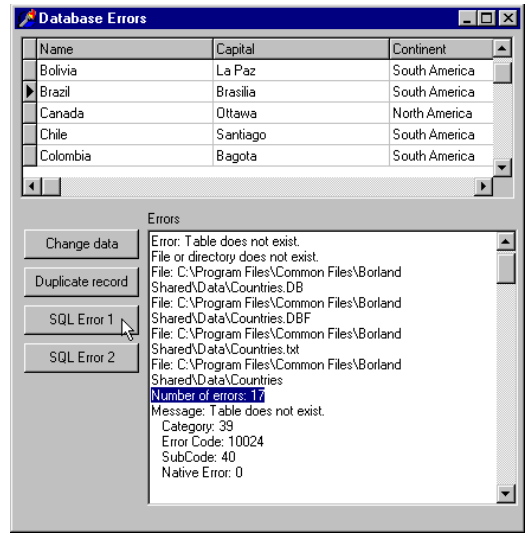


Figure 10.22: The third button of the DBError form generates an exception with 17 database errors! Image from the original book.

Multi User Paradox Applications[270]

Up to now we've seen the development of applications running on a stand-alone computer. In Chapter 11 we'll see how to use SQL servers, which allow you to create applications for a large number of users. An in-between solution, when you have a limited number of users (usually no more than a dozen) working on a set of data at the same time, is to use local Paradox or Access files shared on a network.

In the final part of this chapter I'll cover a few techniques you can use when sharing Paradox data in a multiuser environment. As part of this discussion I'll cover a few related techniques, such as packing tables, the BDE callback function, crash recovery, and concurrency.

270 This entire section is also specifically tied to Paradox pessimistic locking logic (not available in relational databases) and low level BDE calls. Not really applicable to today Delphi.

Low-Level BDE

Before we proceed, we need to focus on the role of the BDE and on some of its low-level features, which are not available within Delphi components. As mentioned in the last chapter, the Borland Database Engine is the means for accessing database data in Delphi (unless you use a custom dataset). This engine takes the requests from your Delphi programs and, using an appropriate driver, translates them into commands recognized by the specific database you are using.

The BDE is actually a set of DLLs (which must be installed along with the applications) that collectively offer a low-level API to programmers, better known as IDAPI (Independent Database Application Programming Interface). Of course, Delphi programmers usually don't call the functions of this API (just as they usually don't call the Windows API directly), but use components that wrap the most common calls, such as the `TTable`, `TQuery`, `TDatabase`, and `TSession` components, to name just a few. Only the data-aware components are part of the VCL and do not relate directly to the BDE.

At times, however, you might need to use some low-level features of the BDE that aren't available in Delphi components, exactly as happens with Windows API functions. We'll use a few low-level BDE functions in the final part of this chapter. To be able to use them, you need to understand at least the foundations of the BDE and the terms used in its API and help file.

---

note The BDE help file is installed along with the BDE and is now linked with the rest of the Delphi help file. If you cannot find it there, you can look in the Program Files/Borland/Common Files/BDE directory (or in the directory where you've installed the BDE).

Every application accessing the BDE is considered a client, with a separate connection to the BDE. This connection is often described as a *session*; and the global `Session` component inside the VCL provides you with a default connection, so you generally don't have to care about this. Technically the global `Session` component calls the `DbiInit` function to accomplish this.

The BDE handles the requests from each client, each session, using a separate context. A single application might even ask for multiple independent connections by using multiple `TSession` components. This is generally required for multithreaded database access, as you'll see in Chapter 16, because if the BDE doesn't create a separate context it can get confused when it gets two simultaneous requests (which can happen only in a multithreading application).

In low-level BDE calls, several parameters (in the form of custom handles, unrelated to the Windows handles) are commonly required. Here is a short list:

- The database handle (`HDBIDB`) is the handle to the current database the application is working with. You can get the value of this handle using the `DBHandle` property of the dataset components or the `Handle` property of the `TDatabase` component.
- The cursor handle (`HDBICur`) is the handle to the result set, the view of the data of the table or query. As the term *cursor* indicates, you can scroll through the data of this view, since access to the data of the cursor takes place one record at a time. You can get the value of this handle using the `Handle` property of the Table and Query components.

note The use of a cursor to access database data is typical in SQL servers. The BDE, however, also applies the same technique to Paradox and other local tables, to provide uniform access to all databases. Traditionally Paradox, dBase, and other local formats were intended for a record-oriented approach.

Every time you call a BDE function, it will return an error code. Checking these error codes (and if necessary raising an exception) is important; you can accomplish this easily by calling the `Check` function of the VCL (defined in the DBTables unit). In theory, it is even possible to create a complete Delphi database application using direct BDE calls instead of using the data-access components, but except for very peculiar situations, this is a lot of extra work with little or no benefit.

Packing a Local Table

A simple and common example of the direct use of local BDE calls is the packing of a table, the operation that physically removes the deleted records. Delphi's Table component doesn't have this feature built in, probably because the feature is required only by some local databases (it makes no sense in the client/server world).

Packing tables is very important in dBase, where deleted records are kept in the table until it is packed. In Paradox it is usually less important, because the database engine can reuse physical locations of the deleted records for new ones. To pack a dBase table you can simply use the `DbiPackTable` function, as in the following code, extracted by the `PackDBaseTable` procedure of the DbPAck example:

```
Table.Close;
// reopen in exclusive mode
Table.Exclusive := True;
Table.Open;
```

```
// pack the table
Check (DBIPackTable (Table.DBHandle,
Table.Handle, nil, nil, True));
// remove the exclusive mode
Table.Close;
Table.Exclusive := False;
```

As you can see in the code above, before calling this function you have to open the table in exclusive mode, which may require closing it beforehand. As an alternative to passing the cursor handle as the second parameter, you can also set it to `nil` and pass the table name as the third parameter and the constant `szDBase` as the fourth parameter.

note To call the BDE functions within a program, you need to add a `uses` statement referring to the BDE unit. Delphi still provides unit aliases for BDE, so if you need Delphi 1 compatibility you can still refer to the units DbiTypes, DbiProcs, and DbiErrs.

While the BDE provides a specific function for packing a dBase table, there isn't a corresponding function for Paradox files. As an alternative, you can restructure the table; this forces the BDE to update the actual data in it, removing the records marked for deletion. This restructuring operation can be done with the `DbiDoRestructure` function, which is quite complex to use, because it is a generic, multipurpose function.

The function requires as parameters one or more table descriptors, of type `CRTblDesc` (passed as the third parameter), and the number of descriptors (passed as the second parameter). Here is a sample code excerpt, from the `PackPdoxTable` procedure of the DbPack example, that uses `DbiDoRestructure`:

```
var
  TableDesc: CRTblDesc;
  hDatabase: hDbiDB;
begin
  // get the database handle and close the table
  hDatabase := Table.DBHandle;
  Table.Close;
  // fill the table descriptor
  FillChar (TableDesc, SizeOf (CRTblDesc), 0);
  with TableDesc do
  begin
    StrPCopy (szTblName, Table.TableName);
    StrPCopy (szTblType, szParadox);
    bPack := True;
  end;
  // restructure the table, packing it
  if hDatabase <> nil then
    Check (DBIDoRestructure (hDatabase, 1,
```

```
      @TableDesc, nil, nil, nil, False));
```

As you can see, most of the fields of the table descriptor and most of the parameters of the function are actually not used; a real restructuring operation would require quite a lot of code. The only real point in the listing above is setting the `bPack` parameter of the table descriptor to `True`, to force the packing operation during the restructuring.

You can find the complete source code of the two routines I've just discussed in the DbPack example. The program simply lists all the Paradox and dBase tables of a given alias (as you can see in Figure 10.23), and it allows a user to select a table and pack it.

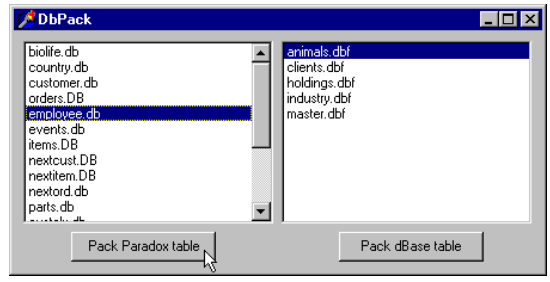


Figure 10.23: The DbPack application allows you to pack Paradox and dBase tables. Image from the original book.

Using Paradox Files on a Network[271]

When you want to share database data among multiple users, you have to keep the shared database files on a disk accessible from every computer. The program can reside on the network drive, but it is executed on the local machine, as the database engine. The BDE and the proper aliases must be installed on each computer. You can generate a simple installation program for your application by using the copy of InstallShield Express[272] available on the Delphi Professional and Client/Server CDs. In any case, you are free to distribute the BDE files, which are also available as a compressed CAB file for Internet-based deployment.

271 Even this can technically still de done today by downloading, installing, and distributing the BDE engine, it's a very old approach I'd recommend against. And if you are still using BDE and Paradox today in applications you build at the time, it's really time to move on to something modern.

272 This installer software no longer ships with Delphi.

There are several BDE settings you have to check when sharing Paradox database files on a network with multiple client applications connected. Here is a list of the most important issues:

- Set LOCAL SHARE to TRUE. This setting can be done with the BDE Configuration Utility, in the System page. This option is required only if the same files are used by other applications that aren't based on the BDE and if you are not using a Novell File Server.
- Use a shared network directory for the database data. This is the value of the Alias in the BDE Configuration Utility or of a temporary alias in the Database component. It is important to note that the network drive you are using *must* be mapped with the same drive letter by all client machines; otherwise, the BDE control gets confused about which tables the applications are sharing. In fact, they refer to the tables in use by using the drive letter name.
- Also use a shared network directory for the network control file, Paradox.NET. This is the value of the Net Dir parameter you can set up using the BDE configuration utility or the `NetFileDir` property of the Session component. This control file holds information about locks, preventing multiple users from editing the same record at the same time, as discussed in the next section of this chapter.

note In case of setup errors, you'll get the confusing error message *Directory is controlled by other .NET file* or *Multiple .NET files in use*.

With these settings everything should work fine, unless a program is not properly closed because of a system crash. There are a few suggestions you can follow to make your applications more robust:

- The `DBISaveChanges` function of the BDE saves all the buffers of the database engine to disk. This function is useful with local tables, but it is vital on a network. A typical approach is to call this function in the `AfterPost` event of a table or the `OnIdle` message of the global `Application` object.
- When there is an error in an index of a Paradox table, you can simply delete the index file and rebuild it. Usually this fixes the problem.
- You can disable disk caching on the drive hosting the database files, so that the data is immediately written to the disk and is not lost in case of a system crash on the file server.
- Finally, you can use a special table repair utility DLL provided by Borland, called TUtil32.DLL for the 32-bit versions of the BDE. There are third-party Delphi

components that call functions of this DLL, which allows you to try to fix a broken Paradox table file.

Concurrency Control

There is nothing you can do about it: when you have multiple users, they'll eventually try to update the same information at the same time, causing an update conflict. Different database servers use different approaches to handle concurrent access to the same data. There are two basic approaches, one exemplified by Paradox and the other by most SQL servers:

- Paradox uses a pessimistic approach. As a user places a record in edit mode, the record is locked. The other users can still read its value, but they cannot place the same record in edit mode.
- Most SQL server database engines use an optimistic approach[273]. They let multiple users edit the same data, and they allow the applications to send back the original and the modified data, so users can check if someone else already made the change. I'll cover this technique in the next chapter (in the section devoted to the `TUpdateSql` component).

When you use Paradox in a networked environment, the lock information for each user is saved in the shared Paradox.NET file (as mentioned in the last section). Even if you don't have a network, you can test this by simply running the same application twice and trying to edit the same record in both windows. Since you have two separate database sessions, one for each instance of the application, the situation is very similar to having two users accessing the database. Figure 10.24 shows an example of the error message displayed when you edit the same record in two copies of a program.

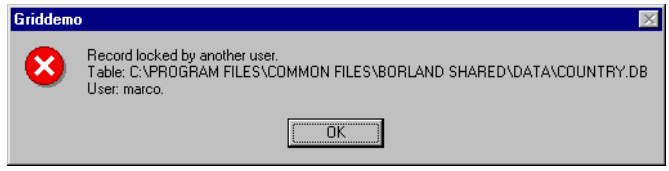


Figure 10.24: The error message displayed by a Delphi application when you try to modify a record locked by another user or program. Image from the original book.

273 I'd say, all of the relational databases now use optimistic conflict resolution. The coverage of the pessimistic approach in this section is quite out of fashion.

There are several things you can do to have more control over the status of a record, either to avoid these kinds of errors or simply to improve the output to the user. If you use explicit calls to the `Edit` method of a table, a very simple solution is to wrap this call inside a `try-except` block. This way, if the record is locked and the system raises an exception, you can warn the user that someone else is modifying the data and ask whether to retry the edit operation (maybe after some time) or skip it.

The problem in these cases is that by default you have little control over the length of the editing operation, so you don't know how long you need to wait before you retry editing the record another user is blocking. The other user might have gone for a cup of coffee without first posting the changes on the record. There are a few techniques you can use to avoid or reduce this problem:

- You can use non–data-aware controls and handle the update operations in code, making them extremely fast (as illustrated in the NonAware example in the last chapter).
- You can force a time-out on the editing operations (using a timer control), so that if a user doesn't post the updates after a given amount of time and is not working on the application anymore, the table editing operation is automatically canceled. If you do this, you should probably save the temporary changes in some local buffer, to let the user restart the editing operation from the exact state in which it was stopped. Otherwise, the user will have to reenter all the data, since canceling the editing operation with data-aware controls forces an update of the controls to reflect the current data in the database.

In some circumstances you might also want to know the status of a record, testing whether it is locked (for example, before a custom update or when you use non–data-aware controls). To check whether you have another table open on the same record, you can simply use the `DbiIsRecordLocked` function. In a multiuser situation, however, this function doesn't help because it checks only the current session. In fact, there is no BDE function to test whether another user has locked a record.

What you can do is mimic the operation Delphi does when the table is set in edit mode. In this case the VCL places a write lock on the record. Doing this operation on the current table (technically, the current cursor) might actually change its status. For this reason, it is better to create a clone cursor first and then apply the function to it:

```
function IsRecordLocked (Table: TTable): boolean;
var
  Locked: BOOL;
  hCur: hDBICur;
  rslt: DBIResult;
```

```
begin
  Table.UpdateCursorPos;
  // test if the record is locked by the current session
  Check (DbiIsRecordLocked (Table.Handle, Locked));
  Result := Locked;
  // otherwise check all sessions
  if (Result = False) then
  begin
    // get a new cursor to the same record
    Check (DbiCloneCursor (Table.Handle, False, False, hCur));
    try
      // try to place a write lock in the record
      rslt := DbiGetRecord (hCur, dbiWRITELOCK, nil, nil);
      // don't call Check: we want to do special actions
      // instead of raising an exception
      if rslt <> DBIERR_NONE then
      begin
        // if a lock error occured
        if HiByte (rslt) = ERRCAT_LOCKCONFLICT then
          Result := True
        else
          // if some other error happened
          Check (rslt); // raise the exception
      end
      else
        // if the function was successful, release the lock
        Check (DbiRelRecordLock (hCur, False));
    finally
      // close the cloned cursor
      Check (DbiCloseCursor (hCur));
    end;
  end;
end;
```

This function is used in the LockTest example, a very simple program you should test by executing multiple copies. The program uses a timer and the `OnDataChange` event of the data source to test the status of the lock. In the form caption, it displays whether the record is in edit mode (in which case we lock it), is locked by another instance of the application, or is available. You can see three copies of the program running with the three different conditions in Figure 10.25.

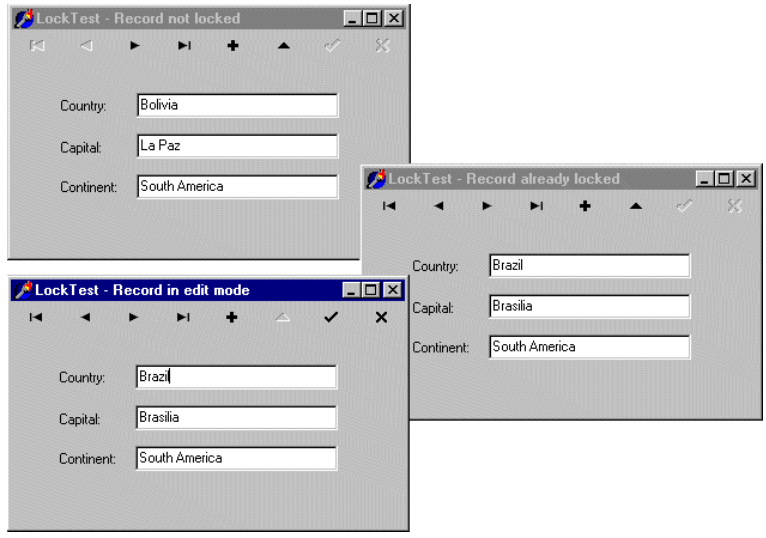


Figure 10.25: The three different states of the LockTest example: edit mode (locked by the current program), locked by another user or instance, and not locked. Image from the original book.

Besides displaying information in the caption, the program disables the three DBEdit controls every time there is a lock placed by another user:

```
procedure TNavigForm.TestLockStatus;
begin
  // if the table is not in edit mode
  if Table1.State in [dsEdit, dsInsert] then
    Caption := 'LockTest - Record in edit mode'
  else if IsRecordLocked (Table1) then
  begin
    DbEdit1.ReadOnly := True;
    DbEdit2.ReadOnly := True;
    DbEdit3.ReadOnly := True;
    Caption := 'LockTest - Record already locked';
  end
  else
  begin
    DbEdit1.ReadOnly := False;
    DbEdit2.ReadOnly := False;
    DbEdit3.ReadOnly := False;
    Caption := 'LockTest - Record not locked';
  end;
end;
```

Database Transactions

Whether you are working with a SQL server or with local database files, you can use *transactions* to make your applications more robust[274]. The idea of a transaction can be described as a series of operations to be considered as a single, "atomic" whole that cannot be split.

An example may help to clarify the concept. Suppose you have to raise the salary of each employee of a company by a fixed rate, as we did in the Total example of the last chapter. Now if during the operation an error occurs, you might want to undo the previous changes. If you consider the operation "raise the salary of each employee" as a single transaction, it should either be completely done or completely ignored. Or consider the analogy with financial transactions—if only part of the operation is performed, because of an error, you might end up with a missed credit or with some extra money!

Working with database operations as transactions serves a useful purpose. You can start a transaction and do several operations that should all be considered parts of a single larger operation; then, at the end, you can either commit the changes or roll back the transaction, discarding all the operations done up to now. Typically you might want to roll back a transaction if an error occurred during its operations.

Handling transactions in Delphi is quite simple. By default each edit/post operation is considered a single transaction, but you can alter this behavior by handling them explicitly. Simply add a Database component to a form or data module, connect each table or query to this form or data module to the Database component, and then use the following three methods of the `TDatabase` class:

- `StartTransaction` marks the beginning of a transaction.
- `Commit` confirms all the updates to the database done during the transaction.
- `Rollback` returns the database to its state prior to starting the transaction.

274 Many RDBMS make the use of transactions compulsory. InterBase is one of them, although database engines like FireDAC can automate the use of transactions behind the scenes.

A Simple Example of Transactions

To show you an example of transaction handling related to Paradox files, I've simply made these three methods into user operations. This is not the standard use of transactions, but it should help you understand how they work. In the next chapter we'll see more complex examples of transactions in a client/server environment.

The form displayed by the Transact example has three buttons in a panel, and it has a grid connected with a simple query, which is in turn connected to a database component. Simply place the database component in the form, give a value to its `DatabaseName` property (which is different from the `Name` property, which is the name of the component), and then choose this database name as the value of the `DatabaseName` property of the Query component. To sum things up, here is a portion of the DFM file of the Transact example:

```
object Database1: TDatabase
  AliasName = 'DBDEMOS'
  Connected = True
  DatabaseName = 'MyData'
  SessionName = 'Default'
  TransIsolation = tiDirtyRead
end
object Query1: TQuery
  BeforeEdit = Query1BeforeEdit
  DatabaseName = 'MyData'
  RequestLive = True
  SQL.Strings = (
    'select * from Employee')
end
```

Now we can look at the code of the example, which is very simple. When the first button is pressed, the program starts a transaction (calling `Database1.StartTransaction`) and enables the other two buttons. The Commit button simply calls the corresponding method of the database object, `Database1.Commit`, after posting any changes, and then enables and disables the buttons properly. The last button, Rollback, should also update the contents of the DBGrid, by calling the `Refresh` method of the Query after calling `Query1.Cancel` and `Database1.Rollback`.

If you try running this program, you'll see that you can post some changes to the database, possibly editing several records, and then simply undo the changes by pressing the Rollback button. You should not even press the Start button, because its code is automatically executed each time you start an edit operation:

```
procedure TForm1.Query1BeforeEdit(DataSet: TDataSet);
begin
```

```
  // start a transaction, if not already started
  if not Database1.InTransaction then
    BtnStartClick (self);
end;
```

Notice that this code is executed *before* the dataset is put in edit mode. You can see the effect of a rollback action in Figure 10.26. As a final note, keep in mind that there is only a single level of transactions, but for multiple transactions on different tables you can use multiple database components.

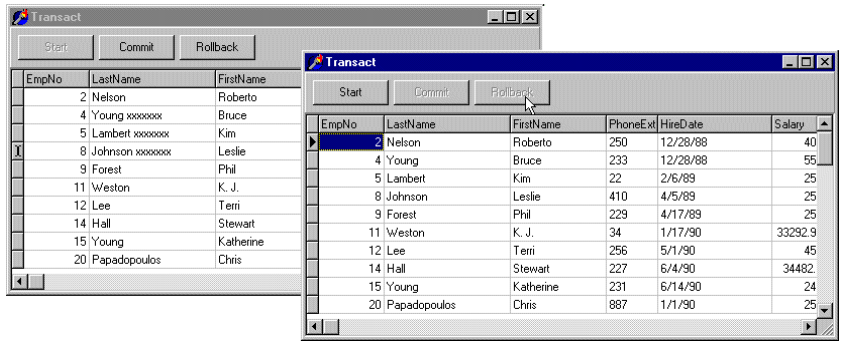


Figure 10.26: The output of the Transact example before (left) and after (right) a rollback operation. Image from the original book.

Transactions can be used on Paradox files, but only on tables that have an index; the BDE handles the transaction by locking all records involved. Not only does this hamper other changes, the BDE might even run out of resources for the locks. For this reason transactions should be limited to non-visual operations, such as posting (rapidly) a series of changes on one or more tables. Again, moving money from one account to another and increasing the salary of each employee are good examples. Transactions also work with other local file formats, such as dBase and FoxPro.

Using Cached Updates as Transactions

An alternative to the use of transactions on local files is the use of cached updates[275]. What is a cached update? You keep the updates in memory and then send them to the physical table all at once. This takes place when you apply the updates, by calling the `ApplyUpdates` method of a dataset (to update a single table) or the same method

275 Cached updated remains a fundamental tool in today's database access. FireDAC's dataset components integrate cached updates, but you can also use the FDMemTable component as an extended cache for a data access component.

of the database (to perform the same operation on multiple tables at once). Don't forget to apply the updates when you are using cached updates, or your changes to the data will be lost!

Cached updates are similar to transactions in that you can cancel the updates just as you can roll back a transaction, skipping the local changes the user has done. To show you an example of this approach I've transformed the Transact application into a new program, called CacheUpd. The program has no database component, and the query component has the `CachedUpdates` property set to `True` and these other settings:

```
object Query1: TQuery
  CachedUpdates = True
  AfterPost = Query1AfterPost
  OnUpdateError = Query1UpdateError
  DatabaseName = 'DBDEMOS'
  RequestLive = True
  SQL.Strings = (
    'select * from Employee')
end
```

The two buttons simply call the `ApplyUpdates` and `CancelUpdates` methods and then disable both buttons. Actually, after applying the updates you should also commit them, to refresh the cache; and you should also stop error messages, as we'll handle them separately:

```
procedure TForm1.BtnApplyClick(Sender: TObject);
begin
  try
    // apply the updates and empty the cache
    Query1.ApplyUpdates;
    Query1.CommitUpdates;
    // set buttons
    BtnApply.Enabled := False;
    BtnCancel.Enabled := False;
  except;
    // silent exception
  end;
end;
```

When the first update is posted to the local cache, the handler of the `AfterPost` event enables the buttons once more. Also, if the user exits from the application while there are still pending updates, we ask for confirmation:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  // if there are pending changes, ask the user what to do
  if Query1.UpdatesPending and
      (MessageDlg ('Apply the pending updates?',
```

```
      mtConfirmation, [mbYes, mbNo], 0) = mrYes) then
    Query1.ApplyUpdates;
end;
```

As the user edits a record, the BDE places the usual lock on it, but it immediately removes the lock once the updates are sent to the local cache. For this reason, two different users can both have some updates in the local memory. The first user to apply the update physically sends the data to the database, while the second is stopped by an error indicating that another user has modified the data. In this case Delphi raises the specific `OnUpdateError` event of the dataset, which passes the handler several parameters:

```
procedure Query1UpdateError (DataSet: TDataSet;
  E: EDatabaseError; UpdateKind: TUpdateKind;
  var UpdateAction: TUpdateAction);
```

The first parameter is the dataset, the second the error that will be displayed to the user, the third a description of the update (which can be `ukModify`, `ukInsert`, and `ukDelete`), and the last the action you want to perform (the default value `uaFail`, `uaAbort`, `uaSkip`, `uaRetry`, or `uaApplied`). A program can look at the data related to the error and determine which action to perform, although it is generally very difficult to fix a problem. All we can do is use the `OldValue` and `NewValue` properties of each field to evaluate error conditions and the `UpdateStatus` function to determine the update operation (modify, insert, or delete). To fix an error you can set the `NewValue` to a proper value and try to reapply the changes, although doing this might create an infinite loop.

In the CacheUpd example I've used the `UpdateStatus` function to show the current status of each record in the status bar as the user moves from one record to another:

```
procedure TForm1.Query1AfterScroll(DataSet: TDataSet);
begin
  // show the record update status in the status bar
  case Query1.UpdateStatus of
    usUnmodified:
      StatusBar1.SimpleText := 'Non Modified';
    usModified:
      StatusBar1.SimpleText := 'Modified';
    usInserted:
      StatusBar1.SimpleText := 'Inserted';
  end;
end;
```

When an error occurs, the program shows a complex secondary form, which lists in a string grid all the fields of the current record, which were modified by the user. The grid is part of a simple dialog box that is created at run time and is initialized in the `FormCreate` method by filling the first row with a description of each column.

But the real initialization of the form takes place when an error occurs, as described before. Here is the rather long but fully commented listing of the `OnUpdateError` event handler of the query:

```
procedure TForm1.Query1UpdateError(DataSet: TDataSet;
  E: EDatabaseError; UpdateKind: TUpdateKind;
  var UpdateAction: TUpdateAction);
var
  strDescr: string;
  I, nRow: Integer;
begin
  nRow := 0;
  // create the dialog box
  ErrorsForm := TErrorsForm.Create (nil);
  try
    // set the caption to a description of the record
    ErrorsForm.Caption := 'Record: ' +
      DataSet.FieldByName('LastName').AsString;

    // for each modified field
    for I := 0 to DataSet.FieldCount - 1 do
      if DataSet.Fields [I].OldValue <>
          DataSet.Fields [I].NewValue then
        begin
          // add a row to the string grid
          Inc (nRow);
          ErrorsForm.StringGrid1.RowCount := nRow + 1;
          // copy the data to the new row
          with ErrorsForm.StringGrid1, DataSet.Fields[I] do
          begin
            Cells [0, nRow] := FieldName;
            Cells [1, nRow] := string (OldValue);
            Cells [2, nRow] := string (NewValue);
          end;
        end;

    // if new items were added, show the dialog
    if (nRow > 0) and
      (ErrorsForm.ShowModal = mrOk) then
    begin
      // revert the record and hide the message
      (DataSet as TQuery).RevertRecord;
      UpdateAction := uaAbort
    end
    else
      // skip the record, keeping it in the cache
      UpdateAction := uaSkip;
  finally
    ErrorsForm.Free;
  end;
end;
```

The effect of this program is visible only when you run two copies of it, make changes on the same record in both windows, and then apply the updates in both. Do not apply the updates in one window before doing the edits in the other one, or the system will automatically update the value when the editing operation starts. You can see an example of the secondary dialog box displayed in case of an error in Figure 10.27.

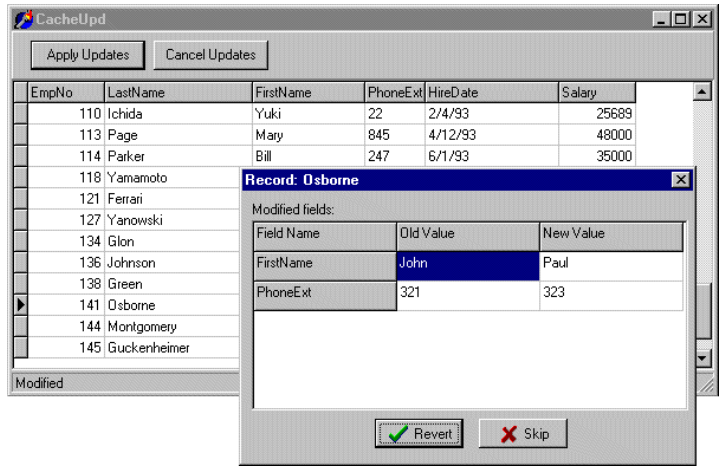


Figure 10.27: The error dialog box displayed when there is a conflict in the updates with the CacheUpd example. Notice the description of the record in the status bar of the main form. Image from the original book.

note If your tables use an ID to identify records, using cached updates might get you in trouble, since different applications see different views of the data (whatever they have in memory, not what is saved on disk). A common solution is to have a separate table with a counter. This is also a typical technique in client/server applications.

What's Next?

In this chapter, we've explored many advanced features of Delphi database programming, which apply both to local databases and to SQL servers. We've explored the structure of database applications based on multiple forms, the use of data modules and the Data Dictionary when building complex applications, and we've even built an MDI program.

Since many Delphi programmers tend to use Paradox tables, a lot of coverage was devoted to using these file-based database in a multiuser networked environment, but also to error handling, concurrency, and other related topics.

However, Access tables are getting very popular as local files, and Delphi 5 support for ADO makes it very simple to use Access and other databases. This will be the topic of the Chapter 12, which will use data access components that don't require the BDE. More components of this type, the InterBase Express components, will be discussed in the next chapter, along with a detailed introduction to client/server development with Delphi.

Chapter 11: Client/Server Programming

In the last two chapters we've examined Delphi's support for database programming, focusing mainly on the use of local files (particularly Paradox) that might or might not be shared over a network. This chapter moves on to the use of SQL server databases, focusing on client/server development[276]. A single chapter cannot cover this complex topic in detail, so I'll simply introduce it from the perspective of the Delphi developer and add some tips and hints. The next chapter extends our discussion of client/server programming to Microsoft's ADO support.

The RDBMS (Relational DataBase Management System), or SQL server, that we'll focus on is InterBase, simply because it is included in the Client/Server edition of

276 While client/server programming is much more actual than the previous local database chapters, some of the components and specific techniques used here are not up to date.

Delphi (a Local version of InterBase is available also in Delphi Professional[277]). I won't try to assess whether this server is better or more robust than competing (and often more expensive) alternatives but simply use InterBase to test the code snippets and the examples presented along with the text.

In a rapid application development (RAD) tool such as Delphi, you can indeed take the same components and code developed for a local database application and use them in a client/server environment. However, this handy feature may prove to be dangerous to beginners, as a standard technique that works well for local access might become extremely inefficient in a client/server application.

note Most of the information in this chapter applies only to the Enterprise (formerly Client/Server) edition of Delphi[278]. Of course, the general information about SQL can be useful whenever you are writing queries against local files, and the basic concepts of client/server architecture apply even if you are accessing a server using ODBC or other techniques available in Delphi Professional.

An Overview of Client/Server Programming

The database applications in previous chapters used the BDE to access data stored in files either on the local machine or on a networked computer. In both cases we used a file server, whose only role was to store the file on a hard disk, because the database engine (the BDE) was running exclusively on the computer that also hosted the application. In this configuration, when we query one of the tables, its data is first copied into a local cache of the BDE and then processed.

As an example, consider taking a table like Employee (part of the InterBase IBLocal sample database[279], which ships with Delphi), adding thousands of records to it, and placing it on a networked computer working as a file server. If we want to know the

277 Today all Delphi editions include the Developer edition of InterBase and the embedded versions, IBLite and IBToGo (which is a single product, with features depending on the license).

278 This is not true any more, even if FireDAC features depend on the SKU, access to the local InterBase engine is available in all Delphi editions.

279 The InterBase sample database hasn't changed much over the years, the same sample tables still exists today.

highest salary paid by the company, we can open a Table component[280] (`EmpTable`) connected with the database table and run this code:

```
EmpTable.Open;
EmpTable.First;
MaxSalary := 0;
while not EmpTable.Eof do
begin
  if EmpTable.FieldByName ('Salary').AsCurrency > MaxSalary then
    MaxSalary := EmpTable.FieldByName ('Salary').AsCurrency;
  EmpTable.Next;
end;
```

The effect of this approach is to move all the data of the (large) table from the networked computer to the local machine, an operation that might take minutes. Because Delphi includes a Query component[281], you might think of using the following SQL code to compute this maximum value:

```
select Max(Salary) from Employee
```

In case of a Paradox table, this query would be processed by the local SQL engine of the BDE, and the entire data set of the table would still have to be moved from the networked computer to the local one, with similarly poor performance. But if you use InterBase and let the server execute the SQL code, only the result set—a single number—will need to be transferred to the local computer.

note The two code excerpts above are part of the GetMax example, which includes some code to time the two approaches. Using the Table component on the small Employee table takes about ten times longer than using the query, even if the InterBase server is installed on the client computer.

If you want to store a large amount of data on a central computer and avoid moving the data to client computers for processing, the only solution is to let the central computer manipulate the data and send back to the client only a limited amount of information. This is the foundation of client/server programming.

In general, you'll use an existing program on the server (an RDBMS) and write a custom client application that connects to it. Sometimes, however, you might even want to write both a custom client and a custom server, as in three-tier applications. Delphi support for this type of program—the MIDAS architecture[282]—is covered in Chapter 21.

---

280 You could use FireDAC FDTable to build a similar demo project.

281 Or, better, FireDAC FDQuery component.

282 Now called DataSnap.

The *upsizing* of an application—that is, the transfer of data from local files to a SQL server database engine—is generally done for performance reasons and to allow for larger amounts of data. Going back to the previous example, in a client/server environment the query used to select the maximum salary would be computed by the RDBMS, which would send back to the client computer only the final result, a single number! With a powerful server computer (such as a multiprocessor Sun SparcStation[283]), the total time required to compute the result might be minimal.

However, there are also other reasons to choose a client/server architecture[284]:

The amount of data: A Paradox table cannot exceed 2GB, but even around 300MB you might start having serious speed problems, and errors in the indexes become more frequent.

The need for concurrent access to the data: Paradox uses the Paradox.NET file to keep track of which user is accessing the various tables and records. The Paradox approach to handling multiple users is based on *pessimistic locking*. When a user starts an editing operation on a record, none of the other users can do the same (to avoid any update conflict), as we saw in the last chapter. In a system with tens of users, this might lead to serious problems, because a single user might block the work of many others. SQL server databases, by contrast, generally use *optimistic locking*, an approach that allows multiple users to work on the same data and delays the concurrency control until the time the users send back some updates.

Protection and security: An RDBMS usually has many more protection mechanisms than the simple password you can add to a Paradox table. When your application is based on files, a malicious or careless user might simply delete those vital files. When SQL servers are based on robust operating systems, instead, they provide multiple levels of protection, make backups easier to do, and often allow only the database administrator to modify the structure of the tables.

Programmability: An RDBMS database can host business rules, in the form of stored procedures, triggers, table views, and other techniques we'll discuss in this chapter. Choosing how to divide the application code between the client and the server is one of the main issues of client/server programming.

Transaction Control: We saw in the last chapter that Paradox and the BDE offer limited support for transactions, but the transaction support provided by an

---

283 Well, that was a great computer at the time the original book was written...

284 These reasons are still accurate today. Protection of data, also from unauthorized access and even more in case of sensitive data a company must legally protect, is more important these days and InterBase offers extensive encryption support, both for data at rest and in transit. The same is available also for the local IBToGo engine.

RDBMS database is generally much greater. This is another important aspect of the overall robustness of the system.

Client/Server and Delphi

Now let's consider how Delphi fits into the client/server picture. How does Delphi help us build client/server applications? As I've mentioned, you can still use all the components and techniques discussed in the last two chapters, although in some cases alternative approaches will help you leverage the power of the RDBMS your application is dealing with.

The Database Component

In local applications, programmers usually refer to the database by indicating the alias of the file path in the `DatabaseName` property of the Table and Query components. An alternative is to use the Database[285] component to define a local alias and then let all the DataSet components refer to this local alias.

As an example, consider the components of the GetMax application discussed earlier:

```
object Database1: TDatabase
  AliasName = 'IBLOCAL'
  Connected = True
  DatabaseName = 'IB'
  LoginPrompt = False
  Params.Strings = (
    'USER NAME=SYSDBA'
    'PASSWORD=masterkey')
  SessionName = 'Default'
end
object EmpTable: TTable
  DatabaseName = 'IB'
  TableName = 'EMPLOYEE'
end
object EmpQuery: TQuery
  DatabaseName = 'IB'
```

285 In FireDAC you'd use an FDConnection component to manage the overall connection to the database, with datasets referring to it. The FDConnection component has a very powerful component editor to configure and customize the database connection.

```
  SQL.Strings = (
    'select Max(Salary) from Employee ')
end
```

In a client/server application, using the Database component is almost mandatory, as it is required to define connectivity and login parameters (the user name and password, as you can see in the `Params` property above) and to handle transactions.

Keep in mind that the Database component establishes a connection with the RDBMS, representing one of the clients of the system. As such, on most servers it requires a license, and your organization is typically paying for a fixed number of licenses. If the same application or the same computer uses multiple connections to the server, it can count as multiple clients! Fortunately, by setting the `KeepConnection` property of the Database component, you can specify whether to keep the database connection active even when there is no active DataSet component using the connection. If your program can fetch some data and then operate on it locally, disconnecting from the server might help you conserve licenses[286].

The Role of the BDE

What is the role of the BDE in this architecture? In a client/server application built with Delphi 4 or earlier versions, the client programs still need to interact with a local copy of the BDE, installed on the client machine. Using the new Delphi 5 ADO or InterBase Express components, you can avoid installing the BDE on the client[287] (but you won't be able to use its features, of course).

Let's discuss the traditional approach first. (It will still be very common with Delphi 5, anyway.) The BDE doesn't know how to handle the RDBMS; it uses some further drivers, called SQL Links, to perform this operation. As an alternative, the BDE can also interact with ODBC drivers. Inprise provides native BDE drivers for InterBase, Oracle, Informix, MS SQL, Sybase, and DB2.

If the BDE is still required on the local machines, it can actually be very efficient. For example, when you use the pass-through mode for queries, the BDE doesn't try to interpret the SQL code but passes it directly to the RDBMS server. This allows

---

286 Similar considerations still applies today, in different areas. For example, sharing connections among multiple threads with connection pooling (a feature included in FireDAC).

287 This is true for FireDAC, which is a library fully written in Delphi and embedded into your executable (unless you use runtime packages). FireDAC requires no deployment other than the client library for the specific database you want to use.

you to use a server's specific SQL commands and also to speed up the execution. The pass-through mode is activated using the BDE Administrator utility[288].

Having the BDE between the client and the server can also help in building applications designed to work with multiple servers[289]. In practice, however, it's not easy to do this and still obtain the best performance, because of differences in the SQL dialects understood by each SQL server. In particular, data types are handled differently by the various servers. If the same table were placed on two servers that have data type differences, Delphi would need to use two different `TField` objects.

Also keep in mind that the BDE generally treats data with a record-oriented approach typical of local files, rather than the set-oriented approach of SQL servers.

note An interesting aspect of the BDE is its ability to perform heterogeneous joins; that is, it can run SELECT statements on multiple tables of different databases (using different SQL servers and local tables). This can be useful, as many servers offer no support for connecting to external tables, but you should keep in mind that to perform this operation the BDE often needs to fetch the entire content of the tables involved in executing the query on the client computer.

There are two alternatives to using the BDE: the direct use of the specific server's API, as in the InterBase Express components; or the use of a different database engine, as in the combination of ADO with OLE DB. Using the server API can result in better performance, but the application will generally not be portable to another SQL server. The native Delphi 5 dataset components for InterBase certainly make the server API approach appealing, and I'll discuss them later in this chapter (after an introduction to client/server programming based on the traditional approach and the BDE).

Another alternative, covered in the next chapter, is the use of ADO instead of the BDE[290]. As we'll see, the advantage of ADO is that its engine is part of the Windows 2000 operating system, so you can take it for granted (if not now, at least in the near future). Also, using ADO might be a good choice in conjunction with Microsoft database technology (including MS Access and MS SQL Server).

If you plan to use Oracle, I think the BDE and ADO are equally good alternatives: although the future of ADO certainly looks brighter, the BDE support for the Object Relational model of Oracle 8.0 is probably superior. Since Oracle and Microsoft are

288 Most modern database access libraries allow passing the queries to the database with little or no upfront processing.

289 This is certainly more true with FireDAC, which offers universal data access across dozens of RDBMS and at the same time can optimize access and performances to each of them.

290 The importance of classic ADO and of the associated components is more limited these days. I'll add some specific comments in the next chapter.

battling each other over database issues (including, for example, the SQL extensions used to introduce the object-relational model) there might be surprises in this area. Keep in mind, anyway, that if Oracle is your definitive choice you'll also be able to use direct components similar to those Delphi provides for InterBase (see, for example, Direct Oracle Access at `www.allroundautomations.nl`[291]).

From Local to Client/Server

Now we can start focusing on particular techniques useful for client/server programming. Keep in mind that the general goal is to distribute the workload properly between the client and the server and reduce the network bandwidth required to move information back and forth.

The foundation of this approach is good database design, which involves both table structure and appropriate data validation and constraints, or business rules. Enforcing the validation of the data on the server is important, as the integrity of the database is one of the key aims of any program. However, the client side should include data validation as well, to improve the user interface and make the input and the processing of the data more user-friendly. It makes little sense to let the user enter invalid data and then receive an error message from the server, when we can prevent the wrong input in the first place.

note If you use a CASE tool for the definition of the database, or import the definition in such a tool afterwards, you can use Delphi's Case Wizard[292] to generate a corresponding data dictionary and have the field objects created at design time automatically import the constraints specified on the server.

Unidirectional Cursors

In local databases, tables are sequential files whose order is either the physical order or is defined by an index. By contrast, SQL servers work on logical sets of data, not related to a physical order. A *relational* database server handles data according to the relational model, a mathematical model based on set theory.

291 The company is still active today, offering tools and components for Oracle.

292 This is another tool that was tied to the BDE and the data dictionary, and has been long been removed from the product.

What is important for the present discussion is that in a relational database, the records (sometimes called tuples) of a table are identified not by position but exclusively through a primary key, based on one or more fields. Once you've obtained a set of records, the server adds to each of them a reference to the following one, which makes it fast to move from a record to the following one but terribly slow to move back to the previous record. For this reason, it is common to say that an RDBMS uses a *unidirectional* cursor.

Connecting such a table or query to a DBGrid control makes it very slow when browsing the grid backwards. Actually, the BDE helps a lot, as it keeps in a local cache the records already loaded in the table[293]. Thus, when we move to following records, they are requested from the SQL server, but when we go back the BDE jumps in and provides the data. In other words, the BDE makes these cursors fully bidirectional, although this might use quite a lot of memory.

note The simple case of a DBGrid used to browse an entire table is common in local programs but should generally be avoided in a client/server environment. It's better to filter out only part of the records and only the fields you are interested in. Do you need to see a list of names? Return all those starting with the letter A, then those with B, and so on, or ask the user for the initial letter of the name.

If proceeding backward might result in problems, keep in mind that jumping to the last record of a table is even worse; usually this operation implies fetching all the records[294]!

A similar situation applies to the `RecordCount` property of data sets. Computing the number of records often implies moving them all to the client computer. This is the reason why the thumb of the vertical scrollbar of the DBGrid works for a local table but not for a remote one. If you need to know the number of records, run a separate query to let the server (and not the client) compute it. For example, you can see how many records will be selected from the Employee table if you are interested in those records having a salary field higher than 50,000:

```
select count(*)
from Employee
where Salary > 50000
```

293 FireDAC also caches the query results, but in ways that are very powerful and can be extensively customized for the best performance.

294 This can now be avoided by smart data access layers like FireDAC by reversing the data query access, starting form the last record and moving backwards.

note Using the SQL instruction `count(*)` is a handy way to compute the number of records returned by a query. Instead of the `*` wildcard, we could have used the name of a specific field, as in `count(First_Name)`, possibly combined with either `distinct` or `all`, to count only records with different values for the field or all the records having a non-null value.

Table and Query Components in Client/Server

In Delphi there are two components you use to access an existing database table, Table and Query[295]. When building client/server applications, programmers tend to use the Query component exclusively, but that is certainly not mandatory and there are cases in which using the simpler Table component has no drawback. Here's a quick look at the pros and cons of both components:

- While the Table component should not be used to access a large table[296], it can work perfectly well with a small lookup table. By opening a Table component you don't transfer the entire content of the table to the local machine; the data is moved only when you access specific records.
- Consider also that with the Table component, the BDE asks the server first for the table structure and then for the table data[297]. These two steps are necessary for setting up the proper internal structures of the BDE, and they are not executed by the Query component. If you activate the BDE's Schema Caching feature, the logical structure of the table will be kept locally, saving this extra step. Of course, this might create problems if the logical structure of the table changes on the server.
- One problem with the Table component is that the BDE mimics a bidirectional cursor by caching the data locally. With a Query component, instead, you can

295 Again, FDTable and FDQuery have matching features and also similarly named properties for easier migration of existing BDE code to FireDAC. For this purpose, Delphi includes a specific script to help migrate your code, powered by the reFind tool. This tool uses regular expressions to find specific text in your code and DFM files and replace references to the BDE components with references to the FireDAC equivalents. Because the tool is powered by a script you can edit, a few iterations might be needed for migrating large applications.

296 With FireDAC offering better optimizations, this is no longer the case. However, in most database structures, the fields in a table have references to other tables and a query can fetch the data from multiple tables offering a more complete data view to an end user. Therefore table access is not that commonly used.

297 This is also true for FireDAC, but it happens also for queries. You can keep a local copy fo the the dataset configuration and even store the `FieldDefs` in the configuration (in the DFM file) to reduce the metadata access.

specify whether you want this caching or not with the `Unidirectional` property[298].

- Another point to consider is that you can generally edit the result of a simple query, sending the data back to the SQL server. This is accomplished by setting the `RequestLive` property to `True`. For more complex queries, however, you'll need to use an UpdateSQL component[299], something we'll discuss later in this chapter.
- When trying to minimize the data moved between the server and the client, you need to consider the size of each record as well as the total number of them. When you select only a few fields with a query, only part of the data is considered. A Table component, instead, always entails transferring the entire record to the local machine, even if you've filtered out some fields using the Fields editor. The same problem takes place when you ask for a live query (by setting the `RequestLive` property). In this case the BDE needs to see the entire record in order to send back the proper update commands. This means that selecting all the records of a table with a live query is equivalent to using the Table component.
- The Query component is not limited to `select` SQL statements; you can also use it to insert or delete records. When the Query component returns a dataset, you generally activate it with the `Open` method (or with the equivalent operation, setting the `Active` property to `True`). When the Query component is used to perform an operation on the server, you activate it by calling the `ExecSQL` method[300].

Parametric Queries and Null Values

Parametric queries are a very useful technique. Essentially, they allow you to run multiple queries with different result sets while the server only needs to work out the access strategy for solving the query once.

You can force this initial preparation of the query access strategy by calling the `Prepare` method of a Query component. With this operation the server receives the

---

298 FireDAC offers these features but also additional options in this regard. Sing unidirectional cursors can save local resources and be an important optimization. You can also combine a unidirectional cursors and an in memory dataset, like FDMemTable.

299 The FDUpdateSQL component has a similar role, but FireDAC ability to generate updates automatically is significantly higher than what the BDE used to offer.

300 To run a query command, FireDAC FDQuery offers a compatible `ExecSQL` method, but also a more specific `Execute` method.

query, checks its syntax, and while compiling it determines how to use indexes and other access techniques[301]. If the query is then executed multiple times, it will be faster as these initial operations have already been executed once for all. Of course, you should call `Prepare` again if you change the SQL text of the query. Also, remember to call `Unprepare` at the end, to free some BDE resources.

Note that some powerful SQL servers can do the same operation by caching the requests and automatically determining that you are sending the same request twice. If the server is smart enough, preparing the query might result in little or no performance gain.

note When you write parametric queries against a SQL server, you should consider `null` values with care. In fact, to test for a `null` value you should not write a `field = null` test, but use the specific expression `field is null` instead.

Using Table and Query Filters

One way to limit the amount of data returned by a table is to filter it. Using the `Filter` property of the Table component you can specify a condition similar to the `where` clause of a query. When you work with local databases, the filter is applied by the BDE, but with a SQL server the BDE passes the condition to the server in the query generated for the table. This makes filtered tables very portable between local and client/server applications.

note The situation is different if you filter the records in the Pascal code, using the `OnFilterRecord` event. In this case all the records are sent to the client application, which does its own custom filtering.

If you use a filter with a Query component, the filtering operation will always be performed locally by the BDE, even when you are working with a SQL server. In this case, the BDE asks the server for the entire result set of the query. This would be reasonable only when the user of the application changes the filtering condition often. For a query, only the local filter will be modified, and the data in the local cache will be used. For a table, the BDE will generate an updated query to be executed.

301 Preparing queries still makes sense in most data access frameworks, even if this is often automated and happens without issuing a specific command.

Getting Started with Local InterBase

Now that we've looked at the theory and some of the most common pitfalls of client/server programming, we can start looking at some practical examples introducing Local InterBase, the single-user version of the Inprise[302] RDBMS included in the Professional and Enterprise editions of Delphi. This local version of the SQL server engine is useful for developing and testing a client/server application on a single computer[303].

note Among the advantages of using InterBase are its simple administration tools, which are Windows applications, its limited "footprint," and its good performance on large amounts of data.

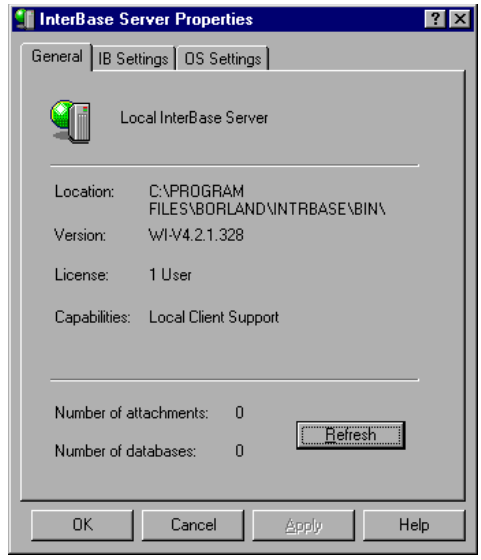
After installing Local InterBase (available on the Delphi 5 CD), you'll be able to activate the server from the Windows Start menu[304]. (The IBServer.EXE program is in the Bin directory of InterBase, usually under the Program Files\Borland\IntrBase path or Program Files\InterBase Corp\InterBase, depending on your installation.) When the server is active, you'll see a corresponding icon in the Tray Icon area of the Windows TaskBar. The menu connected with this icon allows you to configure InterBase and activate its automatic startup. Double-clicking on the icon displays status information, as you can see in Figure 11.1.

There are two main tools you can use to interact directly with InterBase. One is the Server Manager application, which can be used to administer both a local and a remote server, and the other is Windows Interactive SQL (or WISQL).

302 The temporary company name used by Borland for a few years, before chancing it back to Borland and before selling Delphi to Embarcadero (after creating the CodeGear business unit).

303 This version is now the developer edition, which ships with Delphi. It's most visible practiccal limitation, compared to the full server, is that you need to restart it every 48 hours.

304 InterBase installation is now an option in the Delphi installer and the database is generally installed and configured as a service, so you can open the Windows Services app to turn it on (and to restart it, as mentioned above).



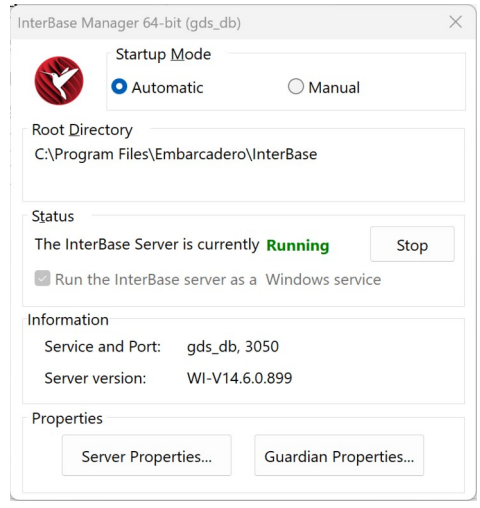


Figure 11.1: The status information displayed by InterBase when you double-click on its tray icon. The License and Capabilities information indicate this is actually Local InterBase. Images from the original book and one captured of the InterBase Manager for the InterBase 2020 Update 6 version that ships with Delphi 12.

Server Manager can be used for administering local or remote InterBase databases and servers. With Server Manager, you can manage database security (authorize new users, change user passwords, and remove user authorizations), back up a database, perform maintenance tasks, see database statistics, and execute other related operations. Figure 11.2 shows an example of using the Server Manager to add a user.

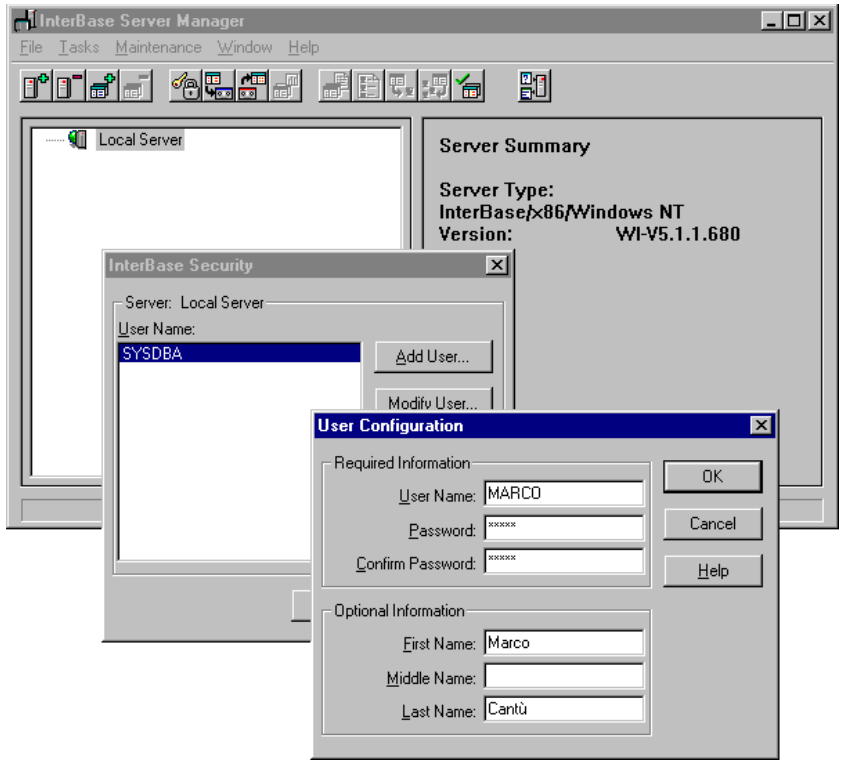


Figure 11.2: The InterBase Server Manager tool, while adding a new user. Image from the original book.

The Windows InterBase ISQL (Interactive SQL) application, available in the InterBase\Bin directory, can be used to execute a SQL statement on a local or remote InterBase server[305]. You can start WISQL, connect to an existing local or remote database, and enter a SQL statement. For example, you could connect to the IBLOCAL alias and enter this statement:

```
select First_Name, Last_Name
from employee
where Job_Code = "Eng"
```

This SQL command outputs the first and last name of every employee in the Engineering (*Eng*) department, as you can see in Figure 11.3. Windows ISQL can be used

305 InterBase today ships also with the IBConsole app, a very powerful client application for managing, configuring, creating data structure and trying out queries against any version of InterBase (Developer, Server, Desktop or embedded). IBConsole fully replaces Windows ISQL, while the command line version ISQL still exists and can be handy, for example, when working on a remote Linux server.

to view the contents of a database, but its real role is in database setup and maintenance. You can define new tables, add indexes, write stored procedures, and so on. You can do all these operations using the Data Definition portion of SQL.

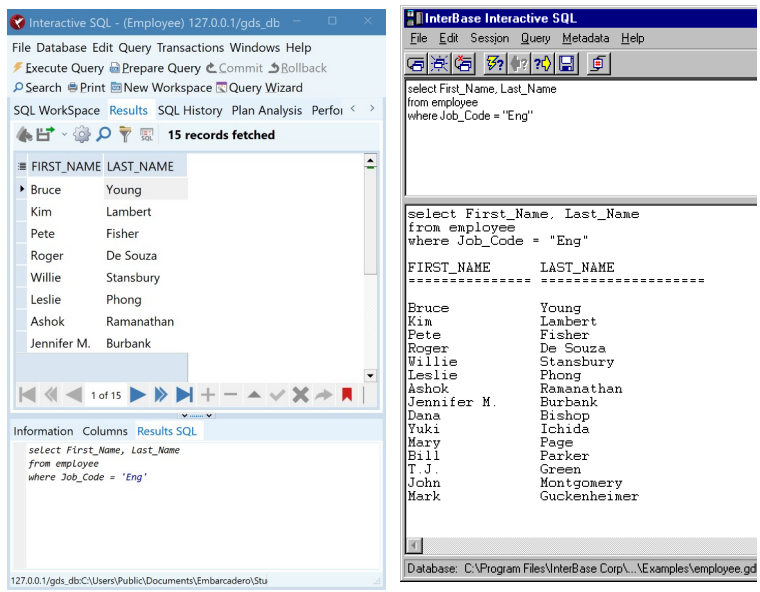


Figure 11.3: The result of executing a SQL statement in the Windows ISQL application. Image from the original book. You can also see here the current Interactive SQL, which is part of IBConsole, running the same query (the query editor and results are in separate tabs).

note As an alternative, you can use a generic database front end to see and modify an InterBase database. For example, you can use Delphi's Database Explorer and Database Desktop to navigate through existing tables or databases[306], as well as to insert and delete records and modify existing values. For instance, with the Database Explorer you can execute a SQL statement similar to the one in the WISQL example. By simply selecting the Data and SQL tabs, you can see all of the data or select just some fields or records.

An alternative to these Windows-based server-management applications is to use some command-line tools that perform similar tasks. These tools (ISQL for queries, GBACK for backups, GFIX for fixing problems, and a few others) are handy for batch processing and when you are working in a Unix or Linux environment. (However, you can still use the simpler Windows-based tools from a client Windows machine connected to the Unix server.)

306 While these tools don't exist any more, same metadata navigation features are now available in the Data Explorer pane within the Delphi IDE.

SQL: The Data Definition Language

RDBMS packages are generally based so closely on SQL (Structured Query Language, commonly pronounced "sequel") that they are often called SQL servers. SQL is defined by an ANSI/ISO committee, although many servers use custom extensions to the last official standard (called SQL-92 or SQL2). Recently many servers have started adding object extensions, which should be part of the future SQL3 standard.

Contrary to what its name seems to imply, SQL is used not only to query a database and manipulate its data, but also to define it. SQL actually consists of two areas: a Data Definition Language (DDL), including the commands for creating databases and tables; and a Data Manipulation Language (DML), including the query commands.

The DDL commands are generally used only when designing and maintaining a database; they are not used directly by a client application. The starting point is the `create database` command, which has a very simple syntax:

```
create database "mddb.gdb";
```

This command creates a new database (in practice, a new GDB file) in the current directory or in the indicated path. Using WISQL, you can also create a database using the File ➢ Create Database menu command[307]. In the above statement notice the final semicolon, used in WISQL as a command terminator. The opposite operation is `drop database`, and you can also modify some of the creation parameters with `alter database`.

note In general, client programs should not operate on metadata, an operation that in most organizations would compete with the database administrator's responsibilities. I've added these calls to a simple Delphi program (called DdlSample) only to let you create new tables, indexes, and triggers in a sample database. You can use that example while reading the following sections. As an alternative, you can type the commands in the Windows Interactive SQL application.

307 There is now a feature-rich and easy-to-use Create Database wizard in IBConsole.

Data Types

After creating the database, you can start adding tables to it with the `create table` command. In creating a table, you have to specify the data type of each field. SQL includes a number of data types, although it is less rich than Paradox and other local databases. Table 11.1 lists SQL standard data types and some other types available on most servers[308].

Table 11.1: The Data Types Used by SQL

Data Type	Standard	Usage
`char, character(n)`	Yes	Indicates a string of `n` characters. Specific servers or drivers can impose a length limit (32,767 characters for InterBase).
`int, integer`	Yes	An integer number, usually four bytes but platform dependent.
`smallint`	Yes	A smaller integer number, generally two bytes.
`float`	Yes	A floating-point number.
`double precision`	Yes	A high-precision floating-point number.
`numeric (precision, scale)`	Yes	A floating-point number, with the indicated precision and scale.
`date`	No	A date. The implementation of this data type varies from server to server.
`blob`	No	An object that holds a large amount of binary data (BLOB stands for Binary Large OBject).
`varchar`	No	A variable-size string used to avoid the space consumption of a fixed large string.

note The `TStringField` class in Delphi can distinguish between `char` and `varchar` types, indicating the actual type in a property and fixing some the problems of using a `char` that's not padded with trailing spaces in the `where` clause of an `update` statement.

308 There are many more data types available in most servers these days, but covering the enhancements to SQL is really beyond the scope of these footnotes.

Programmers who are used to Paradox and other local engines will probably notice the absence of a logical or Boolean type, of date and time fields (the `date` type in InterBase holds both date and time), and of an `AUTOINC` type, which offers a common way to set up a unique ID in a table. The absence of a logical type can create a few problems when upsizing an existing application. As an alternative, you can use a `smallint` field with 0 and 1 values for `True` and `False`, or you can use a domain, as explained in the next section. An AutoInc type is present in some servers, such as Microsoft SQL Server, but not in InterBase. This type can be replaced by the use of a generator, as we'll discuss later on.

Domains

Domains can be used to define a sort of custom data type on a server. A domain is based on an existing data type, possibly limited to a subset (as in a Pascal subrange type). A domain is a useful part of a database definition, as you can avoid repeating the same range check on several fields, and you can make the definition more readable at the same time.

As a simple example, if you have multiple tables with an address field, you can define a type for this field and then use this type wherever an address field is used:

```
create domain AddressType as char(30);
```

The syntax of this statement also allows you to specify a default value and some constraints, with the same notation used when creating a table (as we'll see in the next section). This is the complete definition of a Boolean domain:

```
create domain boolean as smallint default 0
  check (value between 0 and 1);
```

Using and updating a domain (with the `alter domain` call) makes it particularly easy to update the default and checks of all the fields based on that domain at once. This is much easier than calling `alter table` for each of the tables involved.

Creating Tables

In the `create table` command, after the name of the new table, you indicate the definition of a number of columns (or fields) and some table constraints. Every column has a data type and some further parameters:

- `not null` indicates that a value for the field must always be present (this parameter is mandatory for primary keys or fields with unique values, as described below).
- `default` indicates the default value for the field, which can be any of the following: a given constant value, `null`, or `user` (the name of the user who has inserted the record).
- One or more constraints, optionally with a name indicated by the `constraint` keyword. Possible constraints are primary `key`, `unique` (which indicates that every record must have a different value for this field), `references` (to refer to a field of another table), and `check` (to indicate a specific validity check).

Here is an example of the code you can use to create a table with simple customer information:

```
create table customer (
  cust_no integer not null primary key,
  firstname varchar(30) not null,
  lastname varchar(30) not null,
  address varchar(30),
  phone_number varchar(20)
);
```

In this example we've used `not null` for the primary key and for the first and last name fields, which cannot be left empty. The table constraints can include a primary key using multiple fields, as in:

```
create table customers (
  cust_no integer not null,
  firstname varchar(30) not null,
  ...
  primary key (cust_no, name)
);
```

note The most important constraint is the `constraint reference`, which allows you to define a *foreign key* for a field. A foreign key indicates that the value of the field refers to a key in another table (a master table). This relationship makes the existence of the field in the master table mandatory. In other words, you cannot insert a record referring to a nonexistent master field; nor can you destroy this master field while there are other tables referencing it.

Once you've created a table you can remove it with the `drop table` command, an operation that might fail if the table has some constrained relationships with other tables.

Finally, you can use `alter table` to modify the table definition, removing or adding one or more fields and constraints. However, you cannot modify the size of a field

(for example, a `varchar` field) and still keep the current contents of the table. You should move the contents of the resized field into temporary storage, drop the field, add a new one with the same name and a different size, and finally move back the data.

Indexes

The most important thing to keep in mind about indexes is that they are not relevant for the definition of the database and do not relate to the mathematical relational model. An index should be considered simply a suggestion to the RDBMS on how to speed up data access.

In fact, you can always run a query indicating the sort order, which will be available independently from the indexes (although the RDBMS can generate a temporary index). Of course, defining and maintaining too many indexes might require a lot of time; if you don't know exactly how the server will be affected, simply let the RDBMS create the indexes it needs.

The creation of an index is based on the `create index` command:

```
create index cust_name on customers (name);
```

You can later remove the index by calling `drop index`. InterBase also allows you to use the `alter index` command to disable an index temporarily (with the `inactive` parameter) and re-enable it (with the `active` parameter).

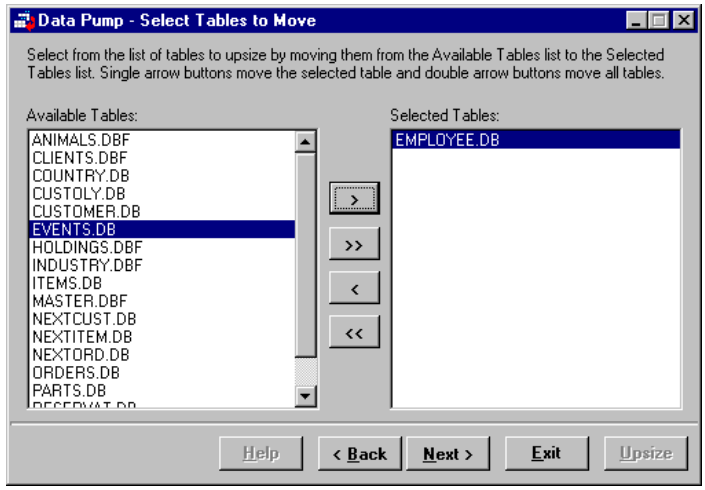
Views

Besides creating tables, the database allows you to define views of a table. A view is defined using a `select` statement and allows you to create persistent *virtual* tables mapped to the physical ones. From Delphi, views look exactly the same as tables.

Views are a handy way to access the result of a join many times, but they also allow you to limit the data that specific users are allowed to see (restricting access to sensitive data). When the `select` statement that defines a view is simple, the view can also be updated, actually updating the physical tables behind it; otherwise, if the `select` statement is complex, the view will be read-only.

Migrating Existing Data[309]

There are two alternatives to defining a database by manually writing the DDL statements. One option is to use a CASE tool to design the database and let it generate the DDL code. The other is to port an existing database from one platform to another, possibly from a local database to a SQL server. The Enterprise version of Delphi includes a tool to automate this process, the Data Pump Wizard.



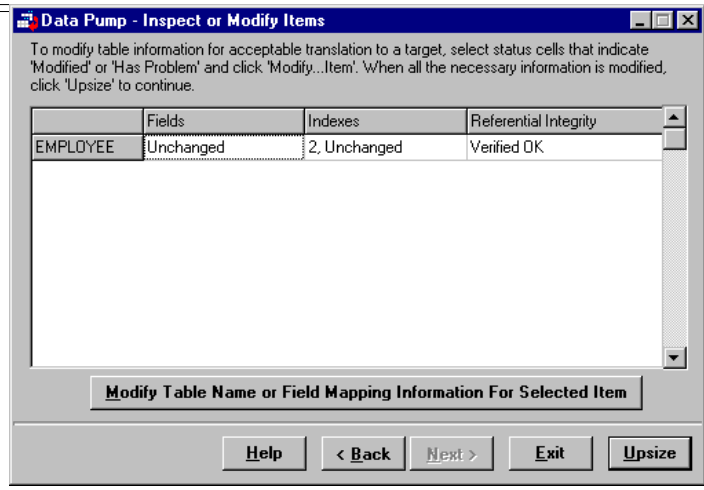
The aim of this tool is to extract the structure of a database and recreate it for a different platform. Before starting the Data Pump, you should use BDE Administrator to create an alias for the database you want to create. Using Data Pump is quite simple: You select the source alias and the target alias; then you select the tables to move (shown next).

When you select a table (for example, the EMPLOYEE table) and click Next, the Data Pump Wizard will verify whether the upsizing operation is possible. After few seconds the wizard will display a list of the tables and let you modify a few options.

If the field conversion is not straightforward, the Data Pump will show the message "*Has Problems*" or "*Modified*." After modifying the options, if necessary, you can press the Upsize button to perform the actual conversion. By selecting a field, you can verify how the wizard plans to translate it; then, clicking the Modify Table Name or Field Mapping Information For Selected Item button, you can change the actual definition.

---

309 FireDAC's BatchMove architectures offes the ability of creating *data pumps* fairly easily and highly customizable way. It require very a little coding, outside of components configuration.



An alternative to the use of the Data Pump Wizard (available only in Delphi Enterprise) is the BatchMove component, which does a default conversion of the tables and cannot be fine-tuned. Finally, you can simply use the Database Desktop, create a new table on the server, and click the Borrow Struct button to extract the table definition from an existing local table.

SQL: The Data Manipulation Language

The SQL commands within the Data Manipulation Language are commonly used by programmers, so I'll describe them in more detail. There are four main commands: `select`, `insert`, `update`, and `delete`. All these commands can be activated using a Query component, but only `select` returns a result set. For the other commands you should open the query using the `ExecSQL` method instead of `Open` (or the `Active` property).

Select

The `select` statement is the most common and well-known SQL command; it's used to extract data from one or more tables (or views) of a database. In its simplest form, the command is

```
select <fields> from <table>
```

In the `<fields>` section you can indicate one or more fields of the table, separated by commas, use the `*` symbol to indicate all the table fields at once, or even specify an operation to apply to one or more fields. Here is a more complex example:

```
select upper(name), (lastname || “,” || firstname) as fullname
from customers
```

In this code, `upper` is a server function that converts all the characters to uppercase, the double pipe symbol (`||`) is the string-chaining operator, and the optional `as` keyword gives a new name to the overall expression involving the first and last name.

By adding a `where` clause, you can use the `select` statement to specify which records to retrieve as well as which fields you are interested in:

```
select *
from customers
where cust_no = 100
```

This command selects a single record, the one corresponding to the customer whose ID number is 100. The `where` clause is followed by one or more selection criteria, which can be joined using the `and`, `or`, and `not` operators. Here is an example:

```
select *
from customers
where cust_no=100 or cust_no=200
```

The selection criteria can contain functions available on the server and use standard operators, including +, –, >, <, =, <>, >=, and <=. There are also few other special SQL operators:

- `is null` tests whether the value of the field is defined.
- `in <list>` returns True if the value is included in a list following the operator.
- `between <min> and <max>` indicates whether the value is included in the range.

Here is an example of these operators:

```
select *
from customers
where address is not null and cust_no between 100 and 150
```

Another powerful operator, used to perform pattern matching on strings, is `like`. For example, if you want to look for all names starting with the letter *B,* you can use this statement:

```
select *
from employee
```

```
where last_name like "B%"
```

The % symbol indicates any combination of characters and can also be used in the middle of a string. For example, this statement looks for all the names starting with *B* and ending with *n*:

```
select *
from employee
where upper(last_name) like "B%N"
```

The use of upper makes the search case-insensitive and is required because the like operator performs a case-sensitive matching. An alternative to like is the use of the containing and starting with operators. Using like on an indexed field with InterBase might produce a very slow search, as the server won't always use the index. If you are looking for a match in the initial portion of a string, it is better to use the starting with expression, which enables the index and is much faster.

Another option is to sort the information returned by the select statement by specifying an order by clause, using one or more of the selected fields:

```
select *
from employee
order by lastname
```

The asc and desc operators can be used for ascending and descending order. The default is ascending.

An important variation of the select command is given by the distinct clause, which removes duplicated entries from the result set. For example, you can see all the cities where you have customers with this expression:

```
select distinct city
from customer
```

The select command can also be used to extract aggregate values, computed by standard functions:

- avg computes the average value of a column of the result set (works only for a numeric field).
- count computes the number of elements in the result set; that is, the number of element satisfying the given condition.
- max and min compute the highest and lowest values of a column in the result set.
- sum computes the total of the values of a column of the result set. (It works only for numeric fields.)

These functions are applied to the result set, usually to a specific column, excluding the `null` values. This statement computes the average salary:

```
select avg(salary)
from employee
```

Another important clause is `group by`, which lets you aggregate the elements of the result set according to some criterion before computing aggregate values with the functions listed above. For example, you might want to determine the maximum and average salary of the employees of each department:

```
select max (salary), avg (salary), department
from employee
group by department
```

Notice that all the noncalculated fields must appear in the `group by` clause. The following is not legal:

```
select max (salary), lastname, department
from employee
group by department
```

note When you extract aggregate values, it is better to use an alias for the result field with the `as` keyword. This makes it easier to refer to the resulting value in your Delphi code.

The aggregate values can also be used to determine the records in the result set. The aggregate functions cannot be used in the `where` clause, but they are placed in a specific `having` section. The following statement returns the highest salary of each department, but only if the value is above 40,000:

```
select max(salary) as maxsal, department
from employee
group by department
having max(salary) > 40000
```

Another interesting possibility is to nest a `select` statement within another one, forming a subquery. Here is an example, used to return the highest-paid employee (or employees):

```
select firstname, lastname
from employee
where salary = (select max(salary) from employee)
```

We could not have written this code with a single statement, since adding the name in the query result would have implied adding it to the `group by` section as well.

Inner and Outer Joins

Up to now our example `select` statements have worked on single tables, a serious limitation for a relational database. The operation of merging data from multiple source tables is called a *table join*. The SQL standard supports two types of joins, called *inner* and *outer*.

An inner join can be written directly in the `where` clause:

```
select *
from <table1>, <table2>
where <table1.keyfield>=<table2.externalkey>
```

This is a typical example of an inner join used to extract all the fields of each table involved. An inner join is handy for tables with a one-to-one relationship (one record of a table corresponding only to one record of the second table). Actually, the standard syntax should be the following, although the two approaches usually generate the same effect:

```
select *
from <table1> left join <table2>
on <table1.keyfield>=<table2.externalkey>
```

An outer join, instead, can be specifically requested with the statement:

```
select *
from <table1> left outer join <table2>
on <table1.keyfield>=<table2.keyfield>
```

The main difference from an inner join is that the selected rows of an outer join will not consider the null fields of the second table. There are other types of joins, including these: the self-join, in which a table merges with itself; the multi-join, which involves more than two tables; and the Cartesian product, a join with no `WHERE` condition, which merges each row of a table with each row of the second one, usually producing a huge result set. The inner join is certainly the most common form.

Insert

The `insert` command is used to add new rows to a table or an updatable view. When you insert a new record in a DBGrid connected with a SQL server table, the BDE generates an `insert` command and sends it to the server. Besides this implicit use, there are several cases in which you'll want to write explicit SQL `insert` calls (including the use of cached updates, which we'll discuss later in this chapter).

Unless you add a value for each field, you should list the names of the fields you are actually providing, as in the following code:

```
insert into employee (empno, lastname, firstname, salary)
values (0,"brown", "john", 10000)
```

You can also insert in a table the result set of a `select` statement (if the fields of the target table have the same structure of the selected fields), with this syntax:

```
insert into <table> <select statement>
```

Update

The `update` command modifies one or more records of a table or view. Delphi generates an `update` call every time you edit data with visual controls connected to a table or a live query on a SQL server. Again, there are also cases where you'll want to use the `update` statement directly.

In an `update` statement you can indicate which record to modify, by using a `where` condition similar to that of a `select` statement. For example, you can change the salary of a specific employee with this call:

```
update employee
set salary = 30000
where emp_id = 100
```

note A single `update` instruction can update all the records that satisfy a given condition. An incorrect `where` clause could unintentionally update many records, and no error message would be displayed.

The `set` clause can indicate multiple fields, separated by commas, and it can use the current values of the fields to compute the new values. For example, the following statement gives a nice raise to the employees hired before January 1, 1990:

```
update employee
set salary = salary * 1.20
where hiredate < "01-01-1990"
```

Delete

The `delete` command is equally simple (although its misuse can be quite dangerous). Again, you generally remove records using a visual component, but you can also issue a SQL command like the following:

```
delete from employee
where empid = 120
```

You simply indicate a condition identifying the records to delete. If you issue this SQL command with a Query component (calling `ExecSQL`), you can then use the `RowsAffected` property to see how many records were deleted. The same applies to the `update` commands.

Using SQL Builder

As we've seen, SQL has a great many commands, particularly in relation to `select` statements. And we haven't actually seen them all! While the DDL commands are generally used by a database administrator, or only for the initial definition of the database, DML commands are commonly used in everyday Delphi programming work.

To help with the development of correct SQL statements, Delphi Enterprise includes a tool called SQL Builder[310]. You easily activate it by right-clicking on a Query component.

Originally, added in Delphi 4 to replace the more limited Visual Query Builder found in earlier versions, SQL Builder is a two-way tool; you can use it both to create the text of a query and to obtain the graphical representation of an existing one (even if you've changed the original text).

Using SQL Builder is very simple. You choose the database you want to work on, and then you select one or more tables, placing them in the work area. After selecting the proper parameters, as explained below, you can use the command Query ➢ Run Query (or F9) to see the result of the query or the command Query ➢ Show SQL (F7) to see the source code of the `select` statement you've generated.

In the selected tables you can simply mark the fields you want to see in the result set. The check box near the name of the table selects all of its fields. But the real

---

310 There is no graphical query builder in Delphi today.

power of SQL Builder lies in two features. First, you can drag a field from one table onto another table to join them, as shown in Figure 11.4.

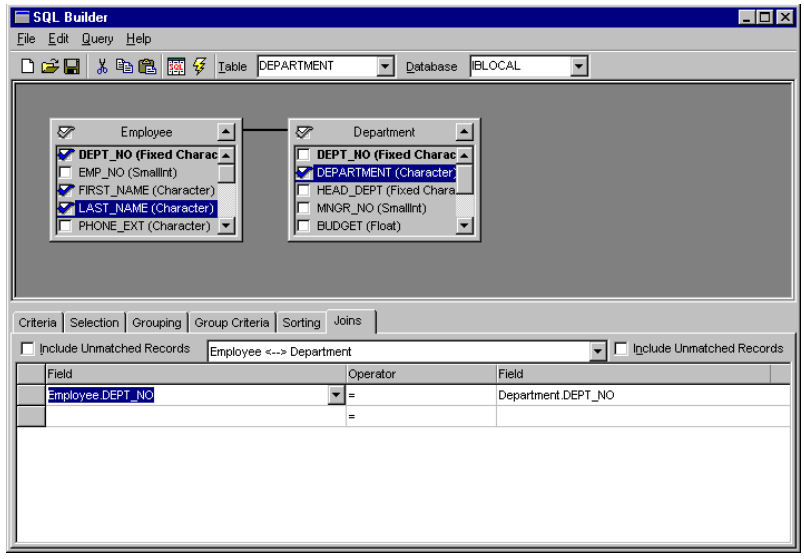


Figure 11.4: Two tables displayed as joined in SQL Builder. Image from the original book.

The other powerful feature is the Query notebook, the multipage control at the bottom of the SQL Builder window. Here is a short description of each of the pages:

- The *Criteria* page indicates the selection criteria of the `where` clause. By selecting one of the fields of the result table, you can indicate a comparison against a fixed value or another field, and you can use `like`, `is null`, `between`, and other operators. Using the local menu of the grid present in this page, you can also activate the `exist` operator or an entire SQL expression. This page allows you to combine multiple conditions with the `and`, `or`, and `not` operators, but it doesn't allow you to specify a precedence among these operators by adding parentheses.
- The *Selection* page lists all the fields of the result set and allows you to give them an alias. With the local menu you can also introduce aggregate functions (`sum`, `count`, and so on). Finally, the upper-left check box indicates the `distinct` condition.
- The *Grouping* page corresponds to the `group by` clause. SQL Builder automatically groups all the fields that are used in the aggregate functions, as required by the SQL standard.

- The *Group Criteria* page corresponds to a `having` clause, which is available in conjunction with aggregate functions. The operations are similar to those of the *Selection* page and are activated by using the local menu.
- The *Sorting* page corresponds to the `order by` clause. You simply select the field you want to sort and then select the ascending or descending sort.
- The *Joins* page is the last but probably the most powerful, as it allows you to define join conditions, beyond the simple dragging of a field from one table to another in the work area. This page allows you to fine-tune the join request by indicating its type (`INNER` or `OUTER`) and selecting conditions other than the equality test.

To better understand how to use the SQL Builder, we can build an actual example based on the sample InterBase database installed by Delphi (and corresponding to the LocalIB alias). The example is in the SqlBuilder directory and its form has a Query, a DataSource, and a DBGrid component, connected as usual. The `DatabaseName` property of the Query component is set to *IBLocal* and a right-click on the component activates SQL Builder, as shown in Figure 11.5.

We want to create a query including the first and last name, department, title, and salary of each employee. This operation requires two joins. Choose the Employee, Department, and Job tables. Click on the Dep_No field of the Department table and drag the cursor over the Dep_No field of the Employee table. Similarly, connect the Job table with the Employee table using the three fields Job_Code, Job_Grade, and Job_Country.

After creating the joins, select the fields you want to see in the result set: First_ Name, Last_Name and Salary from the Employee table; Department from the Department table; and Job_Title from the Job table. Finally, move to the Sorting page of the Query notebook and select Department.Department from the Output Fields list to sort the result set by department.

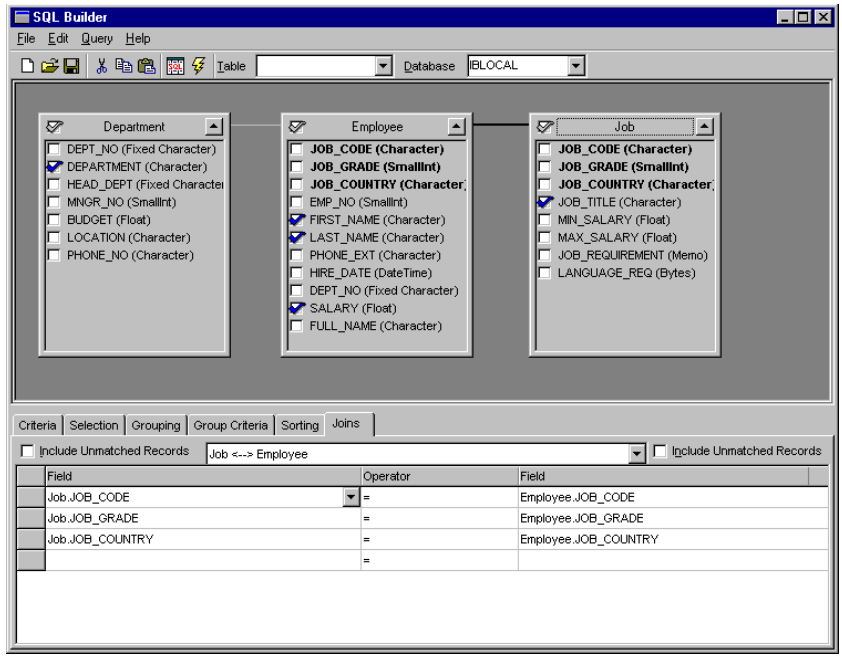


Figure 11.5: A complex join displayed by SQL Builder. Image from the original book.

The following should be the generated SQL:

```
select employee.first_name, employee.last_name,
  department.department, job.job_title, employee.salary
from employee employee
   inner join department department
   on (department.dept_no = employee.dept_no)
   inner join job job
   on (job.job_code = employee.job_code)
   and (job.job_grade = employee.job_grade)
   and (job.job_country = employee.job_country)
order by department.department
```

We might add an extra `where` clause to choose only the employee with a high salary. Simply move to the Selection page, select the Employee.Salary field, go to the column Operator >= and type the value **100,000**. Executing the query you'll see a limited number of records, and looking at the SQL source you'll see the extra statement:

```
where employee.salary >= 100000
```

Finally, note that it is possible to export and import the SQL code from a plain text file. Simply by closing SQL Builder you will also save the text of the query in the `SQL` property of the related Query component.

note When working with ADO queries or InterBase Express ones, you won't have the power of SQL Builder available. You can, however, prepare the query with SQL Builder using a generic `TQuery` component and then copy the SQL code to the actual query component you want to use.

Server-Side Programming

At the beginning of this chapter I underlined the fact that one of the objectives of client/server programming—and one of its problems—is the division of the workload between the computers involved. When you activate SQL statements from the client, the burden falls on the server to do most of the work. However, you should try to use `select` statements that return a large result set, to avoid jamming the network.

Besides accepting DDL and DML requests, most RDBMS servers allow you to create routines directly on the server using the standard SQL commands plus their own server-specific extensions (which are generally not portable). These routines typically come in two forms, stored procedures and triggers.

Stored Procedures

Stored procedures are like the global functions of a Delphi unit and must be explicitly called by the client side. Stored procedures are generally used to define routines for data maintenance, to group sequences of operations you need in different circumstances, or to hold complex `select` statements.

Like Pascal procedures, stored procedures can have one or more typed parameters and a return value. As an alternative to returning a value, a stored procedure can also return a result set, the result of an internal `select` statement.

The following is a stored procedure written for InterBase, which receives a date in input and computes the highest salary among the employees hired on that date:

```
create procedure maxsaloftheday(ofday date)
returns (maxsal decimal(8,2))
as
```

```
begin
  select max(salary)
  from employee
  where hiredate = :ofday
  into :maxsal;
end
```

Notice the use of the `into` clause, which tells the server to store the result of the `select` statement in the `MaxSal` return value. To modify or delete a stored procedure you can later use the `alter procedure` and `drop procedure` commands.

Looking at this stored procedure, you might wonder what its advantage is compared to the execution of a similar query activated from the client. The difference between the two approaches is not in the result you obtain but in its speed. A stored procedure is compiled on the server in an intermediate and faster notation when it is created, and the server will determine at that time the strategy it will use to access the data. By contrast, a query is compiled every time the request is sent to the server (although the server can cache it and avoid recompiling two identical requests). For this reason a stored procedure can replace a very complex query, provided it doesn't change too often!

From Delphi you can activate a stored procedure returning a result set by using either a Query or a Stored Procedure component. With a Query you can use the following SQL code:

```
select *
from MaxSalOfTheDay ("01/01/1990")
```

It's generally easier to use StoredProc when the procedure has multiple or complex parameters. This component lists the stored procedures available on the server and has an easy-to-use dialog box for the definition of the parameters, as shown in Figure 11.6.

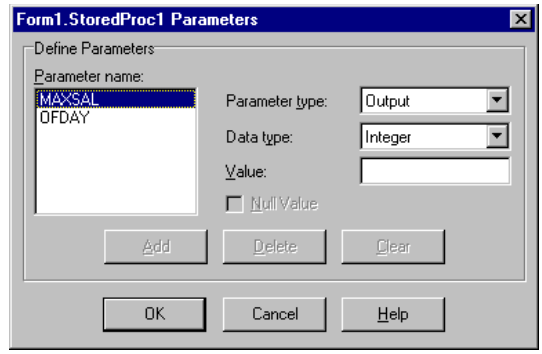


Figure 11.6: The Params property editor of the StoredProc component. Image from the original book.

Triggers (and Generators)

Triggers behave more or less like Delphi events and are automatically activated when a given *event* occurs. Triggers can have specific code or call stored procedures. In both cases the execution is done completely on the server. Triggers are used to keep data consistent, checking new data in more complex ways than a constraint check allows, and to automate the side effects of some input operations (such as creating a log of previous salary changes when the current salary is modified).

Triggers can be fired by the three basic data update operations: `insert`, `update`, and `delete`. When you create a trigger, you indicate whether it should fire before or after one of these three actions.

As an example of a trigger, we can use a generator or sequence to create a unique index in a table. Many tables use a unique index as primary key. SQL servers don't have an AutoInc field, unlike Paradox and other local databases. Because multiple clients cannot generate unique identifiers, we can rely on the server to do this. Almost all SQL servers offer a counter you can call to ask for a new ID, which you should later use for the table. InterBase calls these automatic counters *generators*, while Oracle calls them *sequences*. Here is the sample InterBase code:

```
create generator cust_no_gen;
...
gen_id (cust_no_gen, 1);
```

The `gen_id` function then extracts the new unique value of the generator passed as first parameter, with the second parameter indicating how much to increase (in this case one).

At this point you can add a trigger to a table, an automatic handler for one of the table's events. A trigger is similar to the event handler of the Table component, but you write it in SQL and execute it on the server, not on the client. Here is an example:

```
create trigger set_cust_no for customers
before insert position 0 as
begin
  new.cust_no = gen_id (cust_no_gen, 1);
end
```

This trigger is defined for the Customer table and is activated each time a new record is inserted. The `new` symbol indicates the new record we are inserting. The `position` option indicates the order of execution of multiple triggers connected to the same event. Triggers with the lowest values will be executed first.

Inside a trigger you can write DML statements that also update other tables, but watch out for updates that end up reactivating the trigger, creating an endless recursion. You can later modify or disable a trigger by calling the `alter trigger` statement or `drop trigger`.

---

note Triggers fire automatically for specified events. If you have to make many changes in the database using batch operations, the presence of a trigger might slow down the process. If the input data has already been checked for consistency, you can temporarily drop the trigger. These batch operations are often coded in stored procedures, but stored procedures generally cannot issue DDL statements, like those required for dropping and resetting the trigger. In this situation you can define a view based on a simple `select * from table` command, thus creating an alias for the table. Then you can let the stored procedure do the batch processing on the table and apply the trigger to the view (which should also be used by the client program).

Live Queries and Cached Updates

When working with local data it is very common to use grids and other visual controls, edit the data, and send it back to the database. We've already seen that using a DBGrid might cause a few problems when working with an RDBMS, as moving on the grid might send numerous data requests to the server, creating a huge amount of network traffic.

When you use the Query component to connect to some data, you cannot edit the data unless its `RequestLive` property is set to `True`[311]. If you are working with local tables, the query is always elaborated by the BDE with the Local SQL engine. The BDE will allow for a live query only if it is quite simple: All joins should be outer joins; there cannot be a `distinct` key; there can be no aggregation, no `group by` or `having` clause, no subqueries, and no `order by` unless supported by an index; and there are few other rules you can find in Delphi's Help.

If you are working with a SQL server, setting a live query will put the BDE in control of the query, instead of the server. When connected to a SQL server, a live query behaves like a Table component. (So it makes sense to use the table anyway, in these cases.)

note Most SQL servers, including InterBase, allow you to define updatable views based on the result of a `select` statement that the Local SQL engine of the BDE won't consider updatable. Then you can simply hook a Table component to the view, letting the SQL server do the work and bypassing the Local SQL engine of the BDE.

If the BDE determines that the data set cannot be updated[312], it sets the `CanModify` property to `False`. The DataSource component checks this value before allowing an editing operation. A solution to this problem is to avoid the use of data-aware controls, as discussed in the last chapter, and use specific SQL queries to update, insert, and delete records.

A better approach is to automate this process (retaining the capabilities of the data-aware controls) by using the UpdateSQL component together with the Query component. The UpdateSQL can be used only in conjunction with cached updates, a topic discussed in the last chapter. The basic idea is that the update operations are kept in a local cache until the program calls the `ApplyUpdates` method of the Query component. This operation corresponds to the execution of a series of `update`, `insert`, and `delete` SQL operations on the server, using the data in the cache. The required SQL commands are held by the UpdateSQL component, which has a design-time editor you can use to generate these SQL commands almost automatically.

Cached updates solve the live queries issue, reduce network traffic, define a standard way to solve updates conflicts, and reduce the server load, but they require more memory on the client computer.

311 FireDAC FDQuery FetchOptions property offers a way more granular control on how the data is loaded and kept in memory.

312 FireDAC is generally much smarter in offering automatic updates on more complex queries than the BDE could handle.

The UpdateSQL Component

The role of the UpdateSQL component[313] is to provide a query with the update statements required to make its result set editable. Its key properties are `DeleteSQL`, `InsertSQL`, and `ModifySQL`, but the most important element is the `UpdateObject` property of the related Query component. The update SQL statements are executed when you apply the cached updates, sending the changes to the server. Because cached updates maintain the information on the original records, the updates usually indicate which record to update by passing the original data. This is the only way we have to identify a record on a SQL server, and this technique also helps the server to track any updates on the same record done by other users.

All this setup might seem to imply a lot of work, but it is actually very simple. After you've written a query, you can connect the UpdateSQL component to it and activate the component editor, as shown in Figure 11.7.

This component editor has two sections. The first page indicates the criteria used to generate the SQL statements for adding, deleting, or modifying records. With a join you can select the table to update and the fields involved. When you've completed this step, click the Generate SQL button and the editor will move to the second page, where you can inspect the generated SQL code for the three operations.

313 FDUpdateSQL is the matching component in FireDAC, but it's less commonly used as a separate component, as FireDAC datasets can handle more update scenarios automatically.

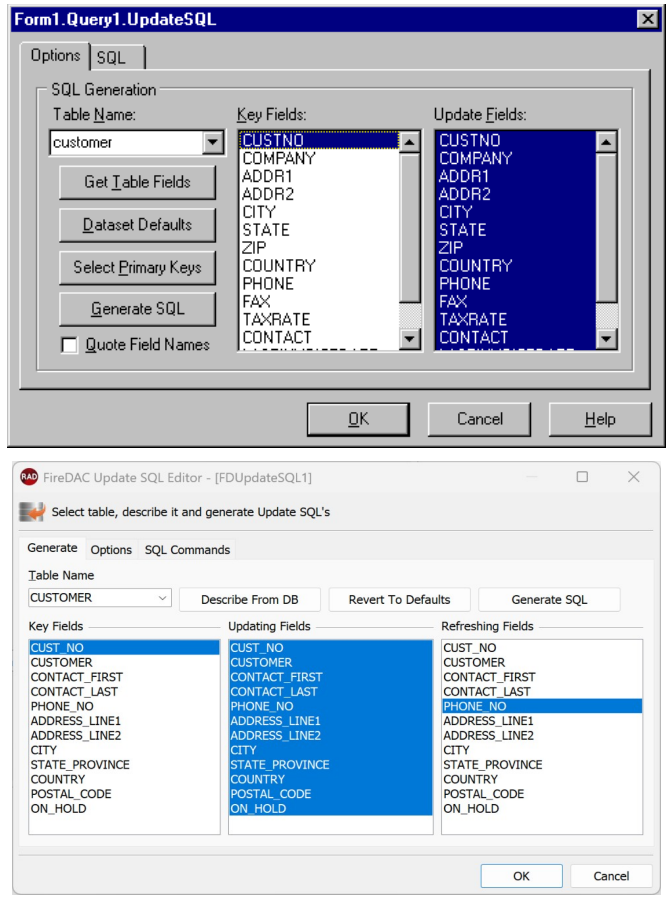


Figure 11.7: The UpdateSQL component editor in action. Image from the original book and fro Delphi 12, based on FireDAC FDUdpateSQL component editor.

The UpdateSQL Example

To demonstrate the real power of the UpdateSQL component, I've built a complex example called UpdateSQL, based on the Employee, Department, and Job tables of the IBLocal database we've used in the past. The example is based on the query shown back in Figure 11.5, which is placed along with the other data access components in a data module.

Here is the textual description of the UpdateSQL component of the example:

```
object EmpUpdate: TUpdateSQL
  ModifySQL.Strings = (
    'update EMPLOYEE'
    'set'
```

```
    '  FIRST_NAME = :FIRST_NAME,'
    '  LAST_NAME = :LAST_NAME,'
    '  SALARY = :SALARY,'
    '  DEPT_NO = :DEPT_NO,'
    '  JOB_CODE = :JOB_CODE,'
    '  JOB_GRADE = :JOB_GRADE,'
    '  JOB_COUNTRY = :JOB_COUNTRY'
    'where'
    '  EMP_NO = :OLD_EMP_NO')
  InsertSQL.Strings = (
    'insert into EMPLOYEE'
    '  (FIRST_NAME, LAST_NAME, SALARY, DEPT_NO, JOB_CODE,'
    '  JOB_GRADE, JOB_COUNTRY)'
    'values'
    '  (:FIRST_NAME, :LAST_NAME, :SALARY, :DEPT_NO, :JOB_CODE, '
    '  :JOB_GRADE, :JOB_COUNTRY)')
  DeleteSQL.Strings = (
    'delete from EMPLOYEE'
    'where'
    '  EMP_NO = :OLD_EMP_NO')
  end
```

To delete the employee records, the program uses a stored procedure, which is already available in the sample database and is connected to the following component:

```
object spDelEmployee: TStoredProc
  DatabaseName = 'AppDB'
  StoredProcName = 'DELETE_EMPLOYEE'
  ParamData = <
    item
      DataType = ftInteger
      Name = 'EMP_NUM'
      ParamType = ptInput
    end>
end
```

The `OnUpdateRecord` event of the Query component uses the stored procedure instead of the default UpdateSQL component for deleting records. Here is the code of the event handler:

```
procedure TdmData.qryEmployeeUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  // when deleting the record, use the stored procedure
  if UpdateKind = ukDelete then
  begin
    // assign emp_no value
    with dmData do
      spDelEmployee.Params[0].Value := qryEmployeeEMP_NO.OldValue;
    try
```

```
      // invoke stored procedure that tries to delete employee
      dmData.spDelEmployee.ExecProc;
      UpdateAction := uaApplied; // success
    except
      UpdateAction := uaFail;
    end;
  end
  else
  try
    // apply updates
    dmData.EmpUpdate.Apply(UpdateKind);
    UpdateAction := uaApplied;
  except
    UpdateAction := uaFail;
  end;
end;
```

Notice that because we perform the update operation directly, we must indicate in the `UpdateAction` parameter whether it succeeds or not. This code is part of the data module. The main form, visible at run time in Figure 11.8, has a couple of extra features. If the user closes the form with any updates pending, the `OnCloseQuery` event of the form displays a warning message, allowing the user to apply the updates or skip them:

```
procedure TMainForm.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
var
  Res: Integer;
begin
  with dmData do
    if qryEmployee.UpdatesPending then
    begin
      Res := MessageDlg (CloseMsg, mtInformation,
        mbYesNoCancel, 0);
      if Res = mrYes then
        AppDB.ApplyUpdates ([qryEmployee]);
      CanClose := Res <> mrCancel;
    end;
end;
```

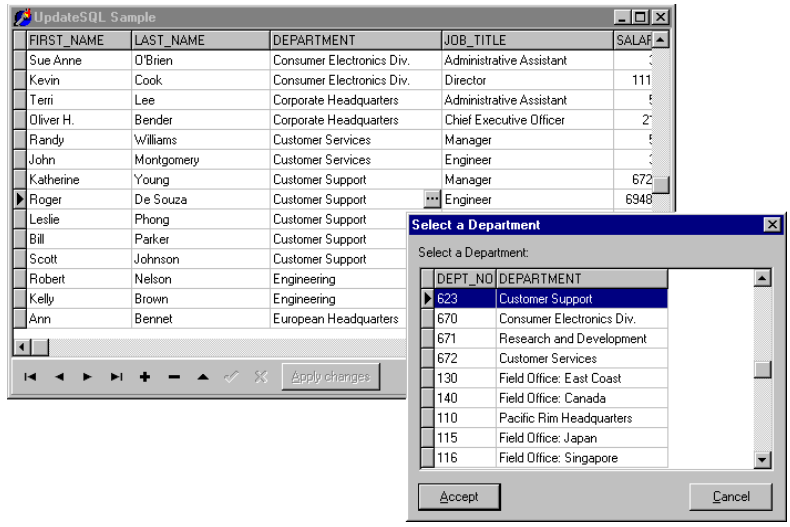


Figure 11.8: The main form of the UpdateSql example along with a secondary form. Image from the original book.

The second feature is the use of a secondary form to update the fields that are related to other tables—the fields involved in the joins. The program uses two secondary dialog boxes, which get the data from other two Query components. The dialog boxes are displayed when the user clicks on the ellipsis button of the DBGrid control, in the `OnEditButtonClick` event. Here is the first part of this event handler, related to the selection of the department:

```
procedure TMainForm.DBGrid1EditButtonClick(Sender: TObject);
begin
  // check if this is the department field
  if DBGrid1.SelectedField = dmData.qryEmployeeDEPARTMENT then
    with TfrmDepartments.Create(self) do
    try
      dmData.qryDepartment.Locate('DEPT_NO',
        dmData.qryEmployeeDEPT_NO.Value, []);
      if ShowModal = mrOk then
        with dmData do
        begin
          if not (qryEmployee.State in [dsEdit, dsInsert]) then
            qryEmployee.Edit;
          qryEmployeeDEPT_NO.Value :=
            qryDepartment.Fields[0].Value;
          qryEmployeeDEPARTMENT.Value :=
            qryDepartment.Fields[1].Value;
        end;
    finally
      Free;
    end
```

```
  else // similar code for the job fields...
```

Finally, the Apply button simply calls the `ApplyUpdates` method if there are pending updates and then refreshes the data of the query:

```
procedure TMainForm.btnApplyClick(Sender: TObject);
begin
  with dmData do
    if qryEmployee.UpdatesPending then
    begin
      AppDB.ApplyUpdates([qryEmployee]);
      // refresh the data
      qryEmployee.Close;
      qryEmployee.Open;
      btnApply.Enabled := False;
    end;
end;
```

If you run this program, you'll notice that even if the underlying query is read-only, you can change data directly in the DBGrid, as you would do with a regular Table component. The visual operations you do are temporarily stored in the cache; then, when you issue the update operation, the UpdateSQL and the StoredProc components provide the actual code. Also keep in mind that the salary field has some constraints (defined in the sample database), so you have to change it carefully to avoid errors on the server when the changes are applied.

Update Conflicts

When you are working with local tables, using cached updates might cause concurrency problems. A plain edit operation usually places a lock on the table, so that the other users cannot modify the same record until the first user has posted the updates. The previous chapter covered locking and concurrency issues in detail.

When working with SQL servers, however, the default locking behavior is optimistic. Multiple users can update the same records, and only when the data is sent back does the server verify the original data of the record before updating it, potentially raising an error. More precisely, the `update` statement uses one or more original fields to locate the record you want to update. If you use all fields and another user has changed the record, then the server will not find the original record and will cause an update error.

You can manually control this behavior either in the code of the UpdateSQL component (indicating to include all the fields read in the query) or by using the `UpdateMode` property of the Table and Query components. The default value,

`upWhereAll`, indicates that the update query will have a `where` clause with all the original fields of the record. In many cases, the fact that another user has modified a field different from those we have modified is not an error. We can set the `upWhereChanged` mode to let Delphi generate an exception and show an error message only if the current and the other user have both modified the same fields. The third alternative is to use the key field only to identify the record, which means that update conflicts will be ignored and that the last user posting the data will simply override any previous change. As you can imagine, this is generally an option to avoid in a client/server, multi-user environment.

Using Transactions

Another topic related to updates in a client/server environment is the use of transactions. In the last chapter we introduced the concept of transactions (multiple updates considered as a single atomic operation), but there are further details specific to working with SQL servers.

In Delphi we use the Database component to handle transactions and set the transaction isolation level using the `TransIsolation` property. When one user starts a transaction and modifies data, should such changes be visible to other users? And what happens if the user rolls back the transaction? To such questions there isn't a universal answer; every programmer should try to answer them according to the requirements or business rules of the application. There are three alternative values for transaction isolation in the BDE:

- `tiDirtyRead` makes the updates of a transaction immediately visible to other transactions and users even before they are committed. This is the only possibility for local databases, which have very limited transaction support.
- `tiReadCommitted` makes available to other transactions only the updates already committed.
- `tiRepeatable Read` hides every other transaction started by other users after the current one. Following repeat calls within a transaction will always produce the same result, as if the database took a snapshot of the data when the current transaction started.

Most but not all SQL servers support only the most advanced levels. The default choice should be `tiReadCommitted`, which is quite powerful but not too heavy on the SQL server (as it adds very few internal locks).

As a general suggestion, transactions should involve only a minimal number of updates (only those strictly related and part of a single atomic operation) and

should be kept short in time. You should avoid transactions that wait for user input to complete them, as the user might be temporarily gone and the transaction might remain active for a long time. Using `update` statements on multiple records and using cached updates can help us make the transactions small and fast.

To further inspect transactions and experiment with the update mode of the Table component, you can use the TranSample application. As you can see in Figure 11.9, you can simply use the radio buttons to choose the different alternatives, and click the push buttons on the right of the toolbar to manually start, commit, and rollback a transaction. To get a real idea of the different effects, you should run multiple copies of the program (provided you have enough licenses on your InterBase server).

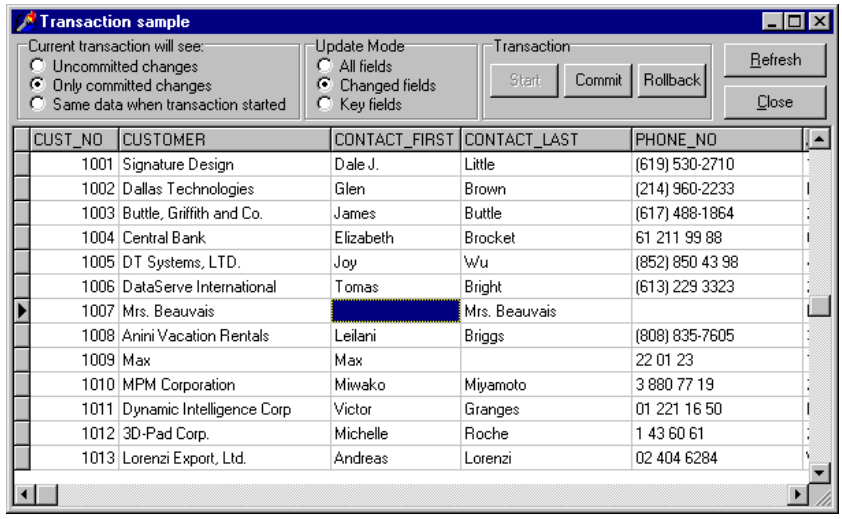


Figure 11.9: The TranSample application allows you to test the transaction isolation of a database and the update modes of a table. Image from the original book.

InterBase Express

All the examples built up to now have used the BDE to reach the InterBase server. As mentioned at the beginning of this chapter, Delphi 5 includes new components specifically designed to access an InterBase database. This technology is called

InterBase Express (or IBX for short)[314]. Borland's promise is that applications using these components work better and faster, giving you more control over the specific features of InterBase. I have no reason to doubt this. The problem is that an application built with this approach is inherently not portable to a different SQL server.

The IBX components include custom dataset components. These inherit from the base `TDataSet` class, can use all the common Delphi data-aware controls, provide a field editor and all the usual design-time features, and can be used in the new Data Module Designer, but they don't require the BDE. You can actually choose among multiple dataset components:

- IBTable resembles the Table component and allows you to access a single table or view.
- IBQuery resembles the Query component and allows you to execute a SQL query, returning a result set. The IBQuery component can be used together with the IBUpdateSQL component to obtain a live (or editable) dataset.
- IBStoredProc resembles the StoredProc component and allows you to execute a stored procedure.
- IBDataSet allows you to work with a live result set obtained by a executing a `select` query. It basically merges IBQuery with IBUpdateSQL in a single component.

Many other components in InterBase Express don't belong to the dataset category:

- IBDatabase mimics the standard Database component.
- IBTransaction allows complete control over transactions.
- IBSQL lets you execute SQL statements without the overhead of a dataset control.
- IBDatabaseInfo is used for querying the database structure and status.
- IBSQLMonitor is used for debugging the system.
- IBEvents receives events posted by the server.

This large group of components provides greater control over the database server than you can have with the BDE. For example, having a specific transaction component allows you to manage multiple concurrent transactions over one or multiple databases, as well as a single transaction spanning multiple databases.

---

314 IBX is still available in Delphi today, although the general recommendation is to use FireDAC for it's additional features and flexibility.

Up and Running

As a first simple example, I've taken the IbEmpl program discussed earlier and recreated it using the minimum InterBase Express components required. After replacing the Query component with an IBQuery, I had to add two more components: IBTransaction and IBDatabase. Any IBX application requires at least an instance of each of these two components. You cannot set database connections in a dataset (as you can do with a plain Query), and at least a transaction object is required to open a query.

Here are the key properties of these components in the IbEmpl2 example:

```
object IBTransaction1: TIBTransaction
  Active = False
  DefaultDatabase = IBDatabase1
end
object IBQuery1: TIBQuery
  Database = IBDatabase1
  Transaction = IBTransaction1
  CachedUpdates = False
  SQL.Strings = (
    'SELECT * FROM EMPLOYEE')
end
object IBDatabase1: TIBDatabase
  DatabaseName = 'C:\Program Files\Common Files\
    Borland Shared\Data\employee.gdb'
  Params.Strings = (
    'user_name=SYSDBA'
    'password=masterkey')
  LoginPrompt = False
  IdleTimer = 0
  SQLDialect = 1
  TraceFlags = []
end
```

The changes don't take too much time to perform, and if you are accessing the same database table as in the BDE-based program you won't need to change the data-aware components at all, but only hook the DataSource component to `IBQuery1`. Because I'm not using the BDE, I had to type in the pathname of the InterBase database. However, not everyone in the world has the Program Files folder, which depends on the local version of Windows, and of course the Borland sample data files could have been installed in any other location of the disk. We'll solve these problems in the next example.

note Notice that I've embedded the password in the code, a very naïve approach to security. Not only can anyone run the program, but someone could even extract the password by looking at the hexadecimal code of the executable file. I used this approach so I wouldn't need to keep typing in my password while testing a program, but in a real application you should require your users to do so if they care about the security of their data.

Building a Live Query

The IbEmpl2 example has a query that doesn't allow editing. To activate editing you need to use an IBTable component or add to the query an IBUpdateSQL component, even if the query is very simple. Usually the BDE does the behind-the-scenes work that lets you edit the result set of a simple query, but we are not using the BDE now.

The relationship between the IBQuery and IBUpdateSQL components is the same as between the Query and UpdateSQL components. To highlight this, I've taken the main form of the UpdateSql example discussed earlier in this chapter and ported it to the InterBase Express components, building the UpdSql2 example. I've simply copied the two components from the original example, pasted them into an editor, changed the type of the object, and copied the resulting text into a new form. The properties are so similar that I had only to ignore a couple of missing ones (the `DatabaseName` and the `UpdateMode` properties).

At this point I simply added an IBDatabase and an IBTransaction component, a data source and a grid, and my program was up and running. The key element of these components, in fact, is their SQL code, which is attached to the `SQL` property of the query and the `ModifySQL`, `DeleteSQL`, and `InsertSQL` properties of the update component.

However, this time I've made the reference to the database a little more flexible. Instead of typing in the database name at design time, I've extracted it from the Windows Registry (where Borland saves it while installing the programs). This is the code executed when the program starts:

```
uses
  Registry;

procedure TForm1.FormCreate(Sender: TObject);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  try
    Reg.RootKey := HKEY_LOCAL_MACHINE;
```

```
    Reg.OpenKey(
      '\Software\Borland\Borland Shared\Data', False);
    IBDatabase1.DatabaseName :=
      Reg.ReadString('Rootdir') + '\employee.gdb';
  finally
    Reg.CloseKey;
    Reg.Free;
  end;
  EmpDS.DataSet.Open;
end;
```

This is actually a nice example of the use of the `TRegistry` class of the VCL, a topic I'll cover again briefly in Chapter 19.

The new feature of this example, compared to the last version, is the presence of a transaction component. As I've already said, the InterBase Express components make the use of a transaction component compulsory. Simply adding a couple of buttons to the form to commit or roll back the transaction would be enough, because a transaction starts automatically as you edit any dataset attached to it.

I've also improved the program a little by adding an ActionList component to it. This includes all the standard database actions and adds two custom actions for transaction support, Commit and Rollback. Both actions are enabled when the transaction is active:

```
procedure TForm1.ActionUpdateTransactions(Sender: TObject);
begin
  acCommit.Enabled := IBTransaction1.InTransaction;
  acRollback.Enabled := acCommit.Enabled;
end;
```

When executed, they perform the main operation but also need to reopen the dataset in a new transaction (which can also be done by "*retaining*" the transaction context):

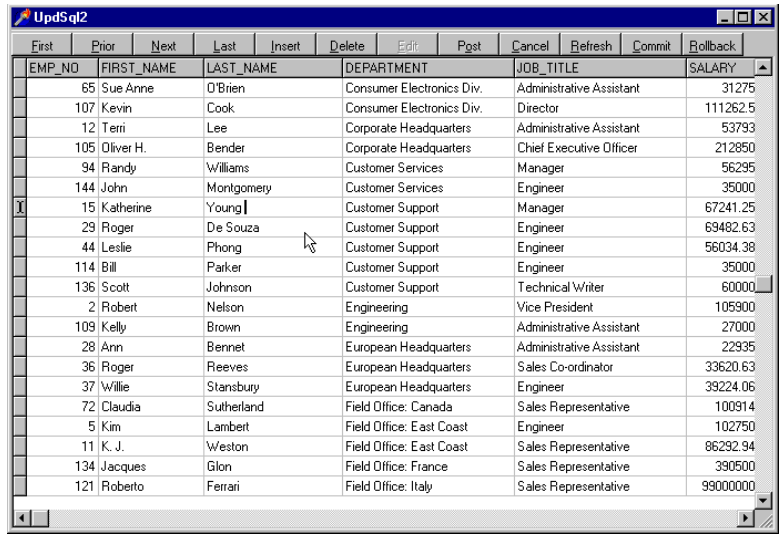
```
procedure TForm1.acCommitExecute(Sender: TObject);
begin
  IBTransaction1.CommitRetaining;
end;

procedure TForm1.acRollbackExecute(Sender: TObject);
begin
  IBTransaction1.Rollback;
  // reopen the dataset in a new transaction
  IBTransaction1.StartTransaction;
  EmpDS.DataSet.Open;
end;
```

note Be aware that InterBase closes any opened cursors when a transaction ends, which means you have to reopen them and re-fetch the data even if you haven't made any changes. When committing data, instead, you can ask InterBase to retain the "transaction context"—not to close open data sets—by issuing a `CommitRetaining` command. With the forthcoming version 6.0 of InterBase, you will also be able to issue a `RollbackRetaining` command. The reason for this behavior depends on the fact that a transaction corresponds to a snapshot of the data. Once a transaction is finished, you are supposed to read the data again to refetch records that may have been modified by other users.

The last operation refers to a generic dataset and not a specific one because I'm going to add a second alternative dataset to the program. The actions are connected to a text-only toolbar, as you can see in Figure 11.10. The program opens the data set at startup and automatically closes the current transaction on exit, after asking the user what to do, with the following `OnClose` event handler:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
  nCode: Integer;
begin
  if IBTransaction1.InTransaction then
  begin
    nCode := MessageDlg ('Commit Transaction? (No to rollback)',
      mtConfirmation, mbYesNoCancel, 0);
    case nCode of
      mrYes: IBTransaction1.Commit;
      mrNo: IBTransaction1.Rollback;
      mrCancel: Action := caNone; // don't close
    end;
  end;
end;
```



EMP_NO	FIRST_NAME	LAST_NAME	DEPARTMENT	JOB_TITLE	SALARY
65	Sue Anne	O'Brien	Consumer Electronics Div.	Administrative Assistant	31275
107	Kevin	Cook	Consumer Electronics Div.	Director	111262.5
12	Terri	Lee	Corporate Headquarters	Administrative Assistant	53793
105	Oliver H.	Bender	Corporate Headquarters	Chief Executive Officer	212850
94	Randy	Williams	Customer Services	Manager	56295
144	John	Montgomery	Customer Services	Engineer	35000
15	Katherine	Young	Customer Support	Manager	67241.25
29	Roger	De Souza	Customer Support	Engineer	69482.63
44	Leslie	Phong	Customer Support	Engineer	56034.38
114	Bill	Parker	Customer Support	Engineer	35000
136	Scott	Johnson	Customer Support	Technical Writer	60000
2	Robert	Nelson	Engineering	Vice President	105900
109	Kelly	Brown	Engineering	Administrative Assistant	27000
28	Ann	Bennet	European Headquarters	Administrative Assistant	22935
36	Roger	Reeves	European Headquarters	Sales Co-ordinator	33620.63
37	Willie	Stansbury	European Headquarters	Engineer	39224.06
72	Claudia	Sutherland	Field Office: Canada	Sales Representative	100914
5	Kim	Lambert	Field Office: East Coast	Engineer	102750
11	K. J.	Weston	Field Office: East Coast	Sales Representative	86292.94
134	Jacques	Glon	Field Office: France	Sales Representative	390500
121	Roberto	Ferrari	Field Office: Italy	Sales Representative	99000000

Figure 11.10: The output of the UpdSql2 example. Image from the original book.

An alternative to using the IBQuery and IBUpdateSQL components is to use the IBDataSet component, which combines the two. An InterBase dataset, in fact, is a live query with a complete set of SQL statements for all the main operations. The differences between using the two components and the single one are minimal. Using IBQuery and IBUpdateSQL is probably better when porting an existing application based on the two equivalent BDE components, even if porting the program directly to the IBDataSet component doesn't really require a lot of extra work.

In the UpdSql2 example I've provided both alternatives, so that you can test the differences yourself. Here is part of the DFM description of the dataset component:

```
object IBDataSet1: TIBDataSet
  Database = IBDatabase1
  Transaction = IBTransaction1
  CachedUpdates = False
  BufferChunks = 32
  DeleteSQL.Strings = (
    'delete from EMPLOYEE
    'where'
    '  EMP_NO = :OLD_EMP_NO')
  InsertSQL.Strings = (
    'insert into EMPLOYEE'
    '  (FIRST_NAME, LAST_NAME, SALARY, DEPT_NO, JOB_CODE, JOB_GRADE, '
    '  JOB_COUNTRY)'
    'values'
    '  (:FIRST_NAME, :LAST_NAME, :SALARY, :DEPT_NO, :JOB_CODE, ' +
    '  :JOB_GRADE, :JOB_COUNTRY)')
  SelectSQL.Strings = (...)
```

```
  UpdateRecordTypes = [cusUnmodified, cusModified, cusInserted]
  ModifySQL.Strings = (...)
end
```

If you connect the `IBQuery1` or the `IBDataSet1` components to the data source and run the program, you'll see that the behavior is identical. Not only do the components have a similar effect; the available properties and events are also very similar.

Client/Server Optimization

Just as you need a debugger to test a Delphi application (a topic discussed in Chapter 18), you need some tools to test how a client/server application behaves and to speed it up if possible. In particular, it is very important to look at the information moving from the client to the server (the explicit SQL requests our program does and those added by the BDE) and from the server to the client (the actual data). This is what the SQL Monitor tool included in Delphi Enterprise is for.

Using SQL Monitor

As you can see in Figure 11.11, the central window of SQL Monitor[315] shows a list of the low-level commands sent to the server. The bottom portion of the window shows the selected line of the above list on multiple rows, which helps when the line is too long.

315 The SQLMonitor was specifically tied to the BDE and no longer exists. There is now a FireDAC Monitor available in the Delphi IDE Tools menu.

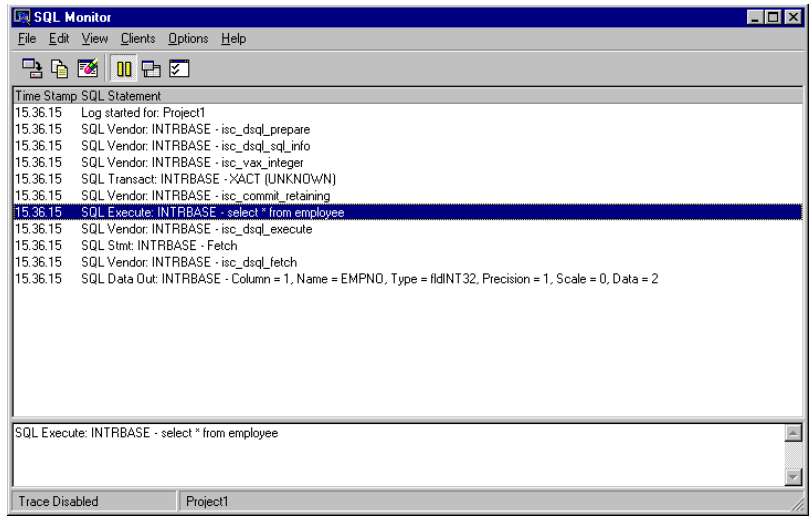


Figure 11.11: The SQL Monitor running. Image from the original book.

To use SQL Monitor, simply select the client program you want to inspect. Then set the proper trace options (by using the corresponding speed button or the Options ➢ Trace Options command). The available options are listed in Table 11.2.

Table 11.2: The Trace Options of the SQL Monitor

Trace Option	Meaning
Prepared Query Statement	Enables tracing of the SQL statements every time they are prepared.
Executed Query Statement	Traces all the SQL statements sent to the server.
Input Parameters	Shows input parameters as they become available. This is important for testing whether the parameters are correct.
Fetched Data	Shows the data sent by the server (a very slow operation).
Statement Operations	Shows the requests preceding the execution of a SQL statement, such as the allocation, preparation, and parsing of the input.
Connect/Disconnect	Shows the connection and disconnection events. This is an important test when the `KeepConnection` of the `Database` component is set to `False`, as the client won't maintain the connection with the server but will establish it only as needed (with the side effect of reducing the number of licenses required, in some cases). Looking at the frequency of these events might help you understand if it is better to keep the connection active or not.
Transactions	Traces the transactions, including those activated automatically by the BDE if you don't use transactions directly.
Blob I/O	Shows the data about BLOB fields.
Miscellaneous	Traces other operations that don't fit any of the above categories.

Vendor Errors	Shows server error messages.
Vendor Calls	Shows client API calls.

SQL Monitor is useful for seeing if the SQL statements sent by the BDE to the server are correct, but it also helps you see how many operations are done behind the scenes. Along with the time-stamp information for each operation, the number of operations can give some clue about your application's speed (although you should remember that the presence of SQL Monitor slows down the connection quite a lot).

In other words, SQL Monitor should be your guide in determining how to speed up your client/server application, using some of the tricks described in this chapter. At the same time, however, it takes a lot of experience and a good understanding of SQL to interpret its output properly.

As an example of the use of SQL Monitor, we can test what happens when we use the `Filter` property of a Table component. In a new project, simply place a Table, a DataSource, and a DBGrid. Select a database and a table (for example, the Employee table of IBLocal) and set the `Filtered` property to `True` and the `Filter` property to `EmpNo>20`. If you now run the program, SQL Monitor will show you that the `select` statement generated by the BDE has a `where` clause corresponding to the filter. You can see this situation in Figure 11.12.

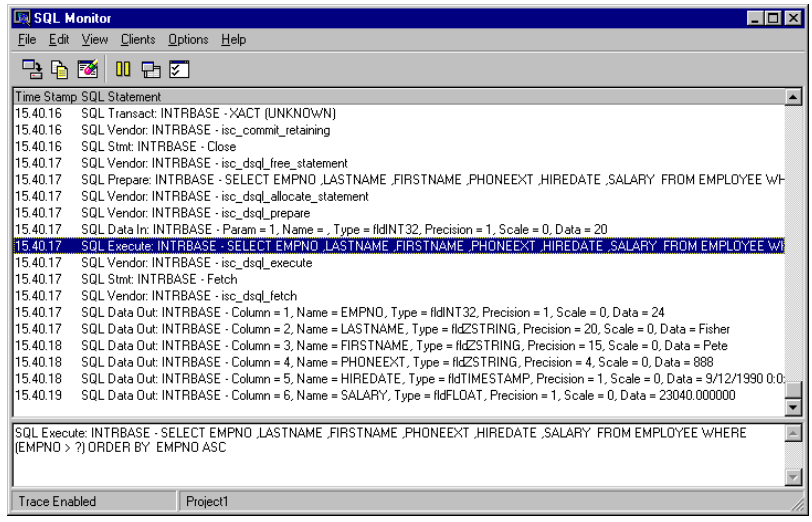


Figure 11.12: SQL Monitor showing SQL statements generated by a Table component. Image from the original book.

Monitoring InterBase Express

SQL Monitor works by using a hook into the BDE architecture. For this reason, you cannot use it with applications based on the InterBase Express components. Instead, however, you can simply embed in your application a copy of the IBSQL-Monitor component and produce a custom log.

You can even write a more generic monitoring application, as I've done in the Ibx-Mon example. I've placed in its form a monitoring and a RichEdit control, and written the following handler for the `OnSQL` event:

```
procedure TForm1.IBSQLMonitor1SQL(EventText: String);
begin
  if Assigned (RichEdit1) then
    RichEdit1.Lines.Add (TimeToStr (Now) + ': ' + EventText);
end;
```

The `if Assigned` test can be useful when receiving a message during shutdown, and it is required when you add this code directly inside the application you are monitoring.

To receive the messages from other applications (or from the current one), you have to turn on the tracing options of the IBDatabase component. In the UpdSql2 example (discussed earlier, in the section "Building a Live Query") I turned them all on:

```
object IBDatabase1: TIBDatabase
  ...
  TraceFlags = [tfQPrepare, tfQExecute, tfQFetch, tfError, tfStmt,
    tfConnect, tfTransact, tfBlob, tfService, tfMisc]
```

If you run the two examples at the same time, the output of the IbxMon program will list the details about the UpdSql2 program's interaction with InterBase, as you can see in Figure 11.13.

Figure 11.13: The output of the IbxMon example, based on the IBMonitor component. Image from the original book.

Performance Tuning

Besides using the SQL Monitor to determine the potential bottlenecks in your applications, there are several things you can do to speed up your client/server programs. The key element to keep in mind—as I've stressed many times in this chapter—is to reduce the network traffic, by reducing the result sets returned by the server both in the number of records and in the size of each.

Besides a good overall database design and a good Delphi implementation of it, there are many settings you can check. The following tips might come in handy, but they won't help as much as a better design!

- In InterBase you can set an automatic sweep (or "garbage collection") interval. The operation is also automatically performed when you do a backup. Because a sweep slows down the database, it should not be done too frequently. However, if you never do it, the database will keep track of many leftover deleted records, reducing the overall performance and using extra memory.
- Use indexes on the fields used more often, particularly if you sort the result set on them. Keep in mind, though, that a good RDBMS will add at least temporary

indexes for you. Using indexes can speed up queries quite a lot, particularly if the indexed fields are used to join two tables.

- If you sort a field in descending order, a corresponding descending index might help.
- If you're an expert user, you might examine the *query plan*, the approach used by the server to perform a query, which is displayed (for example) when you use WISQL. The query plan will show you whether the SQL server is using indexes. In some cases, you might need to modify some complex queries to help the query optimizer built into the RDBMS.
- Check the server settings, including its cache, to obtain the best overall performance. The operating system cache on the server computer might help as well. In InterBase, if you want to perform all the updates physically, you can set the Forced Writes option in the Maintenance ➢ Database Properties menu of the InterBase Server Manager.
- Whenever possible, avoid an excessive use of transactions and try to keep them short and focused. Use cached updates instead of transactions (or together with them) to let the client computer do some more work for you, and skip some costly server operations.
- Handle transactions directly, disabling the auto-commit feature of the BDE; to do this, set `SQLPASSTHRU MODE` to `SHARED NOAUTOCOMMIT`. (You can set this and other BDE features described in this list with the BDE Administrator program.)
- If you have no licensing problems, set the `KeepConnection` property of the Database component to `True`.
- Set `TRACE MODE` to `0` when you are not debugging, to avoid having the drivers send trace strings to the debugger and slowing down the operations. When working with InterBase Express components, call `DisableMonitoring` in your initialization code to disable tracing.
- Enable schema caching (set `ENABLE SCHEMA CACHE` to `TRUE`). This setting reduces the time required to open a table, as the client doesn't need to ask for the metadata. You can also use the Delphi `FieldDefs` and `StoreDefs` properties of the Table component to store the metadata directly in the client program.
- With Microsoft and Sybase SQL Servers, try to set the `PACKETSIZE` parameter to a minimum of 4K, also modifying the corresponding value on the server. With these servers, also check that the `DRIVER FLAGS` parameter is set to `0`. If it is `2048`, queries will be executed in asynchronous mode and will be much slower.

- With ORACLE, DB/2, and ODBC drivers, try to fine-tune the `ROWSET SIZE` parameter until you obtain the best performance.
- With the InterBase driver, if you don't use explicit transactions, set the `DRIVER FLAGS` parameter to `4096`. This value enables *soft commits*, meaning that after each commit or rollback operation the open cursors won't have to be refreshed.

What's Next?

This chapter has presented an introduction to client/server programming with Delphi. We saw what the key issues are, looked at the most important features of the SQL language, and delved a little into some interesting areas of client/server programming. A complete discussion of client/server in Delphi would probably require an entire separate book.

The same can be said of ADO programming, which is only introduced in the next chapter. ADO is the interface to the database engine Microsoft is promoting (called OLE DB). Because we're working on the Windows platform, we should know at least a little about it no matter what database we are using.

Chapter 12: Using ADO

When Delphi was originally designed, Borland owned Paradox and Visual dBase and, therefore, built a single data-access engine shared by all of its products. This engine is the BDE I've discussed in the last three chapters[316]. The BDE was designed to connect to many SQL servers, including Oracle and InterBase, and to access other data source via ODBC, the first Microsoft data-access interface. ODBC, however, was originally very slow, and some of the drivers for specific database formats were not bug free.

Over time, however, three things happened. First, Borland sold its end-user database products. Second, Microsoft's role as a database provider grew larger, with both MS Access and MS SQL Server gaining more and more acceptance in the industry. Whether this happened for technical reasons or because Microsoft has better marketing is not important. As a Delphi programmer, you cannot always choose the technologies you are going to work with, and the odds are your programs will have to work with Microsoft databases more frequently. Third, Microsoft

316 As people say, this is now history. None of the programs mentioned here have had an actual role for many years now.

improved its data-access strategy, introducing DAO, RDO, OLE DB, and ADO[317]. We'll see what these acronyms stand for in the next section.

Borland has acknowledged these changes over the last several years by adding specific BDE drivers for MS Access. Using the BDE, it is possible to connect to an Access database, but this implies using two database engines at once, when an Access connection through ADO is a more direct link. There wasn't much more Borland could do, however, because Microsoft hasn't been very open in documenting how to use its database engines, particularly the JET Engine used by Visual Basic programmers to connect with Access databases. Furthermore, JET Engine doesn't manage data in a consistent way as the BDE does, making it hard to have both engines cooperate smoothly.

Microsoft's ActiveX Data Objects (ADO), the new high-level data interface[318], change this perspective again, and Delphi 5 now supports this technology *directly* using specific DataSet components. These components don't use the BDE and don't require its installation on the computers of your clients. Instead, these components require the existence of Microsoft's ADO and OLE DB database engines on your clients' machines. Because ADO and OLE DB are installed by default with Windows 2000 and Windows 98, the headaches of installing a database engine with your application will probably decrease in the future as these platforms gradually replace the older operating systems. However, Windows 95 and Windows NT 4 don't necessarily have ADO installed, so you will probably have to distribute and install the ADO engine with your application or require the end user to install ADO before installing your application, in order for your application to run on these older platforms[319]. Even when ADO is already installed on the system, your installation process will probably need to configure ADO for your application's needs. Using ADO components isn't so much a matter of shedding BDE installation and configu-

---

317 There is now a new "generation" of Microsoft data access technologies, under the name of ADO.NET. As the name implies, this is a new architecture part of the .NET Framework. However, ADO.NET is not really part of the core operating system like previous data access technologies from Microsoft, which is why the classic ADO and low-level ODBC have remained relevant up and are in use today.

318 It was new at the time, 25 years ago. However, ADO is still a supported technology and Delphi still has native components for it.

319 While most computers have ADO installed these days, it might be a different version than you are targeting, making this installation steps still a worry. A download link is listed in the next footnote. Notice that, by contrast, while BDE and ADO required the engine installation, FireDAC code is 100% Delphi and all of the required code can be compiled into your executable program, with the only external dependency being the database client library – a requirement in any case.

ration chores as exchanging them with ADO installation and configuration chores. For further information, check out www.microsoft.com/data[320].

Before we discuss Delphi 5 ADO components, however, let me recap the key concepts hidden behind Microsoft data technologies.

Microsoft's Way to the Data

Please bear with me if this section sounds more like a Microsoft marketing pitch full of acronyms than a technical description, but when building programs for Windows, we have to live in the Microsoft world.

Microsoft's strategy for providing access to any kind of data is called *Universal Data Access*. The idea is to have a single interface that lets programmers use their preferred tools to access relational databases and other less structured data sources (such as mail messages or spreadsheets).

If Universal Data Access is the idea, in practice this strategy is accomplished by installing the *Microsoft Data Access Components* (MDAC) on a Win32 system. Windows 2000 will have MDAC as part of the system, but the components can also be downloaded from Microsoft's Web site, www.microsoft.com[321].

note Delphi 5's CD includes an installation of MDAC you have to install to be able to use the ADO components, unless you have the ADO support already installed on your system[322].

Now, to make things more confusing, MDAC includes *ActiveX Data Objects* (ADO), OLE DB, and Open Database Connectivity (ODBC) support.

320 The URL now redirects to Microsoft SQL Server. The download for ADO is at https://www.microsoft.com/en-us/download/details.aspx?id=21995 and I recommend searching for "MDAC" as the term "ADO" is now used for one of the Azure features and the old meaning is almost ignored.

321 A good download link was in the previous footnote. As mentioned, MDAC is the terms you should use to look for information.

322 Needless to say the CD is long gone, but also need the to install MDAC, which is generally already installed on most Windows computers.

ADO and OLE DB

ADO provides the actual API that programmers have to target to build solutions according to Microsoft's strategy. ADO is designed with the goal of being the only data interface needed for any programming tasks. I really doubt that all of the Delphi developers will move to ADO, but it is certainly a technology to consider[323].

If I were a Microsoft spokesperson, I'd say that "the primary benefits of ADO are ease of use, high speed, low-memory overhead, a small disk footprint, minimal network traffic in key scenarios, and a minimal number of layers between the front end and data store—all to provide a lightweight, high-performance interface." This is quoted from the Microsoft literature on ADO. Technically, ADO provides an alternative to the BDE using a COM interface, although when using Delphi 5 ADO components, you don't need to understand COM.

While ADO provides an interface, the underlying data access is performed by another layer, called OLE DB. This is a system-level programming interface and can be considered as the successor to ODBC, which is still available as an alternative way to access data. OLE DB extends ODBC by providing access to non-relational data sources, including mainframe and hierarchical databases as well as e-mail and file system stores.

You may wonder why Microsoft has introduced the ADO layer instead of letting programmers access OLE DB. The primary reason is the complexity of OLE DB. ADO encapsulates over sixty OLE DB interfaces, which are not straightforward to relate, into about 20 simple objects. When you use ADO, you don't have to worry about interfaces, memory allocations, reference counting, or class factories (all of which are issues with OLE DB).

An alternative answer is that ADO is targeted for high-level programming tools (such as Visual Basic) that aren't able to deal with COM Interfaces via lower-level VTables. Instead, they created some ActiveX Objects that wrap OLE DB functionality, exposing to the programmers only the most important concepts in a simpler and easy-to-use way. With ADO, any tools supporting ActiveX technology (whether they are fully featured compilers or simple scripting languages) can actually join the UDA model. In fact, you can write ADO programs without the specific components Delphi 5 provides. However, the ready-to-use ADO dataset components allow you to program ADO using the same techniques with which you are familiar.

---

323 Using ADO was suggestion at the time, certainly not today. Microsoft hasn't added any extra feature, and compared to FireDAC the old ADO support is very limited. If you are using ADO, a migration to FireDAC is recommended and there are specific scripts and extra classes that can help you move your existing ADO code to FireDAC.

note Here is some more terminology: OLE DB may be looked at as a set of components consisting of *data providers,* which contain and expose data; *data consumers,* which use data; and *service components,* which process and transport data (such as query processors and cursor engines).

ADO Objects

ADO architecture is built around a few objects, for which Delphi provides wrapper components. Here is a short list of the key ADO objects:

- The *Connection* object offers the means of accessing the data source, using *connection strings* to locate data providers, manage the related session, and handle transactions.
- A *Command* object allows you to operate on a data source (managed by a Connection object), exposing a way to query, add, delete, or update the data. A Command may have one or more *Parameter* objects. Among the nonrelational databases that ADO supports, different data providers support different command strings (and not all of them support SQL commands).
- A *Recordset* object is the result of a Query command; that is, a cache of the records returned by the query. A Recordset allows navigation and editing of the data. Each row of a Recordset is made of multiple *Field* objects.

Other ADO objects include the Error object and the Errors collection, Property objects (which encompass the properties of the objects and dynamic properties attached to them at run time), and the Properties collection. Finally, there are events, which are notifications that an operation is about to occur or has been completed.

Reading this short description, you've probably realized that the ADO architecture is not very different from Delphi's own dataset architecture. Even with these similarities, there are indeed many differences between the ADO objects and the Delphi components. For this reason, even though it is possible to program these objects directly, Delphi 5 wraps the ADO objects with familiar components, providing an easy solution. Mapping the ADO concepts into VCL dataset concepts probably wasn't so simple, considering that the ADO DB unit of the VCL responsible for this has well over 5,000 lines of source code.

note Delphi 5 ADO components are included in Delphi Enterprise and can be bought separately by owners of Delphi Professional.

Delphi 5 ADO Components

Delphi ADO components[324] inherit from the `TDataSet` class. The key benefit of this is the integration with the Delphi IDE and the possibility of using all the default and third-party data-aware controls for displaying and editing the data.

There are several ADO DataSet components in Delphi 5. There are three components you can use for writing database applications:

- The `TADOConnection` component wraps the ADO connection object, providing connection strings, login, and transactions. The features of this component are similar to those of `TDatabase`.
- The `TADOCommand` component wraps the ADO Command object, providing a way to execute a query that doesn't return a dataset.
- The `TADODataSet` component wraps both the ADO Command and the ADO Recordset objects at once. As with any other dataset, you issue a command related to one table or querying multiple tables and receive as a result a set of records.

These components are all you need to use ADO in your Delphi applications. However, the way you use them is quite different from the traditional Delphi programming with `TTable` and `TQuery` components. The available properties are different, and so is the programming style. Because this would make it hard to port existing programs to ADO, Delphi 5 also includes some specialized ADO data-access components, with properties and features corresponding to standard DataSet components. The three components are `TADOTable`, `TADOQuery`, and `TADOStoredProc`.

---

note Beside the new dataset components, ADO support in Delphi 5 includes some new field types, as mentioned in Chapter 9. The specific field classes include `TWideStringField` for strings based on Unicode characters; `TGuidField` for convenient storage of Globally Unique Identifiers; `TVariantField` for fields based on the variant data type (that is, without a predefined type); `TInterfaceField` and the derived `TIDispatchField` host; a generic COM interface; and a COM `IDispatch` interface.

---

324 Delphi's ADO components have later been renamed "dbGo", because Microsoft claimed the terms ADO was proprietary and couldn't be used without permission. The actual component names haven't changed, though.

A Practical ADO Primer

Now that you know the key concepts and components you can use for ADO programming, it's time to look at code. I'll first build a very simple example, called AdoPrimer, which uses an ODBC connection to retrieve a dBase table of the DBDEMOS database. The program is based on a data module, to which I've added an ADOConnection component. Actually, the connection component is not required, because you can use the `ConnectionString` property of an ADODataSet component.

note Using a specific `ConnectionString` instead of linking the dataset to an ADOConnection means that the Recordset will create a new connection object. In this case, if you open multiple data sets from the same program you'll end up with multiple connections, wasting resources and, eventually, client access licenses. If you're going to retrieve more than one Recordset from an ADO source, you should always use a shared ADOConnection object.

Setting up the connection requires indicating how to access the data and where it is physically located. The `ConnectionString` property of the ADOConnection component (as that of the ADODataSet component) activates the specific editor you can see in Figure 12.1. This is also the default editor of the component. You have basically two choices: you can build a custom connection string, or you can store one in a Microsoft Data Link file (`.UDL`) and refer to the file in the connection string.

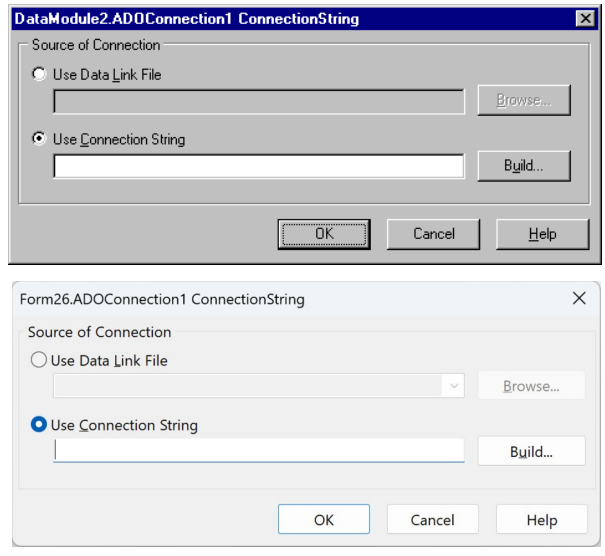


Figure 12.1: The editor of the ConnectionString property and of the ADOConnection component. Images captured in Delphi 5 and Delphi 12.

If you choose the Connection String radio button and press the Build button, you'll see the Data Link properties dialog box provided by Microsoft. This same dialog box is activated when you are editing a Data Link file. The Data Link dialog has multiple pages. In the first page (*Provider*), you select an OLE DB provider. Moving to the next page (*Connection*), you'll see different elements depending on your choice of provider. Figure 12.2 shows the available options for an ODBC provider.

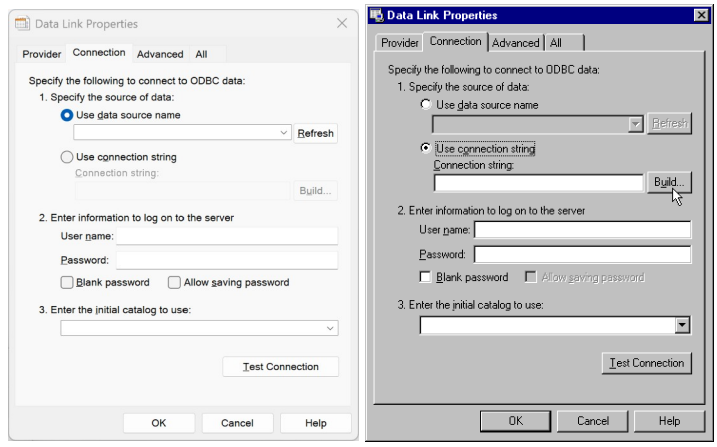


Figure 12.2: The Data Link properties dialog box provided by Microsoft. Images from the original book and captured in a modern version of Windows.

Again, you can select a type of data source (*Use data source name*) or indicate a specific database (*Use connection string*). In the second case, if you press the Build button, you'll be asked to select a type of data source first and a specific file or directory next. I've selected dBase and the Delphi demos data directory. Finally, you can modify network connections and access permissions in the third page (*Advanced*) and check whether everything looks fine in the final page (*All*). The settings I've used in the example produce the following string (reformatted to make it readable):

```
Provider=MSDASQL.1;
Persist Security Info=False;
Mode=Read|Write;
Connect Timeout=15;
Extended Properties="
  DSN=dBASE Files;
  DBQ=c:\PROGRAM FILES\COMMON FILES\BORLAND SHARED\DATA;
  DefaultDir=c:\PROGRAM FILES\COMMON FILES\
    BORLAND SHARED\DATA;
  DriverId=533;
  MaxBufferSize=2048;
  PageTimeout=5;";
Locale Identifier=1033
```

Once this is set, you can test the connection string by using the button in Connections page of the Data Link dialog box or by toggling the value of the `Connected` property of the component. I've also disabled the `LoginPrompt` property, because dBase has no password support.

After setting the ADOConnection component, we can now add to the data module an ADODataSet. This component has a `Connection` property you can use to refer to the ADOConnection, as an alternative to setting up a specific `ConnectionString`. Next, you have to specify the command you want to send to the database. The type of the command is indicated by the `CommandType` property, which indicated how to interpret the `CommandText` property, which has the actual command. For example, we can access the `CLIENTS.DB` table using the following values:

```
object ADODataSet: TADODataSet
  Active = True
  Connection = ADOConnection
  CommandText = 'clients'
  CommandType = cmdTable
end
```

You can also refer directly to a stored procedure (with the `cmdStoredProc` command type), but most of the time, you'll use the `cmdText` command type and write a SQL query in the command text. The same two `CommandText` and `CommandType` properties are also available for the ADOCommand component. As I mentioned earlier, an ADODataSet is a special type of ADO command returning a recordset.

If you plan to use a query, you can use the editor of the `CommandText` property, a very simple query builder as you can see in Figure 12.3. In this case, I've selected a table and all of its fields. Notice that this editor is displayed even if the command type is different than `cmdText`, which could possibly lead to some confusion.

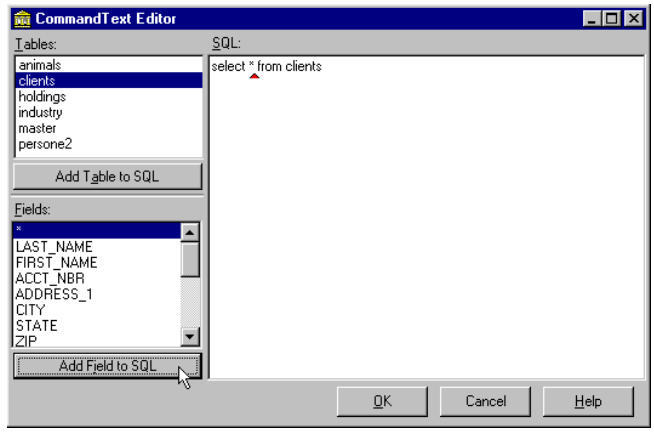


Figure 12.3: The editor of the CommandText property of the ADODataSet and ADOCommand components. Image from the original book.

At this point, we can simply add a data source component to the data module, obtaining the data diagram shown in Figure 12.4[325]. All these components are on the data module. The form is linked to it with a `uses` statement and includes a DBNavigator and a DBGrid, both connected with the data source on the data module.

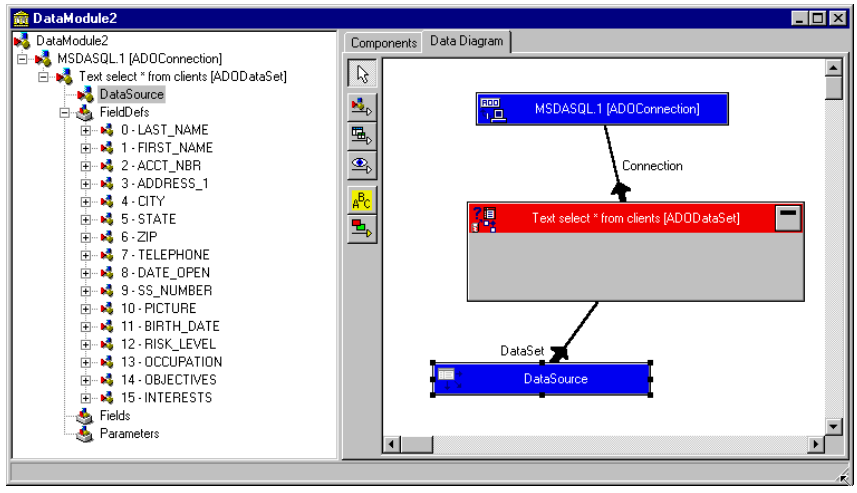


Figure 12.4: The data diagram for the AdoPrimer demo program. Image from the original book.

325 As already mentioned, the Data Diagram view doesn't exist any more.

From Paradox to Access

Before we delve into some of the more advanced features, we should look at an example built in the previous chapters and ported to ADO. The first example I've converted is DbAware, from Chapter 9, because this program generates a new table from scratch. Later on I'll build an example used to convert existing tables from DBDEMOS, and I'll port other programs built in the previous chapters.

note Delphi 5 demo database data actually includes an Access MDB database that is similar to the DBDEMOS database. Still, I'll show you the conversion process, because your programs will need to be ported by converting your own custom tables. It is also possible to use a Paradox OBDC driver to access to the DBDEMOS tables using ADO, but I felt using MS Access was a better choice for discussing ADO features. You shouldn't need to use ADO if you plan to use the Paradox database file format, because the BDE defines the standard for Paradox database file support.

In general, when you are porting existing applications to ADO, you might benefit from using the `TADOTable` and `TADOQuery` components. These components don't have exactly the same properties as the corresponding BDE components, but they do have some common elements.

The ADO table component has an intuitive `TableName` property, which is mapped to the `CommandText` property of the ADO dataset. It also provides support for a master-detail connection among tables (using the `MasterSource` and `MasterFields` properties).

The ADO query component has an intuitive `SQL` property, which is mapped again to the `CommandText` property, and a `DataSource` for building a master detail structure or otherwise getting values for the parameters from another dataset.

Both classes provide very little over the `TADODataset` component, which in turn adds very little to the `TADOCustomDataset` that all three components inherit from. This last class is the real workhorse for the ADO dataset facility provided in Delphi 5.

Using ADOTable

After this introduction, let's get back to the example. As I was saying, I've converted the DbAware example from Chapter 9 into the DbAware2 example. The only operation I had to do was to open the original form, open the DFM file, and change the line

```
object Table1: TTable
```

into

```
object Table1: TADOTable
```

As you save or compile the converted example, Delphi will prompt you to modify the type of the Table1 object in the Pascal file, a handy new feature that appears when you change the type of a component[326]. The Delphi request is visible in Figure 12.5.

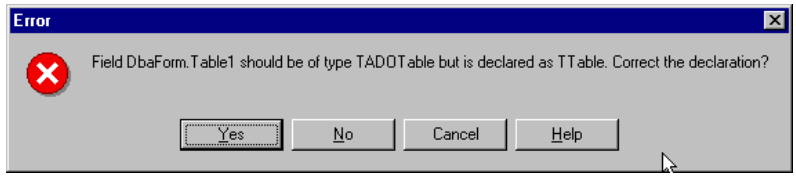


Figure 12.5: When you change the type of an object in a DFM file, Delphi 5 asks you to convert the form field in the Pascal code, as well. Image from the original book.

By converting the DFM file back into the actual form, you'll see that Delphi doesn't recognize the `Database` property. This is not a problem, because the ADO table uses the `Connection` property for this. I've added to the program a connection component pointing to the `MdData.mdb` file in the Data folder of the chapter examples. By the way, I've obtained this link though a UDL file you'll find in the same Data directory.

note All the link files and connection strings of the example programs throughout this book refer to their data using absolute path names. These path names reflect the default folder (directory) structure that is created when you "unpack" the self-extracting archive downloaded from the Sybex Web site[327]. If you move the programs to a different folder structure, you'll have to fix these links before you can run the examples. It is certainly possible to fix this problem by setting the connection at run time, but in this case, you won't be able to work with the live data at design time anyway.

The real problem is that you'll get another error for the field definitions. The ADO table component doesn't allow you to use field definitions to create a sample table, as in the original example. As a solution, we have to perform the table creation (over an existing database) using SQL code. As discussed in the DDL (Data Definition Language) section of the last chapter, you can use a SQL command to create a table.

326 This Delphi feature isn't available any more today. I don't know how we lost it. Today the IDE will suggest you to remove the incorrect field declaration, but it won't automatically convert it to the correct type.

327 When you download it from GitHub, you are going to have the code in a folder of your choice, so you'll have to update this absolute links.

I have added an ADOCommand component, which is hooked to the same connection, and have entered this text manually in the editor of the `CommandText` property:

```
create table workers (
  firstname TEXT(30),
  lastname TEXT(30),
  department INTEGER,
  branch TEXT (20),
  senior YESNO,
  hiredate DATETIME);
```

The data types listed here are those available in MS Access, not generic SQL types. In the following table, you can see a list of available MS Access data types:

Type	Description
TEXT(size)	A string up to 255 characters
MEMO or LONGTEXT	A string up to 1.2 gigabytes
BYTE	A number in the range 0 to 255
SHORT	A short integer (2 bytes)
LONG or INTEGER	An integer number (4 bytes)
COUNTER	An integer automatically incremented for each new record
SINGLE	A 4-byte floating point value
DOUBLE	An 8-byte floating point value
CURRENCY	An 8-byte numeric value with 4 decimal digits (as Delphi's own currency type)
GUID	A COM GUID (128 bits)
DATETIME	A double value holding a date and time
BIT or YESNO	Boolean value or one single bit
BINARY(size)	Binary data of a given size
LONGBINARY	Large binary data

After setting this command in the connection string, I've replaced the code of the `OnCreate` event handler of the form, with the following:

```
procedure TDbaForm.FormCreate(Sender: TObject);
var
  TablesList: TStringList;
begin
  // read table names form database
  TablesList := TStringList.Create;
  try
    ADOConnection.GetTableNames (TablesList);
    // check if the table already exists
    if TablesList.IndexOf (Table1.TableName) < 0 then
      // create it
```

```
      ADOCommand.Execute;
    // open the new or existing table
    Table1.Open;
  finally
    TablesList.Free;
  end;
end;
```

To see whether the table exists, this procedure uses the `GetTableNames` method of the connection component, retrieving all the tables and checking whether the table referenced by the ADOTable component is one of them. If not, then the command is executed, resulting in the creation of the table. Having done this, the converted example runs as smoothly as the original one did. You can see its output in Figure 12.6, although there is nothing special to see: all the differences, in fact, are behind the scenes.

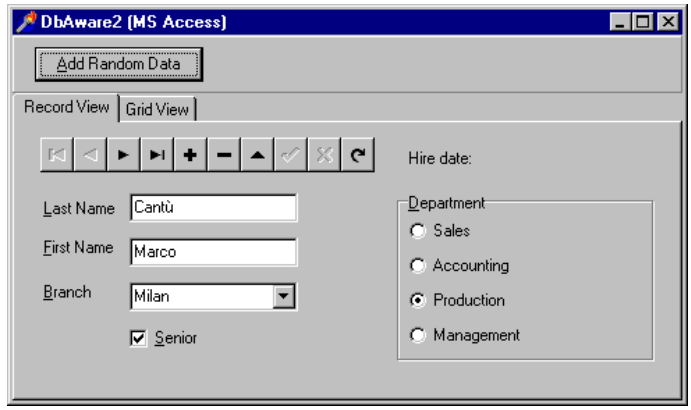


Figure 12.6: The DbAware2 example, based on an MS Access table created on the fly. Image from the original book.

Copying Tables

To be able to convert some of the other BDE-based examples to ADO, or at least recycle some of their source code, I decided to move more database tables from DBDEMOS to an Access database[328]. I'll show you the conversion process, instead of using the converted DBDEMOS database included in Delphi, because your pro-

328 While I'd discourage copying tables from Paradox to Access, some of the concept explained in this section still make sense. As already mentioned in previous chapters, FireDAC include a very powerful and flexible Batch Move architecture for mapping data sources and moving data, not only between databases, bu also to and from other formats.

grams will need to be ported by converting your own custom tables. This example provides some guidelines and ready-to-use code. The simple program I wrote to do that, Bde2Ado, isn't terribly interesting from a user perspective, but it allows me to discuss some more techniques or (at least) to generalize the techniques described in the last section.

The Bde2Ado example borrows its start-up code from the Tables example of Chapter 9. In fact, as it starts, it fills a combo box with the names of the available BDE aliases and a list box with the names of the tables of the selected database. The only difference is that now, in the `GetTablesNames` call, I filter only Paradox tables (second parameter) and don't show the extensions (third parameter):

```
Session.GetTableNames (ComboBox1.Text, '*.db',
  False, False, ListBox1.Items);
```

As you can see in Figure 12.7, the operations after selecting a table are performed in three manual steps: generating the Create Table SQL statement (the Get Structure button), executing the statement (the Create Table button), and copying the data (Move Data button). The first operation reads the field definitions from the Paradox table and generates the SQL code in the text of the memo. This way, if the code is not appropriate, you can fix it (as I mentioned, this is not supposed to be a program for end users but for developers).

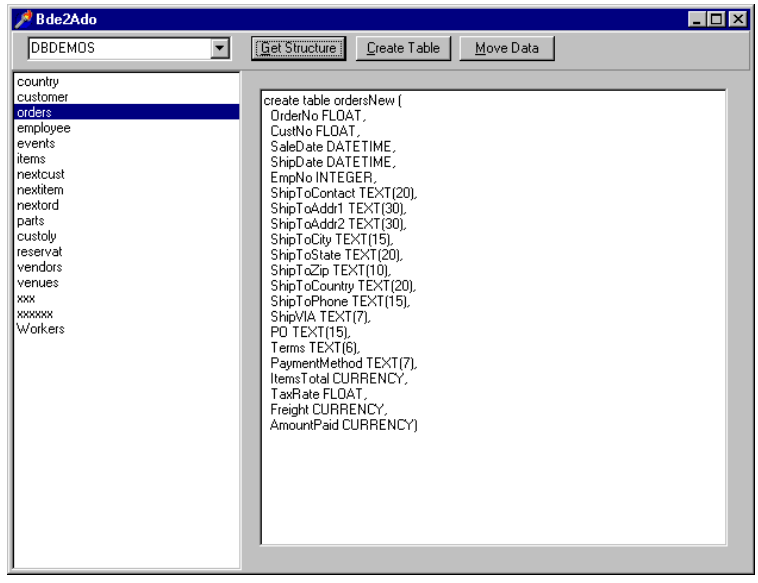


Figure 12.7: The Bde2Ado program, with a create SQL statement generated by pressing the first button. Image from the original book.

The event handler for the first button starts by determining the name of the table. The procedure uses the original name unless that table is already present, in which case it appends the *new* word to the name (one or more times until the table name is unique). For example if there is already a `MyTable` and a `MyTableNew`, it creates a `MyTableNewNew` table:

```
procedure TForm1.btnGetStructureClick(Sender: TObject);
begin
  ...
  // find a new table name
  AdoTable.TableName := (BdeTable.TableName);
  // check if the table already exists
  while TableExists (AdoTable.TableName) do
    AdoTable.TableName := AdoTable.TableName + 'New';
  Memo1.Lines.Add ('create table ' +
    AdoTable.TableName + ' (');
```

The `TableExists` method uses the `GetTableNames` method of the ADOConnection component we've already seen in the last example. Once it has the table name, the program generates the first line of the command, with the *'create table'* string. Next we have to add one line for each field, indicating the field name and its type (in this example, using MS Access data types). This is accomplished by scanning the field definitions:

```
  // btnGetStructureClick continued...
  // get field information
  BdeTable.FieldDefs.Update;
  for I := 0 to BdeTable.FieldDefs.Count - 1 do
  begin
    strField := '  ' +
      BdeTable.FieldDefs[I].Name + ' ' +
      AdoTypeName (BdeTable.FieldDefs[I]);
    // add comma or parenthesis
    if I < BdeTable.FieldDefs.Count - 1 then
      strField := strField + ','
    else
      strField := strField + ')';
    Memo1.Lines.Add (strField);
  end;
```

The actual work is done by the custom `AdoTypeName` function (a more accurate name might have been *AccessTypeName*). It checks the type of the field and returns a corresponding definition. Here is its initial portion:

```
function AdoTypeName (fdef: TFieldDef): string;
begin
  case fdef.DataType of
    ftString: Result := 'TEXT(' +
      IntToStr (fdef.Size) + ')';
```

```
    ftSmallint: Result := 'SMALLINT';
    ftInteger: Result := 'INTEGER';
    ftWord: Result := 'WORD';
    ftBoolean: Result := 'YESNO';
```

With this code, we can build a SQL statement similar to the one in Figure 12.7. The code of the other two buttons is simpler. The second one simply takes the text of the memo, after a user might have manually modified it, and executes it:

```
procedure TForm1.btnCreateTableClick(Sender: TObject);
begin
  ADOCommand.CommandText := Memo1.Text;
  ADOCommand.Execute;
end;
```

After a table is available, with the same structure of the original one, the last button moves the actual records. Because we cannot use the BatchMove component outside the BDE world, we simply scan the table manually, moving fields one by one. This is certainly not the most effective approach, but it can help us solve differences in the implementations: even if the field types are similar, the actual record structures in memory might be slightly different. By using the field names of the source table, the code works even if the destination table has extra fields (but not in the opposite case):

```
procedure TForm1.btnMoveDataClick(Sender: TObject);
var
  I: Integer;
begin
  BdeTable.Open;
  AdoTable.Open;
  try
    // for each record
    while not BdeTable.Eof do
    begin
      // new record
      AdoTable.Insert;
      // for each field
      for I := 0 to BdeTable.Fields.Count - 1 do
        with BdeTable.Fields[I] do
          AdoTable.FieldByName (Name).Value := Value;
      // post and move on
      AdoTable.Post;
      BdeTable.Next;
    end;
  finally
    BdeTable.Close;
    AdoTable.Close;
  end;
end;
```

Master/Detail Structures

Both the ADODataSet and ADOTable components have the same support of the master-detail relationships in the Table component. You can set a *master* data source and link some of the fields. To prove this is really the case, I've built a simple master/detail/detail structure using the common customers/orders/items tables of the DBDEMOS database, after converting them to Access with the Bde2Ado program I've just described.

To connect two ADOTable components, you set the `MasterFields` and `MasterSource` properties, as in the following DFM file:

```
object ADOTable2: TADOTable
  Connection = ADOConnection
  IndexFieldNames = 'CustNo'
  MasterFields = 'CustNo'
  MasterSource = DataSource1
  TableName = 'orders'
end
```

With an ADODataSet component, you use the DataSource and MasterFields properties instead:

```
object ADODataSet3: TADODataSet
  Connection = ADOConnection
  CommandText = 'items'
  CommandType = cmdTable
  DataSource = DataSource2
  IndexFieldNames = 'OrderNo'
  MasterFields = 'OrderNo'
end
```

Notice that the dataset is hooked to a table; when you use a query, you have to use the parameters, such as what we've done for BDE queries. You can see the output of this program in Figure 12.8. By using a data module, you can also define diagrams, such as in traditional master/detail programs, but in this case, I've built the program simply by placing the data-access components on the form.

note When you want to join two tables, you can either write a query with a join statement (as discussed in the last chapter) or use a master/detail structure. An alternative is to let the user build a table that includes an extra field corresponding to the detail table. This is supported in some object-relational database servers (such as Oracle 8) and is also available in ADO using a technique called *Data Shaping*. Data Shaping allows you to define on the server side calculated fields and fields referring to detail tables, among other things. For example, you can use this ADO command: `SHAPE {select * from customer} APPEND ({select * from orders} AS Orders RELATE CustNo TO CustNo)` to embed the orders table as a dataset field of the customer table. There is an example of this approach in the Ado\Shape folder of Delphi's Demos directory. Delphi's MIDAS architecture and the ClientDataSet component provide similar support for nested tables.

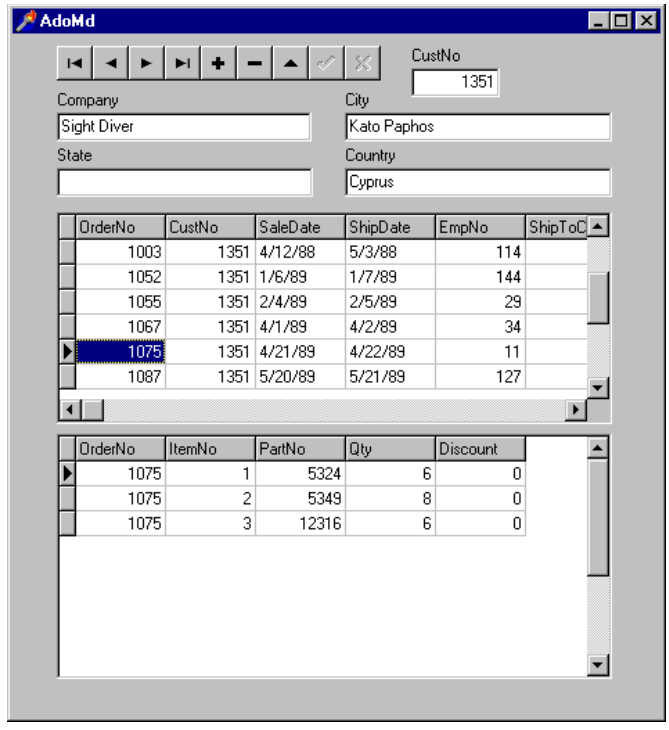


Figure 12.8: The AdoMd example demonstrates master/detail relationships among ADO dataset components. Image from the original book.

More ADO Features

We have discussed only the basics of ADO up to this point. However, I can't cover all the elements of database programming with ADO, because that would take an entire book in itself, and only owners of the Enterprise edition of Delphi would benefit from the discussion. However, a few specific features are important. To highlight some of these features, I'll build a few more examples.

Cursors and Optimization

The process of optimizing a database application is inherently quite specific to the type of database (local or SQL server) and the driver used to access the data; so I cannot discuss every optimization feature of ADO. However, there are still some features you can test when looking for improvements.

A key element that affects speed in the ADO model is the location of the *cursor* on the client or on the server, indicated by the `CursorLocation` property of the ADODataset component. Of course, this is particularly important for client/server operations:

- Using a client-side cursor means moving all of the data to the client before you start using it. This procedure may slow down the operation initially, but you will see a faster response time once the data is local. You can also perform a number of operations on the local data, such as sorting, without requesting the data from the server again. The client-side cursor behaves like a local cache.
- Using a server-side cursor means fetching only the required records, requesting more from the server as the user browses the data. This can make the application very responsive at the beginning, but in some cases, performance might drop. For example, re-sorting the records means executing a new request to the database. Also note that you cannot use a server-side cursor if the data is local, such as in a local Access table.

The cursor location is strictly connected to the cursor type, as indicated by the `CursorType` property. A `ctOpenForwardOnly` cursor type, for example, allows only unidirectional operations, which are the only ones typically supported by SQL servers (as discussed in Chapter 11). In some SQL servers, moving back to look at a past record might cause the entire query to be executed once more and all the previous records to be fetched again. However, if you know in advance you're going to scroll the entire recordset in one direction (as you would do, say, when building a

report on paper or on the Web), you'll get better performance by using a forward-only cursor (as the system doesn't have to cache data it is not going to use again).

Other cursor types (`ctKeyset`, `ctDynamic`, and `ctStatic`) determine how updates from other users will affect the data we are reading:

- A *Dynamic* cursor is the most flexible cursor type; it allows you to see records added, updated, or deleted by other users as well as to scroll forward and backward through your data. It also supports bookmarks (but only if the data provider supports them, too).
- A *Keyset* cursor operates in a similar way to a dynamic cursor, but you can't see records added or deleted by other users.
- A *Static* cursor is the most restrictive one: it gives you a *snapshot* of the data at the time of your request, and you won't see changes made by other users. This is typically used to produce reports and to elaborate a fixed set of data.

The way the data is retrieved is affected also by the `ExecuteOptions` (where you can ask for asynchronous operations, if a data provider supports them). For example, if you set the size of the `Cache` and the `eoAsyncFetch` value for the `ExecuteOptions` the ADO dataset fetches immediately the initial quantity specified in the `Cache` property and then retrieves the remaining rows asynchronously.

Finally, if you want to read records in memory to work on them without refreshing the user interface, you can set the `BlockReadSize` property of the ADODataSet component. When the value is greater than zero, the `State` of the dataset will be `dsBlockRead`, and data-aware controls won't be refreshed. The advantage is extra speed in fetching multiple records.

Indexes and Sorting

Using a client cursor, you can create temporary (or *internal*) indexes. This can speed up finding, sorting, and filtering operations. To create a temporary index, you need to add the `Optimize` dynamic property to the proper ADO field object.

What's a dynamic property? Dynamic properties in ADO are optional properties, which might or might not be present. In practice, `Properties` is a collection of variant values, with each value having a name. Many ADO objects have this collection property.

Although the Delphi ADO fields don't specifically include the `Optimize` property, adding it to the proper field object is not as difficult as it might seem.

Because Delphi doesn't offer this feature, you might think it is very complex to use it. On the contrary, the ADODataSet component exposes the internal `Recordset` ADO object, which has a `Fields` property holding the collection of ADO fields. Each field has a Properties collection you can access to set and get the value of each item. Here it is in code terms:

```
AdoDataSet.Recordset.Fields[I].
  Properties['Optimize'].Set_Value (True);
```

This code is used by the AdoSort program to create a temporary index on the field number `I` of the table. Once you've set up an index, it can be used by the `Sort` and `Filter` properties and by the `Find` method. (When you sort the table on a field, a temporary index is created anyway, making the manual operation almost useless. There is one small advantage, however: if you create the temporary index manually, you can be sure it will be preserved and used by multiple operations. If you rely on automatic indexing, the index might be re-created each time.)

Clearly, one of the aims of the AdoSort example is to sort. ADO datasets have a `Sort` property where you can list the fields to use in the sorting operation, possibly with a client-side cursor. You can assign to this property the name of a field or multiple names.

In the AdoSort example, a list box is filled with the names of the fields as the form is created:

```
procedure TFormSort.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to AdoDataSet.FieldDefs.Count - 1 do
    ListFields.Items.Add (AdoDataSet.FieldDefs [I].Name)
end;
```

A user can select a field and drag it over one of the edit boxes below it, copying the text to it:

```
procedure TFormSort.Edit1DragDrop(Sender, Source: TObject;
  X, Y: Integer);
begin
  (Sender as TEdit).Text := (Source as TListBox).Items [
    (Source as TListBox).ItemIndex];
end;
```

The read-only edit boxes can also be cleared by clicking them.

When you press the Sort button, all of the fields in which names are listed in the edit boxes will be used in the sorting operation. (The operation is timed using the `GetTickCount` Windows API.) The program appends the `'DESC'` keyword to the field

name to ask for a descending order (as in Figure 12.9). You could improve this code by making the references to the components more generic, but it works:

```
procedure TFormSort.btnSortClick(Sender: TObject);
var
  t: Cardinal;
  strSort: string;
begin
  t := GetTickCount;
  strSort := Edit1.Text;
  if CheckBox1.Checked then
    strSort := strSort + ' DESC';
  if Edit2.Text <> '' then
    strSort := strSort + ',' + Edit2.Text;
  if CheckBox2.Checked then
    strSort := strSort + ' DESC';
  if Edit3.Text <> '' then
    strSort := strSort + ',' + Edit3.Text;
  if CheckBox3.Checked then
    strSort := strSort + ' DESC';
  AdoDataSet.Sort := strSort;
  Caption := 'AdoSort - ' + IntToStr (GetTickCount - t);
end;
```

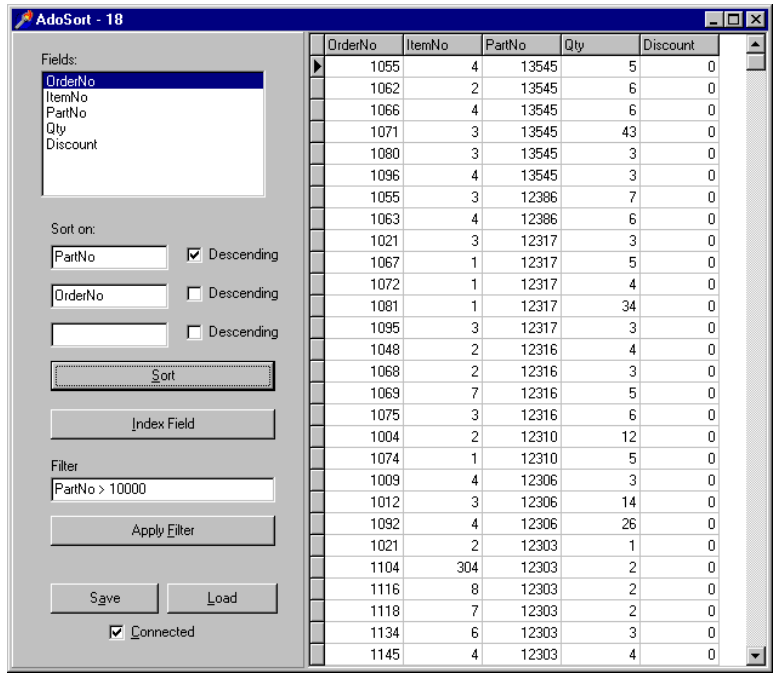


Figure 12.9: The output of the AdoSort example, with a partially descending index. Image from the original book.

If you turn on the index for a field before sorting on it, you'll see a limited time difference compared with a direct sorting. After you've done the operation once, however, the index is kept in memory, and the difference will disappear.

Filtering

The last feature demonstrated by the AdoSort program is a filter. In an ADO Recordset object, the single `Filter` property can be used for three different operations, which Delphi exposes with three different properties:

- The `Filter` property holds the usual series of conditions, such as a SQL Where clause. This is similar to the `Filter` property of a Table and can have multiple conditions separated by `AND` and `OR` operators.
- The `FilterBookmarks` property allows you to pass a collection of bookmarks to the records you want to see. It is a kind of record selector.
- The `FilterGroup` property allows you to filter records depending on their status. You can choose the modified records (only for batch mode updates), the records affected by the last insert or delete operations, and so on.

The actual filtering is enabled only after setting the `Filtered` property to `True`. The AdoSort example uses the most traditional approach, allowing you to move the text of an edit box to the `Filter` property.

A Snapshot of the Data

When you have some data in a recordset, you might want to work on it without being connected to the database, for example, on a laptop. ADO allows you to do this by letting you save a recordset to a local file and then reload it.

note The same feature is also available in Delphi's ClientDataSet component and is called the *Briefcase model*. The ClientDataSet is described in Chapter 21.[329]

In the AdoSort example, you can use the buttons at the bottom of the side panel to save the current snapshot of the recordset to a file and to reload one of these files.

329 The ClientDataSet is still an option, but FireDAC's FDMemTable does the same offering a lot more features and control.

When you save the data, you save the current situation, with the current filters and sorting. This way you can save only the data you are really interested in.

To save the data, you call the `SaveToFile` method, which requires a filename as its parameter. Some of its features are not intuitive: you cannot save to an existing file, the file remains open and is kept updated until you close the dataset (becoming a kind of persistent cache), and you should use a client cursor in case the provider doesn't support this streaming operation. Here is the code:

```
procedure TFormSort.btnSaveClick(Sender: TObject);
begin
  if SaveDialog.Execute and not
      FileExists (SaveDialog.FileName) then
    AdoDataSet.SaveToFile (SaveDialog.FileName);
end;
```

If you want to make the file available, you should first close the dataset, using the check box at the bottom:

```
procedure TFormSort.cbConnectedClick(Sender: TObject);
begin
  AdoDataSet.Active := cbConnected.Checked;
end;
```

After repeating this operation one or more times, you can close the dataset and reload one of the file-based snapshots:

```
procedure TFormSort.btnLoadClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    AdoDataSet.LoadFromFile (OpenDialog.FileName);
  cbConnected.Checked := True;
end;
```

Finding, Summing, and Locking Records

To locate a record within the current record set, you can use the standard `Locate` method. As with a BDE dataset, you can look for values of one or more fields, passed in a variant array. In the AdoEmpl example (which is based on the Employee table moved to Access from DBDEMOS), the Find button has this code attached:

```
procedure TAdoEmplForm.btnFindClick(Sender: TObject);
begin
  if not AdoTable.Locate (
      'LastName', EditName.Text, []) then
    MessageDlg ('Name not found', mtError, [mbOk], 0);
```

```
end;
```

Another feature, which was used in the Total example from Chapter 9, is the ability to compute the sum of employee salaries. Instead of doing this using a specific SQL query, the program scans the table, as we've already seen many times. In past examples we've disabled and enabled the user interface, with code like this:

```
AdoTable.DisableControls;
try
  // code
finally
  AdoTable.EnableControls;
```

This time, instead, we can use the block-size reading technique, which disables the visual controls and should also increase the performance:

```
AdoTable.BlockReadSize := 10;
try
  // code
finally
  AdoTable.BlockReadSize := 0;
```

The program also lets you experiment with lock types, although MS Access support is limited compared with what you can do in ADO. We've seen in the last chapter that Paradox locks the records you are editing. ADO allows you to specify a lock strategy, using the `LockType` property of the dataset. You can choose between a pessimistic lock, in which two users cannot edit the same record at the same time, and an optimistic lock, in which every user can supersede any other user's changes.

As you click the Lock check box, the status of the `LockType` property changes:

```
procedure TAdoEmplForm.cbLockClick(Sender: TObject);
begin
  AdoTable.Close;
  if not cbLock.Checked then
    AdoTable.LockType := ltPessimistic
  else
    AdoTable.LockType := ltOptimistic;
  AdoTable.Open;
end;
```

By activating the lock (or even if you don't activate it!) and editing the same record from two instances of the program, you should be able to see the error message shown in Figure 12.10. Notice that the effect you get by running the program depends on whether the program is also using transactions: by disabling them, you'll see a different message.

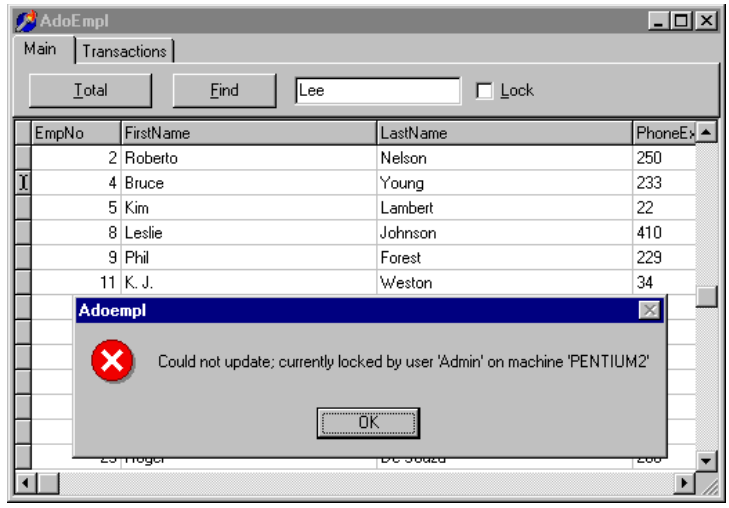


Figure 12.10: The error message caused by the locks in the AdoEmpl example. Image from the original book.

note Keep in mind that Access will give you an error message only when the record is posted, not when the editing operation starts. This is certainly not user-friendly. Also, modifying a record places a lock on that and some neighboring records, because Access uses a page-lock strategy. So if a user is editing a record, another user won't be allowed to modify that record or the next few records before or after it.

Handling Transactions in ADO

Like the BDE, ADO allows you to handle transactions. To demonstrate how ADO does this, I've taken the buttons of the Transact example and their code and added them to the AdoEmpl example, on the second page of the Page Control used to build a toolbar with two pages.

The code is very similar to the earlier BDE example, with only the name of the method having changed. (Quite an odd situation, I have to say.) The `StartTransaction`, `Commit`, and `Rollback` methods of the Database component become the `BeginTrans`, `CommitTrans`, and `RollbackTrans` methods of the AdoConnection component:

To remove the code used to enable and disable the buttons, here is the core of the transaction that handles support in the AdoEmpl example:

```
procedure TAdoEmplForm.BtnCommitClick(Sender: TObject);
begin
```

```
  if AdoTable.State = dsEdit then
    AdoTable.Post;
  AdoConnection.CommitTrans;
end;

procedure TAdoEmplForm.BtnRollbackClick(Sender: TObject);
begin
  AdoTable.Cancel;
  AdoConnection.RollbackTrans;
  // refresh
  AdoTable.Requery;
end;

procedure TAdoEmplForm.BtnStartClick(Sender: TObject);
begin
  AdoConnection.BeginTrans;
end;
```

We saw in the last chapter that the Database component allows us to indicate the transaction isolation level (the *isolation* among users' transactions in a multi-user environment) using the `TransIsolation` property. In ADO, you can control how transactions interact with each other by setting the `IsolationLevel` property of the ADOConnection component before opening the connection. Three important options are

- `ilReadCommitted` (the default value) and the equivalent `ilCursorStability` mean that we can see only committed changes made by other users. This is usually the preferred isolation level.
- `ilReadUncommitted` and the equivalent `ilBrowse` mean that we can see changes made by other users that haven't been committed. Because other users may change their minds and roll back the transaction, we cannot rely on the validity of the data we are working with, which limits this setting to few cases.
- `ilRepeatableRead` means that you won't see changes made by other transactions, even if they are committed, unless you reexecute the query. This is not possible with the `ilIsolated` value, which is the most restrictive isolation level.

Even though ADO allows a fine-grained setting for the transaction isolation level, most the providers (the databases) support fewer alternatives. You might end up with a different level of isolation than you request.

Custom Events

Beyond the usual dataset events, there are several other interesting events of the ADO components. The Delphi dataset has *Before* and *After* events; ADO has *Will* and *Complete* events.

For example, the connection component has the `OnWillConnect` and `OnConnectComplete` events and the `OnWillExecute` and `OnExecuteComplete` events. You might use `OnWillExecute` to log every command issue to the server or to filter out commands that users are not authorized to use (such as a *delete* command). This same component also has many events related to transactions.

The ADODataSet component has *Will* and *Complete* events for the change field, change record, change record set, and move operations. It has also an `OnFetchProgress` and an `OnFetchComplete` event you can use to implement a progress bar while waiting for a large dataset to load.

What's Next?

In this chapter, I've introduced the key ideas related to the use of Microsoft's ADO interface and OLE DB infrastructure for database programming. As Microsoft integrates these components with the operating system, you should expect ADO to become more and more popular.

Borland's decision to support ADO with native DataSet components lets you move existing programs easily and lets you adopt a new technology (ADO) while still using the existing architecture provided by the `TDataSet` class of the VCL. Although this is not the only possible way of working with ADO, it is certainly a good choice.

The other advantage of following this approach is that you can get rid of the BDE on the client computers, with (eventually) you having to do less installation on the client machines. For the near future, however, you should assume you'll have to install ADO on older operating systems that don't already have it. ADO even supports remote access to the data, in a 3-tier architecture. From what I've seen, however, this architecture is more limited than Borland's own MIDAS technology.

This chapter ends the part of the book devoted to database programming, but I'll focus on this topic again, discussing multi-threaded queries in Chapter 17, introducing the use of databases in Internet programming in Chapter 20, and discussing MIDAS in Chapter 21.

Chapter 13: Creating Components

While most Delphi programmers are probably familiar with using existing components, at times it can also be useful to write our own components or to customize existing ones. One of the most interesting aspects of Delphi is that creating components is simple. For this reason, even though this book is intended for Delphi application programmers and not Delphi tool writers, this chapter will cover the topic of creating components and introduce Delphi add-ins, such as component and property editors.

This chapter gives an overview of writing Delphi components and presents a number of simple examples. There is not enough space to present very complex components, but the ideas in this chapter will cover all the basics to get you started.

note You'll find a lot more information about writing components in *Delphi Developer's Handbook*, including how to build data-aware components and many other advanced techniques.

Extending the VCL

When you write a new component, you always extend one of the existing classes of the VCL[330]. To do this, you use many features of the Object Pascal language that component users seldom need. If you still have doubts about advanced Object Pascal features, you may want to review the overall description of the language presented in Part I of the book. Chapter 3 presented an overview of the role of properties, methods, and events, and Chapter 4 introduced the structure of the VCL. If you skipped those chapters or do not feel confident with the basic ideas about the VCL, read them before continuing with this chapter.

Delphi components are classes, and the VCL is the collection of all the classes defining Delphi components. Each time you add a new package with some components to Delphi, you actually extend the VCL with a new class. This new class will be derived from one of the existing component-related classes, adding new capabilities to those it inherits.

You can derive a new component from an existing component or from an *abstract component class*—one that does not correspond to a usable component. The VCL hierarchy includes many of these intermediate classes to let you choose a default behavior for your new component and to change its properties.

Component Packages

Components are added to component packages. Each component package is basically a DLL (a dynamic link library) with a BPL extension (which stands for Borland Package Library).

Packages come in two flavors: design-time packages used by the Delphi IDE, and run-time packages optionally used by applications. The type of the package is determined by the design-only or run-only package option. When you attempt to install a package, the IDE checks whether it has the design-only or run-only flags, and

330 The overall component architecture hasn't changed much since the early days of Delphi. While there are additional features, the foundations remain the same.

decides whether to let the user install the package and if it should be added to the list of run-time packages. Since there are two non-exclusive options, each with two possible states, there are four different kinds of component packages—two main variations and two special cases:

- Design-only component packages can be installed in the Delphi environment. These packages usually contain the design-time parts of a component, such as its property editors and the registration code. Often they can also contain the components themselves, although this is not the most professional approach. The code of the components of a design-only package is usually statically linked into the executable file, using the code of the corresponding Delphi Compiled Unit (DCU) files. Keep in mind, however, that it is also technically possible to use a design-only package as a run-time package.
- Run-only component packages are used by Delphi applications at run time. They cannot be installed in the Delphi environment, but they are automatically added to the list of run-time packages when they are required by a design-only package you install. Run-only packages usually contain the code of the component classes, but no design-time support (this is done to minimize the size of the component libraries you ship along with your executable file). Run-only packages are important because they can be freely distributed along with applications, but other Delphi programmers won't be able to install them in the environment to build new programs.
- Plain component packages (having neither the design-time–only nor the run-time–only option set) cannot be installed and will not be added to the list of run-time packages automatically. This might make sense only for utility packages used by other packages, but they are certainly rare.
- Packages with both flags set can be installed and are automatically added to the list of run-time packages. Usually these packages contain components requiring little or no design-time support (apart from the limited component registration code). Keep in mind, however, that users of applications built with these packages can use them for their own development.

note The file names of Delphi's own design-only packages start with the letters DCL (for example DCLSTD50.BPL); file names of run-only packages start with the letters VCL[331] (for example, VCL50.BPL). You can follow the same approach for your own packages, if you want.

331 This is true only for VCL related component packages, FireMonkey related component packages start with FMX and other subsystem use a different naming.

In Chapter 1 we discussed the effect of packages on the size of a program's executable file. Now we'll focus on building packages, since this is a required step in creating or installing components in Delphi.

When you compile a run-time package, you produce both a dynamic link library with the compiled code (the BPL file) and a file with only symbol information (a DCP file), including no compiled machine code. The latter file is used by the Delphi compiler to gather symbol information about the units that are part of the package without having access to the unit (DCU) files, which contain both the symbol information and the compiled machine code. This reduces compilation time and allows you to distribute just the packages without the pre-compiled unit files. The pre-compiled units are still required to statically link the components into an application. Distribution of pre-compiled DCU files (or source code) may make sense depending on the kind of components you develop. We'll see how to create a package after we've discussed some general guidelines and built our very first component.

note DLLs are executable files containing collections of functions and classes, which can be used by an application or another DLL at run time. The typical advantage is that if many applications use the same DLL, only one copy needs to be on the disk or loaded in memory, and the size of each executable file will be much smaller. This is what happens with the new Delphi packages, as well. Chapter 14 looks at DLLs and packages in more detail.

Rules for Writing Components

Some general rules govern the writing of components. You can find a detailed description of most of them in the Delphi *Component Writer's Guide* help, which is required reading for Delphi component writers[332].

Here is my own summary of the rules for component writers:

- Study the Object Pascal language with care. Particularly important concepts are inheritance, method overriding and overloading, the difference between public and published sections of a class, and the definition of properties and events.
- Study the structure of the VCL class hierarchy and keep a graph of the classes at hand (such as the one included with Delphi).

332 This is still available as part of the Delphi docwiki, see https://docwiki.embarcadero.com/RADStudio/en/Component_Writer%27s_Guide_Index.

- Follow the standard Delphi naming conventions. There are several of them for components, as we will see, and following these rules makes it easier for other programmers to interact with your components and further extend them.
- Keep components simple, mimic other components, and avoid dependencies. These three rules basically mean that a programmer using your components should be able to use them as easily as preinstalled Delphi components. Use similar property, method, and event names whenever possible. If users don't need to learn complex rules about the use of your component (that is, if the dependencies between methods or properties are limited) and can simply access properties with meaningful names, they'll be happy.
- Use exceptions. When something goes wrong, the component should raise an exception. When you are allocating resources of any kind, you must protect them with `try-finally` blocks and destructor calls.
- To complete a component, add a bitmap to it, to be used by Delphi's Component Palette. If you intend your component to be used by more than a few people, consider adding a Help file as well.
- Be ready to write *real* code and forget about the visual aspects of Delphi. Writing components generally means writing code without visual support (although Class Completion can speed up the coding of plain classes quite a lot).
- The exception to the rule above, in Delphi 5, is that you can use frames to write components visually.
- Use a third-party component writing tool to build your component or to speed up their development.
- The most powerful third-party tool for creating Delphi components I know of is the Component Development Kit (CDK) from Eagle Software (`http://www.eagle-software.com`).[333]

333 This tool is no longer available. I don't think there is any tool for writing Delphi components these days.

The Base Component Classes

To build a new component you generally start from an existing one, or from one of the base classes of the VCL. In both cases your component is in one of three broad categories of components (introduced in Chapter 4), set by the three basic classes of the component hierarchy:

- `TWinControl` is the parent class of any component based on a window. Components that descend from this class can receive the input focus and get Windows messages from the system. You can also use their window handle when calling API functions. When creating a brand-new window control, you'll generally inherit from the derived class `TCustomControl`, which has a few extra useful features (particularly some support for painting the control).
- `TGraphicControl` is the parent class of visible components that have no Windows handle (which saves some Windows resources). These components cannot receive the input focus or respond to Windows messages directly. When creating a brand-new graphical control, you'll inherit directly from this class (which has a set of features very similar to `TCustomControl`).
- `TComponent` is the parent class of all components (including the controls) and can be used as a direct parent class for non-visual components.

In the rest of the chapter, we will build some components using various parent classes, and we'll look at the differences among them. We'll start with components inheriting from existing components or classes at a low level of the hierarchy, and then we'll see examples of classes inheriting directly from the ancestor classes mentioned above.

note When you derive a new component from these high-level classes of the VCL hierarchy, it already inherits some properties that are common to all components. Refer to Figure 4.4 to see the properties defined in some of the high-level VCL classes.

Building Your First Component

As we saw in Chapter 3, it is very simple to take an existing class, turn it into a non-visual component, and build a simple package to host it. Building components is actually an important activity for Delphi programmers. The basic idea is that any time you need the same behavior in two different places in an application, or in two

different applications, you can place the shared code inside a class—or, even better, a component.

In this section I'll just introduce a couple of simple components to give you an idea of the steps required to build one and to show you different things you can do to customize an existing component with a limited amount of code.

The Fonts Combo Box

Many applications have a toolbar with a combo box you can use to select a font. If you often use a customized combo box like this, why not turn it into a component? It would probably take less than a minute. To begin, close any active projects in the Delphi environment and start the Component Wizard either by choosing Component ➢ New Component or by selecting File ➢ New to open the Object Repository and then choosing the Component in the New page. As you can see in Figure 13.1, the Component Wizard requires the following information:

- The name of the ancestor type: the component class you want to inherit from. In this case we can use `TComboBox`.[334]
- The name of the class of the new component you are building; we can use `TMdFontCombo`.
- The page of the Component Palette where you want to display the new component, which can be a new or an existing page. We can create a new page, called *Md*.
- The filename of the Pascal unit where Delphi will place the source code of the new component; we can type `MdFontBox`.
- The current search path (which should be set up automatically).

334 In the current version of the Component Wizard, the first page offers a selection of the UI library to write a component for (and language in case of RAD Studio) and a second page offers the selection of the base class to inherit from, with support for search. The third page, visible in the second part of Figure 13.1, has the remaining options.

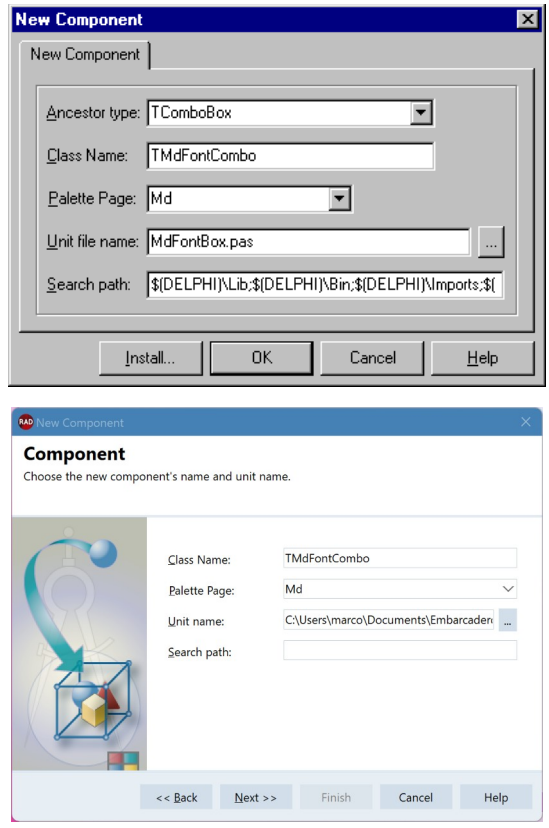


Figure 13.1: The definition of the new TMdFontCombo component with the Component Wizard. Images captured in Delphi 5 and Delphi 12.

Click on the OK button, and the Component Wizard will generate the following simple Pascal source file with the structure of your component. The Install button can be used to install the component in a package immediately. Let's look at the code first, and then discuss the installation:

```
unit FontBox;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;[335]

type
```

335 Today you'll get a list with the following unit in the `uses` statement: `System.SysUtils`, `System.Classes`, `Vcl.Controls`, and `Vcl.StdCtrls`.

```
  TMdFontCombo = class (TComboBox)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Md', [TMdFontCombo]);
end;

end.
```

One of the key elements of this listing is the class definition, which begins by indicating the parent class. The only other relevant portion is the `Register` procedure. In fact, you can see that the Component Wizard does very little work.

note Starting with Delphi 4, the `Register` procedure *must* be written with an uppercase R. The reason is apparently due to C++Builder compatibility (identifiers in C++ are case-sensitive).[336]

note Use a naming convention when building components. All the components installed in Delphi should have different class names. For this reason most Delphi component developers have chosen to add a two- or three-letter signature prefix to the names of their components. I've done the same, using *Md* to identify components built in this book. The advantage of this approach is that you can install my `TMdFontCombo` component even if you've already installed a component named `TFontCombo`. Notice that the unit names must also be unique for all the components installed in the system, so I've applied the same prefix to the unit names. To check the prefixes used by the most important component vendors and other developers you can refer to the Web site `http://developers.href.com/dpr`.[337]

That's all it takes to build a component. Of course, in this example there isn't a lot of code. We need only copy all the system fonts to the `Items` property of the combo box at startup. To accomplish this, we might try to override the `Create` method in the class declaration, adding the statement `Items := Screen.Fonts`. However, this is

336 This is still true today!

337 To my knowledge, this web site doesn't exist any more.

not the correct approach. The problem is that we cannot access the combo box's `Items` property before the window handle of the component is available. And the component cannot have a window handle until its `Parent` property is set, and the `Parent` property isn't set in the constructor, but later on.

For this reason, instead of assigning the new strings in the `Create` constructor, we must perform this operation in the `CreateWnd` procedure, which is called to create the window control after the component is constructed, its `Parent` property is set and its window handle is available. Again, we execute the default behavior, and then we can write our custom code. I could have skipped the `Create` constructor and written all the code in `CreateWnd`, but I decided to use both startup methods to demonstrate the difference between them. Here is the declaration of the component class:

```
type
  TMdFontCombo = class (TComboBox)
  public
    constructor Create (AOwner: TComponent); override;
    procedure CreateWnd; override;
  published
    property Style default csDropDownList;
    property Items stored False;
  end;
```

And here is the source code of its two new methods:

```
constructor TMdFontCombo.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  Style := csDropDownList;
end;

procedure TMdFontCombo.CreateWnd;
begin
  inherited CreateWnd;
  Items.Assign (Screen.Fonts);
end;
```

Notice that besides giving a new value to the component's `Style` property, in the `Create` method, I've redefined this property by setting a value with the `default` keyword. We have to do both operations because adding the `default` keyword to a property declaration has no direct effect on the property's initial value. Why specify a property's default value then? Because properties that have a value equal to the default are not streamed with the form definition (and they don't appear in the textual description of the form, the DFM file). The `default` keyword tells the streaming code that the component initialization code will set the value of that property.

The other redefined property, `Items`, is set as a property that should not be saved to the DFM file at all, regardless of the actual value. This is obtained with the `stored` directive followed by the value `False`. The component and its window are going to be created again when the program starts, so it doesn't make any sense to save in the DFM file information that will be discarded later on (to be replaced with the new list of fonts).

note We could have even written the code of the `CreateWnd` method to copy the fonts to the combo box items only at run time. This can be done by using statements such as this:
```
if not (csDesigning in ComponentState) then
```
But for this first component we are building, the less efficient but more straightforward method described above offers a clearer illustration of the basic procedure.

Creating a Package

Now we have to install the component in the environment, using a package. For this example, we can either create a new package or use an existing one, like the default "users package," as we did in Chapter 3. Creating a new package is quite simple, anyway.

In each case, choose the Component ➢ Install Component menu command. The resulting dialog box has a page to install the component into an existing package, and a page to create a new package. In this last case, simply type in a filename and a description for the package. Clicking OK opens the Package Editor[338] (see Figure 13.2), which has two parts:

- The Contains list indicates the components included in the package (or, to be more precise, the units defining those components).
- The Requires list indicates the packages required by this package. Your package will generally require the `vcl50` package[339] (the main run-time package that contains the fundamental parts of the Delphi VCL), but it might also need the `vcldb50` package (which includes most of the database-related classes) if the components of the new package do any database-related operations.

338 The package editor as a stand alone tool doesn't exist any more. The same features (with the contains and requires sections) are available in the project manager pane when a package is the active project.

339 These days the reference to system packages can be written without specifying the version number, that is using `vcl` and not `vcl50`. This makes the code easier to migrate to newer versions of Delphi. Also in terms of the name of the actual BPL file generated, you can automatically append the internal package number matching the current product version.

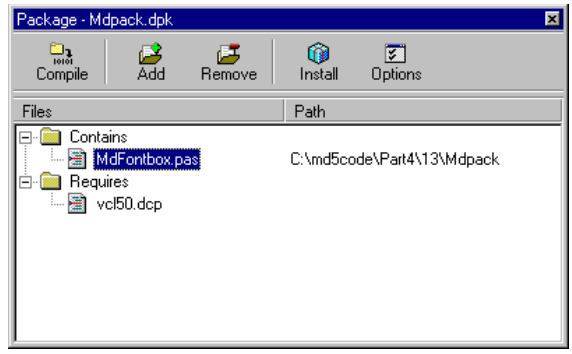


Figure 13.2: The Package Editor. Image from the original book. These same features are now part of the Project Manager when the active project is a package.

If you add the component to the new package we've just defined, and then simply compile the package and install it (using the two corresponding toolbar buttons of the package editor), you'll immediately see the new component show up in the *'Md'* page of the Component Palette. The `Register` procedure of the component unit file told Delphi where to install the new component. By default, the bitmap used will be the same as the parent class, because we haven't provided a custom bitmap (we will do this in later examples). Notice also that if you move the mouse over the new component, Delphi will display as a hint the name of the class without the initial letter *T*.

What's Behind a Package?

What is behind the package we've just built? The package editor basically generates the source code for the package project: a special kind of DLL built in Delphi. The package project is saved in a file with the DPK (for Delphi PacKage) extension. A typical package project looks like this:

```
package MdPack;

{$R *.RES}
{$ALIGN ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
...
{$DESCRIPTION 'Mastering Delphi Package'}
{$IMPLICITBUILD ON}

requires
  vcl50;

contains
  MdFontBox in 'MdFontBox.pas';

end.
```

As you can see, Delphi uses specific language keywords for packages: the first is the `package` keyword (which is similar to the `library` keyword I'll discuss in Chapter 14). This keyword introduces a new package project. Then comes a list with all the compiler options, some of which I've omitted from the listing. Usually the options for a Delphi project are stored in a separate file; packages, by contrast, include all the compiler options directly in their source code. Among the compiler options there is a `DESCRIPTION` compiler directive, used to make the package description available to the Delphi environment. In fact, after you've installed a new package, its description will be shown in the Packages page of the Project Options dialog box, a page you can also activate by selecting the Component ➤ Install Packages menu item. This dialog box is shown in Figure 13.3.

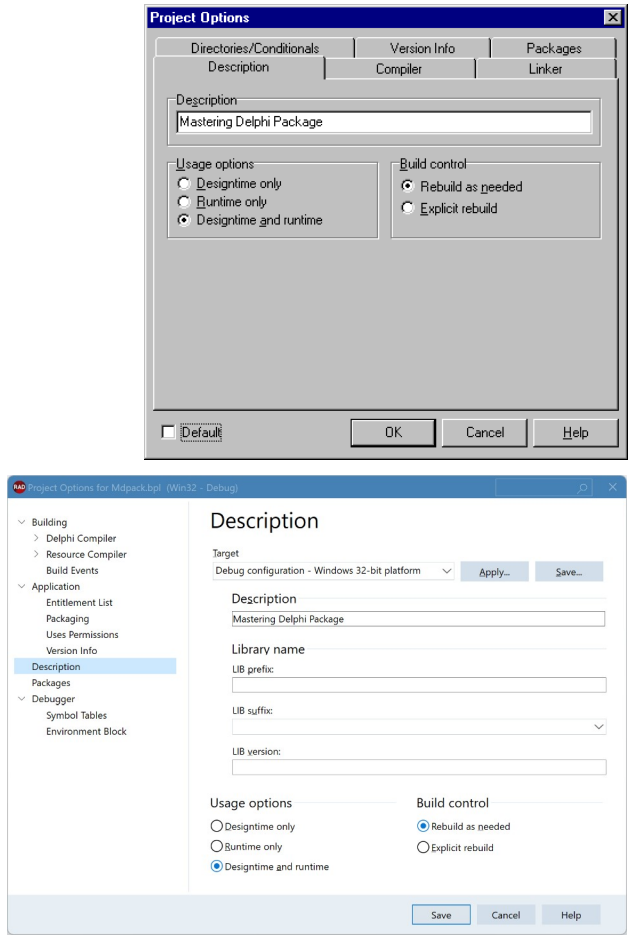


Figure 13.3: The Project Options for packages. You can see the new package we've just created. Image from the original book and in Delphi 12.

Besides common directives like the `DESCRIPTION` one, there are other compiler directives specific to packages. The most common of these options are easily accessible through the Options button of the Package Editor. After this list of options come the `requires` and `contains` keywords, which list the items displayed visually in the two pages of the Package Editor. Again, the first is the list of packages required by the current one and the second is a list of the units installed by this package.

What is the technical effect of building a package? Besides the DPK file with the source code, Delphi generates a BPL file with the dynamic link version of the package and a DCP file with the symbol information. In practice, this DCP file is the sum of the symbol information of the DCU files of the units contained in the package.

At design time, Delphi requires both the BPL and DCP files, because the first has the actual code of the components created on the design form and the symbol information required by the Code Insight technology. If you link the package dynamically (using it as a run-time package), the DCP file will also be used by the linker, and the BPL file should be shipped along with the main executable file of the application. If you instead link the package statically, the linker refers to the DCU files, and you'll need to distribute only the final executable file.

For this reason as a component designer you should generally distribute at least the BPL file, the DCP file, and the DCU files of the units contained in the package and any corresponding DFM files, plus a help file. As an option, of course, you might also make available the source code files of the package units (the PAS files) and of the package itself (the DPK file).

Installing the Components of This Chapter

Having built our first package, we can now start using the component we've added to it. Before we do so, however, I should mention that I've extended the `MdPack` package to include all of the components we are going to build in this chapter, including different versions of the same component. I suggest you install this package. The best approach is to copy it into a directory of your path, so that it will be available both to the Delphi environment and to the programs you build with it. I've collected all the component source code files and the package definition in a single sub-directory, called MdPack. This allows the Delphi environment to refer only to one directory when looking for the package and the DCU files.

Remember, anyway, that if you compile an application using the packages as run-time DLLs, you'll need to install these new libraries on your clients' computers. If you instead compile the programs by statically linking the package, the DLL will be required only by the development environment, and not by the users of your applications.

note Besides creating and installing single packages, Delphi can handle collections of packages. The Package Collection Editor (PCE.EXE, in the Delphi/Bin directory) allows you to place multiple packages in a single DPC (Delphi Package Collection) file[340]. This file can then be installed in Delphi in the same way you install a stand-alone package. The Package Collection Editor allows you to specify a complex installation, with several support files, and even to let users choose the directory where they want to install the compiled packages, the source code files, and all the other support files.

Using the Fonts Combo Box

Now you can create a new Delphi program to test the Font combo box. Move to the Component Palette, select the new component, and add it to a new form. A traditional-looking combo box will appear. However, if you open the Items Property Editor, you'll see a list of the fonts installed on your computer. To build a simple example, I've added a Memo component to the form with some text inside it. The FontBoxDemo program has very little code. When a user selects a new font in the combo box, the new value is used as the Memo component's font:

```
procedure TForm1.MdFontCombo1Change(Sender: TObject);
begin
  // activate the new selection
  Memo1.Font.Name := MdFontCombo1.Text;
end;
```

At the beginning, the reverse action is performed; the name of the Memo component's font is displayed in the combo box:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // select the item corresponding to the current font
  MdFontCombo1.ItemIndex :=
    MdFontCombo1.Items.IndexOf (Memo1.Font.Name);
end;
```

The aim of this simple program (see Figure 13.4 for its output) is only to test the behavior of the new component we have built. The component is still not very useful —we could have added a couple of lines of code to a form to obtain the same effect— but looking at a couple of simple components should help you get an idea of what is involved in component building.

340 The concept of Package Collection and the specific editor are no longer part of Delphi.

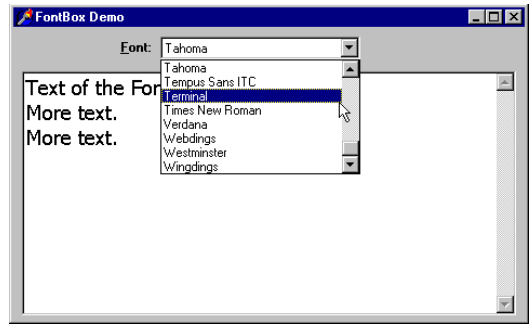


Figure 13.4: The output of the FontBoxDemo example. Image from the original book.

Creating Compound Components

The next component I want to focus on is a digital clock. This example has some interesting features. First, it embeds a component (a Timer) in another component; second, it shows the live-data approach.

Since the digital clock will provide some text output, I considered inheriting from the `TLabel` class. However, this would allow a user to change the label's caption—that is, the text of the clock. To avoid this problem, I simply used the `TCustomLabel` component as the parent class. A `TCustomLabel` object has the same capabilities as a `TLabel` object, but few published properties. In other words, a `TCustomLabel` subclass can decide which properties should be available and which should remain hidden.

note Most of the Delphi components, particularly the Windows-based ones, have a `TCustomXxx` base class, which implements the entire functionality but exposes only a limited set of properties. Inheriting from these base classes is the standard way to expose only some of the properties of a component in a customized version. In fact, you cannot hide public or published properties of a base class.

Besides re-declaring some of the properties of the parent class, `TMdClock` has one new property of its own, `Active`. This property indicates whether or not the clock is working. As you might have guessed, the clock contains a `TTimer` component. The timer is not made public through a property, because I don't want programmers to access it directly. Instead, I made the `Enabled` property of the Timer available, wrap-

ping it inside the `Active` property of the digital clock. Here is the full type declaration for the component:

```
type
  TMdClock = class (TCustomLabel)
  private
    FTimer: TTimer;
    function GetActive: Boolean;
    procedure SetActive (Value: Boolean);
  protected
    procedure UpdateClock (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
  published
    property Align;
    property Alignment;
    property Color;
    property Font;
    property ParentColor;
    property ParentFont;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property Transparent;
    property Visible;
    property Active: Boolean
      read GetActive write SetActive;
  end;
```

Notice that we need methods both to write and to read the value of the `Active` property, because the value of the property is not local data, but rather refers to a member of the embedded component, the Timer:

```
function TMdClock.GetActive: Boolean;
begin
  Result := FTimer.Enabled;
end;

procedure TMdClock.SetActive (Value: Boolean);
begin
  FTimer.Enabled := Value;
end;
```

To create the Timer, we have to override the constructor of the clock component. The `Create` method calls the corresponding method of the base class and creates the Timer object, installing a handler for its `OnTimer` event:

```
constructor TMdClock.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  // create the internal timer object
```

```
  FTimer := TTimer.Create (Self);
  FTimer.OnTimer := UpdateClock;
  FTimer.Enabled := True;
end;
```

The code doesn't set a value for the Timer's `Interval` property, simply because the timer's default interval of 1000 milliseconds is appropriate. We don't need a destructor, simply because the `FTimer` object has our `TMDClock` component as owner (as indicated by the parameter of its `Create` constructor), so it will be destroyed automatically when the clock component is destroyed.

The key piece of the component's code is the `UpdateClock` procedure, which is just one statement:

```
procedure TMdLabelClock.UpdateClock (Sender: TObject);
begin
  // set the current time as caption
  Caption := TimeToStr (Time);
end;
```

This method uses `Caption`, which is an unpublished property, so that a user of the component cannot modify it in the Object Inspector. The result of this statement is to display the current time. This happens continuously, because the method is connected to the Timer's `OnTimer` event.

The Component Palette Bitmaps

Before installing this second component, we can take one further step: define a bitmap for the Component Palette. If we fail to do so, the Palette uses the bitmap of the parent class, or a default object's bitmap if the parent class is not an installed component (as is the case of the `TCustomLabel`). Defining a new bitmap for the component is easy, once you know the rules. You can create one with the Image Editor[341] (as shown in Figure 13.5), starting a new project and selecting the `DCR` (Delphi Component Resource) project type.

341 As already mentioned, the Image Editor is no longer part of Delphi. You can use any graphic editor.

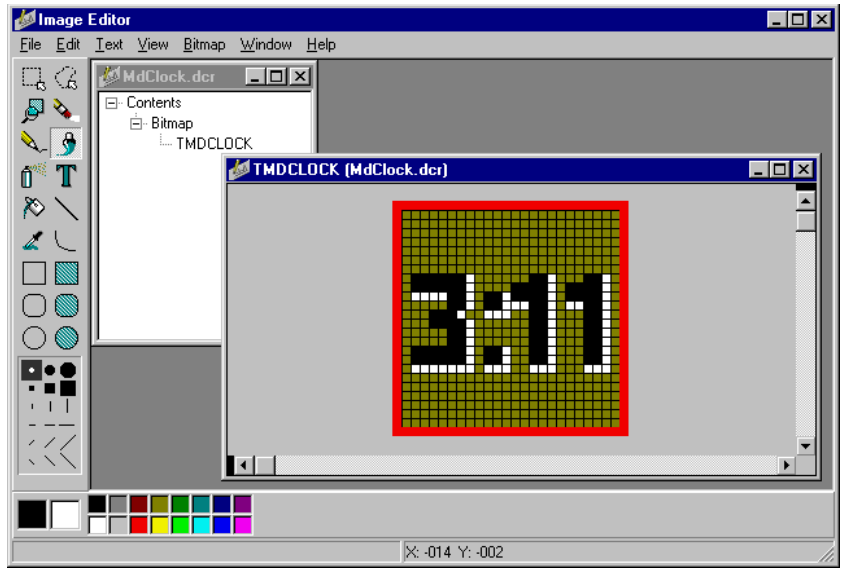


Figure 13.5: The definition of a component palette bitmap in Delphi's Image Editor. Image from the original book. This tool is no longer available.

note DCR files are simply standard RES files with a different extension. If you prefer, you can create them with any resource editor, including the Borland Resource Workshop[342], which is certainly a more powerful tool than the Delphi Image editor. When you finish creating the resource file, simply rename the RES file to use a DCR extension.

Now, we can add a new bitmap to the resource, choosing a size of 24 × 24 pixels. Now we are ready to draw the bitmap. The other important rules refer to naming. In this case, the naming rule is not just a convention, it is a requirement so that the IDE can find the image for a given component class:

- The name of the bitmap resource must match the name of the component, including the initial *T*. In this case, the name of the bitmap resource should be TMDCLOCK. The name of the bitmap resource must be uppercase. This is mandatory.
- If you want the Package Editor to recognize and include the resource file, the name of the DCR file must match the name of the compiled unit that defines the component. In this case, the file name should be MdClock.DCR. If you manually include the resource file, via a `$R` directive, you can give it the name you like, and also use a RES or DCR file with multiple palette icons.

342 The Borland Resource Workshop has also been removed from the product.

When the bitmap for the component is ready, you can install the component in Delphi, by using the Package Editor's Install Package toolbar button. After this operation, the Contains section of the editor should list both the PAS file of the component and the corresponding DCR file (the list of DCR files was not available in Delphi 3). In Figure 13.6 you can see all the files (including the DCR files) of the final version of the Md4Pack package. If the DCR installation doesn't work properly, you can manually add the `{$R unitname.dcr}` statement in the package source code.

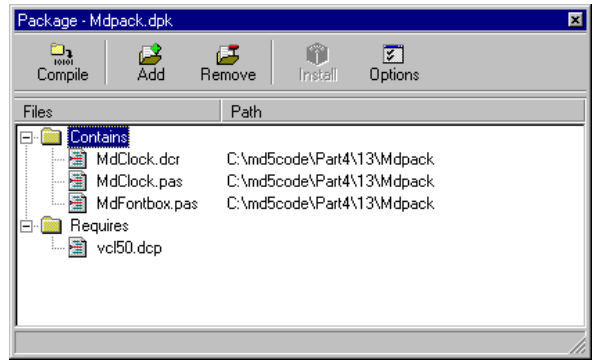


Figure 13.6: The Contains section of the Package Editor shows both the units that are included in the package and the component resource files. Image from the original book.

Building Compound Components with Frames

Instead of building the compound component in code and hooking up the event of the timer manually, we could have obtained a similar effect by using a frame. Frames make the development of compound components with custom event handlers a visual (and simpler) operation. You can share this frame by adding it to the Repository or by creating a template using the Add to Palette command of the shortcut menu of the frame itself.

As an alternative, you might want to share the frame by placing it into a package and registering it as a component. Technically this is not too difficult. You add a `Register` procedure to the frame's unit, add the unit to a package, and build it. The new component/frame will be in the Component Palette, like any other component. However when you place this component/frame on a form you won't see its sub-components and won't be able to interact with them as design time. The running program will behave as expected, but the limited design time support makes this approach less then ideal.

An Active Button

The Windows interface is evolving toward a new standard, including components that become highlighted as the mouse cursor moves over them. Delphi provides similar support in many of its built-in components, but what does it take to mimic this behavior for a simple button component that you create? This might seem a complex task to accomplish, but it is not.

Usually, the development of a component can be extremely simple, once you know which virtual function to override or which message to hook onto. The next component, the `TMdActiveButton` class, demonstrates this by handling some internal Delphi messages to accomplish its task in a very simple way.

---

note How did I determine which messages were the right ones to use, when they are almost completely undocumented? By studying the VCL source code. That's generally a good way to become an expert component builder. You can also study some of the advanced Delphi books available.

The ActiveButton component handles the `cm_MouseEnter` and `cm_MouseExit` internal Delphi messages, which are received when the mouse cursor enters or leaves the area corresponding to the component:

```
type
  TMdActiveButton = class (TButton)
  protected
    procedure MouseEnter (var Msg: TMessage);
      message cm_mouseEnter;
    procedure MouseLeave (var Msg: TMessage);
      message cm_mouseLeave;
  end;
```

The code you write for these two methods can do whatever you want. For this example I've simply decided to toggle the bold style of the font of the button itself. You can see the effect of moving the mouse over one of these components in Figure 13.7.

```
procedure TMdActiveButton.MouseEnter (var Msg: TMessage);
begin
  Font.Style := Font.Style + [fsBold];
end;

procedure TMdActiveButton.MouseLeave (var Msg: TMessage);
begin
  Font.Style := Font.Style - [fsBold];
end;
```

You can add other effects at will, including enlarging the font itself, making the button the default, or increasing its size a little. The best effects usually involve colors,

but you should inherit from the `TBitBtn` class to have this support (`TButton` controls have a fixed color).

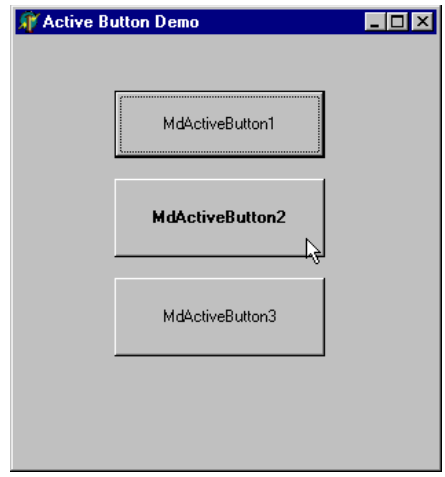


Figure 13.7: An example of the use of the ActiveButton component. Image from the original book.

A Complex Graphical Component

The graphical component I want to build is an arrow component. You can use such a component to indicate a flow of information, or an action, for example. This component is quite complex, so I'll show you the various steps instead of looking directly at the complete source code. The component I've added to the MdPack package is only the final version of this process, which will demonstrate a number of important concepts:

- The definition of new enumerated properties, based on custom enumerated data types.
- The use of properties of `TPersistent`-derived classes, such as `TPen` and `TBrush`, and the issues related to their creation and destruction, and to handling their `OnChange` events internally in our component.
- The implementation of the `Paint` method of the component, which provides its user interface and should be generic enough to accommodate all the possible val-

ues of the various properties, including its `Width` and `Height`. The `Paint` method plays a substantial role in this graphical component.

- The definition of a custom event handler for the component, responding to user input (in this case, a double-click on the point of the arrow). This will require direct handling of Windows messages and the use of the Windows API for graphic regions.
- The registration of properties in Object Inspector categories and the definition of a custom category.

Defining an Enumerated Property

After generating the new component with the Component Wizard and choosing `TGraphicControl` as the parent class, we can start to customize the component. The arrow can point in any of four directions: up, down, left, or right. An enumerated type expresses these choices:

```
type
  TMdArrowDir = (adUp, adRight, adDown, adLeft);
```

This enumerated type defines a private data member of the component, a parameter of the procedure used to change it, and the type of the corresponding property. Two more simple properties are `ArrowHeight` and `Filled`, the first determining the size of the arrowhead and the second whether to fill the arrowhead with color:

```
TMdArrow = class (TGraphicControl)
  private
    fDirection: TMdArrowDir;
    fArrowHeight: Integer;
    fFilled: Boolean;
    procedure SetDirection (Value: TMd4ArrowDir);
    procedure SetArrowHeight (Value: Integer);
    procedure SetFilled (Value: Boolean);
  published
    property Width default 50;
    property Height default 20;
    property Direction: TMd4ArrowDir
      read fDirection write SetDirection default adRight;
    property ArrowHeight: Integer
      read fArrowHeight write SetArrowHeight default 10;
    property Filled: Boolean
      read fFilled write SetFilled default False;
```

note A graphic control has no default size, so when you place it in a form, its size will be a single pixel. For this reason it is important to add a default value for the `Width` and `Height` properties and set the class fields to the default property values in the constructor of the class.

The three custom properties are read directly from the corresponding field and are written using three Set methods, all having the same standard structure:

```
procedure TMdArrow.SetDirection (Value: TMdArrowDir);
begin
  if fDirection <> Value then
  begin
    fDirection := Value;
    ComputePoints;
    Invalidate;
  end;
end;
```

Notice that we ask the system to repaint the component (by calling `Invalidate`) only if the property is really changing its value and after calling the `ComputePoints` method, which computes the triangle delimiting the arrowhead. Otherwise, the code is skipped and the method ends immediately. This code structure is very common, and we will use it for most of the *Set* procedures of properties.

We must also remember to set the default values of the properties in the component's constructor:

```
constructor TMdArrow.Create (AOwner: TComponent);
begin
  // call the parent constructor
  inherited Create (AOwner);
  // set the default values
  fDirection := adRight;
  Width := 50;
  Height := 20;
  fArrowHeight := 10;
  fFilled := False;
```

In fact, as mentioned before, the default value specified in the property declaration is used only to determine whether to save the property's value to disk. The `Create` constructor is defined in the public section of the type definition of the new component, and it is indicated by the `override` keyword. It is fundamental to remember this keyword; otherwise, when Delphi creates a new component of this class, it will call the constructor of the base class, rather than the one you've written for your derived class.

Property Naming Conventions

In the definition of the Arrow component, notice the use of several naming conventions for properties, access methods, and fields. Here is a summary:

- A property should have a meaningful and readable name.
- When a private data field is used to hold the value of a property, the field should be named with an *f* (field) at the beginning, followed by the name of the corresponding property.
- When a function is used to change the value of the property, the function should have the word *Set* at the beginning, followed by the name of the corresponding property.
- A corresponding function used to read the property should have the word *Get* at the beginning, again followed by the property name.

These are just guidelines to make programs more readable. The compiler doesn't enforce them. These conventions are described in the *Delphi Component Writers Guide* and are followed by the Delphi's class completion mechanism.

Writing the Paint Method

Drawing the arrow in the various directions and with the various styles requires a fair amount if code. To perform custom painting, you override the `Paint` method and use the protected `Canvas` property.

Instead of computing the position of the arrowhead points in drawing code that will be executed often, I've written a separate function to compute the arrowhead area and store it in an array of points defined among the private fields of the component as:

```
fArrowPoints: array [0..3] of TPoint;
```

These points are determined by the `ComputePoints` private method, which is called every time some of the component properties change. Here is an excerpt of its code:

```
procedure TMdArrow.ComputePoints;
var
  XCenter, YCenter: Integer;
begin
  // compute the points of the arrowhead
```

```
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;
  case FDirection of
    adUp: begin
      fArrowPoints [0] := Point (0, FArrowHeight);
      fArrowPoints [1] := Point (XCenter, 0);
      fArrowPoints [2] := Point (Width-1, FArrowHeight);
    end;
  // and so on for the other directions
```

The code computes the center of the component area (simply dividing the `Height` and `Width` properties by two) and then uses it to determine the position of the arrowhead. Besides changing the direction or other properties, we need to refresh the position of the arrowhead when the size of the component changes. What we can do is to override the `SetBounds` method of the component, which is called by the VCL every time the `Left`, `Top`, `Width`, and `Height` properties of a component change:

```
procedure TMdArrow.SetBounds(ALeft, ATop, AWidth, AHeight: Integer);
begin
  inherited SetBounds (ALeft, ATop, AWidth, AHeight);
  ComputePoints;
end;
```

Once the component knows the position of the arrowhead, its painting code becomes simpler. Here is an excerpt of the `Paint` method:

```
procedure TMdArrow.Paint;
var
  XCenter, YCenter: Integer;
begin
  // compute the center
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;

  // draw the arrow line
  case FDirection of
    adUp: begin
      Canvas.MoveTo (XCenter, Height-1);
      Canvas.LineTo (XCenter, FArrowHeight);
    end;
    // and so on for the other directions
  end;

  // draw the arrow point, eventually filling it
  if FFilled then
    Canvas.Polygon (fArrowPoints)
  else
    Canvas.PolyLine (fArrowPoints);
end;
```

You can see an example of the output of this component in Figure 13.8.

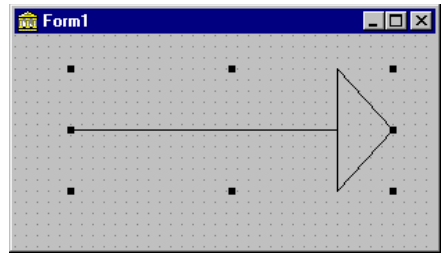


Figure 13.8: The output of the Arrow component. Image from the original book.

Adding TPersistent Properties

To make the output of the component more flexible, I've added to it two new properties, defined with a class type (specifically, a `TPersistent` data type, which defines objects that can be automatically streamed by Delphi). These properties are a little more complex to handle, because the component now has to create and destroy these internal objects (as we did with the internal Timer of the clock component). This time, however, we also export the internal objects using some properties, so that users can directly change them from the Object Inspector. To update the component when these sub-objects change, we'll also need to handle their internal `OnChange` property. Here is the definition of the two new `TPersistent`-type properties and the other changes to the definition of the component class:

```
type
  TMdArrow = class (TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    ...
    procedure SetPen (Value: TPen);
    procedure SetBrush (Value: TBrush);
    procedure RepaintRequest (Sender: TObject);
  published
    property Pen: TPen
      read FPen write SetPen;
    property Brush: TBrush
      read FBrush write SetBrush;
  end;
```

The first thing to do is to create the objects in the constructor and set their `OnChange` event handler:

```
constructor TMdArrow.Create (AOwner: TComponent);
begin
```

```
  ...
  // create the pen and the brush
  FPen := TPen.Create;
  FBrush := TBrush.Create;
  // set a handler for the OnChange event
  FPen.OnChange := RepaintRequest;
  FBrush.OnChange := RepaintRequest;
end;
```

These `OnChange` events are fired when one of the properties of these sub-objects changes; all we have to do is to ask the system to repaint our component:

```
procedure TMdArrow.RepaintRequest (Sender: TObject);
begin
  Invalidate;
end;
```

You must also add to the component a destructor, to remove the two graphical objects from memory (and free their system resources):

```
destructor TMdArrow.Destroy;
begin
  FPen.Free;
  FBrush.Free;
  inherited Destroy;
end;
```

The properties related to these two components require some special handling: instead of copying the pointer to the objects, we should copy the internal data of the object passed as parameter. The standard `:=` operation copies the pointers, so in this case we have to use the `Assign` method instead. Here is one of the two Set procedures:

```
procedure TMdArrow.SetPen (Value: TPen);
begin
  FPen.Assign(Value);
  Invalidate;
end;
```

Many `TPersistent` classes have an `Assign` method that should be used when we need to update the data of these objects. Now, to actually use the pen and brush for the drawing, you have to modify the `Paint` method, setting the `Pen` and the `Brush` properties of the component `Canvas` to the value of the internal objects before drawing any line:

```
procedure TMdArrow.Paint;
begin
  // use the current pen and brush
  Canvas.Pen := FPen;
```

```
Canvas.Brush := FBrush;
```

You can see an example of the new output of the component in Figure 13.9.

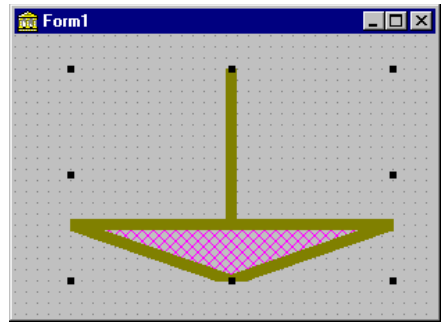


Figure 13.9: The output of the Arrow component with a thick pen and a special hatch brush. Image from the original book.

Defining a New Custom Event

To complete the development of the Arrow component, let's add a custom event. Most of the time, new components use the events of their parent classes. For example, in this component, I've made some standard events available simply by redeclaring them in the published section of the class:

```
type
  TMdArrow = class (TGraphicControl)
  published
    property OnClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
```

Thanks to this declaration, the above events (originally declared in a parent class) will now be available in the Object Inspector when the component is installed.

Sometimes, however, a component requires a custom event. To define a brand-new event, you first need to add to the class a field of the type of the event. This type is actually a method pointer type (see Chapter 3 for details). Here is the definition I've added in the private section of the `TMd4Arrow` class:

```
fArrowDblClick: TNotifyEvent;
```

In this case I've used the `TNotifyEvent` type, which has only a `Sender` parameter and is used by Delphi for many events, including `OnClick` and `OnDblClick` events. Using this field I've defined a very simple published property, with direct access to the field:

```
property OnArrowDblClick: TNotifyEvent
  read fArrowDblClick write fArrowDblClick;
```

Notice again the standard naming convention, with event names starting with *On*. The `fArrowDblClick` method pointer is activated (executing the corresponding function) inside the specific `ArrowDblClick` dynamic method. This happens only if an event handler has been specified in the program that uses the component:

```
procedure TMdArrow.ArrowDblClick;
begin
  if Assigned (FArrowDblClick) then
    FArrowDblClick (self);
end;
```

This method is defined in the protected section of the type definition to allow future sub-classes to both call and change it. Basically, the `ArrowDblClick` method is called by the handler of the `wm_LButtonDblClk` Windows message, but only if the double-click took place inside the arrow's point. To test this condition, we can use some of the Windows API's region functions.

note A *region* is an area of the screen enclosed by any shape. For example, we can build a polygonal region using the three vertices of the arrow-point triangle. The only problem is that to fill the surface properly, we must define an array of `TPoints` in a clockwise direction (see the description of the `CreatePolygonalRgn` in the Windows API Help for the details of this strange approach). That's what I did in the `ComputePoints` method.

Once we have defined a region, we can test whether the point where the double-click occurred is inside the region by using the `PtInRegion` API call. You can see the complete source code of this procedure in the following listing:

```
procedure TMdArrow.WMLButtonDblClk (
  var Msg: TWMLButtonDblClk); // message wm_LButtonDblClk;
var
  HRegion: HRgn;
begin
  // perform default handling
  inherited;

  // compute the arrowhead region
  HRegion := CreatePolygonRgn (
    fArrowPoints, 3, WINDING);
  try
```

```
    // check whether the click took place in the region
    if PtInRegion (HRegion, Msg.XPos, Msg.YPos) then
      ArrowDblClick;
  finally
    DeleteObject (HRegion);
  end;
end;
```

Registering Property Categories

We've added to this component some custom properties and a new event. If you arrange the properties in the Object Inspector by category, all the new elements will show up in the generic Miscellaneous category. Of course, this is far from ideal, but we can easily register the new properties in one of the available categories (listed in Delphi's Help file).

We can register a property (or an event) in a category by calling one of the four overloaded versions of the `RegisterPropertyInCategory` function[343] (defined in the DsgnIntf unit), and specifying the property name, its type, or the property name and the component it belongs to. For example, we can add the following lines to the `Register` procedure of the unit to register the `OnArrowDblClick` event in the Input category and the `Filled` property in the Visual category:

```
uses
  DsgnIntf;

procedure Register;
begin
  RegisterComponents('Md', [TMdArrow]);
  RegisterPropertyInCategory (
    TInputCategory, TMdArrow, 'OnArrowDblClick');
  RegisterPropertyInCategory (
    TVisualCategory, TMdArrow, 'Filled');
end;
```

We can also take one further step and create a brand-new category for the specific properties of our component, which can make it much simpler for a user to locate its specific features. To accomplish this we simply inherit a new class from the generic `TPropertyCategory` class and override a class function, `Name`. The `Name` will be used to indicate the category in the Object Inspector, while a second class function you can override, `Description`, is apparently not used.

---

343 Property categories are still available but seldom used, and not many components manage them as described here. I'd say you can safely skip using them.

Here is the code of the custom property category I've defined for the Arrow component:

```
interface

type
  TArrowCategory = class (TPropertyCategory)
    class function Name: string; override;
    class function Description: string; override;
  end;

implementation

class function TArrowCategory.Name: string;
begin
  Result := 'Arrow';
end;

class function TArrowCategory.Description: string;
begin
  Result := 'Properties of the Mastering Delphi Arrow component';
end;

procedure Register;
begin
  ...
  RegisterPropertyInCategory (
    TArrowCategory, TMdArrow, 'Direction');
  RegisterPropertyInCategory (
    TArrowCategory, TMdArrow, 'ArrowHeight');
  RegisterPropertyInCategory (
    TArrowCategory, TMdArrow, 'Filled');
end;
```

Notice that the `Filled` property was already registered in another existing category. This is not a problem, because the same property can show up multiple times in the Object Inspector under different categories, as you can see in Figure 13.10.

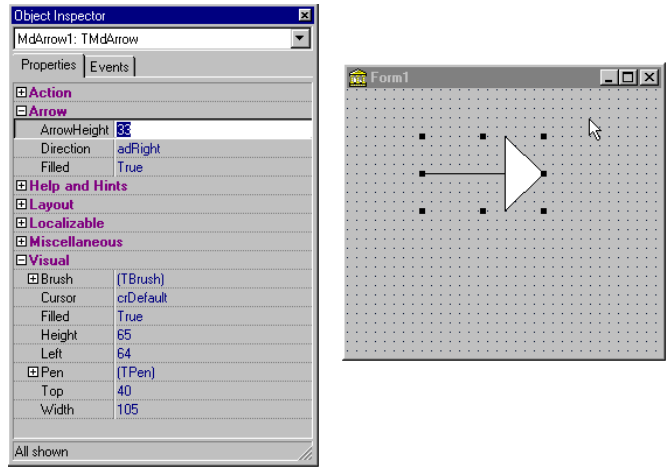


Figure 13.10: The Arrow component defines a custom property category, *Arrow*, as you can see in the Object Inspector. Image from the original book.

To test the arrow component I've written a very simple example program, ArrowDemo, which allows you to modify most of its properties at run time. This type of test, after you have written a component or while you are writing it, is very important.

Customizing Windows Controls

One of the most common ways of customizing existing components is to add some predefined behavior to their event handlers. Every time you need to attach the same event handler to components of different forms, you should consider adding the code of the event right into a subclass of the component. An obvious example is that of edit boxes accepting only numeric input. Instead of attaching to each of them a common `OnChar` event handler, we can define a simple new component. This component, however, won't handle the event; events are for component users only. Instead, the component can either handle the Windows message directly or override a method, as described in the next two sections.

Overriding Message Handlers: The Numeric Edit Box

To customize an edit box component to restrict the input it will accept, all you need to do is handle the `wm_Char` Windows messages that occur when the user presses any but a few specific keys (namely, the numeric characters).

One way to respond to a message for a given window (whether it's a form or a component) is to create a new *message-response* method that you declare using the `message` keyword. Delphi's message-handling system makes sure that your message-response method has a chance to respond to a given message before the form or component's default message handler does. As we'll see in the next section, instead of creating a new method (as we'll do here) you can override an existing virtual method that responds to a given message. Below is the complete code of the `TMdNumEdit` class:

```
type
  TMdNumEdit = class (TCustomEdit)
  private
    fInputError: TNotifyEvent;
  protected
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
  public
    procedure WmChar (var Msg: TWmChar); message wm_Char;
    constructor Create (Owner: TComponent); override;
  published
    property OnInputError: TNotifyEvent
      read fInputError write fInputError;
    property Value: Integer
      read GetValue write SetValue default 0;
    property AutoSelect;
    property AutoSize;
    property BorderStyle;
    // and so on...
```

This component inherits from `TCustomEdit` instead of `TEdit` so that it can hide the `Text` property and surface the Integer `Value` property instead. Notice that I don't create a new field to store this value, because we can use the existing (but now unpublished) `Text` property. To do this we'll simply convert the numeric value to and from a text string. The `TCustomEdit` class (or actually the Windows control it wraps) automatically paints the information from the `Text` property on the surface of the component:

```
function TMdNumEdit.GetValue: Integer;
begin
```

```
  // set to 0 in case of error
  Result := StrToIntDef (Text, 0);
end;

procedure TMdNumEdit.SetValue (Value: Integer);
begin
  Text := IntToStr (Value);
end;
```

The most important method is the response for the `wm_Char` message. In the body of this method, the component filters out all the non-numerical characters and raises a specific event in case of an error:

```
procedure TMdNumEdit.WmChar (var Msg: TWmChar);
begin
  if not (Char (Msg.CharCode) in ['0'..'9'])
    and not (Msg.CharCode = 8) then
  begin
    Msg.CharCode := 0;
    if Assigned (fInputError) then
      fInputError (Self);
  end;
end;
```

This method checks each character as the user enters it, testing for numerals and the Backspace key (which has an ASCII value of 8). The user should be able to use Backspace in addition to the system keys (the arrow keys and Del), so we need to check for that value. We don't have to check for the system keys, because they are surfaced by a different Windows message, `wm_SysChar`.

That's it. Now if you place this component on a form, you can type something in the edit box and see how it behaves. You might also want to attach a method to the `OnInputError` event to provide feedback to the user when a wrong key is typed.

Overriding Dynamic Methods: The Sound Button

Our next component, `TMdSoundButton`, plays one sound when you press the button and another sound when you release it. The user specifies each sound by modifying two `String` properties that name the appropriate WAV files for the respective sounds. Once again, we need to intercept and modify some system messages (`wm_LButtonDown` and `wm_LButtonUp`), but instead of handling the messages by writing a new message-response method, we'll override the appropriate *second-level* handlers.

note When most VCL components handle a Windows message, they call a *second-level* message handler (usually a dynamic method), instead of executing code directly in the message-response method. This makes it simpler for you to customize the component in a derived class. Typically, a second-level handler will do its own work and then call any event handler that the component user has assigned.

Here is the code of the `TMdSoundButton` class, with the two `protected` methods that override the second-level handlers, and the two string properties that identify the sound files. You'll notice that in the property declarations, we read and write the corresponding private fields without calling a Get or Set method, simply because we don't need to do anything special when the user makes changes to those properties.

```
type
  TMdSoundButton = class(TButton)
  private
    FSoundUp, FSoundDown: string;
  protected
    procedure MouseDown(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
    procedure MouseUp(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
  published
    property SoundUp: string
      read FSoundUp write FSoundUp;
    property SoundDown: string
      read FSoundDown write FSoundDown;
  end;
```

There are several reasons why overriding existing second-level handlers is generally a better approach than handling straight Windows messages. First, this technique is more sound from an object-oriented perspective. Instead of duplicating the message-response code from the base class and then customizing it, you're overriding a virtual method call that the VCL designers planned for you to override. Second, if someone needs to derive another class from one of your component classes, you'll want to make it as easy for them to customize as possible, and overriding second-level handlers is less likely to induce strange errors (if only because you're writing less code). Finally, this will make your component classes more consistent with the VCL—and therefore easier for someone else to figure out. Here is the code of the two second-level handlers:

```
uses
  MMSystem;

procedure TMdSoundButton.MouseDown(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  inherited MouseDown (Button, Shift, X, Y);
```

```
  PlaySound (PChar (FSoundDown), 0, snd_Async);
end;

procedure TMdSoundButton.MouseUp(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  inherited MouseUp (Button, Shift, X, Y);
  PlaySound (PChar (FSoundUp), 0, snd_Async);
end;
```

In both cases, you'll notice that we call the inherited version of the methods *before* we do anything else. For most second-level handlers, this is a good practice, since it ensures that we execute the standard behavior before we execute any custom behavior.

Next, you'll notice that we call the `PlaySound` Win32 API function to play the sound. You can use this function (which is defined in the MmSystem unit to play either WAV files or system sounds, as the SoundB example demonstrates. Here is a textual description of the form of this sample program (from the DFM file):

```
object MdSoundButton1: TMdSoundButton
  Caption = 'Press'
  SoundUp = 'RestoreUp'
  SoundDown = 'RestoreDown'
end
```

note Selecting a proper value for these sound properties is far from simple. Later in this chapter I'll show you how to add a property editor to the component to make the operation simpler.

A Non-Visual Dialog Component

The next component we'll examine is completely different from the ones we have seen up to now. After building window-based controls and simple graphic components, I'm now going to build a non-visual component.

The basic idea is that forms are components. When you have built a form that might be particularly useful in a number of projects, you can add it to the Object Repository or make a component out of it. The second approach is more complex than the first one, but it makes using the new form easier, and it allows you to distribute the form without its source code. As an example, I'll build a component based on a custom dialog box, trying to mimic as much as possible the behavior of standard Delphi dialog box components.

The first step in building a dialog box in a component is to write the code of the dialog box itself, using the standard Delphi approach. Just define a new form and work on it as usual. When a component is based on a form, you can almost visually design the component. Of course, once the dialog box has been built, you have to define a component around it in a non-visual way.

The standard dialog box I want to build is based on a list box, because it is common to let a user choose a value from a list of strings. I've customized this common behavior in a dialog box and then used it to build a component. The simple `ListBoxForm` form I've built has a list box and the typical OK and Cancel buttons, as shown in its textual description:

```
object MdListBoxForm: TMdListBoxForm
  BorderStyle = bsDialog
  Caption = 'ListBoxForm'
  object ListBox1: TListBox
    OnDblClick = ListBox1DblClick
  end
  object BitBtn1: TBitBtn
    Kind = bkOK
  end
  object BitBtn2: TBitBtn
    Kind = bkCancel
  end
end
```

The only method of this dialog box form relates to the double-click event of the list box, which closes the dialog box as though the user clicked the OK button:

```
procedure TMdListBoxForm.ListBox1DblClick(Sender: TObject);
begin
  ModalResult := mrOk;
end;
```

Once this form works, we can start changing its source code, adding the definition of a component and removing the declaration of the global variable for the form.

note For components based on a form, you can use two Pascal source code files: one for the form, and the other for the component encapsulating it. It is also possible to place both the component and the form in a single unit, as I've done for this example. In theory it would be even nicer to declare the form class in the implementation portion of this unit, hiding it from the users of the component. In practice this is not a good idea. To manipulate the form visually in the Form Designer, the form class declaration must appear in the interface section of the unit. The rationale behind this behavior of the Delphi IDE is that, among other things, this constraint minimizes the amount of code the module manager has to scan to find the form declaration, an operation required often to maintain the synchronization of the visual form with the form class definition.

The most important of these operations is the definition of the `TMdListBoxDialog` component, a non-visual component. What determines that this component is non-visual is that its immediate ancestor class is `TComponent`. The component has three published properties and a public one. These are the three published properties:

- `Lines` is a `TStrings` object, which is accessed via two methods, `GetLines` and `SetLines`. This second method uses the `Assign` procedure to copy the new values to the private field corresponding to this property. This internal object is initialized in the `Create` constructor and destroyed in the `Destroy` method.
- `Selected` is an integer that directly accesses the corresponding private field. It stores the selected element of the list of strings.
- `Title` is a string used to change the title of the dialog box.

The public property is `SelItem`, a read-only property that automatically retrieves the selected element of the list of strings. Notice that this property has no storage and no data: it simply accesses other properties, providing a virtual representation of data:

```
type
  TMdListBoxDialog = class (TComponent)
  private
    FLines: TStrings;
    FSelected: Integer;
    FTitle: string;
    function GetSelItem: string;
    procedure SetLines (Value: TStrings);
    function GetLines: TStrings;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function Execute: Boolean;
    property SelItem: string read GetSelItem;
  published
    property Lines: TStrings read GetLines write SetLines;
    property Selected: Integer read FSelected write FSelected;
    property Title: string read FTitle write FTitle;
  end;
```

Most of the code of this example is in the `Execute` method, a function that returns `True` or `False` depending on the modal result of the dialog box. This is consistent with the `Execute` method of most standard Delphi dialog box components. The `Execute` function creates the form dynamically, sets some of its values using the component's properties, shows the dialog box, and if the result is correct, updates the current selection:

```
function TMdListBoxDialog.Execute: Boolean;
```

```
var
  ListBoxForm: TListBoxForm;
begin
  if FLines.Count = 0 then
    raise EStringListError.Create ('No items in the list');
  ListBoxForm := TListBoxForm.Create (self);
  try
    ListBoxForm.ListBox1.Items := FLines;
    ListBoxForm.ListBox1.ItemIndex := FSelected;
    ListBoxForm.Caption := FTitle;
    if ListBoxForm.ShowModal = mrOk then
    begin
      Result := True;
      Selected := ListBoxForm.ListBox1.ItemIndex;
    end
    else
      Result := False;
  finally
    ListBoxForm.Free;
  end;
end;
```

Notice that the code is contained within a `try-finally` block, so if a run-time error occurs when the dialog box is displayed, the form will be destroyed anyway. I've also used exceptions to raise an error if the list is empty when a user runs it. This error is by design, and using an exception is a good technique to enforce it. The other methods of the component are quite straightforward. The constructor creates the `FLines` string list, which is deleted by the destructor; the `GetLines` and `SetLines` methods operate on the string list as a whole, and the `GetSelItem` function (listed below) returns the text of the selected item:

```
function TMdListBoxDialog.GetSelItem: string;
begin
  if (Selected >= 0) and (Selected < FLines.Count) then
    Result := FLines [Selected]
  else
    Result := '';
end;
```

Of course, since we are manually writing the code of the component and adding it to the source code of the original form, we have to remember to write the `Register` procedure.

Using the Non-Visual Component

Once you've done that and the component is ready, you must provide a bitmap. For non-visual components, bitmaps are very important because they are used not only

for the Component Palette, but also when you place the component on a form. Now let's prepare the bitmap, install the component, and write a simple project to test it. The form of this test program has a button, an edit box, and our new non-visual component, as you can see in Figure 13.11.

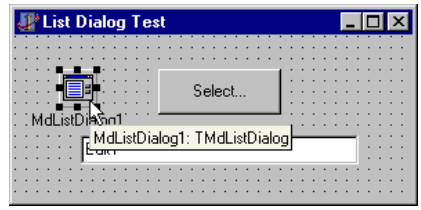


Figure13.11: The form of the ListDialDemo example, with the new non-visual component. Image from the original book.

Now you can write a few lines of code, corresponding to the `OnClick` event of the button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // select the text of the edit,
  // if corresponding to one of the strings
  MdListDialog1.Selected :=
    MdListDialog1.Lines.IndexOf (Edit1.Text);
  // run the dialog and get the result
  if MdListDialog1.Execute then
    Edit1.Text := MdListDialog1.SelItem;
end;
```

That's all you need to run the dialog box we have placed in the component, as you can see in Figure 13.12. As you've seen, this is an interesting approach to the development of some common dialog boxes.

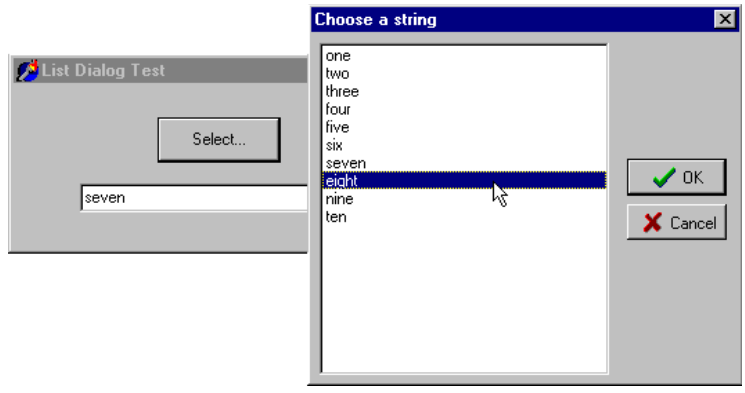


Figure 13.12: The ListDialDemo example shows the dialog box I've encapsulated in the ListDial component. Image from the original book.

Defining Custom Actions

Besides defining custom components, you can define and register new standard actions, which will be made available in the Action Editor of the Action List component. Creating new actions is not complex. You have to inherit from the `TAction` class and override some of the methods of the base class[344].

There are basically three methods to override. The `HandlesTarget` function returns whether the action object wants to handle the operation for the current target, by default the control with the focus. The `UpdateTarget` procedure can set the user interface of the controls connected with the action, eventually disabling the action if the operation is currently not available. Finally, you can implement the `ExecuteTarget` method to determine the actual code to execute, so that the user can simply select the action and doesn't have to implement it.

To show you this approach in practice, I've implemented the three cut, copy, and paste actions for a list box, in a way similar to what the VCL does for an edit box (although I've actually simplified the code a little). I've written a base class, which has some common code to handle the status of the actions, and three derived classes with the `ExecuteTarget` code. Here are the four classes:

344 As mentioned earlier, I think the ActionList architecture (and it's extensions with the ActionManager) is a very important element of the VCL architecture). Therefore, the ability of creating custom actions is relevant.

```
type
  TMdListAction = class (TAction)
  public
    function HandlesTarget (Target: TObject): Boolean; override;
    procedure UpdateTarget (Target: TObject); override;
  end;

  TMdListCutAction = class (TMdListAction)
  public
    procedure ExecuteTarget(Target: TObject); override;
  end;

  TMdListCopyAction = class (TMdListAction)
  public
    procedure ExecuteTarget(Target: TObject); override;
  end;

  TMdListPasteAction = class (TMdListAction)
  public
    procedure UpdateTarget (Target: TObject); override;
    procedure ExecuteTarget (Target: TObject); override;
  end;
```

The `HandlesTarget` method is implemented only for the base class, and it activates the actions only if the target control is a list box and this list box has the focus:

```
function TMdListAction.HandlesTarget (Target: TObject): Boolean;
begin
  Result := (Target is TListBox) and
    TListBox(Target).Focused;
end;
```

The `UpdateTarget` method, instead, has two different implementations. The default one is provided by the base class and used by the copy and cut actions. These actions are enabled only if the target list box has at least one item and an item is currently selected. The status of the Paste action, instead, depends on the Clipboard status:

```
procedure TMdListAction.UpdateTarget (Target: TObject);
begin
  Enabled := ((Target as TListBox).Items.Count > 0) and
    ((Target as TListBox).ItemIndex >= 0);
end;

procedure TMdListPasteAction.UpdateTarget (Target: TObject);
begin
  Enabled := Clipboard.HasFormat (CF_TEXT);
end;
```

Finally, the three `ExecuteTarget` methods simply perform the corresponding actions on the target list box:

```
procedure TMdListCopyAction.ExecuteTarget (Target: TObject);
begin
  with Target as TListBox do
    Clipboard.AsText := Items [ItemIndex];
end;

procedure TMdListCutAction.ExecuteTarget(Target: TObject);
begin
  with Target as TListBox do
  begin
    Clipboard.AsText := Items [ItemIndex];
    Items.Delete (ItemIndex);
  end;
end;

procedure TMdListPasteAction.ExecuteTarget(Target: TObject);
begin
  (Target as TListBox).Items.Add (Clipboard.AsText);
end;
```

Once you've written this code in a unit and added it to a package (in this case the MdPack package), the final step is to register the new custom actions in a given category. This is indicated as the first parameter of the `RegisterActions` procedure[345], while the second is the list of action classes to register:

```
procedure Register;
begin
  RegisterActions ('ListBox',
    [TMdListCutAction, TMdListCopyAction, TMdListPasteAction],
    nil);
end;
```

To test the use of these three custom actions I've written the ListTest example. This program has two list boxes and a toolbar with three buttons connected with the three custom actions and an edit box for entering new values. The program allows a user to cut, copy, and paste list box items. Nothing special, you might think, but the strange fact is that the program has absolutely no code!

Writing Property Editors

Writing components is certainly an effective way to customize the Delphi environment, helping developers to build applications faster without requiring a detailed

345 This is now in the `System.Actions` unit.

knowledge of low-level techniques. The Delphi environment is also quite open to extensions. In particular, it is easy to extend the Object Inspector by writing custom property editors and to extend the Form Designer by adding component editors.

note Along with these techniques, there are a number of internal interfaces Delphi offers to add-on tool developers. Using these interfaces requires an advanced understanding of how the Delphi environment works and a fairly good knowledge of many advanced techniques. As these advanced techniques are not needed by every Delphi developer, they are not discussed in this book. You can find a detailed description of the traditional Delphi design-time interfaces in *Delphi Developer's Handbook* (Sybex, 1998). As far as I know, there are no books yet detailing the ToolsApi introduced in Delphi 4[346]. You can find technical information and some examples of these techniques on my Web site, along with links to other sites where these techniques are presented.

Every property editor must be a subclass of the abstract `TPropertyEditor` class, which is defined in the DsgnIntf (*Design Interface*) system unit. However, Delphi already defines some specific property editors for strings (the `TStringProperty` class), integers (the `TIntegerProperty` class), characters (the `TCharProperty` class), enumerations (the `TEnumProperty` class), sets (the `TSetProperty` class), and many others for colors, fonts, pens, string lists, and so on.

In any custom property editor, you have to redefine the `GetAttributes` function so it returns a set of values indicating the capabilities of the editor. The most important attributes are `paValueList` and `paDialog`. The `paValueList` attribute indicates that the Object Inspector will show a combo box with a list of values (eventually sorted if the `paSortList` attribute is set) provided by overriding the `GetValues` method. The `paDialog` attribute style activates an ellipsis button in the Object Inspector which executes the `Edit` method of the editor.

An Editor for the Sound Properties

The sound button we built earlier had two sound-related properties: `SoundUp` and `SoundDown`. These were actually strings, so we were able to display them in the Object Inspector using a default property editor. However, requiring the user to type the name of a system sound or an external file is not very friendly, and it's a bit error-prone.

346 There is documentation, if not books, covering Delphi IDE ToolsAPI. A good reference is https://github.com/Embarcadero/OTAPI-Docs

When you need to select a file for a string property, you can reuse an existing property editor, the `TMPFilenameProperty` class. All you have to do is register this editor for the property using the special `RegisterPropertyEditor` procedure, as in:

```
RegisterPropertyEditor (
   TypeInfo (string), TDdhSoundButton,
  'SoundUp', TMPFileNameProperty);
```

This editor allows you to select a file for the sound, but we want to be able to choose the name of a system sounds as well. As described earlier, system sounds are predefined names of sounds connected with user operations, associated with actual sounds files in the Sound page of Windows Control Panel. For this reason, instead of using this simple approach I'll build a more complex property editor. My editor for sound strings allows a user to either choose a value from a drop-down list or display a dialog box from which to load and test a sound (from a sound file or a system sound). For this reason, the property editor provides both an `Edit` and a `GetValues` method:

```
type
  TSoundProperty = class (TStringProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure GetValues(Proc: TGetStrProc); override;
    procedure Edit; override;
  end;
```

note The default Delphi convention is to name a property editor class with a name ending with *Property* and all component editors with a name ending with *Editor*.

The `GetAttributes` function combines both the `paValueList` (for the drop down list) and the `paDialog` attributes (for the custom edit box), and also sorts the lists and allows the selection of the property for multiple components:

```
function TSoundProperty.GetAttributes:
  TPropertyAttributes;
begin
  // editor, sorted list, multiple selection
  Result := [paDialog, paMultiSelect,
    paValueList, paSortList];
end;
```

The `GetValues` method simply calls [I guess I originally wanted to write "many times", as there is a single procedure passed as parameter, and the program calls it over and over- Marco] the procedure it receives as parameter multiple times, once for each string it wants to add to the drop-down list (as you can see in Figure 13.13):

```
procedure TSoundProperty.GetValues(Proc: TGetStrProc);
begin
  // provide a list of system sounds
  Proc ('Maximize');
  Proc ('Minimize');
  Proc ('MenuCommand');
  Proc ('MenuPopup');
  Proc ('RestoreDown');
  Proc ('RestoreUp');
  Proc ('SystemAsterisk');
  Proc ('SystemDefault');
  Proc ('SystemExclamation');
  Proc ('SystemExit');
  Proc ('SystemHand');
  Proc ('SystemQuestion');
  Proc ('SystemStart');
  Proc ('AppGPFault');
end;
```

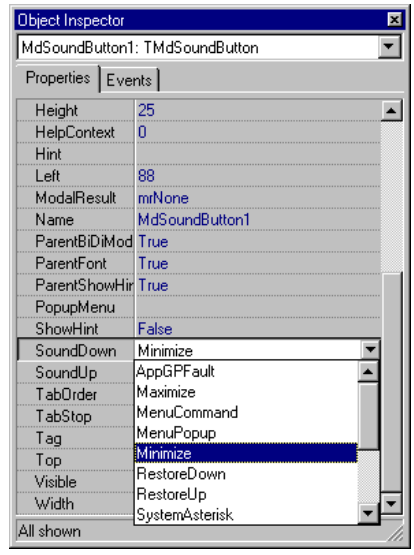


Figure 13.13: The list of sounds provides a hint for the user, who can also type in the property value or double-click to activate the editor (shown later in Figure 13.14). Image from the original book.

note A better approach would be to extract these values from the Windows Registry, where all these names are listed.

note If you want to further extend this example, Delphi 5 allows you to add graphics to the drop-down list displayed in the Object Inspector—if you can decide which graphics to attach to particular sounds.

The `Edit` method is very straightforward, as it simply creates and displays a dialog box. You'll notice that we could have just displayed the Open dialog box directly, but we decided to add an intermediate step to allow the user to test the sound. This is similar to what Delphi does with graphic properties. You open the preview first, and load the file only after you've confirmed that it's correct. The most important step is to load the file and test it before you apply it to the property. Here is the code of the `Edit` method:

```
procedure TSoundProperty.Edit;
begin
  SoundForm := TSoundForm.Create (Application);
  try
    SoundForm.ComboBox1.Text := GetValue;
    // show the dialog box
    if SoundForm.ShowModal = mrOK then
      SetValue (SoundForm.ComboBox1.Text);
  finally
    SoundForm.Free;
  end;
end;
```

The `GetValue` and `SetValue` methods called above are defined by the base class, the string property editor. They simply read and write the value of the current component's property that we are editing. As an alternative, you can access the component you're editing by using the `GetComponent` method (which requires a parameter indicating which of the selected components you are working on—0 indicates the first component). When you access the component directly, you also need to call the `Modified` method of the `Designer` object (a property of the base class property editor). We don't need this `Modified` call in the example, as the base class `SetValue` method does this automatically for us.

The `Edit` method above displays a dialog box, a standard Delphi form that is built visually, as always, and added to the package hosting the design-time components. The form is quite simple; a ComboBox displays the values returned by the `GetValues` method, and four buttons allow you to open a file, test the sound, and terminate the dialog box by accepting the values or canceling. You can see an example of the dialog box in Figure 13.14. Providing a drop-down list of values *and* a dialog box for editing a property causes the Object Inspector to display only the arrow button that indicates a drop-down list, and to omit the ellipsis button to indicate that a dialog box editor is available.

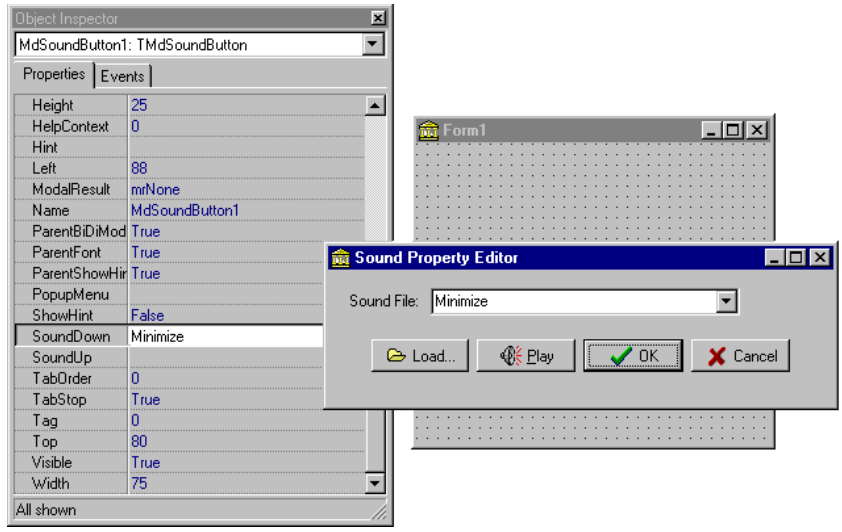


Figure 13.14: The sound property editor's form displays a list of available sounds, but it also lets you load a file and hear the selected sound. Image from the original book.

The first two buttons of the form each have a simple method assigned to their `OnClick` event:

```
procedure TSoundForm.btnLoadClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    ComboBox1.Text := OpenDialog1.FileName;
end;

procedure TSoundForm.btnPlayClick(Sender: TObject);
begin
  PlaySound (PChar (ComboBox1.Text), 0, snd_Async);
end;
```

Unfortunately, I've not found a simple way to determine whether a sound is properly defined and is available The `PlaySound` function returns an error code when played synchronously, but only if it can't find the default system sound it attempts to play if it can't find the sound you asked for. If the requested sound is not available, it plays the default system sound and doesn't return the error code. `PlaySound` looks for the sound in the Registry first, and if it doesn't find the sound there, checks to see if the specified sound file exists.

Installing the Property Editor

After you've written this code, you can install the component and its property editor in Delphi. To accomplish this, you have to add the following statement to the `Register` procedure of the unit:

```
procedure Register;
begin
  RegisterPropertyEditor (TypeInfo(string),
    TMdSoundButton, 'SoundUp', TSoundProperty);
  RegisterPropertyEditor (TypeInfo(string),
    TMdSoundButton, 'SoundDown', TSoundProperty);
end;
```

This call registers the editor specified in the last parameter for use with properties of type `string` (the first parameter), but only for a specific component and for a property with a specific name. These last two values can be omitted to provide more general editors. Registering this editor allows the Object Inspector to show a list of values and the dialog box called by the `Edit` method.

To install this component we can simply add its source code file into an existing or new package. Instead of adding this unit and the others of this chapter to the `MdPack` package, I built a second package, containing all the add-ins built in this chapter. The package is named `MdDesPk` (which stands for "Mastering Delphi design package"). What's new about this package is that I've compiled it using the `{$DESIGNONLY}` compiler directive. This directive is used to mark packages that interact with the Delphi environment, installing components and editors, but are not required at run time by applications you've built.

note The source code of all of the add-on tools is in the MdDesPk sub-directory, along with the code of the package used to install them. There are no examples demonstrating how to use these design-time tools, because all you have to do is select the corresponding components in the Delphi environment and see how they behave.

The property editor's unit uses the SoundB unit, which defines the `TMdSoundButton` component. For this reason the new package should refer to the existing package. Here is its initial code (I'll add other units to it later in this chapter):

```
package MdDesPk;

{$R *.RES}
{$ALIGN ON}
...
{$DESCRIPTION 'Mastering Delphi DesignTime Package'}
{$DESIGNONLY}
```

```
requires
  vcl50,
  Mdpack;

contains
  PeSound in 'PeSound.pas',
  PeFSound in 'PeFSound.pas' {SoundForm};
```

Writing a Component Editor

Using property editors allows the developer to make a component more user-friendly. In fact, the Object Inspector represents one of the key pieces of the user interface of the Delphi environment, and Delphi developers use it quite often. However, there is a second approach you can adopt to customize how a component interacts with Delphi: write a custom component editor.

Just as property editors extend the Object Inspector, component editors extend the Form Designer. In fact, when you right-click on a form at design time, you see some default menu items, plus the items added by the component editor of the selected component. Examples of these menu items are those used to activate the Menu Designer, the Fields Editor, the Visual Query Builder, and other editors of the environment. At times, displaying these special editors becomes the default action of a component when it is double-clicked.

Common uses of component editors include adding an About box with information about the developer of the component, adding the component name, and providing specific wizards to set up its properties.

Sub-Classing the TComponentEditor Class

A component editor should inherit from the `TComponentEditor` class. This class has four virtual methods you can override (plus a couple of less important methods I've decided to skip here):

- `GetVerbCount` returns the number of menu items to add to the local menu of the Form Designer when the component is selected.
- `GetVerb` is called once for each new menu item, and it should return the text that should go in the local menu for each.

- `ExecuteVerb` is called when one of the new menu items is selected. The number of the item is passed as parameter.
- `Edit` is called when the user double-clicks on the component in the Form Designer to activate the default action.

Once you get used to the idea that a verb is nothing but a new menu item with a corresponding action to execute, the names of the methods of this interface become quite intuitive. This interface is actually much simpler than those of property editors we've seen before.

A Component Editor for the ListDialog

Now that I've introduced the key ideas about writing component editors, we can look at an example, an editor for the ListDialog component built earlier. In my component editor I simply want to be able to show an About box, add a copyright notice to the menu (an improper but very common use of component editors), and allow users to perform a special action: previewing the dialog box connected with the dialog component). I also want to change the default action to simply show the About box after a beep (which is not particularly useful, but it demonstrates the technique).

To implement this property editor, the program must override the four methods listed above:

```
uses
  DsgnIntf;

type
  TMdListCompEditor = class (TComponentEditor)
    function GetVerbCount: Integer; override;
    function GetVerb(Index: Integer): string; override;
    procedure ExecuteVerb(Index: Integer); override;
    procedure Edit; override;
  end;
```

The first method simply returns the number of menu items I want to add to the local menu:

```
function TMdListCompEditor.GetVerbCount: Integer;
begin
  Result := 3;
end;
```

This method is called only once, before displaying the menu. The second method, instead, is called once for each menu item, so in this case it is called three times:

```
function TMdListCompEditor.GetVerb (
  Index: Integer): string;
begin
  case Index of
    0: Result := 'MdTabbedList (©Cantù)';
    1: Result := '&About this component...';
    2: Result := '&Preview...';
  end;
end;
```

The effect of this code is to add the menu items to the local menu of the form, as you can see in Figure 13.15. Selecting any of these menu items simply activates the `ExecuteVerb` method of the component editor:

```
procedure TMdListCompEditor.ExecuteVerb (
  Index: Integer);

begin
  case Index of
    0..1: MessageDlg (
      'This is a simple component editor'#13 +
      'built by Marco Cantù'#13 +
      'for the book "Mastering Delphi"',
      mtInformation, [mbOK], 0);
    2: with Component as TMdListDialog do
      Execute;
  end;
end;
```

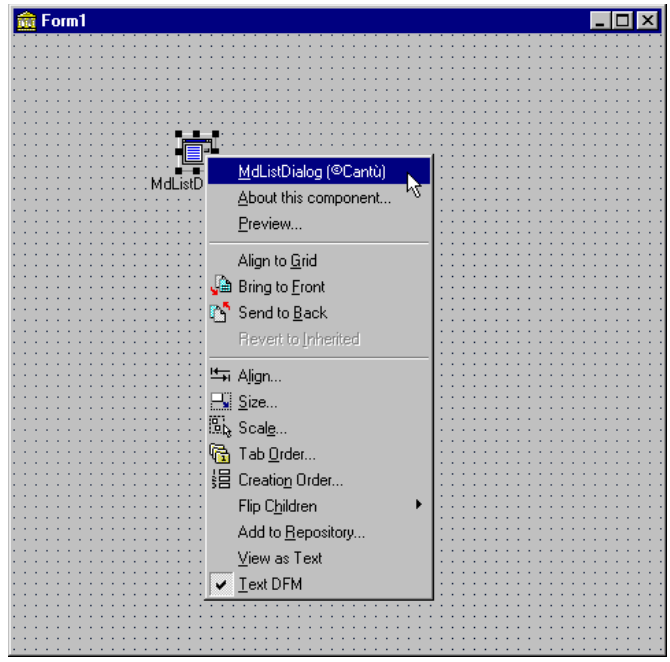


Figure 13.15: The custom menu items added by the property editor of the ListDialog component. Image from the original book.

I decided to handle the first two items in a single branch of the `case` statement, although I could have skipped the code for the copyright notice item. The other command changes calls the `Execute` method of the component we are editing, determined using the `Component` property of the `TComponentEditor` class. Knowing the type of the component, we can easily access its methods after a dynamic type cast.

The last method refers to the default action of the component and is activated by double-clicking on it in the Form Designer:

```
procedure.Edit;
begin
  // produce a beep and show the About box
  Beep;
  ExecuteVerb (0);
end;
```

Registering the Component Editor

To make this editor available to the Delphi environment we need to register it. Once more we can add to its unit a `Register` procedure and call a specific registration procedure for component editors:

```
procedure Register;
begin
  RegisterComponentEditor (
    TMdListDialog, TMdListCompEditor);
end;
```

I've added this unit to the `MdDesPk` package, which includes all of the design-time extensions of this chapter. After installing and activating this package you can create a new project, place a tabbed list component in it, and experiment with it.

What's Next

In this chapter we have seen how to define various types of properties, how to add events, and how to define and override component methods. We have seen different examples of components, including simple changes to existing ones, new graphical components, and, in the final section, a dialog box inside a component. While building these components, we have faced some new Windows programming challenges. In general, programmers often need to use the Windows API directly when writing new Delphi components.

Writing components is a very handy technique for reusing software, but to make your components easier to use you should try to integrate them as much as possible within the Delphi environment, writing property editors and component editors

There are many more extensions of the Delphi IDE you can write, including custom Wizards. Delphi, in fact, has a quite extensive ToolsApi, which is partially documented in some third party books, including my *Delphi Developer's Handbook* (Sybex, 1998). I've personally built a number of Delphi extensions, some of which are available on my Web site, `www.marcocantu.com`[347].

After discussing components and delving a little into the Delphi environment, the next chapter focuses on Delphi DLLs. We have already met DLLs in many chapters

347 You can see https://www.marcocantu.com/tools/ but also https://github.com/marcocantu/cantools

in the past, and it is time for a detailed discussion of their role and how to build them. In the same chapter I'll also further discuss the use of Delphi packages introduced in the current chapter, as they are a special type of DLLs.

Chapter 14: Dynamic Link Libraries And Packages

Windows executable files come in two flavors: *programs* and *dynamic link libraries* (DLLs). When you write a Delphi application, you typically generate a program file, an EXE. However, Delphi applications often use calls to functions stored in DLLs. Each time you call a Windows API function directly, you actually access a DLL. Delphi allows programmers to use run-time DLLs also for the component library. When you create a package, you basically create a DLL. Delphi can also gen-

erate plain dynamic link libraries. The New page of the Object Repository includes a DLL skeleton generator, which generates very few lines of source code.

It is very simple to generate a DLL in the Delphi environment. However, some problems arise from the nature of DLLs. Writing a DLL in Windows is not always as simple as it seems, because the DLL and the calling program have to agree on calling conventions, parameters' types, and other details. This chapter covers the basics of DLL programming from the Delphi point of view, and it provides some simple examples of what you can place in a Delphi DLL. While discussing the examples I'll also refer to other programming languages and environments, simply because one of the key reasons for writing a procedure in a DLL is to be able to call it from a program written in a different language.

The last part of the chapter will focus on a specific type of dynamic link library, the Delphi package. These packages are not as easy to use as they first seem, and it took Delphi programmers several months to figure out how to leverage them effectively. Here I'm going to share with you some of these interesting tips and techniques.

The Role of DLLs in Windows

Before delving into the development of DLLs in Delphi and other programming languages, I'll give you a short technical overview of DLLs in Windows, highlighting the key elements. We will start by looking at dynamic linking, then see how Windows uses DLLs, explore the differences between DLLs and executable files, and end with some general rules to follow when writing DLLs.

What Is Dynamic Linking?

First of all, you need to understand the difference between static and dynamic linking of functions or procedures. When a subroutine is not directly available in a source file, the compiler adds the subroutine to an internal table, which includes all external symbols. Of course, the compiler must have seen the declaration of the subroutine and know about its parameters and type, or it will issue an error.

After compilation of a normal—*static*—subroutine, the linker fetches the subroutine's compiled code from a Delphi compiled unit (or static library) and adds it to the executable. The resulting EXE file includes all the code of the program and of the units involved. The Delphi linker is smart enough to include only the minimum

amount of code of the units used by the program and to link only the functions and methods that are actually used.

note A notable exception to this rule is the inclusion of virtual methods. The compiler cannot determine in advance which virtual methods the program is going to call, so it has to include them all. For this reason, programs and libraries with too many virtual functions tend to generate larger executable files. While developing the VCL, the Borland developers had to balance the flexibility obtained with virtual functions against the reduced size of the executable files obtained by limiting the virtual functions.

In the case of dynamic linking, which occurs when your code calls a DLL-based function, the linker simply uses the information in the `external` declaration of the subroutine to set up some tables in the executable file. When Windows loads the executable file in memory, it first loads all the required DLLs, and then the program starts. During this loading process, Windows fills the program's internal tables with the addresses of the functions of the DLLs in memory[348]. If for some reason the DLL is not found, the program won't even start, often complaining with nonsense error messages (such as the famous "a device attached to your system is not functioning").

Each time the program calls an external function, it uses this internal table to forward the call to the DLL code (which is now located in the program's address space). Note that this scheme does not involve two different applications. The DLL becomes part of the running program and is loaded in the same address space. All the parameter passing takes place on the stack of the application (since the DLL doesn't have a separate stack).

You can see a sketch of how the program calls statically or dynamically linked functions in Figure 14.1. Notice that I haven't yet discussed compilation of the DLL—because I wanted to focus on the two different linking mechanisms first.

note The term *dynamic linking*, when referring to DLLs, has nothing to do with the late-binding feature of object-oriented programming languages. Virtual and dynamic methods in Object Pascal have nothing to do with DLLs. Unfortunately, the same term is used for both kinds of procedures and functions, which causes a lot of confusion. When I speak of dynamic linking in this chapter, I am referring not to polymorphism but to DLL functions.

348 There is now an exception to this rule for functions maker for delayed binding. The issue is that with load time mapping, if you are using an older version of the library missing a function (maybe because you are using an older version of Windows) the loader will stop with an error due to the missing function. With delayed loading, the DLL is still required to load, but the function mapping is done at the first invocation, causing an error but not preventing the program to run.

There is another, less common approach to using DLLs, which is even more dynamic than the one we have just discussed. In fact, at run time, you can load a DLL in memory, search for a function (provided you know its name), and call the function by name. This approach requires more complex code and is generally slower. On the positive side, you don't need to have the DLL available to start the program. We will use this approach in the DynaCall example later in the chapter.

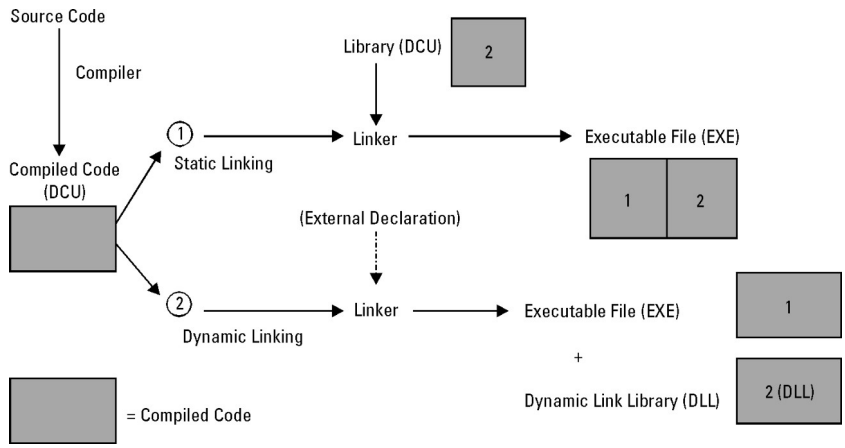


Figure 14.1: Static and dynamic linking in Windows. Graphic from the original book.

What Are DLLs For?

Now that you have a general idea of how DLLs work, we can focus on the reasons for using them in Windows:

- If different programs use the same DLL, the DLL is loaded in memory only once, thus saving system memory. DLLs are mapped into the private address space of each process (each running application), but their code is loaded in memory only once. The operating system will try to load the DLL at the same address in each application's address space, but if that address is not available in a particular application's virtual address space, the DLL code image for that process will have to be relocated. Note that the relocation happens on a per-process basis, not system-wide.
- You can provide a different version of a DLL, replacing the current one. If the subroutines in the DLL have the same parameters, you can run the program with the new version of the DLL without having to recompile it. If the DLL has new

subroutines, it doesn't matter at all. Problems might arise only if a routine in the older version of the DLL is missing in the new one.

Versions of DLL Parameters

The most recent Windows API functions often use as a single parameter the pointer to a data structure, which includes the actual parameters. This approach allows the DLL creator to add new parameters to the data structure, without affecting the existing code. Typically, in these cases, the first parameter of the data structure holds its size, which is used to indicate the *version* of the structure. This way, the DLL can determine which version of the data structure the application refers to, simply by looking at this size/version parameter. This is a useful approach to follow if you think the parameters of a function are likely to change in future versions of the DLL.

For examples, you can look at the Windows API functions for the common dialog boxes (such as `GetOpenFileName` or `ChooseColor`) in the help file, but many other new API functions use the same approach.

These generic advantages apply in several cases. If you have a complex algorithm, or some complex forms required by several applications, you can store them in a DLL. This will let you reduce the executable's size and save some memory when you run several programs using those DLLs at the same time.

The second advantage is particularly applicable to complex applications. If you have a very big program that requires frequent updates and bug fixes, dividing it into several executable files and DLLs allows you to distribute only the changed portions instead of one single large executable. This makes sense for Windows system libraries in particular. If Borland (Inprise) releases a new version of the Database Engine libraries or writes new SQL Links to access other SQL server databases, you won't need to recompile your application to take advantage of the changes.

Another common technique is to use DLLs to store nothing except resources. You can build different versions of a DLL containing strings for different languages and then change the language at run time, or you can prepare a library of icons and bitmaps and then use them in different applications. The development of language-specific versions of a program is particularly important, and Delphi 5 includes support for it through the Integrated Translation Environment (ITE), described in Chapter 19[349].

Another key advantage, as I already mentioned, is that DLLs are independent of the programming language. Most Windows programming environments, including

---

349 This feature is now deprecated, as already mentioned.

most macro languages in end-user applications, allow a programmer to call a subroutine stored in a DLL. This means you can build a DLL in Delphi and call it from C++, Visual Basic, Excel, WordPerfect, and many other Windows applications.

Understanding System DLLs

The Windows system DLLs take advantage of all the key benefits of DLLs I've just highlighted. For this reason, it is worth examining them. First of all, Windows has many system DLLs. The three central portions of Windows—Kernel, User, and GDI—are implemented using DLLs[350].

In Windows 95 and 98, the three key system libraries are duplicated in 16-bit versions (`KRNL386.EXE`, `USER.EXE`, and `GDI.EXE`) and 32-bit versions (`KERNEL32.DLL`, `USER32.DLL`, and `GDI32.DLL`). These two versions often call each other, in a process called *thunking*. In Windows NT (and Windows 2000), the system libraries have only 32-bit code. Other system DLLs are operating-system extensions, such as the DLLs for common dialog boxes and controls, DDE, OLE, device drivers, fonts, ActiveX controls, and many others.

In the case of Windows itself, using DLLs is extremely important. In fact, DLLs are one of the key technical foundations of the Windows operating systems. Since each application uses the system DLLs for anything from creating a window to producing output, every program is linked to those DLLs. Let's take a minute to look at why you might have different versions of the same library.

First, consider device drivers. When you change your printer, you do not need to rebuild your application or even buy a new version of the Windows GDI library, which manages the printer output. You only need to provide a specific driver, which is a DLL called by the GDI, to access your printer. Each printer type has its own driver DLL, which makes the system extremely flexible.

From a different point of view, version handling is important for the system itself. If you have an application compiled for Windows 3.1, you should be able to run it on any Win32 platform. Each version of Windows has different system code (and Win32 16-bit support actually corrects some Windows 3.1 quirks), but since each new version contains the older API functions, the old code still works, even though it cannot take advantage of the new API functions. However, old code can indeed take advantage of new features when an existing function's code changes. An obvious example is the user interface: If you build an application for Windows 3.1, you

350 These three fundamental DLLs are still a the core of Windows, despite the many enhancements to the Windows API and architecture.

can run it on Windows 95 and 98, and it will automatically have different user interface elements. You have not recompiled your program; it used the features of the new system libraries, which were linked dynamically to it.

The system DLLs are also used as system-information archives. For example, the USER DLL maintains a list of all the active windows in the system, and the GDI DLL holds the list of active pens, brushes, icons, bitmaps, and the like. The free memory area of these two system DLLs is usually called "free system resources" and plays a very important role in Windows[351].

Differences between DLLs and EXEs

Now that you know the basic elements of dynamic linking and some of the reasons to use the technique, we can focus on the difference between a normal executable file (an EXE file) and a dynamic link library (a DLL file). For the most part, the internal structure of an EXE file and a DLL file is the same. It is when a DLL is loaded into memory that things change.

As I mentioned earlier, Windows loads the code of a DLL into memory only once. The same happens with an executable file, even if you run multiple copies. In both cases a mechanism for counting module usage ensures that the code of the DLL is discarded when all programs using it terminate.

The key difference between programs and DLLs is that a DLL, even when loaded in memory, is not a running program. It is only a collection of procedures and functions that other programs can call. These procedures and functions use the stack of the calling program (the *calling thread*, to be precise). So another key difference between a program and a library is that a library doesn't create its own stack—it uses the stack of the program calling it. In Win32, because a DLL is loaded into the application's address space, any memory allocations of the DLL or any global data it creates reside in the address space of the main process.

351 Moving towards a full 32-bit system first and a 64-bit operating system now has significantly reduced the limitations of some of these system memory areas, but depleting them can still negative affect performance.

Rules for Delphi DLL Writers

What I've described so far can be summarized in some rules for DLL programmers. A DLL function or procedure to be called by external programs must follow these guidelines:

- It has to be listed in the DLL's `exports` clause. This makes the routine visible to the outside world.
- Exported functions should also be declared as `stdcall`, to use the standard Win32 parameter-passing technique instead of the optimized `register` parameter-passing technique (which is the default in Delphi).
- The types of the parameters of a DLL should be the default Windows types, at least if you want to be able to use the DLL within other development environments. There are further rules for exporting strings, as we'll see in the FirstDll example.
- A DLL can use global data that won't be shared by calling applications. Each time an application loads a DLL, it stores the DLL's global data in its own address space, as we will see in the DllMem example.

Win16 and Win32 DLLs

Another important aspect of DLLs is that in Windows, DLLs come in two different flavors[352]. There are Windows 3.1 (Win16) DLLs, and Windows NT, 95, or 98 (Win32) DLLs. Libraries written with the 16-bit version of Delphi are of the first kind. Libraries compiled with 32-bit versions of Delphi are of the second kind.

Unfortunately, as I've already mentioned, 16-bit and 32-bit DLLs are *not* compatible. For example, you cannot call a 16-bit DLL from a 32-bit Delphi program. This is not a Delphi limitation but a general Windows problem. There is actually a solution: You can use Microsoft's thunk compiler to create the proper entry points for the different DLL type. This is what Windows 95 and 98 do to call 16-bit system libraries

352 While 16-bit support has now been removed, the system DLLs still come in two different flavours, the 64-bit version Windows is now based on and the 32-bit version for compatibility with 32-bit applications. What's important, in the operating system and in your code, is that an executable with a specific "bitness" can only directly loads DLLs with the same "bitness". That is true for packages you load in the Delphi IDE, for third party DLLs, for database client drivers, and any other dynamically linked binary.

from a 32-bit application or to call new 32-bit system libraries from old 16-bit applications.

However, using the thunk mechanism is quite complex and allows your 32-bit applications to run only under Windows 95 and 98, but not under Windows NT. In most cases, a solution to this problem is to create a 16-bit application that accesses the 16-bit DLL, and then use one of the available techniques (memory-mapped files or the `WM_COPYDATA` message) to let the 16-bit application share data with the 32-bit version of your program. Thunking is also very slow to execute, because it requires kernel mode switching.

Although parts of the core of Windows 98 are still made of 16-bit DLLs, this 16-bit world is slowly fading away. For this reason calling 16-bit DLLs from your 32-bit Delphi programs is less and less common, nowadays.

Using Existing DLLs

We have already used existing DLLs in a number of examples in the book, when calling Windows API functions. As you might remember, all the API functions are declared in the system Windows unit. Functions are declared in the `interface` portion of the unit , as shown here:

```
function PlayMetaFile(DC: HDC; MF: HMETAFILE): BOOL; stdcall;
function PaintRgn(DC: HDC; RGN: HRGN): BOOL; stdcall;
function PolyPolygon(DC: HDC; var Points; var nPoints;
  p4: Integer): BOOL; stdcall;
function PtInRegion(RGN: HRGN; p2, p3: Integer): BOOL; stdcall;
```

Then, in the `implementation` portion, instead of providing each function's code, the unit refers to the external definition in a DLL:

```
const
  gdi32 = 'gdi32.dll';

function PlayMetaFile; external gdi32 name 'PlayMetaFile';
function PaintRgn; external gdi32 name 'PaintRgn';
function PolyPolygon; external gdi32 name 'PolyPolygon';
function PtInRegion; external gdi32 name 'PtInRegion';
```

note In Windows.PAS there is a heavy use of the `{$EXTERNALSYM identifier}` directive. This has little to do with Delphi itself; it applies to C++Builder. This symbol prevents the corresponding Pascal symbol from appearing in the C++ translated header file. This helps keep the Delphi and C++ identifiers in sync, so that code can be shared between the two languages.

The external definition of these functions refers to the name of the DLL they use. The name of the DLL must include the DLL extension, or the program will work under Windows 95 and 98 but not under Windows NT. The other element is the name of the DLL function itself. The `name` directive is not necessary if the Pascal function (or procedure) name matches the DLL function name (which is case sensitive).

To call a function that resides in a DLL, you can provide its declaration and external definition, as shown above, or you can merge the two in a single declaration. Once the function is properly defined, you can call it in the code of your Delphi application just like any other function. There is nothing special about the calling syntax; it is just a normal function or procedure call.

As an example, I've written a very simple DLL in C++, with some trivial functions, just to show you how to call DLLs from a Delphi application. I won't explain the C++ code in detail (it is basically C code, anyway) but will focus instead on the calls between the Delphi application and the C++ DLL. In Delphi programming it is common to use DLLs written in C or C++.

---

note With the release of Borland C++ Builder (the Delphi "clone" based on the C++ language), the possibilities of sharing code between C++ and Object Pascal applications have increased exponentially. C++ Builder can directly read Pascal units and use Delphi components. What I'm discussing here is a more generic and traditional approach.

Using a C++ DLL

Suppose you are given a DLL built in C or C++. You'll generally have in your hands a .DLL file (the compiled library itself), an .H file (the declaration of the functions inside the library), and an .LIB file (another version of the list of the exported functions for the C/C++ linker). This .LIB file is totally useless in Delphi, while the .DLL file is used as is, and the .H file has to be translated into a Pascal unit with the corresponding declarations.

In the following listing, you can see the declaration of the C++ functions I've used to build the CppDll library. The complete source code and the compiled version of the C++ DLL and of the source code of the Delphi application using it are in the CppDll directory among the folders you've downloaded. You should be able to compile this code with any C++ compiler; I've tested it only with Borland compilers (Borland C++ 5 and C++ Builder 3 and 4). Here are the C++ declarations of the functions:

```
extern "C" __declspec(dllexport)
int WINAPI Double (int n);
```

```
extern "C" __declspec(dllexport)
int WINAPI Triple (int n);
__declspec(dllexport)
int WINAPI Add (int a, int b);
```

The three functions perform some basic calculations on the parameters and return the result. Notice that all the functions are defined with the `WINAPI` modifier, which sets the proper parameter-calling convention; and they are preceded by the `__declspec(dllexport)` declaration, which makes the functions available to the outside world.

Two of these C++ functions also use the C naming convention (indicated by the `extern "C"` statement), but the third one, `Add`, doesn't. This affects the way we call these functions in Delphi. In fact, the internal names of the three functions correspond to their names in the C++ source code file, except for the `Add` function. Since we didn't use the `extern "C"` clause for this function, the C++ compiler used *name mangling*. This is a technique used to include information about the number and type of parameters in the function name, which the C++ language requires in order to implement function overloading. The result when using the Borland C++ compiler is a funny function name: `@Add$qqsii`. This is actually the name we have to use in our Delphi example to call the `Add` DLL function (which explains why you'll generally avoid C++ name mangling in exported functions, and why you'll generally declare them all as `extern "C"`). The following are the declarations of the three functions in the Delphi CallCpp example:

```
function Add (A, B: Integer): Integer;
  stdcall; external 'CPPDLL.DLL' name '@Add$qqsii';
function Double (N: Integer): Integer;
  stdcall; external 'CPPDLL.DLL' name 'Double';
function Triple (N: Integer): Integer;
  stdcall; external 'CPPDLL.DLL';
```

As you can see, you can either provide or omit an alias for an external function. I've provided one for the first function (there was no alternative, because the exported DLL function name `@Add$qqsii` is not a valid Pascal identifier) and for the second, although in the second case it was unnecessary. If the two names match, in fact, you can omit the `name` directive, as I did for the third function above.

Remember to add the `stdcall` directive to each definition, so that the caller module (the application) and the module being called (the DLL) use the same parameter-passing convention. If you fail to do so, you will get random values passed as parameters, a bug that is very hard to trace.

note When you have to convert a large C/C++ header file to the corresponding Pascal declarations, instead of doing a manual conversion you can use a third-party tool to partially automate the process. One of these tools is HeadConv, written by Bob Swart. You'll find a copy on his Web site, `http://www.drbob42.com`.[353]

If you are not sure of the actual names of the functions exported by the DLL, you can simply select the DLL file in the Windows Explorer, right-click on it, and choose the QuickView command[354]. The viewer that appears lists some of the low-level technical information available for each executable file. What we are interested in right now is the *Export Table* section, as shown in Figure 14.2.

Notice that each of the three functions has a name and an index number (indicated as Ordinal). This index number was generally used for binding DLL functions in 16-bit Windows. In Win32, Microsoft suggests that you bind DLL functions by name.

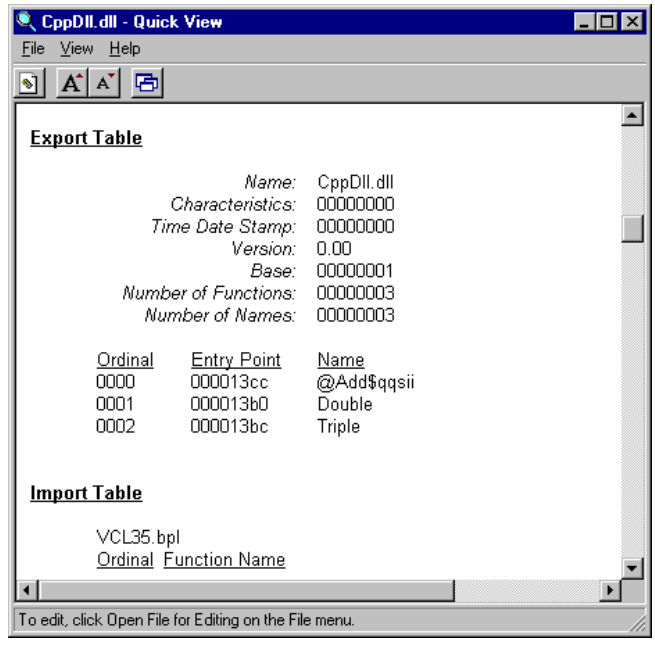


Figure 14.2: Windows QuickView lets you explore a DLL or an EXE file. Here is the Export Table of the file CPPDLL.DLL. Image from the original book – notice that this Windows tool doesn't exist any more.

353 This tool is still around, but a bit old... There several other tools for mapping header functions to Pascal. Once I can recommend is Chet, https://github.com/neslib/Chet.

354 I don't think this tool (or something similar) is available today as part of the core Windows operating system. Third party utilities exist, including Delphi TDump32, covered below.

As an alternative to QuickView you can use the TDump32 command-line program that comes with Delphi, which will give you even more details about the internal structure of the executable file. Note that QuickView and TDump32 can be used for both executable files and DLLs.

To use this C++ DLL I've built a Delphi example, named CallCpp. Its simple form has three buttons to call each function of the DLL, two SpinEdit components for the parameters, and a read-only edit box to show the result of the sum. If you click the first button, the value in the corresponding SpinEdit component is doubled:

```
procedure TForm1.BtnDoubleClick(Sender: TObject);
begin
  SpinEdit1.Value := Double (SpinEdit1.Value);
end;
```

The code for the Triple button is very similar. When you click on the third button, Add, the program adds the values of the two SpinEdit components by calling the third function of the DLL, and it displays them in the edit box. Figure 14.3 shows an example of the output after each button has been pressed once.

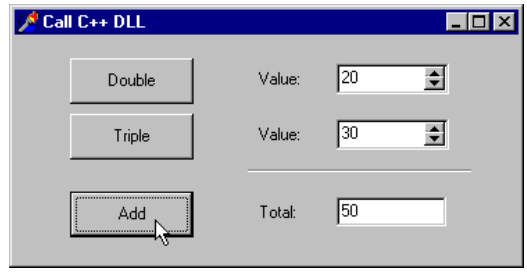


Figure 14.3: The output of the CallCpp example when you have pressed each of the buttons. Image from the original book.

Here is the code of the `OnClick` event handler for the third button:

```
procedure TForm1.BtnAddClick(Sender: TObject);
begin
  Edit1.Text := IntToStr (Add (
    SpinEdit1.Value, SpinEdit2.Value));
end;
```

To run this application, you should have the DLL in the same directory as the project, in one of the directories on the path, or in the Windows or System directories. If you move the executable file to a new directory and try to run it, you'll get a run-time error indicating that the DLL is missing, as you can see in Figure 14.4.

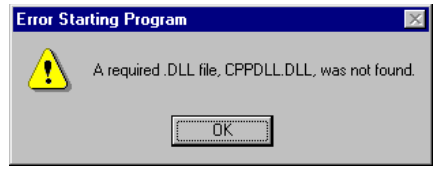


Figure 14.4: The error message displayed when you run the CallCpp example and Windows cannot find the required DLL. Image from the original book.

Creating a DLL in Delphi

Besides using DLLs written in other environments, you can use Delphi to build DLLs that can be used by Delphi programs or with any other development tool that supports DLLs. Building DLLs in Delphi is so easy that you might overuse this feature. In general, I suggest you try to build components and packages instead of plain DLLs. Packages can contain components but also classes in general, allowing you to write object-oriented code and to reuse it effectively. Placing a collection of functions in a DLL is a more traditional approach to programming, even if the functions can encapsulate forms and objects.

note In other words, DLLs by default do not fully support objects, but you can provide your own wrapping, use Delphi packages, or use Microsoft's COM technology (described in the next chapter).

When writing a DLL you generally export subroutines, functions, and procedures. If you want to export classes and methods from a Delphi DLL you should either build a package (as described later in this chapter) if you need to use the library only from Delphi programs or built a COM server (or ActiveX library), as we'll see in the next chapter.

In general, when you build complex Delphi applications, you use object-oriented programming techniques to define your application's structure. If you later divide the application's code among traditional DLLs, you lose this advantage.

It is useful to build libraries of small functions if the same functions have to be called from different environments. In particular, you can write DLLs in a compiled language like Object Pascal, and call them from interpreted environments. Of course, whenever possible, it's best to build the whole program in Delphi.

As I've already mentioned, building a DLL is also useful when a portion of the code of a program is subject to frequent changes. In this case you can frequently replace

the DLL, keeping the rest of the program unchanged. Similarly, when you need to write a program that provides different features to different groups of users, you can distribute different versions of a DLL to different users.

A First Simple Delphi DLL

As a starting point in exploring the development of DLLs in Delphi, I'll show you a very simple library built in Delphi. The primary focus of this example will be to show the syntax you use to define a DLL in Delphi, but it will also illustrate a few considerations involved in passing string parameters. To start, select the File ➢ New command and choose the DLL option in the New page of the Object Repository. This creates a very simple source file that starts with the following definition:

```
library Project1;
```

The `library` statement indicates that we want to build a DLL instead of an executable file. Now we can add routines to the library and list them in an `exports` statement:

```
function Triple (N: Integer): Integer; stdcall;
begin
  Result := N * 3;
end;

function Double (N: Integer): Integer; stdcall;
begin
  Result := N * 2;
end;

exports
  Triple, Double;
```

In this basic version of the DLL, we don't need a `uses` statement; but in general, the main project file includes only the `exports` statement, while the function declarations are placed in a separate unit. In the final source code of the FirstDLL example (the version you've downloaded), I've actually changed the code slightly from the version listed above, to show a message each time a function is called. There are two ways to accomplish this. The simplest is to change the code as follows:

```
uses
  Dialogs;

function Triple (N: Integer): Integer; stdcall;
begin
  ShowMessage ('Triple function called');
```

```
  Result := N * 3;
end;
```

This code requires Delphi to link a lot of VCL code into the application. If you statically link the VCL into this DLL, the resulting size will be about 200KB. The reason is that the `ShowMessage` function displays a VCL form that contains VCL controls and uses VCL graphics classes; and those indirectly refer to things like the VCL streaming system and the VCL application and screen objects. For this simple case, a better alternative is to show the messages using direct API calls, so that the VCL code is not required:

```
uses
  Windows;

function Triple (N: Integer): Integer; stdcall;
begin
  MessageBox (0, 'Triple function called',
    'First DLL', mb_OK);
  Result := N * 3;
end;
```

This change in code brings the size of the application down to only about 20 KB. In the downloaded source code of the FirstDLL example, you'll find both versions of the library, one of which is commented. Changing the commented section you can easily alter the code and do your own experiments.

---

note This huge difference in size underlines the fact that you should not overuse DLLs in Delphi, to avoid compiling the code of the VCL in multiple executable files. Of course, you can reduce the size of a Delphi DLL by using run-time packages, as detailed later in this chapter.

If you run a test program as the CallFrst example (described later) using the API-based version of the DLL, its behavior won't be correct. In fact, you can click the buttons which call the DLL functions several times without first closing the message boxes displayed by the DLL. This happens because the first parameter of the `MessageBox` API call above is zero. Its value should instead be the handle of the program's main form or the application form. We'll make this change, although for a different reason, in a following example, FormDLL.

Overloaded Functions in Delphi DLLs

When you create a DLL in C++, overloaded functions (and generally all the functions compiled with a C++ compiler) use name mangling to generate a different

name for each function, including the type of the parameters right in the name, as we've seen in the CppDll example.

When you create a DLL in Delphi and use overloaded functions (that is, multiple functions using the same name and marked with the `overload` directive), Delphi allows you to export only one of the overloaded functions. It requires you to indicate which one, in the `exports` clause, as indicated by this portion of the FirstDll code:

```
function Triple (C: Char): Integer; stdcall; overload;
begin
  ShowMessage ('Triple (Char) function called');
  Result := Ord (C) * 3;
end;

function Triple (N: Integer): Integer; stdcall; overload;
begin
  ShowMessage ('Triple (Integer) function called');
  Result := N * 3;
end;

exports
  Triple (N: Integer);
```

note The reverse is possible as well: You can import a series of similar functions from a DLL and define them all as overloaded functions in the Pascal declaration. You can refer to the OpenGl.pas unit for a series of examples of this technique.

Exporting Strings from a DLL

In general, functions in a DLL can use any type of parameter and return any type of value. There are two exceptions to this rule:

- If you plan to call the DLL from other programming languages, you should probably try using Windows native data types instead of Delphi-specific types. For example, to express color values you should use Integers or the Windows `ColorRef` type instead of the Delphi native `TColor` type, doing the appropriate conversions (as detailed in the FormDLL described in the next section). Other Delphi types that for compatibility you should avoid using include objects, which cannot be used by other languages at all, and Pascal strings, which can be replaced by `PChar` strings. In other words, every Windows development environment must support the basic types of the API, and if you stick to them your DLL will be usable with other development environments.

- Even if you plan to use the DLL only from a Delphi application, you cannot pass Delphi strings across the DLL boundary without taking some precautions. This is because of the way Delphi manages strings in memory—allocating, reallocating, and freeing them automatically. The solution to the problem is to include the ShareMem system unit both in the DLL and in the program using it. This unit must be included as the first unit of each of the projects.

In the FirstDLL example I've actually included both approaches: One function receives and returns a Pascal string, and another one receives as parameter a `PChar` pointer, which is then filled by the function itself. The first function is very simple:

```
function DoubleString (S: string; Separator: Char): string; stdcall;
begin
  Result := S + Separator + S;
end;
```

The second one is quite complex because `PChar` strings don't have a simple + operator, and they are not directly compatible with characters; the separator must be turned into a string before adding it. Here is the complete code; it uses input and output `PChar` buffers, which are compatible with any Windows development environment:

```
function DoublePChar (BufferIn, BufferOut: PChar;
  BufferOutLen: Cardinal; Separator: Char): LongBool; stdcall;
var
  SepStr: array [0..1] of Char;
begin
  // if the buffer is large enough
  if BufferOutLen > StrLen (BufferIn) * 2 + 2 then
  begin
    // copy the input buffer in the output buffer
    StrCopy (BufferOut, BufferIn);
    // build the separator string (value plus null terminator)
    SepStr [0] := Separator;
    SepStr [1] := #0;
    // append the separator
    StrCat (BufferOut, SepStr);
    // append the input buffer once more
    StrCat (BufferOut, BufferIn);
    Result := True;
  end
  else
    // not enough space
    Result := False;
end;
```

This second version of the code is certainly more complex, but the first one can be used only from Delphi. Moreover, the first version requires us to include the ShareMem unit in the DLL (and in the programs using it) and to deploy the file

BorlndMM.DLL (the name stands for Borland Memory Manager) along with the program and the specific library.

Calling the Delphi DLL

How can we use the library we've just built? We can call it from within another Delphi project or from other environments. As an example, I've built the CallFrst project (stored in the FirstDLL directory).

To access the DLL functions we must declare them as `external`, as we've done with the C++ DLL. This time, however, we can simply copy and paste the definition of the functions from the source code of the Delphi DLL, adding the `external` clause, as in:

```
function Double (N: Integer): Integer;
  stdcall; external 'FIRSTDLL.DLL';
```

This declaration is similar to those used to call the C++ DLL. This time, however, we have no problems with function names. The source code of the example is actually quite simple. Once they are redeclared as `external`, the functions of the DLL can simply be used as if they were local functions. Here are two examples, with calls to the string-related functions:

```
procedure TForm1.BtnDoubleStringClick(Sender: TObject);
begin
  // call the DLL function directly
  EditDouble.Text :=
    DoubleString (EditSource.Text, ';');
end;

procedure TForm1.BtnDoublePCharClick(Sender: TObject);
var
  Buffer: string;
begin
  // make the buffer large enough
  SetLength (Buffer, 1000);
  // call the DLL function
  if DoublePChar (PChar (EditSource.Text),
      PChar (Buffer), 1000, '/') then
    EditDouble.Text := Buffer;
end;
```

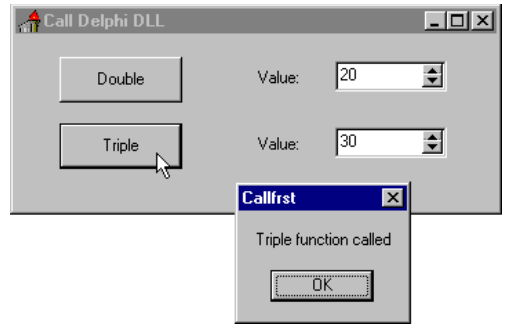


Figure 14.5: The output of the CallFrst example, which calls the DLL we've built in Delphi. Image from the original book.

A Delphi Form in a DLL

Besides writing simple DLLs with functions and procedures, you can place a complete form built with Delphi into a DLL. This can be a dialog box or any other kind of form, and it can be used not only by other Delphi programs, but also by other development environments or macro languages.

To build the FormDLL example, I've built a simple form with three scrollbars you can use to select a color and two preview areas for the resulting pen and brush colors. The form also contains two bitmap buttons and has its `BorderStyle` property set to `bsDialog`.

Aside from developing a form as usual, I've only added two new subroutines to the unit that defines the form. In the `interface` portion of the unit I've added the following declarations:

```
function GetColor (Col: LongInt): LongInt; stdcall;
procedure ShowColor (Col: LongInt;
  FormHandle: THandle; MsgBack: Integer); stdcall;
```

In both subroutines the `Col` parameter is the initial color. Notice that I've passed it as a long integer, which corresponds to the Windows `COLORREF` data type. As mentioned before, using the `TColor` Delphi type might have caused problems with non-Delphi applications: Even though a `TColor` is very similar to a `COLORREF`, these types don't always correspond. When you write a DLL, I suggest you use only the Windows native data types (unless you are sure only Delphi programs will use the DLL).

The `GetColor` function returns the final color (which is the same as the initial color if the user clicks on the Cancel button). The value is returned immediately because

the function shows the form as a modal form. The `ShowColor` procedure, instead, simply displays the form (as a modeless form) and returns immediately. For this reason the form needs a way to communicate back to the calling form. In this case I've decided to pass as parameters the handle for the window of the calling form and the ID of the message to use to communicate back with it.

In the next sections you'll see how to write the code of the two subroutines; and you'll also see what problems arise, particularly when you place a modeless form in a DLL. Of course, I'll also provide a few alternative fixes.

Using the DLL Form as Modal

When you want to place a Delphi component (such as a form) in a DLL, you can only provide functions that create, initialize, or run the component or access its properties and data. The simplest approach is to have a single function that sets the data, runs the component, and returns the result, as in the modal version. Here is the code of the function, added to the `implementation` portion of the unit that defines the form:

```
function GetColor (Col: LongInt): LongInt; stdcall;
var
  FormScroll: TFormScroll;
begin
  // default value
  Result := Col;
  try
    FormScroll := TFormScroll.Create (Application);
    try
      // initialize the data
      FormScroll.SelectedColor := Col;
      // show the form
      if FormScroll.ShowModal = mrOK then
        Result := FormScroll.SelectedColor;
    finally
      FormScroll.Free;
    end;
  except
    on E: Exception do
      MessageDlg ('Error in FormDLL: ' +
        E.Message, mtError, [mbOK], 0);
  end;
end;
```

An important element is the structure of the `GetColor` function. The code creates the form at the beginning, sets some initial values, and then runs the form, eventu-

ally extracting the final data. What makes this different from the code we generally write in a program is the use of exception handling:

- A `try-except` block protects the whole function, so that any exception generated by the function will be trapped, displaying a proper message. The reason for handling every possible exception is that the calling application might be written in any language, in particular one that doesn't know how to handle exceptions. Even when the caller is a Delphi program, it is sometimes useful to use the same protective approach.
- A `try-finally` block protects the operations on the form, ensuring that the form object will be properly destroyed, even when an exception is raised. This kind of code is often used within Delphi programs, as well as in exportable DLLs.

By checking the return value of the `ShowModal` method, the program determines the result of the function. I've set the default value before entering the `try` block to ensure it will always be executed (and also to avoid the compiler warning indicating that the result of the function might be undefined).

Now that we have updated the form and written the code of the unit, we can move to the project source code, which (temporarily) becomes the following:

```
library FormDLL;

uses
  ScrollF in 'SCROLLF.PAS' {FormScroll};

exports
  GetColor;
end.
```

We can now use a Delphi program to test the form we have placed in the DLL. The UseCol example is in the same directory as the previous DLL, FormDLL (and both projects are part of the FormDLL project group, the file `FormDll.BPG`). The form of the UseCol example contains a button to call the `GetColor` function of the DLL. Here is the definition of this function and the code of the `Button1Click` method:

```
function GetColor (Col: LongInt): LongInt;
  stdcall; external 'FormDLL.DLL';

procedure TForm1.Button1Click(Sender: TObject);
var
  Col: LongInt;
begin
  Col := ColorToRGB (Color);
  Color := GetColor (Col)
end;
```

Running this program (see Figure 14.6) displays the dialog box, using the current background color of the main form. If you change the color and click OK, the program uses the new color as the background color for the main form.

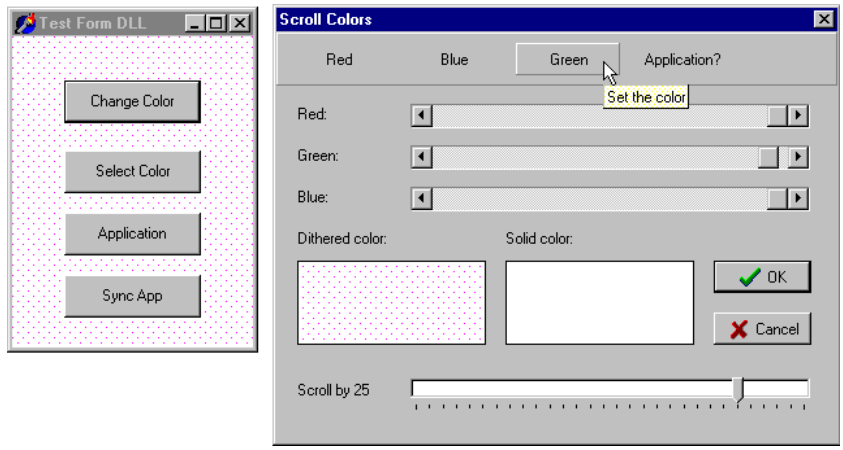


Figure 14.6: The execution of the UseCol test program when it calls the dialog box we have placed in the FormDLL. Image from the original book.

If you execute this as a modal dialog box, almost all the features of the form work fine. You can see the fly-by hints, the flat speed buttons in the toolbar behave properly, and you get no extra entry in the task bar. This might be obvious, but is not what will happen when we use the form inside the DLL as a modeless form. Even with modal forms, however, I recommend synchronizing the application objects of the DLL and executable file, as described in the next section.

A Modeless Form in a DLL

The second subroutine of the FormDLL example uses a different approach. As mentioned, it receives three parameters: the color, the handle of the main form, and the message number for notification when the color changes. These values are stored in the private data of the form:

```
procedure ShowColor (Col: LongInt;
  FormHandle: THandle; MsgBack: Integer); stdcall;
var
  FormScroll: TFormScroll;
begin
  FormScroll := TFormScroll.Create (Application);
  try
    // initialize the data
```

```
    FormScroll.FormHandle := FormHandle;
    FormScroll.MsgBack := MsgBack;
    FormScroll.SelectedColor := Col;
    // show the form
    FormScroll.Show;
  except
    on E: Exception do
    begin
      MessageDlg ('Error in FormDLL: ' +
        E.Message, mtError, [mbOK], 0);
      FormScroll.Free;
    end;
  end;
end;
```

When the form is activated, it checks to see if it was created as a modal form (simply testing the `FormHandle` field). In this case, the form changes the caption and the behavior of the OK button, as well as the overall style of the Cancel button (you can see the modified buttons in Figure 14.7):

```
procedure TFormScroll.FormActivate(Sender: TObject);
begin
  // change buttons for modeless form
  if FormHandle <> 0 then
  begin
    BitBtn1.Caption := 'Apply';
    BitBtn1.OnClick := ApplyClick;
    BitBtn2.Kind := bkClose;
  end;
end;
```

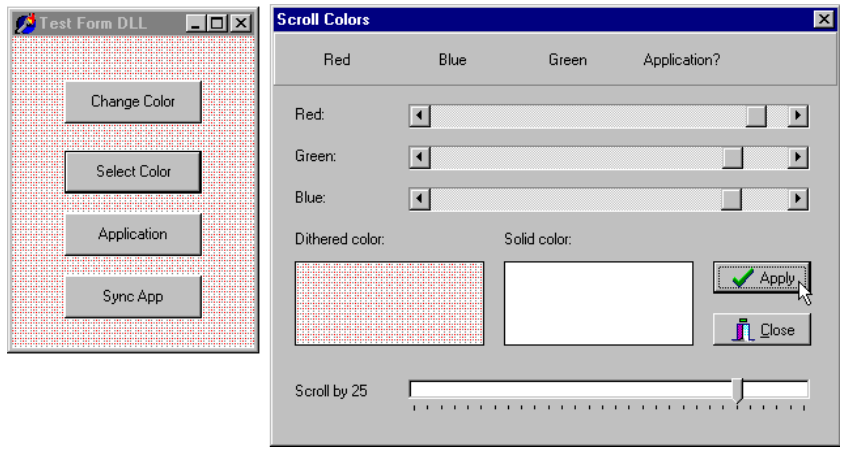


Figure 14.7: When the DLL-based form is used as a modeless form, its buttons are slightly modified (as you can see comparing this image with that of Figure 14.6). Image from the original book.

The `ApplyClick` method I've manually added to the form simply sends the notification message to the main form, using one of the parameters to send back the selected color:

```
procedure TFormScroll.ApplyClick(Sender: TObject);
begin
  // notify to the main form
  SendMessage (FormHandle, MsgBack, SelectedColor, 0);
end;
```

Finally, the form's `OnClose` event destroys the form object:

```
procedure TFormScroll.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  // used by the modeless form
  Action := caFree;
end;
```

Now let us move back to the demo program. The second button of the UseForm example's form has the following code:

```
procedure TForm1.BtnSelectClick(Sender: TObject);
var
  Col: LongInt;
begin
  Col := ColorToRGB (Color);
  ShowColor (Col, Handle, wm_user);
end;
```

The form also has a message-handling method, connected with the `wm_user` message. This method reads the value of the parameter corresponding to the color and sets it:

```
procedure TForm1.UserMessage(var Msg: TMessage);
begin
  Color := Msg.WParam;
end;
```

Running this program produces some strange effects. Basically, the modeless form and the main form are not synchronized, so they both show up in the Windows Taskbar; and when you minimize the main form, the other one remains on the screen. The two forms behave as if they were part of separate applications, and the reason is that two Delphi programs (the DLL and the EXE) have two separate global `Application` objects, and only the `Application` object of the executable file has an associated window.

To test this situation I've added a button to both the main form and the DLL form, showing the numeric value of the `Application` object's handle. Here is the code for one of them:

```
procedure TFormScroll.spApplicationClick(Sender: TObject);
begin
  ShowMessage ('Application Handle: ' +
    IntToStr (Application.Handle));
end;
```

For the form in the DLL you'll invariably get the value 0, while for the form in the executable you get a numeric value determined each time by Windows.

To fix the problem we can add to the DLL an initialization function that passes the handle of the application window to the library. In practice, we copy the `Handle` of the `Application` object of the executable to the same property of the `Application` object of the DLL. This is enough to synchronize the two `Application` objects and make the two forms behave as in a simple Delphi program. Here is the code of the function in the DLL:

```
procedure SyncApp (AppHandle: THandle); stdcall;
begin
  Application.Handle := AppHandle;
end;
```

And here is the call to it in the executable file:

```
procedure TForm1.BtnSyncClick(Sender: TObject);
begin
  SyncApp (Application.Handle);
  BtnSync.Enabled := False;
end;
```

note Assigning the handle of the application object of the DLL is not a workaround for a bug, but a documented operation required by VCL. The VCL `Application` object supports assignment to its `Handle` property (contrary to most other `Handle` properties of the VCL) specifically to allow programmers to tie DLL-based forms into the environment of a host application.

I've connected this code to a button, instead of executing it automatically at startup, to let you test the behavior in the two different cases. Before you press the *Sync App* button, the secondary modeless form behaves oddly. If you close it, synchronize the applications, and then create another instance of the modeless form, it will behave almost correctly. The only visible problem is that the flat speed buttons of the modeless form won't be highlighted when the mouse moves over them. We'll see how to fix this problem using run-time packages at the end of the chapter.

note Technically this behavior of the speed buttons depends on the fact that the controls in the DLL form don't receive the `cm_MouseEnter` and `cm_MouseLeave` messages, because the DLL's `Application.Idle` method is never called. The DLL's `Application` object, in fact, is not running the application's message loop. You can activate it by exporting from the DLL a function that calls the internal `Application.Idle` routine, and call that function from the host application when its message loop goes idle.As I mentioned, however, all these problems and few others can be solved by using run-time packages.

Calling a Delphi DLL from Visual Basic for Applications

We can also display this color dialog box from other programming languages. Calling this DLL from C or C++ is easy. To link the application, you need to generate an import library (using the `IMPLIB` command line utility) and add the resulting `LIB` file to the project. Since I've already used a C++ compiler in this chapter, this time I will write a similar example using Microsoft Word for Windows and Visual Basic for Applications instead.

To start, open Microsoft Word. Then open its Macro dialog box (with the Tools ➢ Macro menu item or a similar command, depending on your version of Word), type a new macro name, such as "**DelphiColor**," and click the Create button. You can now write the BASIC code, which declares the function of our DLL and calls it. The BASIC macro uses the result of the DLL function in two ways. By calling `Insert`, it adds to the current document a description of the color with the amount of Red, Green, and Blue; and by calling `Print` it displays the numeric value in the status bar:

```
Declare Function GetColor Lib "FormDLL"(Col As Long) As Long
Sub MAIN
  NewColor = GetColor(0)
  Print "The code of the color is " + Str$(NewColor)
  Insert "Red:" + Str$(NewColor Mod 256) + Chr$(13)
  Insert "Green:" + Str$(Int(NewColor / 256) Mod 256) + Chr$(13)
  Insert "Blue:" + Str$(Int(NewColor / (256 * 256))) + Chr$(13)
End Sub
```

Unfortunately, there is no easy way to use RGB colors in Word, since Word's color schemes are based on fixed color codes. Here is an example of the output of this macro:

```
Red: 141
Green: 109
Blue: 179
```

The nice thing is that I produced the three lines of text above by running the macro while I was writing the text of this chapter. You can find the text of this macro in the file `WORDCALL.TXT`, in the directory containing this DLL. If you want to test it, remember to first copy the DLL file into one of the directories of the path or into the Windows system directory. Of course, you need Microsoft Word to run this program. However, other Microsoft Office applications (and the macro languages of other office programs) probably require very similar code.

note A better way to integrate Delphi code with Office applications is to use OLE Automation, instead of writing custom DLLs and calling them from the macro language. We'll see examples of OLE Automation in Chapter 16.

Calling a DLL Function at Run Time

Now that we know how to access resources in a DLL at run time, we might want to use this approach to access a function. I've built a very simple example showing this and made it quite flexible. We will look at the example first, and then consider some general cases in which this approach might be useful. The example is named DynaCall and uses the FirstDLL library we built earlier in this chapter (to make the program work, I've copied the DLL into the same folder as the DynaCall example). Instead of declaring the `Double` and `Triple` functions and using them directly, this example obtains the same effect with somewhat more complex code. The advantage, however, is that if new functions are added to the DLL, we won't have to revise the program's source code and recompile it to access those new functions.

The form displayed by this example simply contains a button, an edit box, and a SpinEdit component. Clicking the button executes the only method of the program. First, the method calls the `LoadLibrary` function. Then, if the handle of the library instance is valid, the program calls the `GetProcAddress` API function. This function searches the DLL's exports table, looking for the name of the function passed as a parameter. If `GetProcAddress` finds a match, it returns a pointer to the requested procedure. Now we can simply cast this function pointer to the proper data type and call it. The output of the program and the effect of this call are visible in Figure 14.8. Here is the (quite complex) code:

```
type
  TIntFunction = function (I: Integer): Integer; stdcall;

const
  DllName = 'Firstdll.dll';
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  HInst: THandle;
  FPointer: TFarProc;
  MyFunct: TIntFunction;
begin
  HInst := LoadLibrary (DllName);
  if HInst > 0 then
  try
    FPointer := GetProcAddress (HInst,
      PChar (Edit1.Text));
    if FPointer <> nil then
    begin
      MyFunct := TIntFunction (FPointer);
      SpinEdit1.Value := MyFunct (SpinEdit1.Value);
    end
    else
      ShowMessage (Edit1.Text + ' DLL function not found');
  finally
    FreeLibrary (HInst);
  end
  else
    ShowMessage (DllName + ' library not found');
end;
```

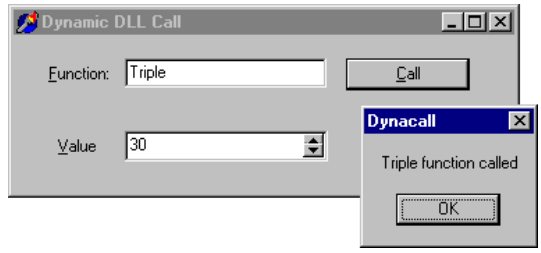


Figure 14.8: The output of the DynaCall program. Image from the original book.

How do you call a procedure in Delphi, once you have a pointer to it? One solution is to convert the pointer to a procedural type and then call the procedure using the procedural-type variable, as in the listing above. Notice that the procedural type you define must be compatible with the definition of the procedure in the DLL. This is the Achilles' heel of this method—there is no check of the parameter types.

What is the advantage of this approach? In theory, you can use it to access any function of any DLL at any time. In practice, it is useful when you have different DLLs with compatible functions or a single DLL with several compatible functions, as in our case. What we can do is to call the `Double` and `Triple` methods simply by entering their names in the edit box. Now, if someone gives us a DLL with a new function

receiving an Integer as parameter and returning an Integer, we can call it simply by entering its name in the edit box. We don't even need to recompile the application.

With this code, the compiler and the linker ignore the existence of the DLL. When the program is loaded, the DLL is not loaded immediately. We might make the program even more flexible and let the user enter the name of the DLL to use. In some cases, this is a great advantage. A program may switch DLLs at run time, something the direct approach does not allow. Note that this approach to loading DLL functions is common in macro languages and is used by many visual programming environments. Also, the code of the Word macro we saw earlier in this chapter uses this approach to load the DLL and to call the external function. Well, you don't want to recompile Word, do you?

Only a system based on a compiler and a linker, such as Delphi, can use the direct approach, which is generally more reliable and also a little bit faster. I think the indirect loading approach of the DynaCall example is useful only in special cases, but it can be extremely powerful.

A DLL in Memory: Code and Data

We can use this technique, based on the `GetProcAddress` API function, to test which memory address of the current process a function has been mapped to, with the following code:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  HDLLInst: THandle;
begin
  HDLLInst := LoadLibrary ('dllmem');
  Label1.Caption := Format ('Address: %p', [
    GetProcAddress (HDLLInst, 'SetData')]);
  FreeLibrary (HDLLInst);
end;
```

This code displays in a label the memory address of the function, within the address space of the calling application: If you run two programs using this code they'll generally both show the same address. This demonstrates that the code is loaded only once at a common memory address. Even if the code is loaded only once, the memory address will be different in case the DLL had to be relocated in one of the processes but not the other, or each process relocated the DLL to a different base address.

If the code of the DLL is loaded only once, what about the global data? Basically each copy of the DLL has its own copy of the data, in the address space of the calling application. However, it is indeed possible to share global data between applications using a DLL. The most common technique for sharing data is to use memory-mapped files. I'll use this technique for a DLL, but it can be used also to share data directly among applications.

This example is called DllMem and uses a project group with the same name, as in past examples of this chapter. The DllMem project group includes the DllMem project (the DLL itself) and the UseMem project (the demo application).

The DLL code has a simple project file, which exports four subroutines:

```
library dllmem;

uses
  SysUtils,
  DllMemU in 'DllMemU.pas';

exports
  SetData, GetData,
  GetShareData, SetShareData;

end.
```

The actual code is in the secondary unit (DllMemU.pas), which has the code of the four routines that read or write two global memory locations. These hold an integer and a pointer to an integer. Here are the variable declarations and the two *set* routines:

```
var
  PlainData: Integer = 0; // not shared
  ShareData: ^Integer; // shared

procedure SetData (I: Integer); stdcall;
begin
  PlainData := I;
end;

procedure SetShareData (I: Integer); stdcall;
begin
  ShareData^ := I;
end;
```

Sharing Data with Memory-Mapped Files

For the data that isn't shared, there isn't anything else to do. To access the shared data, however, the DLL has to create a memory-mapped file and then get a pointer to this memory area. These two operations require two Windows API calls:

- `CreateFileMapping` requires as parameters the filename (or `$FFFFFFFF` to use a virtual file in memory), some security and protection attributes, the size of the data, and an internal name (which must the same to share the mapped file from multiple calling applications).
- `MapViewOfFile` requires as parameters the handle of the memory mapped file, some attributes and offsets, and the size of the data (again).

Here is the source code of the `initialization` section, executed every time the DLL is loaded into a new process space (that is, once for each application that uses the DLL):

```
var
  hMapFile: THandle;

const
  VirtualFileName = 'ShareDllData';
  DataSize = sizeof (Integer);

initialization
  // create memory mapped file
  hMapFile := CreateFileMapping ($FFFFFFFF, nil,
    Page_ReadWrite, 0, DataSize, VirtualFileName);
  if hMapFile = 0 then
    raise Exception.Create ('Error creating memory mapped file');

  // get the pointer to the actual data
  ShareData := MapViewOfFile (
    hMapFile, File_Map_Write, 0, 0, DataSize);
```

When the application terminates and the DLL is released, it has to free the pointer to the mapped file and the file mapping itself:

```
finalization
  UnmapViewOfFile (ShareData);
  CloseHandle (hMapFile);
```

The code of the program using this DLL, UseMem, is very simple. The form of this application has four edit boxes (two with an UpDown control connected), five buttons, and a label. The first button saves the value of the first edit box in the DLL data, getting the value from the connected UpDown control:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SetData (UpDown1.Position);
end;
```

If you press the second button, the program copies the DLL data to the second edit box:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Edit2.Text := IntToStr(GetData);
end;
```

The third button is used to display the memory address of a function, with the source code shown at the beginning of this section, and the last two buttons have basically the same code as the first two, but they call the `SetShareData` procedure and `GetShareData` function.

If you run two copies of this program, you can see that each copy has its own value for the plain global data of the DLL, while the value of the shared data is common. Set different values in the two programs and then get them in both, and you'll see what I mean. This situation is illustrated in Figure 14.9.

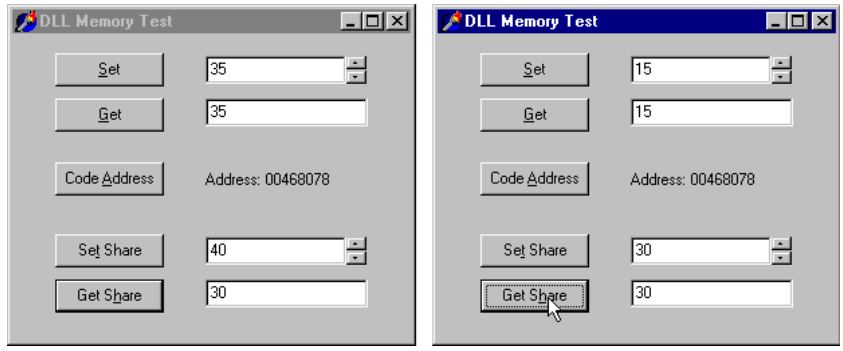


Figure 14.9: If you run two copies of the UseMem program, you'll see that the global data in the USEMEM .DLL is not shared.

note Memory mapped files reserve a minimum of a 64KB range of virtual addresses and consume physical memory in 4KB pages. The 4 byte Integer data in shared memory of the example is rather expensive, especially if you use the same approach for sharing multiple values. If you need to share several variables, you should place them all in a single shared memory area (accessing the different variables using pointers or building record structure for all of them).

Using Delphi Packages

In Delphi, component packages are an important type of DLL. Packages allow you to bundle a group of components and then link the components either statically (adding their compiled code to the executable file of your application) or dynamically (keeping the component code in a DLL, the run-time package that you'll distribute along with your program). We saw in the last chapter how to build a package. Now I want to underline advantages and disadvantages of the two forms of linking for a package. There are many elements to keep in mind:

- Using a package as a DLL makes the executable files much smaller.
- Using a statically linked package allows you to distribute only part of its code. Generally the size of the executable file of an application plus the size of the required package DLLs that it requires is much bigger than the size of the statically linked program. The linker includes only the code actually used by the program, whereas a package must link in all the functions and classes declared in the interface sections of all the units contained in the package.
- If you distribute several Delphi applications based on the same packages, you might end up distributing less code, because the run-time packages are shared. In other words, once the users of your application have the standard Delphi run-time packages, you can ship them very small programs. This even allows you to distribute the programs through the Internet.
- If you run several Delphi applications based on the same packages, you can save some memory space at run time: The code of the run-time packages is loaded in memory only once between the multiple Delphi applications.
- Don't worry too much about distributing a large executable file. Keep in mind that when you make minor changes to a program, you can use any of various tools to create a *patch file*, so that you distribute only a file containing the differences, not a complete copy of the files.

Packages for Versioning of Applications

A very important and often misunderstood element, however, is the distribution of updated packages. When you update a DLL, you can ship the new version, and the executable programs requiring this DLL will generally still work (unless you've removed existing exported functions or changed some of their parameters).

When you distribute a Delphi package, instead, if you update the package changing anything in the interface portion of any unit exported by the package, you'll need to recompile all the applications which use the package. You can only modify code in the implementation section of the units of the package to avoid recompiling the executable files using it.

The DCU files in Delphi have a version tag based on a checksum computed from the interface portion of the unit. When you change the interface portion of a unit, every other unit based on it should be recompiled. The compiler compares the checksum of the unit of previous compilations with the new checksum, and decides whether the dependent unit has to be recompiled. This is why you have to recompile each unit when you get a new version of Delphi, which has modified system units.

A package is a collection of units. In Delphi 3 a checksum of the package, obtained from the checksum of the units it contains and the checksum of the packages it requires, was added as an extra entry function to the package library, so that the executable based on an older version of the package would fail at start-up.

Delphi 4 and 5 have relaxed the runtime constraints of the package. The design time constraints on DCU files still remain identical, though. The checksum of the packages is not checked anymore, and so you can directly modify the units that are part of a package and deploy a new version of the package to be used with the existing executable file. Since methods are referenced by name, you cannot remove any existing method. You cannot even change its parameters, because of name mangling techniques.

Removing a method referenced from the calling program will stop the program during the loading process. However, if you do other changes, the program might fail unexpectedly during its execution. For example, if you replace a component with a similar one, the calling program might still able to access the component in that memory location, although it is now a different component!

If you decide to follow this treacherous road of changing the interface of units in a package without recompiling all the programs using it, you should try to limit your changes at most. When you add new properties or non-virtual methods to the form, you should be able to maintain full compatibility with existing programs already using the package. Also adding fields and virtual methods might affect the internal structure of the class, leading to problems with existing programs which expect a different class data and VMT layout. Of course, this applies to the binary compatibility between the EXE and the BPL. Any change in the interface of a unit of the package, in fact, breaks the DCU/DCP compatibility of any units that refer to yor package.

warning Here I'm referring to the distribution of compiled programs divided between EXE and packages, not to the distribution of components to other Delphi developers. In this latter case the versioning rules are more stringent, and you must take extra care in package versioning.

Having said this, I recommend never change the interface of units exported by your packages. To accomplish this you can add to your package a unit with form creation functions (as in the DLL with forms I've build earlier), and use it to access another unit which defines the form. Although there is no way to *hide* a unit that's linked into a package, if you never used the class defined in a unit directly, but only through other routines, you'll have more flexibility in modifying it. You can also use form inheritance to modify a form within a package wihtout really affecting the original version.

The most stringent rule for pacjages is the followin one used by component writers. For long-term deployment and maintenance of code in packages, plan on having a major release with minor maintenance releases. A major release of your package will require all your client programs to be recompiled from source; the packages file itself should be renamed with a new version number, and the interface section of the units can be modified. Maintenance releases of that package should be restricted to implementation changes to preserve full compatibility with existing executable files and units.

Executable Files and DLLs Sharing the VCL Packages

In the FormDll example, we faced a problem: When you place forms inside a DLL, you don't get the proper behavior for the flat buttons even if you synchronize the two application objects. Moreover, both the executable file and the DLL contain the compiled code of the VCL library, leading to a useless duplication. In the last section we've seen that a radically alternative approach is to compile the form into a package, instead of a DLL.

Another solution to this problem is the use of run-time packages for both the EXE and the DLL, so that no code will be duplicated. As a side effect, there will be only one `Application` object, shared by the program and the DLL, instead of two separate objects, so we don't need the synchronization code any more.

Another simplification to the program comes from the fact that the modeless form inside the DLL can communicate back to the main form by accessing the list of the forms (available to the shared global `Screen` object) or simply using the

`Application.MainForm` property. In the FormDllP example I've changed the second routine exported by the DLL to this simpler version:

```
procedure ShowColor (Col: LongInt); stdcall;
```

I've modified the code by removing the references to the form handle and the call-back message, and I've rewritten the code of the Apply button to use the main form instead of sending a user-defined Windows message to its handle:

```
procedure TFormScroll.ApplyClick(Sender: TObject);
begin
  // access the main form directly
  Application.MainForm.Color := SelectedColor;
end;
```

The problem is that if you now try to write this code (which is not the final version in the source code files), the behavior is not at all what you might expect. The main form and the form in the DLL are not synchronized at all, there are two entries in the Taskbar, and it still has all the other problems of the first version of the FormDll example. The problem lies in the fact that when you run the program the DLL is initialized before the application, so it is the DLL that initializes the Forms unit of the VCL. Within a DLLthe VCL creates the `Application` object but doesn't create the corresponding window.

There are two radically different approaches to this initialization issue: One is to change the initialization order by loading the DLL dynamically after the application has started, the second is to add some extra initialization code in the program.

Dynamically Loading the DLL with Packages

The first solution is demonstrated by the FirstDLLD library and UseDyna example, which dynamically loads the DLL build with run-time packages. The main program loads the DLL at startup, in the `OnCreate` event handler of the form:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  hInstDll := LoadLibrary ('FormDllD.dll');
  if hInstDll <= 0 then
    raise Exception.Create ('FormDllD library not found');
end;
```

In the program I haven't declared the functions exported by the DLL, to avoid the implicit link of the library. Instead I've declared two procedure types:

```
type
  TGetColorProc = function (Col: LongInt): LongInt; stdcall;
  TShowColorProc = procedure (Col: LongInt); stdcall;
```

These types are used for converting the generic pointer returned by the `GetProcAddress` function, as we've already seen in the DynaCall example:

```
procedure TForm1.BtnChangeClick(Sender: TObject);
var
  Col: LongInt;
  GetColorProc: TGetColorProc;
  FPointer: TFarProc;
begin
  FPointer := GetProcAddress (hInstDll, 'GetColor');
  if FPointer = nil then
    raise Exception.Create ('GetColor DLL function not found');
  GetColorProc := TGetColorProc (FPointer);
  // original code
  Col := ColorToRGB (Color);
  Color := GetColorProc (Col);
end;
```

Using dynamic loading the correct approach officially supported by Delphi. Still, you have to call the functions dynamically, which requires a little extra coding.

Fixing the Initialization Code

An alternative solution is to keep the external functions defined in the main program and let the DLL start first and initialize the VCL, and the VCL create the `Application` object without the connected window. In fact, we can add one line of code to the library to ask for the creation of the window of the `Application` object during the library initialization process (before the executable creates its own main objects). We accomplish this by writing the code in the initialization section of one of the units of the DLL:

```
initialization
  Application.CreateHandle;
```

As this code is in the DLL, the application fails to load its icon. The solution is actually very simple. In the `OnCreate` event handler of the main form (in the main program), simply reload the current icon:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // reload the icon of the application
  Application.Icon.Handle :=
    LoadIcon (HInstance, 'MAINICON');
end;
```

Exploring the Structure of a Package

You might wonder if it is possible to know whether a package has been linked in at design-time or used as run-time package. Not only is this possible in Delphi, but you can also explore the overall structure of an application.

A component can use the undocumented `ModuleIsPackage` global variable, declared in the SysInit unit. You shouldn't ever need this, but it is technically possible (for a component) to have different code depending whether it is packaged or not. The following code extracts the name of the run-time package hosting the component, if any:

```
var
  fPackName: string;
begin
  // get package name
  SetLength (fPackName, 100);
  if ModuleIsPackage then
  begin
    GetModuleFileName (HInstance,
      PChar (fPackName), Length (fPackName));
    fPackName := PChar (fPackName) // string length fixup
  end
  else
    fPackName := 'Not packaged';
```

Besides accessing package information from within a component (as in the code above), you can also do so from a special entry point of the package libraries, the `GetPackageInfoTable` function. This function returns some specific package information that Delphi stores as resources and includes in the package DLL. Fortunately, we don't need to use low-level techniques to access this information, since Delphi provides some high-level functions to manipulate it.

There are basically two functions you can use to access package information:

- `GetPackageDescription` returns a string that contains a description of the package. To call this function, you must supply the name of the module (the package library) as the only parameter.
- `GetPackageInfo` doesn't directly return information about the package. Instead, you pass it a function that it calls for every entry in the package's internal data structure. In practice, `GetPackageInfo` will call your function for every one of the package's contained units and required packages. In addition, `GetPackageInfo` sets several flags in an Integer variable.

These two function calls allow us to access internal information about a package, but how do we know which packages our application is using? You could determine this by looking at an executable file using low-level functions, but Delphi helps you again by supplying a simpler approach. The `EnumModules` function doesn't directly return information about an application's modules but allows you to pass it a function, which it calls for each module of the application, the main executable file, and for each of the packages the application relies on.

To demonstrate this approach, I've built a simple example program that displays the module and package information in a TreeView component. Each first-level node corresponds to a module, and within each module I build a subtree that displays the contained and required packages for that module, as well as the package description and compiler flags (`RunOnly` and `DesignOnly`). You can see the output of this example in Figure 14.10.

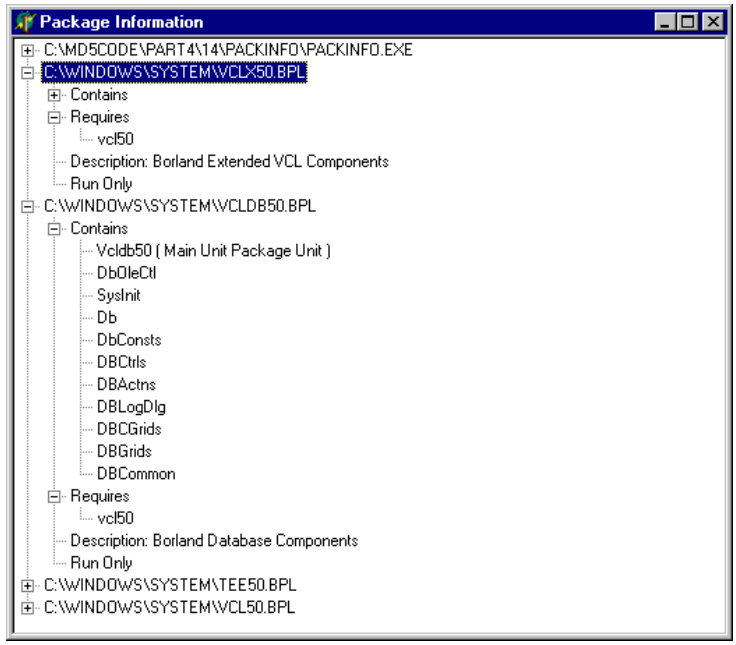


Figure 14.10: The output of the PackInfo example, with the details of the packages it uses. Image from the original book.

In addition to the TreeView component, I've added several other components to the main form, but hidden them from view: a DBEdit, a Chart, and a FilterComboBox. I added these components simply to include more run-time packages in the application, beyond the ubiquitous VCL50.DPL. The only method of the form class is `FormCreate`, which calls the module enumeration function:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  EnumModules(ForEachModule, nil);
end;
```

The `EnumModules` function accepts two parameters. The first is the callback function (in our case, `ForEachModule`), and the second is a pointer to a data structure that the callback function will use (in our case, `nil`, since we didn't need this). The callback function must accept two parameters, an `HInstance` value and an untyped pointer, and must return a `Boolean` value. The `EnumModules` function will in turn call our callback function for each module, passing the instance handle of each module as the first parameter and the data structure pointer (`nil` in our example) as the second:

```
function ForEachModule (HInstance: Longint;
  Data: Pointer): Boolean;
var
  Flags: Integer;
  ModuleName, ModuleDesc: string;
  ModuleNode: TTreeNode;
begin
  with Form1.TreeView1.Items do
  begin
    SetLength (ModuleName, 200);
    GetModuleFileName (HInstance,
      PChar (ModuleName), Length (ModuleName));
    ModuleName := PChar (ModuleName); // fixup
    ModuleNode := Add (nil, ModuleName);

    // get description and add fixed nodes
    ModuleDesc := GetPackageDescription (PChar (ModuleName));
    ContNode := AddChild (ModuleNode, 'Contains');
    ReqNode := AddChild (ModuleNode, 'Requires');

    // add information if the module is a package
    GetPackageInfo (HInstance, nil,
      Flags, ShowInfoProc);
    if ModuleDesc <> '' then
    begin
      AddChild (ModuleNode,
        'Description: ' + ModuleDesc);
      if Flags and pfDesignOnly = pfDesignOnly then
        AddChild (ModuleNode, 'Design Only');
      if Flags and pfRunOnly = pfRunOnly then
        AddChild (ModuleNode, 'Run Only');
    end;
  end;
  Result := True;
end;
```

As you can see in the code above, the `ForEachModule` function begins by adding the module name as the main node of the tree (by calling the `Add` method of the `TreeView1.Items` object and passing `nil` as the first parameter). It then adds two fixed child nodes, which are stored in the `ContNode` and `ReqNode` variables declared in the `implementation` section of this unit.

Next, the program calls the `GetPackageInfo` function and passes it another callback function, `ShowInfoProc`, which I'll discuss shortly. The program adds the details for the main module (see Figure 14.11), simply because this will provide a list of the application's units. At the end of this function, we add more information if the module is a package, such as its description and compiler flags (we know it's a package if its description isn't an empty string).

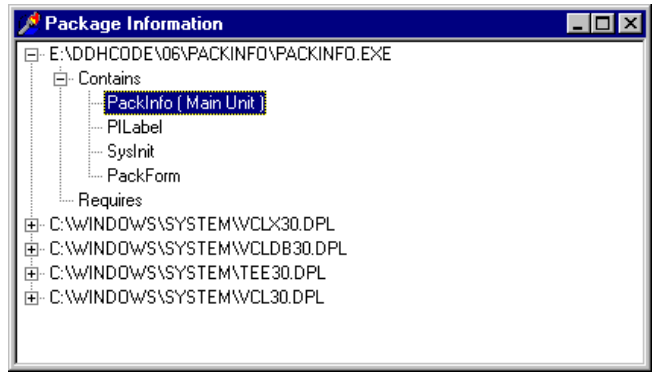


Figure 14.11: The PackInfo example also lists the units that are part of the current application. Image from the original book.

Earlier, I mentioned passing another callback function, the `ShowInfoProc` procedure, to the `GetPackageInfo` function, which in turn calls our callback function for each contained or required package of a module. This procedure creates a string that describes the package and its main flags (added within parentheses), and then inserts that string under one of the two nodes (`ContNode` and `ReqNode`), depending on the type of the module. We can determine the module type by examining the `NameType` parameter. Here is the complete code of our second callback function:

```
procedure ShowInfoProc (const Name: string;
  NameType: TNameType; Flags: Byte; Param: Pointer);
var
  FlagStr: string;
begin
  FlagStr := ' ';
  if Flags and ufMainUnit <> 0 then
    FlagStr := FlagStr + 'Main Unit ';
  if Flags and ufPackageUnit <> 0 then
```

```
    FlagStr := FlagStr + 'Package Unit ';
  if Flags and ufWeakUnit <> 0 then
    FlagStr := FlagStr + 'Weak Unit ';
  if FlagStr <> ' ' then
    FlagStr := ' (' + FlagStr + ')';
  with Form1.TreeView1.Items do
    case NameType of
      ntContainsUnit:
        AddChild (ContNode, Name + FlagStr);
      ntRequiresPackage:
        AddChild (ReqNode, Name);
    end;
end;
```

Here, you'll notice that the `Flags` parameter doesn't contain flag style information, as the online help seems to imply. If you want to investigate this topic further, examine the SysUtils unit.

What's Next

In this chapter we have seen how you can call functions that reside in DLLs created in C++ or other languages, how to create DLLs using Delphi itself, and how to use strings and place Delphi forms inside a library. DLLs have been one of the traditional approaches to writing applications using multiple programming languages and environments. Today, COM and OLE provide more advanced techniques.

Another technique for exporting objects from DLLs is to use packages, although there are many issues related to this technique. When considering DLLs and other alternatives, keep in mind that although Delphi and Windows share many elements, they also have different "views" of programming. Whenever possible, follow the Delphi object-oriented approach over the Windows procedural approach, and you'll probably benefit a lot. Actually, the advantages of object orientation are so important that Microsoft has introduced some object-oriented concepts right into the system. In short, OLE and COM are examples of these built-in techniques.

The following chapters are fully devoted to these topics: COM and OLE, OLE Automation, OLE Documents, and ActiveX controls.

Chapter 15: COM Programming

According to Microsoft, COM technology will have a fundamental role in the evolution of the Windows platform[355]. Microsoft originally used the term *OLE* to refer to this technology, then started using more often the term *COM,* and now calls the current version *COM+*. COM is not a single technology but a basic infrastructure of the operating system, which is applied under many different circumstances. This chapter shows that COM programming is simpler than you probably think. In this chapter, we'll build our first COM object as well as integrate our COM objects with the Windows shell. Type libraries, Automation, and other topics will be covered in the next chapter. I will stick to the basic elements to let you understand the role of this technology without delving heavily into the details.

---

355 Even if Microsoft later introduced the .NET framework, the new WinRT APIs, and many other technologies, COM remains the foundation of Windows programming and is still the foundation or at least a way to interact with newer APIs. In other words, COM is still around and relevant, even if some of its elements (like OLE) are not commonly used these days.

What Is OLE? And What Is COM?

Part of the confusion related to COM technology comes from the fact that Microsoft has used different names for it for marketing reasons. Everything started with Object Linking and Embedding (OLE, for short), which was an extension of the DDE (Dynamic Data Exchange) model. Using the Clipboard allows you to copy some raw data, and using DDE allows you to connect parts of two documents. Object Linking and Embedding allows you to copy the data from a server application to the client application, along with some information regarding the server or a reference to some information stored in the Windows Registry. The raw data might be copied along with the link (Object Embedding) or kept in the original file (Object Linking). Object Linking and Embedding documents were later renamed *OLE Documents* and are now called *Active Documents*.[356]

Microsoft updated OLE to OLE 2 and started adding new features, such as OLE Automation and OLE Controls. The next step was to build the Windows 95 shell using OLE technology and interfaces and then to rename the OLE Controls (previously known also as OCX) as ActiveX controls, changing the specification to allow for lightweight controls suitable for distribution over the Internet.

As this technology was extended and became increasingly important to the Windows platform, Microsoft changed the name to OLE, and then to COM, and now to COM+ for Windows 2000. These changes in naming are only partially related to technological changes and are driven to a large extent by marketing purposes.

What, then, is COM? Basically, the Component Object Model, or COM, is a technology that defines a standard way for a client module and a server module to communicate through a specific interface. Here, "module" indicates an application or a library (a DLL); the two modules may execute on the same computer or on different machines connected via a network. Many interfaces are possible, depending on the role of the client and server, and you can add new interfaces for specific purposes. These interfaces are implemented by server objects. A server object usually implements more than one interface, and all the server objects have a few common capabilities, because they must all implement the `IUnknown` interface.

The good news is that Delphi is fully compliant with COM. When you look at the source code, Object Pascal seems to be easier to use than C++ or other languages for writing COM objects. This simplicity mainly derives from the incorporation of inter-

---

356 I'm not certain which is the name Microsoft uses today, but this mechanism is still available in Microsoft Office applications and similar third party solutoins.

face types into the Object Pascal language[357]. By the way, interfaces are also similarly used to integrate Java with COM on the Windows platform.

The purpose of COM interfaces is to communicate between two software modules, two executable files, or one executable file and a DLL. Implementing COM objects in DLLs is generally simpler, because in Win32, a program and the DLL it uses reside in the same memory address space. This means that if the program passes a memory address to the DLL, the address remains valid. When you use two executable files, COM has a lot of work to do behind the scenes to let the two applications communicate. This mechanism is called *marshaling*. Note that a DLL implementing COM objects is described as an *in-process* server, whereas when the server is a separate executable, it is called an *out-of-process* server. However, when DLLs are executing on another machine (DCOM) or inside a host environment (MTS), they are also out-of-process.

Implementing IUnknown

Before we start looking to an example of COM development, I would like to introduce a few basics of COM. The first is that every COM object must implement the `IUnknown` interface[358]. This is the base interface from which every Delphi interface inherits, and Delphi provides a couple of different classes with ready-to-use implementations of `IUnknown`, including `TInterfacedObject` and `TComObject`. The first can be used to have an internal COM object, while the latter must be used to export an object from a server. As I'll discuss in Chapter 16, several other classes inherit from `TComObject` and provide support for more interfaces, which are required by Automation servers or ActiveX controls.

The `IUnknown` interface has three methods: `Add`, `Release`, and `QueryInterface`. Here is the definition of the `IUnknown` interface (extracted from the System unit):

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID;
```

357 Interfaces can also be used to map to .NET objects and they are used to map to WinRT, which is not complaint with COM in terms of memory management, but it is in terms of methods invocation.

358 This interface can also be used via the newer `IInterface` alias, which makes sense when not using COM – it's a new name, but the functionally is identical.

```
      out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

The `_AddRef` and `_Release` methods are used to implement reference counting. The `QueryInterface` method handles the type information and type compatibility of the objects.

---

note In the code above, you can see an example of an `out` parameter, a parameter passed back from the method to the calling program but without an initial value passed by the calling program to the method. The `out` parameters have been added to Delphi's Object Pascal language specifically to support COM. It's also important to note that while Delphi's language definition for the interface type is designed for compatibility with COM, Delphi interfaces do not require COM. This was already highlighted in Chapter 3, where I build a complex interface-based example with no COM support whatsoever.

You don't usually need to implement these methods, as you can inherit from one of the Delphi classes already supporting them. The most important class is `TComObject`, defined in the ComObj unit. When you build a COM server, you'll generally inherit from this class. Because `TComObject` is a complex class, this excerpt shows only its key elements:

```
type
  TComObject = class(TObject, IUnknown, ISupportErrorInfo)
  private
    FNonCountedObject: Boolean;
    FRefCount: Integer;
  protected
    { IUnknown }
    function IUnknown.QueryInterface = ObjQueryInterface;
    function IUnknown._AddRef = ObjAddRef;
    function IUnknown._Release = ObjRelease;
    { ISupportErrorInfo }
    function InterfaceSupportsErrorInfo(const iid: TIID):
      HResult; stdcall;
  public
    constructor Create;
    destructor Destroy; override;
    procedure Initialize; virtual;
    function ObjAddRef: Integer; virtual; stdcall;
    function ObjQueryInterface(const IID: TGUID; out Obj): HResult;
      virtual; stdcall;
    function ObjRelease: Integer; virtual; stdcall;
    property RefCount: Integer read FRefCount;
  end;
```

This class implements the `IUnknown` interface (using the `ObjAddRef`, `ObjQueryInterface`, and `ObjRelease` methods, as indicated by the method-mapping statements in the protected portion of the class) and the `ISupportErrorInfo` interface (through the `InterfaceSupportsErrorInfo` method). The implementation of reference counting for the `TComObject` class has been extended to support threading. Instead of using `Inc` and `Dec`, the code uses the thread-safe `InterlockedIncrement` and `InterlockedDecrement` API functions, as you can see in the source code of the class:

```
function TComObject.ObjAddRef: Integer;
begin
  Result := InterlockedIncrement(FRefCount);
end;

function TComObject.ObjRelease: Integer;
begin
  Result := InterlockedDecrement(FRefCount);
  if Result = 0 then Destroy;
end;
```

As you can see, the implementation of `Release` destroys the object when there are no more references to it. At first sight, the need to call this method each time you operate on an object seems like a lot of work. However, you might remember from Chapter 3 that when you're using interface variables to refer to objects, Delphi automatically adds the reference-counting calls to the compiled code, which automatically destroys unreferenced objects. This labor-saving feature makes Delphi a convenient tool for COM development.

The most complex method is `QueryInterface`, which in Delphi is actually implemented through the `GetInterface` method of the `TObject` class:

```
function TComObject.ObjQueryInterface(
  const IID: TGUID; out Obj): HResult;
begin
  if GetInterface(IID, Obj) then
    Result := S_OK
  else
    Result := E_NOINTERFACE;
end;
```

The role of the `QueryInterface` method is twofold:

- `QueryInterface` is used for type checking. The program can ask an object the following questions: Are you of the type I'm interested in? Do you implement the interface I want to call? And the specific methods? If the answer is no, the program can look for another object, maybe asking another server.

- If the answer is yes, `QueryInterface` usually returns a pointer to the object, using its reference output parameter (`out`).

To understand the role of the `QueryInterface` method, it is important to keep in mind that a COM object can implement multiple interfaces, as the event `TComObject` does. When you call `QueryInterface`, you might ask for one of the possible interfaces of the object, using the `TGUID` parameter.

Globally Unique IDentifiers

The `QueryInterface` method has a special parameter of the `TGUID` type. This is an ID that identifies any COM server class and any interface in the system. When you want to know whether an object supports a specific interface, you ask the object whether it implements the interface that has a given ID (which for the default OLE interfaces is determined by Microsoft).

Another ID is used to indicate a specific class, a specific server. The Windows Registry stores this ID, with indications of the related DLL or executable file. The developers of an OLE server define the class identifier.

Both of these IDs are known as GUIDs, or *Globally Unique IDentifiers*. If each developer uses a number to indicate its own OLE servers, how can we be sure that these values are not duplicated? The short answer is that we cannot. The real answer is that a GUID is such a long number (with 16 bytes, or 128 bits, or a number with 38 digits!) that it is statistically impossible to come up with two random numbers having the same value. Moreover, programmers should use the specific API call `CoCreateGuid` (directly or through their development environment), to come up with a valid GUID that reflects some system information.

In fact, GUIDs created on machines with network cards are guaranteed to be unique, because network cards contain unique serial numbers that form a base for the GUID creation. GUIDs created on machines with CPU IDs (such as the Pentium III) should also be guaranteed unique, even without a network card. With no unique hardware identifier, GUIDs are statistically unlikely to ever duplicate.

note Besides being careful not to copy the GUID from someone else's program (which can result in two completely different COM objects using the same GUID), you should never make up your own ID by entering a casual sequence of numbers. Windows checks the IDs, and using a casual sequence won't generate a valid ID. An OLE server with an invalid ID is not recognized, and you won't get an error message! Windows also won't include an API or technique to validate a GUID. The risk with creating class IDs or Interface IDs by hand is that you could coincidentally duplicate a GUID that is already in use somewhere else in the system. However, to avoid this problem, simply press Ctrl+Shift+G in the Delphi editor, and you will get a new, properly defined GUID.

Delphi defines a `TGUID` data type (in the System unit) to hold these numbers:

```
type
  TGUID = record
    D1: Integer;
    D2: Word;
    D3: Word;
    D4: array [0..7] of Byte;
  end;
```

This structure is actually quite odd but is required by Windows. You can assign a value to a GUID using the standard hexadecimal notation, as in this code fragment:

```
const
  Class_ActiveForm1: TGUID =
    '{1AFA6D61-7B89-11D0-98D0-444553540000}';
```

If you need to generate a GUID manually and not in the Delphi environment, you can simply call the `CoCreateGuid` Windows API function, as demonstrated by the NewGuid example (see Figure 15.1). This example is so simple that I've decided not to list its code. (You can find the source code for this application along with the chapter's other examples in the Chapter 15 folder on GitHub.)

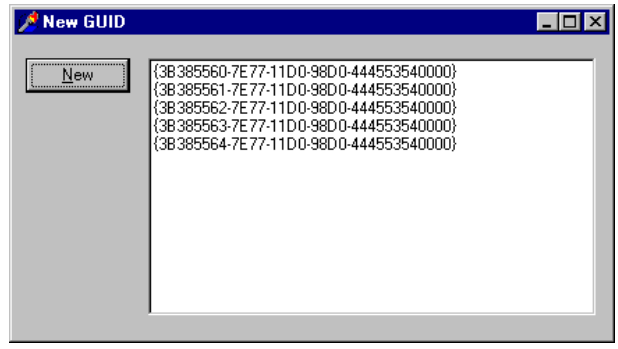


Figure 15.1: An example of the GUIDs generated by the NewGuid example. Values depend on my computer and the time I run this program. Image from the original book.

To handle GUIDs, Delphi provides the `GUIDToString` function and the opposite `StringToGUID` function. You can also use the corresponding Windows API functions, such as `StringFromGuid2`, but in this case, you must use the WideString type instead of the string type. Any time OLE is involved, you have to use the WideString type, unless you use Delphi functions that automatically do the required conversion for you. Actually, OLE API functions use the `PWChar` type (pointer to null-terminated arrays of wide characters), but simply casting a WideString to `PWChar` does the trick.

note Keep in mind that GUIDs come in different flavors. The two most important types are Interface IDs (or IID), which refer to an interface, and Class IDs (or CLSID), which refer to a specific object in a server. These two kinds of IDs both use the GUID style.

The Role of Class Factories

When we register the GUID of a COM object in the Registry, we can use a specific API function to create the object, such as the `CreateComObject` API:

```
function CreateComObject (const ClassID: TGUID): IUnknown;
```

This API function will look into the Registry, find the server registering the object with the given GUID, load it, and, if the server is a DLL, call the `DLLGetClassObject` method of the DLL. This is a function every in-process server must provide and export:

```
function DllGetClassObject (const CLSID, IID: TGUID;
  var Obj): HResult; stdcall;
```

This API function receives as parameters the requested class and interface, and it returns an object in its reference parameter. The object returned by this function is a *class factory*.

Now, what is a class factory? As the name suggests, a class factory is an object capable of creating other objects. Each server can have multiple objects. The server exposes the class factory, and the class factory can create one of these various objects. Each object, then, can have a number of interfaces. One of the many advantages of the Delphi simplified approach to COM development is that the system can provide a class factory for us. For this reason, I'm not going to add a class factory to our simple example.

The call to the `CreateComObject` API doesn't stop at the creation of the class factory, however. After retrieving the class factory, `CreateComObject` calls the `CreateInstance` method of the `IClassFactory` interface. This method creates the

requested object and returns it. If no error occurs, this object becomes the return value of the `CreateComObject` API.

By setting up this mechanism (including the class factory and the `DLLGetClassObject` call), you make it very simple to create objects. `CreateComObject` is just a simple function call with a complex behavior behind the scenes. What's great in Delphi is that the complex mechanism is handled for you by the run-time system. So it's time to start looking in detail at how Delphi makes COM really easy to master.

Class Factories and Other Delphi COM Classes

Besides the `TComObject` class, Delphi includes several other predefined COM classes. We'll use them in the following sections, but here is a list of the most important COM classes of the Delphi VCL:

- `TInterfacedObject`, defined in the System unit, inherits from `TObject` and implements the `IUnknown` interface. It is used only for internal objects.
- `TComObject`, defined in the ComObj unit, inherits from `TObject` and implements both the `IUnknown` interface and the `ISupportErrorInfo` interface. Unlike `TInterfacedObject`, this class also has a related class factory.
- `TTypedComObject`, defined in the ComObj unit, inherits from `TComObject` and implements the `IProvideClassInfo` interface (in addition to the `IUnknown` and `ISupportErrorInfo` interfaces already implemented by the base class, `TComObject`).
- `TAutoObject`, defined in the ComObj unit, inherits from `TTypedComObject` and implements also the `IDispatch` interface.
- `TActiveXControl`, defined in the AxCtrls unit, inherits from `TAutoObject` and implements a number of interfaces (`IPersistStreamInit`, `IPersistStorage`, `IOleObject`, and `IOleControl`, to name just a few).

For each of these classes, Delphi also defines a class factory. The class factory classes form another hierarchy, with the same structure. Their names are `TComObjectFactory`, `TTypedComObjectFactory`, `TAutoObjectFactory`, and `TActiveXControlFactory`. Class factories are important, and every COM server requires them. Usually we simply use class factories by creating an object in the initialization section of the unit defining the corresponding server object class.

A First COM Server

There is no better way to understand COM than to build a simple COM server hosted by a DLL. A library hosting a COM object is indicated in Delphi as an ActiveX library. For this reason we can start the development of this project by selecting File ➢ New, moving to the ActiveX page, and selecting the ActiveX Library option. This generates a project file I've saved as FirstCom. This is the complete source code:

```
library FirstCom;

uses
  ComServ;

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.RES}

begin
end.
```

The four functions exported by the DLL are required for COM compliance and are used by the system as follows:

- To access the class library (`DllGetClassObject`).
- To check whether the server has destroyed all its objects and can be unloaded from memory (`DllCanUnloadNow`).
- To add or remove information about the server in the Windows Registry (`DllRegisterServer` and `DllUnregisterServer`).

You generally don't have to implement these functions, because Delphi provides a default implementation in the ComServ unit. For this reason, in the code of our server we only need to export them.

COM Interfaces and Objects

Now that we have the structure of our COM server in place, we can start developing it. The first step is to write the code of the interface we want to implement in the server. The interface can be very similar to the code of an abstract class, listing all

the methods we want to make available from our server. (I have already discussed Object Pascal interfaces in Chapter 3.) Here is the code of a simple interface, which you should add to a separate unit (called NumIntf in the example):

```
type
  INumber = interface
    ['{B4131140-7C2F-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    procedure Increase; stdcall;
  end;
```

The IID was added to the code by pressing the Ctrl+Shift+G key combination.

After declaring the custom interface, we can add the actual object to the server. To accomplish this, we can use the COM Object Wizard (available in the ActiveX page of the usual File ➢ New dialog box). You can see this Wizard's dialog box in Figure 15.2. Here you should enter the name of the class of the server, the interface you want to implement, and a description. I've disabled the generation of the type library to avoid introducing too many topics at once. You should also choose an instancing and a threading model, as described in the related sidebar.

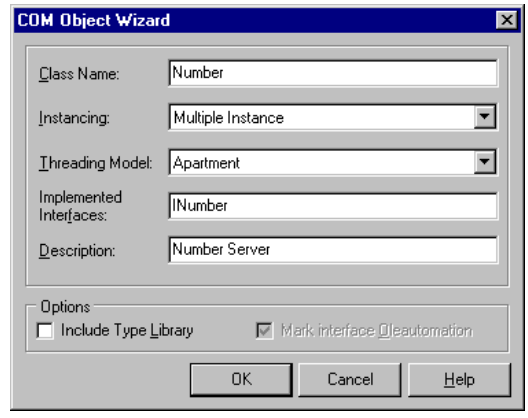


Figure 15.2: The COM Object Wizard. Image from the original book.

The code generated by the COM Object Wizard is actually quite simple. The interface contains the definition of the class to fill with methods and data:

```
type
  TNumber = class(TComObject, INumber)
    protected
      {Declare INumber methods here}
    end;
```

The server class inherits from the `TComObject` class, which I discussed in the last section. In the code generated by the wizard, after the server class comes the definition of the GUID for the server:

```
const
  Class_Number: TGUID =
  '{5B2EF181-3AAE-11D3-B9F1-00000100A27B}';
```

Finally there is some code in the `initialization` section (which uses most of the options we've set up in the wizard's dialog box):

```
initialization
  TComObjectFactory.Create(ComServer, TNumber,
  Class_Number, 'Number', 'Number Server',
  ciMultiInstance, tmApartment);
```

This code creates an object of the `TComObjectFactory` class, passing as parameters the global `ComServer` object, a class reference to the class we've just defined, the GUID for the class, the server name, the server description, and the instancing and threading models we want to use.

The global `ComServer` object, defined in the ComServ unit, is a manager of the class factories available in the server library. It uses its own `ForEachFactory` method to look for the class supporting a given COM object request, and it keeps track of the number of allocated objects. As we've already seen, in fact, the ComServ unit implements the functions required by the DLL to be a COM library.

Having examined the source code generated by the wizard, we can now complete it by adding to the `TNumber` class the methods required for implementing the `INumber` interface. First, write the declaration of the methods:

```
type
  TNumber = class(TComObject, INumber)
  private
    fValue: Integer;
  public
    function GetValue: Integer; virtual; stdcall;
    procedure SetValue (New: Integer); virtual; stdcall;
    procedure Increase; virtual; stdcall;
  end;
```

At this point, simply activate class completion by pressing Shift+Ctrl+C and fill the methods with the proper code. This is so straightforward that I'm not going to list it here; you can find the source code in the FirstCom folder.

COM Instancing and Threading Models

When you create a COM server, you should choose a proper instancing and threading model, which can significantly affect the behavior of the COM server.

Instancing affects only out-of-process servers (any COM server in a separate executable file, rather than a DLL) and can assume three values:

- *Multiple* indicates that when several client applications require the COM object, the system starts multiple instances of the server.
- *Single* indicates that even when several client applications require the COM object, there is only one instance of the server application; it creates multiple internal objects to service the requests.
- *Internal* indicates that the object can only be created inside the server; client applications cannot ask for one.

The second decision relates to the thread support of the COM object, which is valid for in-process servers only (DLLs). The threading model is a joint decision of the client and the server application: if both sides agree on one model, it is used for the connection. If no agreement is found, COM can still set up a connection using marshaling, which can slow down the operations. Also keep in mind that a server must not only publish its threading model in the Registry (as a result of setting the option in the wizard), it must also follow the rules for that threading model in the code. Here are the key highlights of the various threading models:

- The single model indicates no real support for threads. The requests reaching the COM server are serialized, so that the client can perform one operation at a time.
- The apartment model, or single-threaded apartment, indicates that only the thread that created the object can call its methods. This means that the requests for each server object are serialized, but other objects of the same server can receive requests at the same time. For this reason, the server object must take extra care only to access global data of the server (using critical sections, mutexes, or some of the other synchronization techniques described in Chapter 17). This is the threading model generally used for ActiveX controls inside Internet Explorer.
- The Free model, or multithread apartment, indicates that the client has no restrictions, which means that multiple threads can use the same object at

the same time. For this reason, every method of every object must protect itself and the nonlocal data it uses against multiple simultaneous calls. This threading model is more complex to support for a server than the single and apartment models, because even access to the object's own instance data must be handled with thread-safe care.

- The final option, Both, indicates that the server object supports both the apartment model and the free model.

Initializing the COM Object

If you look back at the definition of the `TComObject` class, you will notice it has a non-virtual constructor. (Actually, it has multiple non-virtual constructors, which I've omitted from the listing.) Each `TComObject` constructor calls the virtual `Initialize` method. For this reason, if you want to customize the creation of an object and then initialize it, you should not define a new constructor (which will never be called). What you should do is override its `Initialize` method, as I've done in the `TNumber` class. Here is the final version of this class:

```
type
  TNumber = class(TComObject, INumber)
  private
    fValue: Integer;
  public
    function GetValue: Integer; virtual; stdcall;
    procedure SetValue (New: Integer); virtual; stdcall;
    procedure Increase; virtual; stdcall;
    procedure Initialize; override;
    destructor Destroy; override;
  end;
```

As you can see, I've also overridden the destructor of the class, because I wanted to test the automatic destruction of the COM objects provided by Delphi. Here is the code for this pseudo-constructor and the destructor:

```
procedure TNumber.Initialize;
begin
  inherited;
  fValue := 10;
end;

destructor TNumber.Destroy;
begin
  inherited;
```

```
  MessageBox (0, 'Object Destroyed',
    'TDLLNumber', mb_OK); // API call
end;
```

In the first method, calling the inherited version is good practice, even though the `TComObject.Initialize` method has no code in this version of Delphi. The destructor, instead, must call the base class version. This is the code required to make our COM object work properly and to let us know when an object is actually destroyed.

Testing the COM Server

Now that we've finished writing our COM server object, we can register and use it. To register this server, you can simply compile its code and then use the Run ➢ Register ActiveX Server menu command in Delphi. You do this to register the server on your own machine, with the results you can see in Figure 15.3.

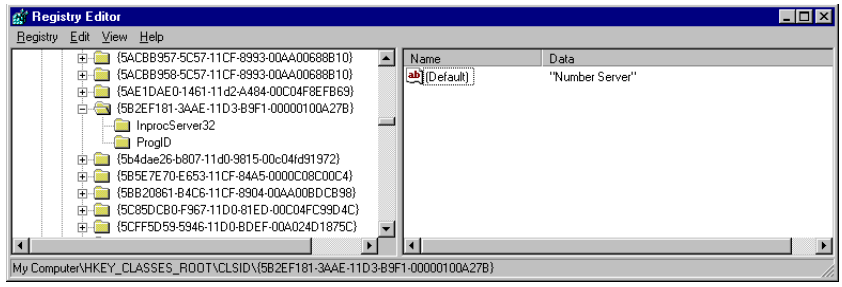


Figure 15.3: The new registered server in Windows RegEdit. Image from the original book.

When you distribute this server, you should install it on the client computers. To accomplish this you can write a REG file to install the server in the Registry. However, this is not really the best approach, because the server already includes a function you can activate to register the server. This function can be activated by the Delphi environment, as we've seen, or in a few other ways:

- You can pass the COM server DLL as a command-line parameter to Microsoft's `RegSvr32.exe` program, found in the Windows/system directory.
- You can use the similar `TRegSvr.exe` demo program that ships with Delphi. (The compiled version is in the Bin directory, and its source code is in the ActiveX sub-directory of the Demos directory.)
- You can let an installation builder program call the registration function of the server.

Having registered the server, we can now turn to the client side of our example. This time the example is called TestCOM and is stored in a separate directory. In fact, the program loads the server DLL through the OLE/COM mechanism, thanks to the server information present in the Registry, so it's not necessary for the client to know which directory the server resides in.

The form displayed by this program is very similar to the one we've used to test the object inside the DLL. In the client program, you must include the source code file with the interface and redeclare the COM server GUID. Of course, the code of the program's `FormCreate` method should be updated to create the required COM objects. The program starts with all the buttons disabled (at design time), and it enables them only after an object has been created. This way, if an exception is raised while creating one of the objects, the buttons related to the object won't be enabled:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // create first object
  Num1 := CreateComObject (Class_Number) as INumber;
  Num1.SetValue (SpinEdit1.Value);
  Label1.Caption := 'Num1: ' + IntToStr (Num1.GetValue);
  Button1.Enabled := True;
  Button2.Enabled := True;

  // create second object
  Num2 := CreateComObject (Class_Number) as INumber;
  Label2.Caption := 'Num2: ' + IntToStr (Num2.GetValue);
  Button3.Enabled := True;
  Button4.Enabled := True;
end;
```

Notice in particular the call to `CreateComObject` and the following `as` cast. The API call starts the COM object-construction mechanism I've already described in detail. This call also dynamically loads the server DLL. The return value is an `IUnknown` object. This object must be converted to the proper interface type before assigning it to the `Num1` and `Num2` fields, which now have the interface type `INumber` as their data type:

```
type
  TForm1 = class(TForm)
    ...
  private
    Num1, Num2 : Inumber;
```

note To downcast an interface to the actual type, *always* use the `as` cast, which for interfaces performs a `QueryInterface` call behind the scenes. This provides some protection, because it raises an exception if the interface you are casting to is not supported by the given object. In the case of interfaces, the `as` cast is the only way to *extract* an interface from another interface. If you write a plain cast of the form `INumber(CreateComObject (Class_Number))`, the program will crash, even if the cast seems to make sense as in the case above. Casting an interface pointer to another interface pointer is an error. Period. Never do it.

In Figure 15.4 you can see the output of this test program, which is very similar to the previous version. Notice that this time, `Num2` shows the initial value of the object at start-up, as set up in its `Initialize` method. Notice also that I've added one more button, which creates a third temporary COM object:

```
procedure TForm1.Button5Click(Sender: TObject);
var
  Num3: INumber;
begin
  // create a new temporary COM object
  Num3 := CreateComObject (Class_Number) as INumber;
  Num3.SetValue (100);
  Num3.Increase;
  ShowMessage ('Num3: ' + IntToStr (Num3.GetValue));
end;
```

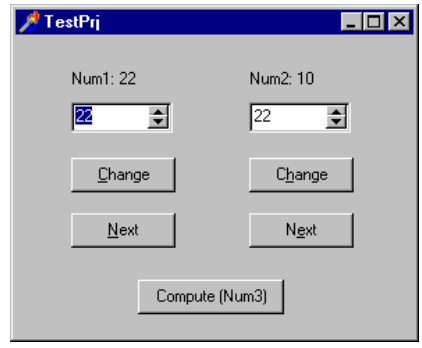


Figure 15.4: The output of the TestCom example, a COM client. Image from the original book.

Pressing this button, you simply get the value of the number following 100. To see why I added this method to the example, you need to press the button a second time, after the message showing the result. Now you get a second message, indicating that the object has been destroyed. This demonstrates that simply letting an interface object go out of scope automatically calls the object's `Release` method, decreases the object's reference count, and destroys the object if its reference count

reaches zero. Chapter 3 described this reference-counting mechanism in more detail.

The same happens to the other two objects as soon as the program terminates. Even if the program doesn't explicitly destroy the two objects in the `FormDestroy` method, they are indeed destroyed, as the message shown by their `Destroy` destructor clearly demonstrates. This happens because they were declared to be of an interface type, and Delphi is going to use reference counting for them.

Using Interface Properties

As a further small step, we can extend the example by adding a property to the `INumber` interface. When you add a property to an interface, you indicate the data type and then the `read` and `write` directives. You can have read-only or write-only properties, but the `read` and `write` clauses must always refer to a method because interfaces don't hold anything else but methods.

Here is the updated interface, which is part of the PropCom example:

```
type
  INumberProp = interface
    ['{B36C5800-8E59-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    property Value: Integer
      read GetValue write SetValue;
    procedure Increase; stdcall;
  end;
```

I've given this interface a new name and, what's even more important, a new interface ID. I could have inherited the new interface type from the previous one, but this would have provided no real advantage. COM by itself doesn't really support inheritance, and from the perspective of COM, all interfaces are different simply because they have a different interface ID. Needless to say, in Delphi we can use inheritance to improve the structure of the code of the interfaces and of the server objects implementing them.

In the PropCom example I've updated the server class declaration simply by writing:

```
type
  TDllNumber = class (TComObject, INumberProp)
  ...
```

This class also has a new server object ID. The client program, saved in the TestProp directory, can now simply use the `Value` property instead of the `SetValue` and `GetValue` methods. Here is a small excerpt from the `FormCreate` method:

```
Num1 := CreateComObject (Class_NumPropServer) as INumberProp;
Num1.Value := SpinEdit1.Value;
Label1.Caption := 'Num1: ' + IntToStr (Num1.Value);
```

The difference between using methods and properties for an interface is only syntactical, because interface properties cannot access private data as class properties can. By using properties, we can make the code a little more readable.

Calling Virtual Methods

We've built a couple of examples based on COM; but you might still feel uncomfortable with the idea of a program calling methods of objects that are created within a DLL. How is this possible if those methods are not exported by the DLL? The COM server, the DLL, creates an object and returns it to the calling application. By doing this, the DLL creates an object with a virtual method table. (Remember that all the interface methods are virtual by default.)

Because every object embeds a pointer to its virtual method table, the main program receives an object, and also a way to work on it, by calling its virtual methods. The main program doesn't need to know the memory address of those methods, because the objects know it, exactly as they do with a polymorphic call. But COM is even more powerful than this: you don't even have to know which programming language was used to create the object, provided its VMT follows the standard dictated by COM.

note The COM-compatible VMT implies also a strange effect. The method names are not important, provided their address is in the proper position in the VMT. This is why you can map a method of an interface to an actual function implementing it.

To round up things, we can say that COM provides a language-independent binary standard for objects. The object you share among modules are compiled, and their VMT has a determined structure, which is determined by COM and not by the development environment you've used.

Using a Shell Interface

In the last section, we built a fully standard COM object, packaged it as an in-process server, and used it from a standard client. However, the COM interface we implemented was a custom interface we'd built. Now we can try to build clients and servers related to the Windows shell interfaces, which are all based on COM. The original Windows API was basically a collection of functions, but all the most recent APIs are generally based on COM.

The following sections use some existing servers that are part of the Windows shell; in this case, we'll write a client application and use the COM servers provided by the system. This case illustrates the difference from the traditional use of the Windows API calls. I'm also going to write some COM servers to be used by the Windows system, particularly the Explorer. This case illustrates the difference from the traditional development of a callback function invoked by the system.

Creating Shortcuts

One of the simplest shell interfaces we can use in a client application is the `IShellLink` interface. This interface relates to Windows shortcuts and allows programmers to access the information of an existing shortcut or to create a new one. In the ShCut example, I'm going to create various types of shortcuts, all referring to the program itself. Of course, once you understand how to do this, you can easily extend the example and create shortcuts for any program or file.

The example has an edit box for the name of the shortcut, a few check boxes, and two buttons. When the Create button is pressed, the text in the edit box is used as the name of a new shortcut, which is placed in the current directory, on the desktop, or in the start menu. These options are not exclusive; a user can create multiple shortcuts at once.

The most important code is at the very beginning of this method. The `CreateComObject` call creates a system object, as indicated by the GUID passed as a parameter. The result of this call (which is an `IUnknown` interface) is converted both to an `IShellLink` interface and to an `IPersistFile` interface:

```
uses
  ComObj, ActiveX, ShlObj, Registry;

procedure TForm1.Button1Click(Sender: TObject);
var
```

```
  AnObj: IUnknown;
  ShLink: IShellLink;
  PFile: IPersistFile;
  FileName: string;
  WFileName: WideString;
  Reg: TRegIniFile;
begin
  // access the two interfaces of the object
  AnObj := CreateComObject (CLSID_ShellLink);
  ShLink := AnObj as IShellLink;
  PFile := AnObj as IPersistFile;
```

Actually, we could have written the three lines of code above using this shorter notation:

```
ShLink := CreateComObject (CLSID_ShellLink) as IShellLink;
PFile := ShLink as IPersistFile;
```

If you look at similar examples built in other languages, you'll notice that to access the `IPersistFile` interface, the programs use custom calls to the `QueryInterface` method. The two `as` expressions basically call `QueryInterface` for us.

Once we have the `IShellLink` interface, we can call some of its methods, such as `SetPath` and `SetWorkingDirectory`:

```
  // get the name of the application file
  FileName := ParamStr (0);
  // set the link properties
  ShLink.SetPath (PChar (FileName));
  ShLink.SetWorkingDirectory (PChar (
    ExtractFilePath (FileName)));
```

Once we've set up the shell link object, we have to save it, depending on the status of the three check boxes, calling the `Save` method of the `IPersistFile` interface of the object. The simplest version is the one used to save the link in the current directory:

```
  // save the file in the current dir
  if cbDir.Checked then
  begin
    // using a WideString
    WFileName := ExtractFilePath (FileName) +
      EditName.Text + '.lnk';
    PFile.Save (PWChar (WFileName), False);
  end;
```

The call to the `Save` method (which creates the physical LNK file) requires a "pointer to wide char" parameter. The simplest way to obtain this is to declare a long string and then cast it to a `PWChar`. Do not try casting a plain string to `PWChar`—the compiler will not complain, but the program won't work!

To create the shortcut on the desktop or in the Start menu, we should first determine the corresponding system folder by looking up the proper value in the Registry. By writing the program this way, we ensure it will work on different versions of Windows and on localized versions as well. Here is the source code for the last two check boxes:

```
// save on the desktop
if cbDesktop.Checked then
begin
  Reg := TRegIniFile.Create(
    'Software\MicroSoft\Windows\CurrentVersion\Explorer');
  WFileName := Reg.ReadString ('Shell Folders', 'Desktop', '') +
    '\' + EditName.Text + '.lnk';
  Reg.Free;
  PFile.Save (PWChar (WFileName), False);
end;
// save in the Start Menu
if cbStartMenu.Checked then
begin
  Reg := TRegIniFile.Create(
    'Software\MicroSoft\Windows\CurrentVersion\Explorer');
  WFileName := Reg.ReadString ('Shell Folders', 'Start Menu', '') +
    '\' + EditName.Text + '.lnk';
  Reg.Free;
  PFile.Save (PWChar (WFileName), False);
end;
```

To look up the information in the Registry I've used the `TRegIniFile` class, although there are other related classes in the VCL, such as the `TRegistry` class. The effect of running this program and pressing the button is that Windows will add a new link in the directory of the project, on the desktop, or in the Start menu. You can see an example of the program in Figure 15.5.

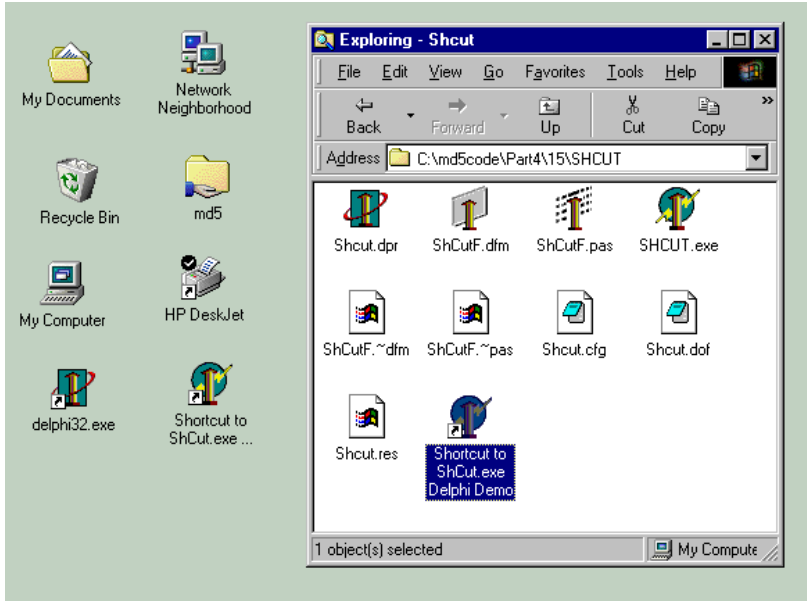


Figure 15.5: The simple user interface of the ShCut example, and two shortcuts created with it in the project folder and on the desktop. Image from the original book.

The "To-Do File" Application

As a second example of integrating a Delphi program with the system shell, I've tried to write a simple real-world application, which uses file dragging and a context menu handler. I'll start with the file dragging first, because this will actually introduce some of the techniques used by the context menu handler.

As I mentioned, this application is actually useful; you can use it to create a sort of "to-do list." It is based on a Paradox table that stores filenames and notes about the files. The form of the application has a DBGrid component showing only a single column containing the filenames and a memo control hosting the notes related to the current file. You can see this form at design time in Figure 15.6.

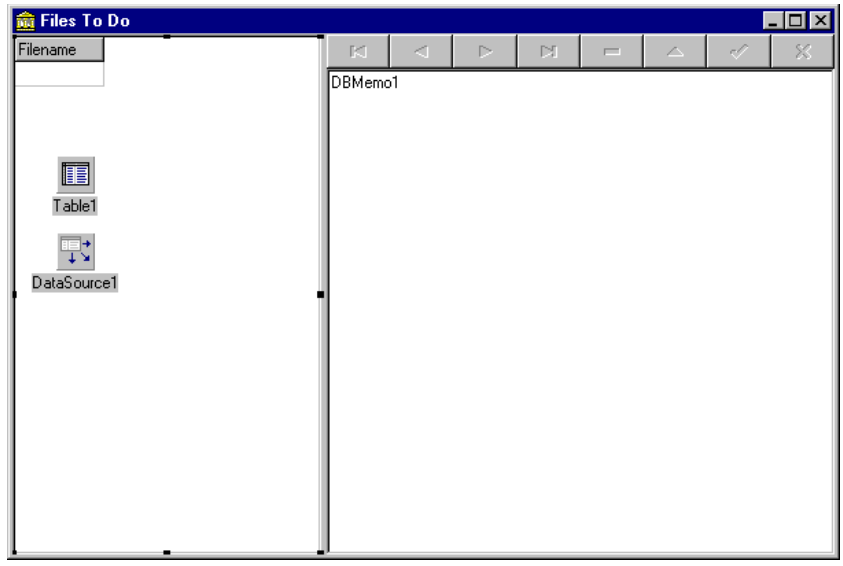


Figure 15.6: The form of the ToDoFile example at design time. Image from the original book.

note Using a single-column DBGrid is the only way in Delphi to show a list of the available records in a listbox format. The alternative, of course, is to fill a listbox with custom code and then manually navigate in the database table when the selection in the listbox changes. This manual approach is, of course, less efficient when we have many records, because the program needs to scan them all to fill the listbox, while the DBGrid loads only the record it currently displays.

Notice that the navigator component has no "new record" button, and the DBGrid is set up as a read-only component. In fact, a user should not be able to create new records except by dragging a file onto the form and is not allowed to change the file-name field in any way (except by deleting it). All the user can do is edit the notes field, entering a description of the operations to be done on the file.

Creating the Database

To create the database table for this example, I've used the `FieldDefs` property to define the structure and set the `StoreDefs` property to `True` to save the table definition along with the form DFM file. The table has two fields, a string field called Filename and a memo field called Notes. Of course, you can also create the table at design time, using the table component's local menu. The program, however, calls the `CreateTable` method in the `OnCreate` event handler, unless this has already been done:

```
procedure TToDoFileForm.FormCreate(Sender: TObject);
begin
  // eventually create the table
  if not Table1.Exists then
    Table1.CreateTable;
  // activate the table
  Table1.Activate;
  // accept dragging to the form
  DragAcceptFiles (Handle, True);
end;
```

Dragging Files to the Form

As you can see in the listing above, the form initialization code also registers the window with the system as a file-dragging target, by calling the `DragAcceptFiles` Windows API function. As a result, the application's cursor changes to the typical "drag accept" icon when a file is dragged over it. You can see an example of this cursor in Figure 15.7.

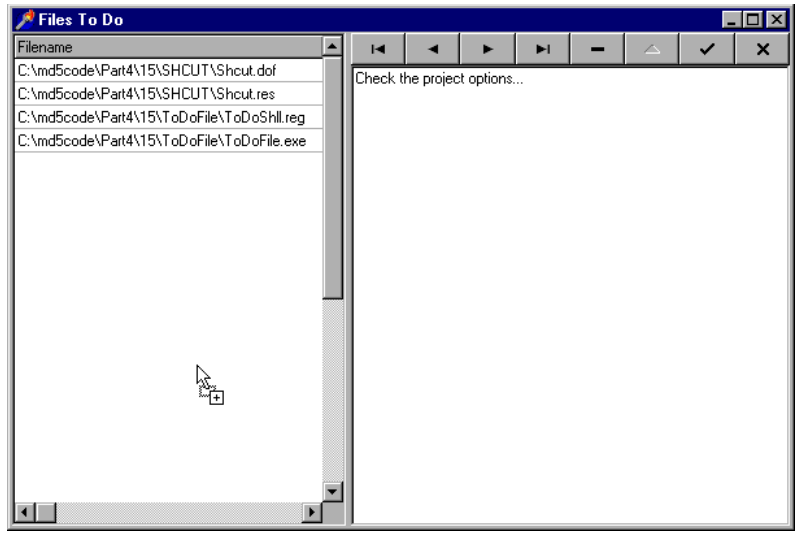


Figure 15.7: The drag-accept cursor displayed by the ToDoFile application as a user drags a file over it. Image from the original book.

When a file-dragging operation is performed, the system sends the window a `wm_DropFiles` message. This message passes (among its other parameters) a handle to a file-drop structure from which you can extract information by using the `DragQueryFile` API function. When this API function is called with the `$FFFFFFFF` parameter, it returns the number of files dragged to the window; when it is called with a numeric parameter it fills a buffer with the name of that file. For this reason,

the code of a `wm_DropFiles` message handler gets the number of files first and then loops for each of the files, as the following listing demonstrates:

```
procedure TToDoFileForm.DropFiles(var Msg: TWmDropFiles);
var
  nFiles, I: Integer;
  Filename: string;
begin
  // get the number of dropped files
  nFiles := DragQueryFile (Msg.Drop, $FFFFFFFF, nil, 0);
  // for each file
  try
    for I := 0 to nFiles - 1 do
    begin
      // allocate memory
      SetLength (Filename, 80);
      // read the file name
      DragQueryFile (Msg.Drop, I, PChar (Filename), 80);
      // normalize file
      Filename := PChar (Filename);
      // add a new record
      Table1.InsertRecord ([Filename, '']);
    end;
  finally
    DragFinish (Msg.Drop);
  end;
  // open the (last) record in edit mode
  Table1.Edit;
  // move the input focus to the memo
  DBMemo1.SetFocus;
end;
```

As you can see in the code above, for every new file the program inserts a new record, with the corresponding filename and an empty field for the notes. Then, for the last file being dragged, the program opens the record in edit mode and moves the focus to the memo control, so that a user can fill the notes for the file.

Creating a Context-Menu Handler

Now that we have the base program running, we can add a shell extension to the system to let the user simply select a file and "send" it to the application without having to do the dragging operation, which is not always handy when there are many programs running. A context menu extension is one of the available Windows

shell extensions and is activated every time a user right-clicks on a file in the Windows Explorer[359].

Technically, a context menu is a COM server exposing an internal object that is going to be created and used by the system. A context-menu COM object must implement two different interfaces, `IContextMenu` and `IShellExtInit`. The first interface defines specific actions for the context menu, such as defining the number of menu items to add and their text, while the second interface defines a way to access the file or files the user is operating on. This is the resulting definition of the COM server object class:

```
type
  TToDoMenu = class(TComObject, IUnknown,
    IContextMenu, IShellExtInit)
  private
    fFileName: string;
  protected
    {Declare IContextMenu methods here}
    function QueryContextMenu(Menu: HMENU; indexMenu,
      idCmdFirst, idCmdLast, uFlags: UINT): HResult; stdcall;
    function InvokeCommand(
      var lpici: TCMInvokeCommandInfo): HResult; stdcall;
    function GetCommandString(idCmd, uType: UINT;
      pwReserved: PUINT; pszName: LPSTR;
      cchMax: UINT): HResult; stdcall;
    {Declare IShellExtInit methods here}
    function IShellExtInit.Initialize = InitShellExt;
    function InitShellExt (pidlFolder: PItemIDList;
      lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
  end;
```

Notice that the class implements the `Initialize` method of the `IShellExtInit` interface with a differently named method, `InitShellExt`. The reason is that I wanted to avoid confusion with the `Initialize` method of the `TComObject` base class, which is the hook we have to initialize the object, as described earlier in this chapter. Let us examine `InitShellExt` method first; it is definitely the most complex one:

```
function TToDoMenu.InitShellExt(pidlFolder: PItemIDList;
  lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
var
  medium: TStgMedium;
  fe: TFormatEtc;
begin
```

---

359 In Windows 11 the UI has changed and this feature remains available in a second level menu. Notice how Windows 11 Explorer first level and second level local menus are build on different technologies, which makes the user experience really odd, in my opinion.

```
  Result := E_FAIL;
  // check if the lpdobj pointer is nil
  if Assigned (lpdobj) then
  begin
    with fe do
    begin
      cfFormat := CF_HDROP;
      ptd := nil;
      dwAspect := DVASPECT_CONTENT;
      lindex := -1;
      tymed := TYMED_HGLOBAL;
    end;
    // transform the lpdobj data to a storage medium structure
    Result := lpdobj.GetData(fe, medium);
    if not Failed (Result) then
    begin
      // check if only one file is selected
      if DragQueryFile (medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
      begin
        SetLength (fFileName, 1000);
        DragQueryFile (medium.hGlobal, 0, PChar (fFileName), 1000);
        // realign string
        fFileName := PChar (fFileName);
        Result := NOERROR;
      end
      else
        Result := E_FAIL;
    end;
    ReleaseStgMedium(medium);
  end;
end;
```

The initial portion of the method transforms the pointer to the `IDataObject` interface, which we receive as a parameter, into the same data structure used in a file drop operation, so that we can read the file information by using the `DragQueryFile` function again. This complex way of coding is actually the simplest one you can use! At the end of this operation, we have the value of the filename. Any selection of multiple files is not accepted.

We can now look at the methods of the `IContextMenu` interface. The first, `QueryContextMenu`, is used to add new items to the local menu of the file. In this case, we add a new menu item (calling the `InsertMenu` API function) only if the ToDoFile application is running. We can determine this by searching for a window corresponding to the `TToDoFileForm` class, which should be unique in the system. The result of the function is the number of items added to the menu:

```
function TToDoMenu.QueryContextMenu(Menu: HMENU;
  indexMenu, idCmdFirst, idCmdLast, uFlags: UINT): HResult;
begin
  // add entry only if the program is running
```

```
  if FindWindow ('TToDoFileForm', nil) <> 0 then
  begin
    // add a new item to context menu
    InsertMenu (Menu, indexMenu,
      MF_STRING or MF_BYPOSITION, idCmdFirst,
      'Send to ToDoFile');
    // return the number of menu items added
    Result := 1;
  end
  else
    Result := 0;
end;
```

Now that items have been added to the menu, a user can select them. While he or she moves over the items, a descriptive message is displayed in the status bar of the Windows Explorer, as you can see in Figure 15.8. The menu ID (`idCmd`) we receive in the `GetCommandString` method is simply the relative number, starting with zero, of the items we have added to the menu. When the cursor is over an item, we simply copy a string with its description to the buffer provided by the system:

```
function TToDoMenu.GetCommandString(idCmd, uType: UINT;
  pwReserved: PUINT; pszName: LPSTR; cchMax: UINT): HRESULT;
begin
  if idCmd = 0 then
  begin
    // return help string for menu item
    strCopy (pszName, 'Add file to the ToDoFile database');
    Result := NOERROR;
  end
  else
    Result := E_INVALIDARG;
end;
```

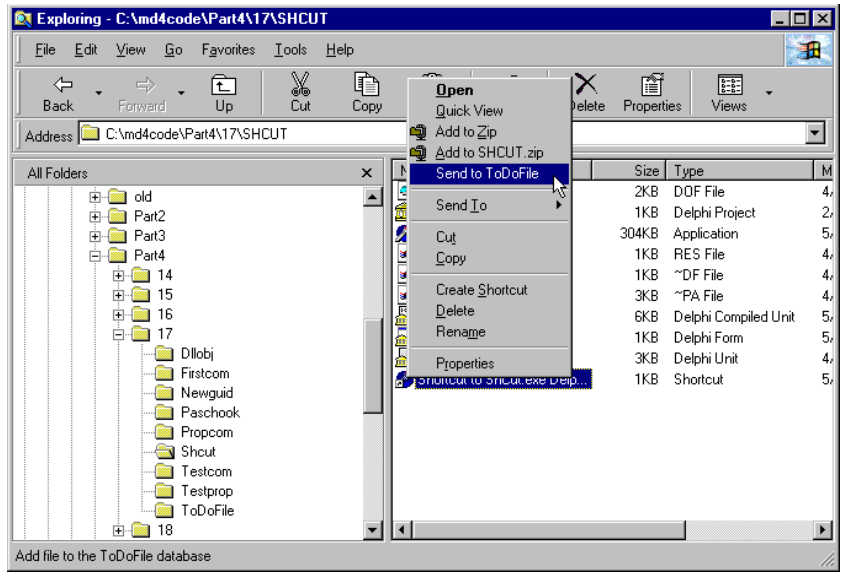


Figure 15.8: As a user moves over the new item of the local menu of the Windows Explorer, a description is displayed in the status bar. Image from the original book.

The final step is the operation to do once a menu item is actually selected. The `InvokeCommand` method receives a pointer to a structure holding the request. This method follows a standard pattern of first checking that the request is valid by looking at the two 16-bit words of the `lpici.lpVerb` value. After these preliminary (but required) steps, we check the value to see which menu item was activated; or, if the context menu has only one item, as in this case, we simply test for a value of zero. The following is the skeleton of the code, before we add the specific action:

```
function TToDoMenu.InvokeCommand (
  var lpici: TCMInvokeCommandInfo): HResult;
begin
  Result := NOERROR;
  // make sure we are not being called by an application
  if HiWord(Integer(lpici.lpVerb)) <> 0 then
  begin
    Result := E_FAIL;
    Exit;
  end;
  // make sure we aren't being passed an invalid argument number
  if LoWord(lpici.lpVerb) > 0 then
  begin
    Result := E_INVALIDARG;
    Exit;
  end;
  // execute the command specified by lpici.lpVerb
  if LoWord(lpici.lpVerb) = 0 then
```

```
  begin
    // actual code still missing here
  end
end;
```

Sending Data to Another Application with wm_CopyData

Because we have the filename the user is operating on, all we have to do in the context-menu handler is send this name to the main form of the ToDoFile application. The problem is that the context-menu handler DLL runs in the Windows Explorer process, so it cannot send the value of a memory pointer to another process. This would simply be useless; as in Windows 32, different applications have separate memory address spaces.

We saw in the last chapter that one way to share data among applications is to use a memory-mapped file. Another technique, which is actually better in this case, is to use the `wm_CopyData` message. This is a special Windows message, which can be used to send a memory buffer to another application: Windows will resolve all the memory conversion problems for us. A program basically fills the `CopyDataStruct` data structure with the data and indicates its length, and then must use the `SendMessage` API to forward it to a destination window. For this reason we need to use `FindWindow` again to get the handle of the main window of the ToDoFile application. Here is the rest of the code of the `InvokeCommand` method:

```
var
  hwnd: THandle;
  cds: CopyDataStruct;
begin
  ...
  if LoWord(lpici.lpVerb) = 0 then
  begin
    // get the handle of the window
    hwnd := FindWindow ('TToDoFileForm', nil);
    if hwnd <> 0 then
    begin
      // prepare the data to copy
      cds.dwData := 0;
      cds.cbData := length (fFileName);
      cds.lpData := PChar (fFileName);
      // activate the destination window
      SetForegroundWindow (hwnd);
      // send the data
      SendMessage (hwnd, wm_CopyData,
        lpici.hWnd, Integer (@cds));
    end;
  end;
```

note Before sending the data, we must activate the destination window by calling the `SetForegroundWindow` API. This is necessary because we are going to activate a window that was created by another thread, something Windows doesn't normally do. Notice also that if you write this call in the ToDoFile application as it receives the `wm_CopyData` message, it will produce no effect at all.

As the context-menu handler sends data to it, the application has to be extended to handle the `wm_CopyData` message. In this event handler we receive the same structure we sent for the other side, although between the send operation done by the context menu handler and the receive operation done by the application. Windows takes care of mapping the data properly to the other address space. As a result, extracting the filename is actually very simple, but keep in mind that this is so only because Windows does a lot of work behind the scenes. Using a plain Windows message other than `wm_CopyData` will never work!

Here is the code I've added to the form of the ToDoFile application. It does several things: It restores the application if it was minimized, retrieves the name of the file, inserts a new record in the database table, copies the filename, and moves the focus to the memo control once more.

```
procedure TToDoFileForm.CopyData(var Msg: TWmCopyData);
var
  Filename: string;
begin
  // restore the window if minimized
  if IsIconic (Application.Handle) then
    Application.Restore;

  // extract the filename from the data
  Filename := PChar (Msg.CopyDataStruct.lpData);
  // insert a new record
  Table1.Insert;
  // set up the file name
  Table1.FieldByName ('Filename').AsString := Filename;
  // move the input focus to the memo
  DBMemo1.SetFocus;
end;
```

Registering the Shell Extension

After writing this shell extension, we must register it. With the usual Run ➢ Register ActiveX command, we can register the server in the system, but we still have to provide some extra information to register it as a shell extension, in this case for any type of file. There are several approaches, besides manually editing the Registry. You can write a REG file, along this line:

```
REGEDIT4

[HKEY_CLASSES_ROOT\CLSID\{CDF05220-DB84-11D1-B9F1-004845400FAA}]
@= "ToDoFile Context Menu"
[HKEY_CLASSES_ROOT\CLSID\{CDF05220-DB84-11D1-B9F1-004845400FAA}
  \InProcServer32]
@= "c:\\md5code\\Part4\\15\\ToDoFile\\ToDoShll.dll"
"ThreadingModel" = "Apartment"

[HKEY_CLASSES_ROOT\*\shellex\ContextMenuHandlers\
  {CDF05220-DB84-11D1-B9F1-004845400FAA}]
@= ""
```

The first part of this file corresponds to the registration already done by installing the server, while the final part adds the server as a context menu handler for all files (as indicated by the `*` symbol under the `HKEY_CLASSES_ROOT` path).

A totally different but much better approach is to add the registration information right into the COM server library. The default registration takes place in the `TComObjectFactory` class, when the `UpdateRegistry` method is executed. We can modify the default registration by inheriting a class from the standard class factory class and overriding this method:

```
type
  TToDoMenuFactory = class (TComObjectFactory)
  public
    procedure UpdateRegistry (Register: Boolean); override;
  end;
```

In this method we should either add the entry in the Registry or delete it, depending on the value of the Boolean parameter:

```
procedure TToDoMenuFactory.UpdateRegistry(Register: Boolean);
var
  Reg: TRegistry;
begin
  inherited UpdateRegistry (Register);

  Reg := TRegistry.Create;
  try
    // register or remove the menu handler
    if Register then
      Reg.CreateKey (
        '\HKEY_CLASSES_ROOT\*\ShellEx\ContextMenuHandler\' +
        GUIDToString (Class_ToDoMenuMenu))
    else
      Reg.DeleteKey (
        '\HKEY_CLASSES_ROOT\*\ShellEx\ContextMenuHandler\' +
        GUIDToString (Class_ToDoMenuMenu));
  finally
    Reg.Free;
```

```
  end;
end;
```

In the initialization section of the COM object unit, we also need to create a new global object of this class instead of the base class factory class:

```
initialization
  TToDoMenuFactory.Create (
    ComServer, TToDoMenu, Class_ToDoMenuMenu,
    'ToDoMenu', 'ToDoMenu Shell Extension',
    ciMultiInstance, tmApartment);
```

Now you can simply register the server and set it up as a context menu handler by using the Delphi Run ➢ Register ActiveX Server menu command, the `RegSrv32` application, or most of the tools used to create installation programs.

What's Next?

In this chapter I have discussed the foundations of Microsoft's COM technology. We've seen how Delphi supports COM and how Delphi makes the development of Explorer extensions very simple.

Next chapter opens up COM to its higher-level techniques, covering Automation, Documents, and Controls. Now that we know the foundations, exploring these COM-technologies will be definitely simpler.

We'll get back to other elements related to COM when discussing Internet and distributed applications, in Chapter 20.

Chapter 16: Automation And ActiveX

After the last chapter, which was devoted to the foundations of Microsoft's COM architecture, it is time to look into some of the actual high-level Windows programming techniques based on COM. We'll start by discussing Automation and the role of Type Libraries. Also, we'll see how to work properly with Delphi data types in Automation servers and clients[360].

Later on we'll focus on the use of the Automation support provided by Microsoft Office applications, made even simpler in Delphi 5 thanks to some ready-to-use components that embed Office server programs and documents.

---

360 While the COM foundations are still very important today, the topics covered in this chapter have loft some of their relevance. In any case, they are still supported by Windows and by Dlephi today.

In the final part of the chapter, we'll explore the use of embedded objects, with the OleContainer component, and the development of OLE Controls or ActiveX Controls. But let's begin with more foundation material.

OLE Automation

In the last chapter, we saw that you can use COM to let an executable file and a library share objects and that this can be used to interact with the Windows shell. Most of the time, however, users want applications that can talk to each other. One of the approaches you can use for this goal is OLE Automation. After presenting a couple of examples that use custom interfaces based on type libraries, I'll cover the development of Word and Excel OLE controllers, showing how to transfer database information to those applications.

---

note The current Microsoft documentation uses the term *Automation* instead of *OLE Automation,* and it uses the terms *active document* and *compound document* instead of *OLE Document*. This book uses this new terminology along with the older "OLE" terminology incorporated into many Delphi component names and other identifiers.

In Windows, applications don't live in separate worlds; users often want them to interact. The Clipboard and DDE offer a very simple way for applications to interact, as users can copy and paste data between applications. However, more and more programs offer an OLE Automation interface to let other programs drive them. Beyond the obvious advantage of programmed automation compared to manual user operations, these interfaces are completely language-neutral, so you can use Delphi, C++, Visual Basic, or a macro language to drive an OLE Automation server regardless of the programming language used to write it.

OLE Automation is very simple to implement in Delphi, thanks to the extensive work the VCL and the compiler do to shield developers from its intricacies. To support OLE Automation, Delphi provides a simple wizard and a powerful Type Library editor, and it supports dual interfaces.

When you use an in-process DLL, the client application can simply use the server and call its methods directly, because they are in the same address space. When you use OLE Automation, the situation is more complex. The client (called the *controller*) and the server are two separate applications running in different address spaces. For this reason, the system must dispatch the method calls using a complex mechanism called *marshaling* (something I won't cover in detail). What is impor-

tant to know is that there are two ways a controller can call the methods exposed by a server:

- It can simply ask for the execution of a method, passing its name in a string, in a way similar to the dynamic call to a DLL. This is what Delphi does when you use a variant to call the OLE Automation server. This technique is very easy to use, but it is rather slow and provides very little compiler type-checking.
- It can import the definition of a Delphi interface for the object on the server and call its methods in a more direct way (simply dispatching a number). This technique, based on interfaces, allows the compiler to check the types of the parameters and produces faster code, but it requires a little more effort from the programmer. Also, you end up binding your controller application to a specific version of the server. A variation of this technique involves the use of dispatch interfaces, based on the definition of the interfaces.

In the following examples, we'll use all these techniques and compare them a little further.

Introducing Type Libraries

The most important difference between the two approaches is that the second generally requires a *Type Library,* one of the foundations of OLE and COM. A Type Library is basically a collection of type information. This collection generally describes all of the elements (the objects, the interfaces, and other type information) made available by a server. The key difference between a Type Library and other descriptions of these elements (such as some C or Pascal code) is that a Type Library is language-independent. The type elements are defined by OLE as a subset of the standard elements of programming languages, and any development tool can use them. Why do we need this information?

As mentioned before, a simple OLE Automation controller can use variants and have no type information about the server it is using. This means that, behind the scenes, every function call has to be dispatched to the server using the `Inovke` method of `IDispatch`, passing the function name as a string parameter, and hoping the name corresponds to an existing function of the server.

Although this sounds difficult, a small code fragment using the old Automation interface of Microsoft Word, registered as `Word.Basic`, illustrates how simple it is for a programmer:

```
var
```

```
  VarW: Variant;
begin
  VarW := CreateOleObject ('Word.Basic');
  VarW.FileNew;
  VarW.Insert ('Mastering Delphi by Marco Cantu');
```

note As we'll see later, Word 97 still registers the `Word.Basic` interface, which corresponds to the internal WordBasic macro language, but it also registers the new interface `Word.Application`, which corresponds to the VBA macro language. We'll also see that Delphi 5 provides some components that simplify the connection with Microsoft Office applications.

These three lines of code start Word (unless it was already running), create a new document, and add a few words to it. You can see the effect of this code in Figure 16.1. A variant is a *type-variant* data type. It can assume as its value different data types, including a COM object supporting the `IDispatch` interface. Variants are type-checked at run time; this is why the compiler can compile the code even if it doesn't know about the methods of the OLE Automation server.

Unfortunately, the Delphi compiler has no way to check whether the methods exists. Doing all the type checks at run time is risky, because if you make even a minor spelling error in a function name, you get no warning whatsoever of your error until you run the program and reach that line of code. For example, if you type `VarW.Isnert`, the compiler will not complain about the misspelling at all, but at run time, you'll get an error. Because it doesn't recognize the name, Word assumes the method does not exist.

Although the OLE `IDispatch` interface supports the approach we've just seen, it is also possible—and safer—for a server to export the description of its interfaces and objects using a Type Library. This Type Library can then be converted by a specific tool (such as Delphi) into definitions written in the language you want to use to write your client or controller program (such as Object Pascal). This makes it possible for a compiler to check whether the code is correct.

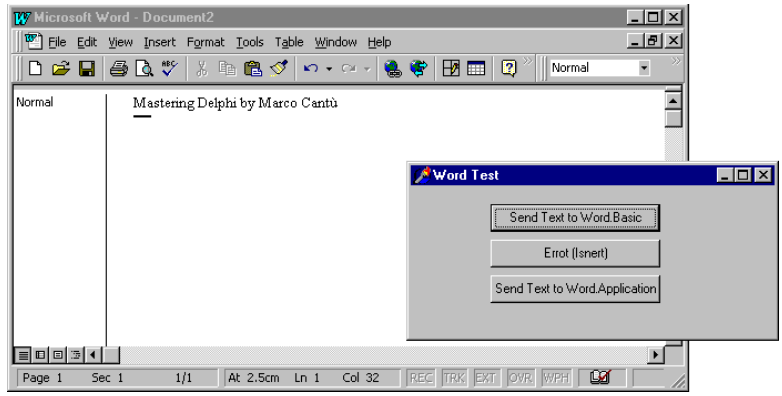


Figure 16.1: This Word document is being created and composed by a Delphi application, WordTest. Image from the original book.

Once the compiler has done its checks, it can use either of two different techniques to send the request to the server. It can use a plain VTable (that is, an entry in an interface type declaration), or it can use a `dispinterface` (dispatch interface). We used an interface type declaration in the last chapter, so it should be familiar. A `dispinterface` is basically a way to map each entry in an interface to a number. Calls to the server can then be dispatched by number. We can consider this an intermediate technique, in between dispatching by function name and using a direct call in the VTable.

note The term *dispinterface* is actually a keyword. A `dispinterface` is automatically generated by the Type library editor for every interface. Along with `dispinterface`, Delphi uses other related keywords: `dispid` indicates the number to associate with each element; `readonly` and `writeonly` are optional specifiers for properties.

The term used to describe this ability to connect to a server in two different ways, using a more dynamic or a more static approach, is *dual interfaces*. This means that in writing an OLE controller you can choose to access the methods of a server in two ways: you can use late binding and the mechanism provided by the `dispinterface`, or you can use early binding and the mechanism based on the VTables, the interface types.

It is important to keep in mind that (along with other considerations) different techniques result in faster or slower execution. Looking up a function by name (and doing the type checking at run time) is the slowest approach, using a `dispinterface` is much faster, and using the direct VTable call is the fastest approach. We'll do this kind of test in the TlibCli example, later in this chapter.

Writing an OLE Automation Server

We'll start by writing an OLE Automation server. To create an OLE Automation object, you can use Delphi's Automation Object Wizard. Simply start with a new application, open the Object Repository by selecting File ➢ New, move to the ActiveX page, and choose Automation Object. In the resulting Automation Object Wizard (shown in Figure 16.2) enter the name of the class (without the initial *T,* because this will be added automatically for you), and click OK. Delphi will now open the Type Library editor.

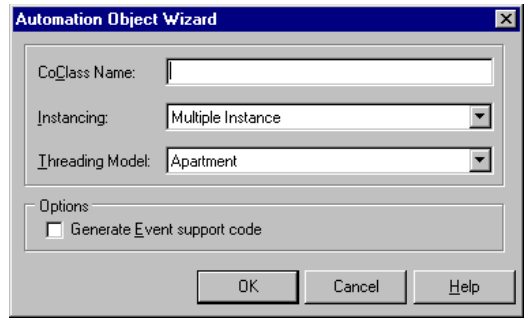


Figure 16.2: Delphi's Automation Object Wizard. Image from the original book.

As you can see in Figure 16.2, Delphi can generate OLE Automation servers that also export events. Simply select the corresponding check box of the Automation Object Wizard, and Delphi will add the proper entries in the Type Library and in the source code it generates.

The Type Library Editor

The Type Library editor is the tool you can use to define a Type Library in Delphi. Figure 16.3 shows its window after I've added some elements to it. The Type Library editor allows you to add methods and properties to the OLE Automation server object we've just created. Once this is done, it can generate both the Type Library (TLB) file and the corresponding Object Pascal source code.

To build a simple example, we can add to the server a property and a method. In the editor, we actually add these two elements to the interface, which should be called `IFirstServer`. Simply select it, and then click the Method button of the toolbar. (The names of these buttons can be displayed by using the local menu of the tool-

bar.) Now you have to give it a name, such as `ChangeColor`. You can type the name either in the Tree View control on the left side of the window or in the Name edit box on the right side. Delphi automatically defines the new method as a function in the Invoke Kind box and (as you'll see on the Parameters page) assigns it an `HRESULT` return value and no parameters. This corresponds to the Pascal definition:

```
procedure ChangeColor; safecall;
```

There are two reasons for this difference in the type of method. The first is that in the IDL language used by COM, all methods are indicated as functions (following the C language style); the second is that Delphi handles the `HRESULT` error codes automatically in every method that uses the `safecall` calling convention.

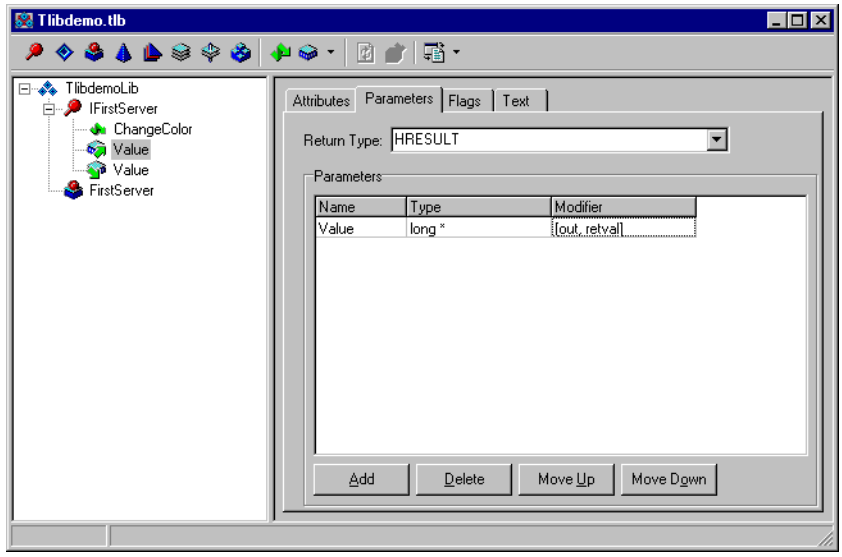


Figure 16.3: The Type Library editor, showing the details of an interface. Image from the original book.

note The methods contained in OLE Automation interfaces in Delphi generally use the `safecall` calling convention. This wraps a `try-except` block around each method and provides a default return value indicating error or success.

Now we can add a property to the interface by clicking the Property button of the Type Library editor's toolbar. Again we can type a name for it, such as `Value`, and select a data type in the Type combo box. Besides selecting one of the many types already listed, you can also enter other types directly, particularly interfaces of other objects. Keep in mind, however, that OLE Automation supports only a subset of

Delphi types. In this example, we can simply select the `long` type, which corresponds to Delphi's `Integer` type.

If you look again in the Parameters page for this example (see Figure 16.4), you can see that both the `Set` and `Get` (actually called `Put` and `Get` in the COM jargon) methods have the `HRESULT` return value.

You can also see that while the `Put` method uses the property's data type as its parameter (as with Delphi properties), the `Get` method uses a pointer to the type as its `out` parameter. This definition corresponds to the following elements of the Pascal interface:

```
function Get_Value: Integer; safecall;
procedure Set_Value(Value: Integer); safecall;
property Value: Integer read Get_Value write Set_Value;
```

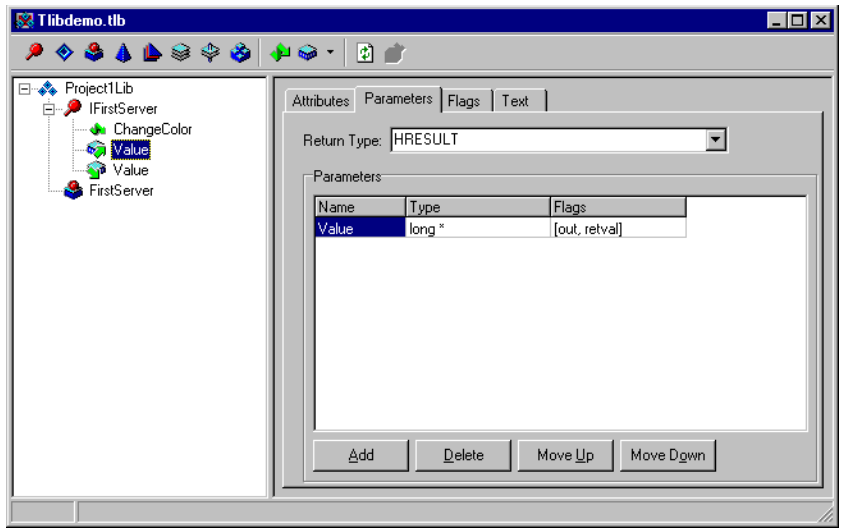


Figure 16.4: The Parameters page of the Type Library editor. Image from the original book.

Clicking the Refresh button on the Type Library editor toolbar generates the Pascal version of the interface. We'll examine it shortly, but first I want you to focus on the Text page of the editor, which includes the definition we've just created, written in the IDL language:

```
interface IFirstServer: IDispatch
{
  [id(0x00000001)]
  HRESULT _stdcall ChangeColor( void );
  [propget, id(0x00000002)]
```

```
  HRESULT _stdcall Value([out, retval] long * Value );
  [propput, id(0x00000002)]
  HRESULT _stdcall Value([in] long Value );
};
```

Fortunately, Delphi's Type Library editor saves you from writing similar code by hand, and the Delphi environment options (in the Type Library page) include a radio button to select Pascal or IDL in the text displayed by the Type Library editor.

The Code of the Server

Now we can close the Type Library editor and save the changes. This operation adds three items to the project: the Type Library file, a corresponding Pascal definition, and the declaration of the server object. The Type Library is connected to the project using a resource-inclusion statement, added to the source code of the project file:

```
{$R *.TLB}
```

You can always reopen the Type Library editor by using the View ➢ Type Library command or by selecting the proper TLB file in the normal File Open dialog box of Delphi.

As mentioned earlier, the Type Library is also converted into an interface definition and added to a new Pascal unit. This unit is quite long, so I've listed in the book only its key elements. The most important part is the new interface declaration:

```
type
  IFirstServer = interface(IDispatch)
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; safecall;
    function Get_Value: Integer; safecall;
    procedure Set_Value(Value: Integer); safecall;
    property Value: Integer read Get_Value write Set_Value;
  end;
```

Then comes the `dispinterface`, which associates a number with each element of the `IFirstServer` interface:

```
type
  IFirstServerDisp = dispinterface
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; dispid 1;
    property Value: Integer dispid 2;
  end;
```

The last portion of the file includes the so-called CoClass (also shown in the Type Library editor), a class used to create an object on the server (and for this reason used on the client side of the application, not on the server side):

```
type
  CoFirstServer = class
    class function Create: IFirstServer;
    class function CreateRemote(
      const MachineName: string): IFirstServer;
  end;
```

All the declarations of this file (there are some others I've skipped) can be considered an internal, hidden implementation support. You don't need to understand them fully in order to write most OLE Automation applications.

Finally, Delphi generates a file with the declaration of the actual object. This unit is added to the application and is the one we'll work on to finish the program. This unit declares the class of the server object, which must implement the interface we've just defined:

```
type
  TFirstServer = class(TAutoObject, IFirstServer)
  protected
    function Get_Value: Integer; safecall;
    procedure ChangeColor; safecall;
    procedure Set_Value(Value: Integer); safecall;
  end;
```

Delphi already provides us with the skeleton code of the methods, so you only need to fill the lines in between. This is the final code of the server object methods of the TLibDemo example:

```
function TFirstServer.Get_Value: Integer;
begin
  Result := ServerForm.Value;
end;

procedure TFirstServer.ChangeColor;
begin
  ServerForm.ChangeColor;
end;

procedure TFirstServer.Set_Value(Value: Integer);
begin
  ServerForm.Value := Value;
end;
```

In this case, the three methods simply refer to a property and two methods I've added to the form. In general, you should not add code related to the user interface

inside the class of the server object. It is better to refer to a user interface element, such as a form class, and let it perform the actions.

I've added a property to the form because I want to change the `Value` property and have a side effect (displaying the value in an edit box). The server object, in this example, simply exposes some properties and methods of the application. Here is the part of the declaration of the `TServerForm` class I've edited manually:

```
type
  TServerForm = class(TForm)
    ...
  private
    CurrentValue: Integer;
  protected
    procedure SetValue (NewValue: Integer);
  public
    property Value: Integer
      read CurrentValue write SetValue;
    procedure ChangeColor;
  end;
```

The implementation of these methods is quite straightforward, and you can easily guess what their code looks like. What's important is the `SetValue` method, which might produce a side effect:

```
procedure TServerForm.SetValue (NewValue: Integer);
begin
  if NewValue <> CurrentValue then
  begin
    CurrentValue := NewValue;
    UpDown1.Position := CurrentValue;
  end;
end;
```

The form of this example has an edit box with an associated UpDown component as well as a couple of buttons to show the current value and change the color. You can see this form at design time in Figure 16.5.

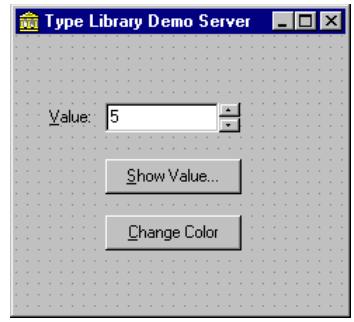


Figure 16.5: The form of the TLibDemo example at design time. Image from the original book.

Registering the Automation Server

The unit containing the server object has one more statement, added by Delphi to the `initialization` section:

```
initialization
  TAutoObjectFactory.Create(ComServer, TFirstServer,
    Class_FirstServer, ciMultiInstance);
end.
```

note In this case, I've selected multiple instancing. For the various instancing styles possible in COM, see the sidebar "COM Instancing and Threading Models" in Chapter 15.

This is not very different from the creation of class factories we saw in the examples of the last chapter. Combined with the call to the `Initialize` method of the `Application` object, which Delphi adds by default to the project source code of any program, the `initialization` code above makes the registration of this server straightforward.

You can add the server information to the Windows Registry by running this application on the target machine (the computer where you want to install the OLE automation server), passing to it the `/regserver` parameter on the command line. You can do this by selecting Start ➢ Run, by using the Explorer or File Manager, or by running the program within Delphi after you've entered a command-line parameter (using the Run ➢ Parameters command). Another command-line parameter, `/unregserver`, is used to remove this server from the Registry.

Writing a Client for Our Server

Now that we have built a server, we can prepare a simple client program to test it. This client can connect to the server either by using variants or by using the new Type Library. This second approach can be implemented manually or by using the new Delphi 5 techniques for wrapping components around Automation servers. We'll actually try out all of these approaches.

Create a new application—I've called it TLibCli—and then open the Type Library file of the server, after (optionally) copying it to the project's directory. Simply save the Type Library file, using Delphi's File ➢ Save menu command, and a new version of the interface declarations will be generated for you. Of course, in this case you could have grabbed the Pascal declarations from the server source code, but I'm trying to follow a more general approach, which also applies when you haven't written the server yet. In fact, you can usually extract the Type Library directly from the executable file of the server or from a DLL shipped with the program.

note Do not add the Type Library to the client application, though, because we are writing the OLE Automation controller, not a server. The Delphi project of a controller should not include the Type Library of the server it connects to.

You can simply refer to the Pascal file generated by the Type Library editor in the code of the main form:

```
uses
  TlibdemoLib_TLB;
```

I've already mentioned that one of the elements of this unit generated by the Type Library is the *creation* class, or CoClass, a special class with two class functions you can use to create a server object locally or remotely (using DCOM). I've already shown you the interface of this class, but here is the implementation:

```
class function CoFirstServer.Create: IFirstServer;
begin
  Result := CreateComObject(Class_FirstServer)
    as IFirstServer;
end;

class function CoFirstServer.CreateRemote(
  const MachineName: string): IFirstServer;
begin
  Result := CreateRemoteComObject(MachineName,
    Class_FirstServer) as IFirstServer;
end;
```

You can use the first of these two functions, `Create`, to create a server object (and possibly start the server application) on the same computer. You can use the second function, `CreateRemote`, to create the server on a different computer, as long as your version of the operating system supports DCOM.

The two functions are simply a shortcut of the `CreateComObject` call, which allows you to create an instance of a COM object if you know its GUID. As an alternative, you can also use the `CreateOleObject` function, which requires as parameter the registered name of the server. There is another difference between these two creation functions: `CreateComObject` returns an object of the `IUnknown` type, whereas `CreateOleObject` returns an object of the `IDispatch` type.

In my example I'm going to use the `CoFirstServer.Create` shorthand. When you create the server object you get as return value an `IFirstServer` interface. You can use it directly or store it in a variant variable. Here is an example of the first approach:

```
var
  MyServer: Variant;
begin
  MyServer := CoFirstServer.Create;
  MyServer.ChangeColor;
```

This code, based on variants, is not very different from that of the first controller we built in this chapter (the one that used Microsoft Word). Here is the alternative code, which has exactly the same effect:

```
var
  IMyServer: IFirstServer;
begin
  IMyServer := CoFirstServer.Create;
  IMyServer.ChangeColor;
```

Interfaces, Variants, and Dispatch Interfaces: Testing the Speed Difference

As I mentioned in the section introducing type libraries, one of the differences between these approaches is speed. It is actually quite complex to assess the exact performance of each technique because there are many factors involved. I've added to the TLibCli example a simple test, just to give you an idea. Here is the code of the test, a loop that accesses the `Value` of the server. The total value is displayed only to fool the optimizer, which might otherwise remove some of the code. The real output of the program relates to the timing, which is determined by calling the

`GetTickCount` API function before and after executing the loop. (Two alternatives are to use Delphi's own time functions, which are slightly less precise, or to use the very precise timing functions of the multimedia support unit, MMSystem.) Here is the code of one of the methods; they are quite similar:

```
procedure TClientForm.BtnIntfClick(Sender: TObject);
var
  I, K: Integer;
  Ticks: Cardinal;
begin
  Screen.Cursor := crHourglass;
  try
    Ticks := GetTickCount;
    K := 0;
    for I := 1 to 100 do
      K := K + IMyServer.Value;
    Ticks := GetTickCount - Ticks;
    ListResult.items.Add (Format (
      'Interface: %d - Seconds %.3f', [K, Ticks / 1000]));
  finally
    Screen.Cursor := crDefault;
  end;
end;
```

With this program you can compare the output obtained by calling this method based on an interface, the corresponding version based on a variant, and even a third version based on a dispatch interface. An example of the output (which is added to a list box so you can do several tests and compare the results) is shown in Figure 16.6. Obviously, the timing depends on the speed of your computer, and you can also alter the results by increasing or decreasing the maximum value of the loop counter.

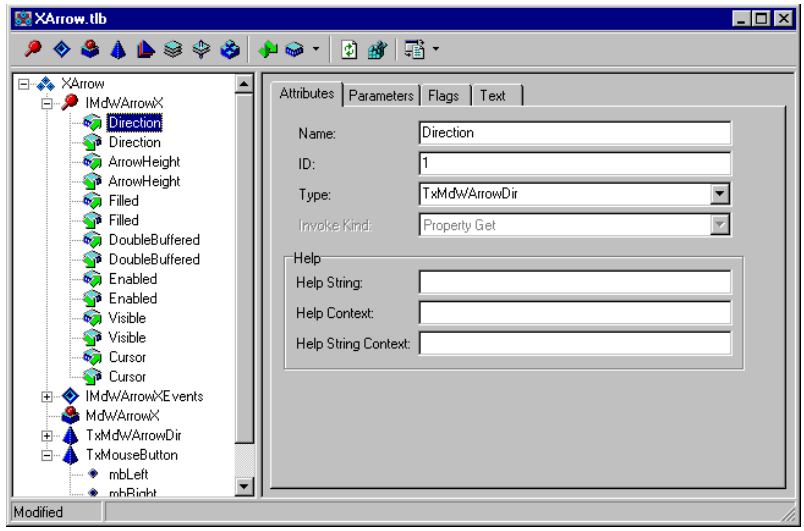


Figure 16.6: The TLibCli OLE Automation controller can access the server in different ways, with different performance results. Notice the server window in the background. Image from the original book.

We've already seen how you can use the interface and the variant. What about the dispatch interface? You can simply declare a variable of the dispatch interface type, in this case:

```
var
  DMyServer: IFirstServerDisp;
```

Then you can use it to call the methods as usual, after you've assigned an object to it by casting the object returned by the CoClass:

```
DMyServer := CoFirstServer.Create as IFirstServerDisp;
```

Looking at the timing and at the internal code of the example, there is apparently very little difference between the use of the interface and of the dispatch interface, because the two are actually connected. In other words, we can say that dispatch interfaces are a technique in between variants and interfaces, but they deliver almost all of the speed of interfaces.

The Scope of Automation Objects

Another important element to keep in mind is the scope of the automation objects. Variants and interface objects use reference-counting techniques, so if a variable that is related to an interface object is declared locally in a method, at the end of the method the object will be destroyed and the server may terminate (if all the objects

created by the server have been destroyed). For example, writing a method with this code produces very little effect:

```
procedure TClientForm.ChangeColor;
var
  IMyServer: IFirstServer;
begin
  IMyServer := CoFirstServer.Create;
  IMyServer.ChangeColor;
end;
```

Unless the server is already active, a copy of the program is created, and the color is changed, but then the server is immediately closed as the interface-typed object goes out of scope. The alternative approach I've used in the TLibCli example is to declare the object as a field of the form and create the COM objects at start-up, as in this procedure:

```
procedure TClientForm.FormCreate(Sender: TObject);
begin
  IMyServer := CoFirstServer.Create;
end;
```

With this code as the client program starts, the server program is immediately activated. At the program termination, the form field is destroyed and the server closes. A further alternative is to declare the object in the form but then create it only when it is used, as in these two code fragments:

```
// MyServerBis: Variant;
if varType (MyServerBis) = varEmpty then
  MyServerBis := CoFirstServer.Create;
MyServerBis.ChangeColor;

// IMyServerBis: IFirstServer;
if not Assigned (IMyServerBis) then
  IMyServerBis := CoFirstServer.Create;
ImyServerBis.ChangeColor;
```

note A variant is initialized to the `varEmpty` type when it is created. If you instead assign the value `null` to the variant, its type becomes `varNull`. Both `varEmpty` and `varNull` represent variants with no value assigned, but they behave differently in expression evaluation. The `varNull` value always propagates through an expression (making it a null expression), while the `varEmpty` value quietly disappears.

The Server in a Component

When creating a client program for our server or any other Automation server, we can use a new Delphi 5 approach, namely, wrapping a component around our COM server. Actually, if you look at the final portion of the `TlibdemoLib_TLB` file, you can find the following declaration:

```
// OLE Server Proxy class declaration
TFirstServer = class(TOleServer)
private
  FIntf: IFirstServer;
  FProps: TFirstServerProperties;
  function GetServerProperties: TFirstServerProperties;
  function GetDefaultInterface: IFirstServer;
protected
  procedure InitServerData; override;
  function Get_Value: Integer;
  procedure Set_Value(Value: Integer);
public
  constructor Create(AOwner: TComponent); override;
  destructor  Destroy; override;
  procedure Connect; override;
  procedure ConnectTo(svrIntf: IFirstServer);
  procedure Disconnect; override;
  procedure ChangeColor;
  property  DefaultInterface: IFirstServer
    read GetDefaultInterface;
  property Value: Integer
    read Get_Value write Set_Value;
published
  property Server: TFirstServerProperties
    read GetServerProperties;
end;
```

This is a new component, derived from `TOleServer`, that the system registers in the `Register` procedure, which is part of the unit. If you add this unit to a package, the new server component will become available on the Delphi Component Palette. You can also import the Type Library of the new server (with the Project ➢ Import Type Library menu command), add the server to the list (by pressing the Add button and selecting the server's executable file), and install it in a new or existing package. The component will be placed in the Servers page of the Palette. The Import Type Library dialog box indicating these operations is visible in Figure 16.7.

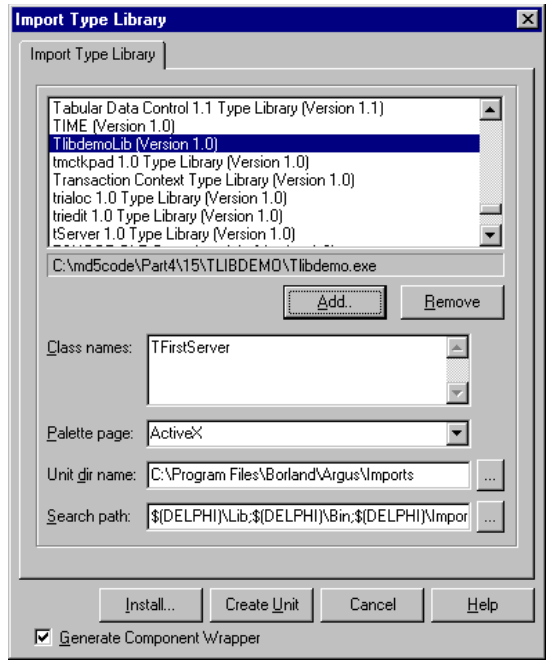


Figure 16.7: The Import Type Library dialog box can be used to import an Automation Server object as a new Delphi component. Image from the original book.

I've created a new package, AutoPack, available in the directory of the TlibDemo project. In this package, I've added the directive `LIVE_SERVER_AT_DESIGN_TIME` in the Directories/Conditionals page of the Project Options dialog box of the package. This enables an extra feature that you don't get by default: at design time the server component will have an extra property that lists as subitems all the properties of the Automation server. You can see an example in Figure 16.8, taken from the simple TLibComp example at design time.

note The `LIVE_SERVER_AT_DESIGN_TIME` directive should be used with care with the most complex Automation servers (including programs such as Word, Excel, PowerPoint, and Visio). In fact this setting requires the application to be in a particular mode before you can use some properties of their automation interfaces. For example, you'll get exceptions if you touch the Word server before a document has been opened in Word. That's why this feature is not active by default in Delphi—it's problematic at design time for many servers.

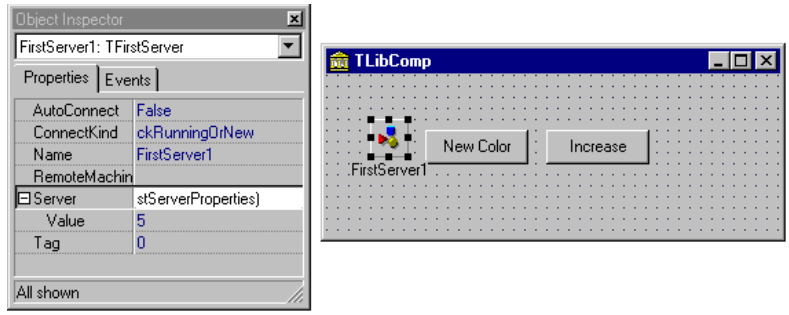


Figure 16.8: A server component, with the live properties at design time. Image from the original book.

As you can see, the Object Inspector shows that the component has few properties. `AutoConnection` indicates when to start up the server component at design time and as soon as the client program starts. As an alternative, the Automation server is started the first time one of its methods is called. Another property, `ConnectKind`, indicates how to establish the connection with the server. It can always start a new instance (`ckNewInstance`), use the running instance (`ckRunningInstance`, which causes an access violation if the server is not already running), or select the current instance or start a new one if none is available (`ckRunningOrNew`). Finally, you can ask for a remote server with `ckRemote` and directly attach a server in the code after a manual connection with `ckAttachToInterface`.

OLE Data Types

OLE and COM do not support all of the data types available in Delphi. This is particularly important for OLE Automation, because the client and the server are often executed in different address spaces, and the system must move the data from one side to the other. Also keep in mind that OLE interfaces should be accessible by programs written in any language.

COM data types include basic data types such as Integer, SmallInt, Byte, Single, Double, WideString, Variant, and WordBool (but not Boolean). Here is the mapping of some basic data types, available in the Type Library editor, to the corresponding Delphi types:

OLE Type	Delphi Type
`BSTR`	`WideString`
`byte`	`ShortInt`

CURRENCY	Currency
DATE	TDateTime
DECIMAL	TDecimal
double	Double
float	Single
GUID	GUID
int	SYSINT
long	Integer
LPSTR	PChar
LPWSTR	PWideChar
short	SmallInt
unsigned char	Byte
unsigned int	SYSUINT
unsigned long	UINT
unsigned short	Word
VARIANT	OleVariant

Notice that SYSINT is currently defined as an Integer, so don't worry about the apparently strange type definition. Besides the basic data types, you can also use OLE types for complex elements such as fonts, string lists, and bitmaps, using the IFontDisp, IStrings, and IPictureDisp interfaces. The following sections describe the details of a server that provides a list of strings and a font to a client.

Exposing Strings Lists and Fonts

The ListServ example is a practical demonstration of how you can expose two complex types, such as a list of strings and a font, from an OLE Automation server written in Delphi. I've chosen these two specific types simply because they are both supported by Delphi.

The `IFontDisp` interface is actually provided by Windows and is available in the ActiveX unit. The AxCtrls Delphi unit extends this support by providing conversion methods like `GetOleFont` and `SetOleFont`. The `IStrings` interface is provided by Delphi in the StdVCL unit, and the AxCtrls unit provides conversion functions for this type (along with a third type I'm not going to use, `TPicture`).

The server we are building has a very simple form containing a list-box component. It includes an automation object built around the following interface:

```
type
```

```
  IListServer = interface (IDispatch)
    ['{323C4A84-E400-11D1-B9F1-004845400FAA}']
    function Get_Items: IStrings; safecall;
    procedure Set_Items(const Value: IStrings); safecall;
    function Get_Font: IFontDisp; safecall;
    procedure Set_Font(const Value: IFontDisp); safecall;
    property Items: IStrings read Get_Items write Set_Items;
    property Font: IFontDisp read Get_Font write Set_Font;
  end;
```

The server object has the same four methods listed in its interface as well as some private data storing the status, the initialization function, and the destructor:

```
type
  TListServer = class (TAutoObject, IListServer)
  private
    fItems: TStrings;
    fFont: TFont;
  protected
    function Get_Font: IFontDisp; safecall;
    function Get_Items: IStrings; safecall;
    procedure Set_Font(const Value: IFontDisp); safecall;
    procedure Set_Items(const Value: IStrings); safecall;
  public
    destructor Destroy; override;
    procedure Initialize; override;
  end;
```

The code of the methods is actually quite simple. The pseudoconstructor creates the internal objects, and the destructor destroys them. Here is the first of the two:

```
procedure TListServer.Initialize;
begin
  inherited Initialize;
  fItems := TStringList.Create;
  fFont := TFont.Create;
end;
```

The `Set` and `Get` methods are quite simple, as well. They copy information from the OLE interfaces to the local data and then from this to the form and vice versa. Here is the code of the two methods of the strings (the other two, for the font, are quite similar, so I've not listed them here):

```
function TListServer.Get_Items: IStrings;
begin
  // get the listbox items, converting them
  GetOleStrings (ListServForm.Listbox1.Items, Result);
end;

procedure TListServer.Set_Items(const Value: IStrings);
begin
```

```
  // convert the strings received as parameter
  SetOleStrings (ListServForm.Listbox1.Items, Value);
end;
```

After we've compiled and registered the server, we can turn our attention to the client application. This simply embeds the Pascal translation of the Type Library of the server, as in the previous example, and then implements an object that uses the interface.

Instead of creating the server when the object starts, the client program creates it when it is required. I've described this technique earlier, but the problem is that because there are several buttons a user can click, and we don't want to impose an order, every event should have a handler like this:

```
if not Assigned (ListServ) then
  ListServ := CoListServer.Create;
```

This kind of code duplication is quite dangerous, so I've decided to use an alternative approach. I've defined a property corresponding to the interface of the server and defined a read method for it. The property is mapped to some internal data I've defined with a different name to avoid the error of using it directly. Here are the definitions added to the form class:

```
private
  fInternalListServ: IListServer;
  function GetListSrv: IListServer;
public
  property ListSrv: IListServer
    read GetListSrv;
```

The implementation of the `Get` method can simply check whether the object already exists. This code is going to be repeated often, but that should not slow down the application noticeably:

```
function TListCliForm.GetListSrv: IListServer;
begin
  // eventually create the server
  if not Assigned (fInternalListServ) then
    fInternalListServ := CoListServer.Create;
  Result := fInternalListServ;
end;
```

The rest of the code of the client application is quite simple, and you can see an example of the program running (along with the server) in Figure 16.9. This is an example of the selection of a font, which is then sent to the server:

```
procedure TListCliForm.btnFontClick(Sender: TObject);
var
```

```
  NewFont: IFontDisp;
begin
  // select a font and apply it
  if FontDialog1.Execute then
  begin
    GetOleFont (FontDialog1.Font, NewFont);
    ListSrv.Font := NewFont;
  end;
end;
```

There are also several methods related to the strings, which you can see by looking at the source code of the program.

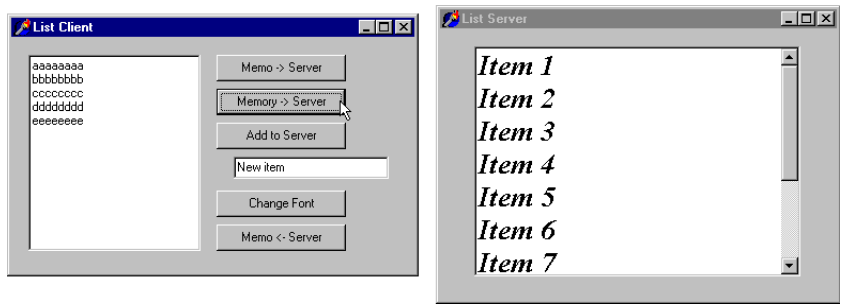


Figure 16.9: The ListCli and ListServ applications share complex data, namely fonts and lists of strings. Image from the original book.

Using Office Programs

So far, we've built both the client and the server side of the OLE Automation connection. If your aim is just to let two applications you've built cooperate, this is certainly a useful technique, although it is not the only one. We've seen some alternative data-sharing approaches in the last two chapters (using memory-mapped files and the `wm_CopyData` message). The real value of OLE Automation is that it is a standard, so you can use it to integrate your Delphi programs with other applications your users own. A typical example is the integration of a program with office applications, such as Microsoft Word and Microsoft Excel, or even with stand-alone applications, such as AutoCAD.

The integration with these applications provides a two-fold advantage:

- You can let your users work in an environment they know, for example, generating reports and memos from database data in a format they can easily manipulate.
- It allows you to avoid implementing complex functionality from scratch, such as writing your own word-processing code inside a program. Instead of just reusing simple components, you can reuse complex applications.

There are also some drawbacks with this approach, which are certainly worth mentioning:

- The user must own the application you plan to integrate with, and they may also need a recent version of it to support all the features you are using in your program.
- You have to learn a new programming language and programming structure, often with limited documentation at hand. It is true, of course, that you are still using Pascal, but the code you write depends on the OLE data types, the types introduced by the server, and in particular, a collection of interrelated classes that are often difficult to understand.
- You might end up with a program that works only with a specific version of the server application, particularly if you try to optimize the calls by using interfaces instead of variants. In particular, Microsoft does not attempt to maintain script compatibility between major releases of Word or other Office applications.

We've already seen a small source code excerpt from the WordTest example, but now I want to complete the coverage of this simple but interesting test program by providing a few extra features.

Sending Data to Microsoft Word

Delphi 5 has simplified the use of Microsoft Office applications by preinstalling some ready-to-use components that wrap the Automation interface of these servers. These components, available in the Servers page of the Palette, have been installed using the same technique I've demonstrated in the last section. What I want to underline here is that the real Delphi 5 innovation lies in this technique of creating components to wrap existing Automation servers, rather than in the availability of the predefined server components.

Technically it is possible to use variants to interact with Automation servers, as we've seen in the section "Introducing Type Libraries." Using interfaces and the type libraries is certainly better because the compiler helps you catch errors in the

source code and produces faster code. Thanks to the new server component, this process is also quite simple.

I've written a program, called DBOffice, which uses Delphi 5 predefined components to send a table to Word and to Excel. In both cases, you can use the application object, the document/worksheet object, or a combination of the two. There are other specialized components, for tasks such as handling Excel charts, but this example will suffice to introduce use of the built-in Office components.

In case of Microsoft Word, I use only a document object with default settings. The code used to send the table to word starts by adding some text to a document:

```
procedure TFormOff.BtnWordClick(Sender: TObject);
begin
  WordDocument1.Activate;
  // insert title
  WordDocument1.Range.Text := 'American Capitals from ' +
    Table1.TableName;
  WordDocument1.Range.Font.Size := 14;
```

This code follows the typical `while` loop, which scans the database table and has the following code inside:

```
while not Table1.EOF do
begin
  // send the two fields
  WordDocument1.Range.InsertParagraphAfter;
  WordDocument1.Paragraphs.Last.Range.Text :=
    Table1.FieldByName ('Name').AsString + #9 +
    Table1.FieldByName ('Capital').AsString;
  Table1.Next;
end;
```

The final part of the code gets a little more complex. It works on a selection and on a row of the table, respectively stored in two variables of the `Range` and `Row` types defined by Word and available in the Word 97 unit.

```
procedure TFormOff.BtnWordClick(Sender: TObject);
var
  RangeW: Word97.Range;
  v1: Variant;
  ov1: OleVariant;
  Row1: Word97.Row;
begin
  // code above...
  RangeW := WordDocument1.Content;
  v1 := RangeW;
  v1.ConvertToTable (#9, 19, 2);
  Row1 := WordDocument1.Tables.Item(1).Rows.Get_First;
  Row1.Range.Bold := 1;
```

```
  Row1.Range.Font.Size := 30;
  Row1.Range.InsertParagraphAfter;
  ov1 := ' ';
  Row1.ConvertToText (ov1);
end;
```

As you can see in the last statement above, in order to pass a parameter, you must first save it in an `OleVariant` variable, because many parameters are passed by reference, so you cannot pass a constant value. This implies that if there are many parameters, you must still define a number of parameters, even if you are fine with the default values. An alternative, often useful, is to use a temporarily variant variable and apply the method on it, because variants don't require a strict type checking on the parameters. This technique is used in the code above to call the `ConvertToTable` method, which has more than 10 parameters.

Building an Excel Table

In the case of Excel, I've used a slightly different approach and worked with the application object. The code creates a new Excel spreadsheet, fills it with a database table, and formats the result. It uses an Excel internal object, Range, which is not to be confused with a similar type available in Word (the reason this type is prefixed with the name of the unit defining the Excel Type Library). Here is the complete code:

```
procedure TFormOff.BtnExcelClick(Sender: TObject);
var
  RangeE: Excel97.Range;
  I, Row: Integer;
  Bookmark: TBookmarkStr;
begin
  // create and show
  ExcelApplication1.Visible [0] := True;
  ExcelApplication1.Workbooks.Add (NULL, 0);
  // fill is the first row with field titles
  RangeE := ExcelApplication1.ActiveCell;
  for I := 0 to Table1.Fields.Count - 1 do
  begin
    RangeE.Value := Table1.Fields [I].DisplayLabel;
    RangeE := RangeE.Next;
  end;
  // add field data in following rows
  Table1.DisableControls;
  try
    Bookmark := Table1.Bookmark;
    try
      Table1.First;
```

```
      Row := 2;
      while not Table1.EOF do
      begin
        RangeE := ExcelApplication1.Range ['A' + IntToStr (Row),
          'A' + IntToStr (Row)];
        for I := 0 to Table1.Fields.Count - 1 do
        begin
          RangeE.Value := Table1.Fields [I].AsString;
          RangeE := RangeE.Next;
        end;
        Table1.Next;
        Inc (Row);
      end;
    finally
      Table1.Bookmark := Bookmark;
    end;
  finally
    Table1.EnableControls;
  end;
  // format the section
  RangeE := ExcelApplication1.Range ['A1', 'E' + IntToStr (Row - 1)];
  RangeE.AutoFormat (3, NULL, NULL, NULL, NULL, NULL, NULL);
end;
```

You can see the effect of this code in Figure 16.10. Notice that in the code I don't handle any events of the Office applications, but many are available. Handling these events was quite complex in the past, but they now become as simple to handle as events of native Delphi components. The presence of these events is a reason to have specific objects for documents and other specific elements: you might want to know when the user closes a document, therefore this is an event of the document object, not of the application object.

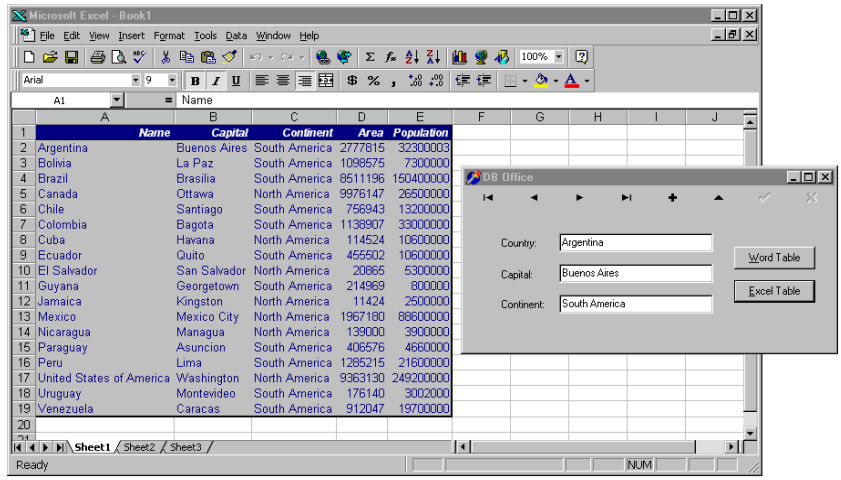


Figure 16.10: The Excel spreadsheet generated by the DbOffice application. Image from the original book.

note When using the Office server components, one of the key problems is the lack of adequate documentation. Although Microsoft distributes some of it with the high-end version of the Office suite, this is certainly not Delphi friendly. A totally alternative approach to solve the problem is to use "Office Partner," a set of components from DeVries Data Systems, Inc. (`www.dvdata.com`). These components map the Office servers, like those available in Delphi, but they also provide extensive property editors that allow you to work visually with the internal structure of these servers. With these property editors, you can create documents, paragraphs, tables, and all the other internal objects even at design time! From my experience, this can really save a lot of time.

Using Compound Documents

Compound Documents, or Active Documents, is Microsoft's name for the technology that allows in-place editing of a document within another one (for example, a picture in a Word document). This is the technology that originated the term OLE, but although it is still in use, its role is definitely more limited than Microsoft envisioned when it was introduced in the early 1990s. Compound documents actually have two different capabilities, o*bject linking* and *embedding* (hence the term OLE):

- Embedding an object in a compound document corresponds to a smart version of the copy and paste operations you make with the Clipboard. The key difference is that when you copy an OLE object from a server application and paste it into a container application, you copy both the data and some information about

the server (its GUID). This allows you to activate the server application from within the container to edit the data.

- Linking an object to a compound document instead copies only a reference to the data and the information about the server. You generally activate object linking by using the Clipboard and making a Paste Link operation. When editing the data in the container application, you'll actually modify the original data, which is stored in a separate file.

Because the server program refers to an entire file (only part of which might be linked in the client document), the server will be activated in a stand-alone window, and it will act upon the entire original file, not just the data you've copied. When you have an embedded object, instead, the container might support visual (or *in-place*) editing, which means that you can modify the object in context, inside the container's main window. The server and container application windows, their menus, and their toolbars are merged automatically, allowing the user to work within a single window on a number of different object types—and therefore with a number of different OLE servers—without leaving the window of the container application.

Another key difference between embedding and linking is that the data of an embedded object is stored and managed by the container application. The container saves the embedded object in its own files. By contrast, a linked object physically resides in a separate file, which is handled by the server exclusively, even if the link refers only to a small portion of the file.

In both cases, the container application doesn't have to know how to handle the object and its data—not even how to display it—without the help of the server. Accordingly, the server application has a lot of work to do, even when you are not editing the data. Container applications often make a copy of the image of an OLE object and use the bitmap to represent the data, which speeds up some operations with the object itself. The drawback of this approach is that many commercial OLE applications end up with bloated files (because two copies of the same data are saved). If you consider this problem along with the relative slowness of OLE and the amount of work necessary to develop OLE servers, you can understand why the use of this powerful approach is still somewhat limited, compared with what Microsoft envisioned a few years ago.

Compound document containers can support OLE in varying degrees. You can place an object in a container by inserting a new object, by pasting or *paste-linking* one from the Clipboard, by dragging one from another application, and so on.

Once the object is placed inside the container, you can then perform operations on it, using the server's available *verbs,* or actions. Usually the *edit verb* is the default

action—the action performed when you double-click on the object. For other objects, such as video or sound clips, *play* is defined as the default action. You can typically see the list of actions supported by the current contained object by right-clicking on it. The same information is available in many programs via the Edit ➢ Object menu item, which has a submenu that lists the available verbs for the current object.

note Delphi provides no *visual* support for building compound document servers. You can always write a server implementing the proper interfaces. Compound document container support, instead, is easily available through the OleContainer component.

The OLE Container Component

To create a simple OLE container application in Delphi, place an OleContainer component in a form. Then select the component, and right-click to activate its local menu, which will have an Insert Object command. When you select this command, Delphi displays the standard OLE Insert Object dialog box. This dialog box allows you to choose from one of the server applications registered on the computer.

Once the OLE object is inserted in the container, the local menu of the control container component will have several more custom menu items. The new menu items include commands to change the properties of the OLE object, insert another one, copy the existing object, or remove it. The list also includes the verbs, or actions, of the object (such as Edit, Open, or Play). Once you have inserted an OLE object in the container, the corresponding server will launch to let you edit the new object. As soon as you close the server application, Delphi updates the object in the container and displays it at design time in the form of the Delphi application you are developing.

If you look at the textual description of a form containing a component with an object inside, you'll notice a `Data` property, which contains the actual data of the OLE object. Although the client program stores the data of the object, it doesn't know how to handle and show that without the help of the proper server (which must be available on the computer where you run the program). This means that the OLE object is *embedded*.

To fully support compound documents, a program should provide a menu and a toolbar or panel. These extra components are important because in-place editing implies a merging of the user interface of the client and that of the server program. When the OLE object is activated in place, some of the pull-down menus of the

server application's menu bar are added to the menu bar of the container application.

OLE menu merging is handled almost automatically by Delphi. You only need to set the proper indexes for the menu items of the container, using the `GroupIndex` property. Any menu item with an odd index number is replaced by the corresponding element of the active OLE object. More specifically, the File (0) and Window (4) pull-down menus belong to the container application. The Edit (1), View (3), and Help (5) pull-down menus (or the groups of pull-down menus with those indexes) are taken by the OLE server. A sixth group, named Object and indicated with the index 2, can be used by the container to display another pull-down menu between the Edit and View groups, even when the OLE object is active. The OleCont demo program I've written to demonstrate these features allows a user to create a new object by calling the `InsertObjectDialog` method of the `TOleContainer` class.

The `InsertObjectDialog` method shows a system dialog box, but it doesn't automatically activate the OLE object:

```
procedure TForm1.New1Click(Sender: TObject);
begin
  if OleContainer1.InsertObjectDialog then
    OleContainer1.DoVerb (OleContainer1.PrimaryVerb);
end;
```

Once a new object has been created, you can execute its primary verb using the `DoVerb` method. The program also displays a small toolbar with some bitmap buttons. I placed some `TWinControl` components in the form to let the user select them and thus disable the OleContainer. To keep this toolbar/panel visible while in-place editing is occurring, you should set its `Locked` property to `True`. This forces the panel to remain present in the application and not be replaced by a toolbar of the server.

To show what happens when you don't use this approach, I've added to the program a second panel, with some more buttons. Because I haven't set its `Locked` property, this new toolbar will be replaced with that of the active OLE server. When in-place editing launches a server application that displays a toolbar, that server's toolbar replaces the container's toolbar, as you can see in the lower part of Figure 16.11.

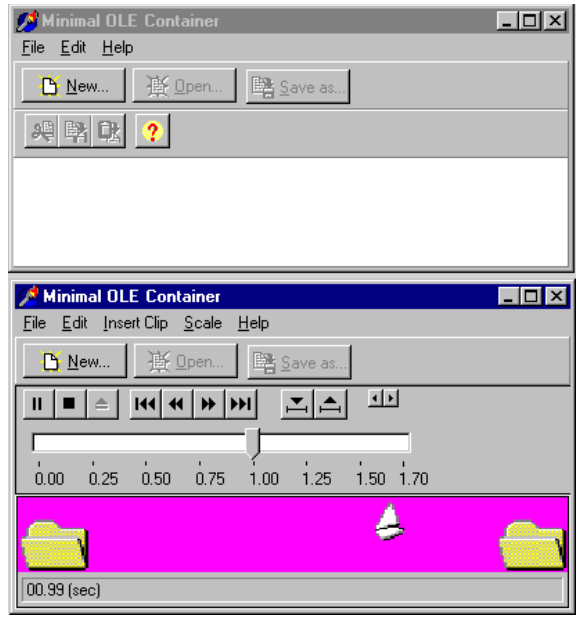


Figure 16.11: The second toolbar of the OleCont example (top) is replaced by the toolbar of the server (bottom). Image from the original book.

note To make all the automatic resizing operations work smoothly, you should place the OLE container component in a panel component and align both of them to the client area of the form.

Another way to create an OLE objects is to use the `PasteSpecialDialog` method, called in the `PasteSpecial1Click` event handler of the example. Another standard OLE dialog box, wrapped in a Delphi function, is the one showing the properties of the object, which is activated with the Object Properties item in the Edit pull-down menu:

```
procedure TForm1.Object1Click(Sender: TObject);
begin
 OleContainer1.ObjectPropertiesDialog;
end;
```

You can see an example of the resulting standard OLE dialog box in Figure 16.12. Obviously, this dialog box changes depending on the nature of the active OLE object in the container.

The last feature of the OleCont program is the support for files. This is actually one of the simplest additions we can make, because the OLE container component already provides file support.

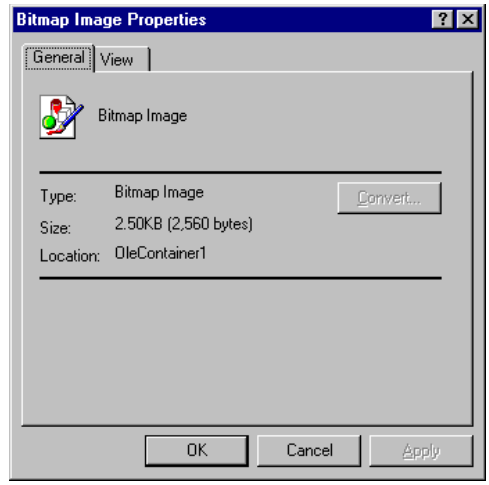


Figure 16.12: The standard OLE Object Properties dialog box, available in the OleCont example. Image from the original book.

Using the Internal Object

In the last program the user determined the type of the internal object created by the program. In this case there is little you can do to interact with the internal objects. Suppose, instead, that you want to embed a Word document in a Delphi application and then modify it by code. You can do this by using OLE Automation with the embedded object, as demonstrated by the WordCont example (the name stands for Word Container).

note Since the WordCont example includes an object of a specific type, a Microsoft Word document, it won't run if you don't have that server application installed. Having a different version of the server might also create problems if the Automation methods used by the client program are not available in that version of the server.

In the form of this example, I've added an OleContainer component, set its `AutoActivate` property to `aaManual` (so that the only possible interaction is with our code), and added a toolbar with a couple of buttons. The code for the two buttons is quite straightforward, once you know that the embedded object corresponds to a Word document:

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
  Document: Variant;
begin
  // activates if not running
  if not (OleContainer1.State = osRunning) then
    OleContainer1.Run;

  // get the document
  Document := OleContainer1.OleObject;
  // first paragraph to bold
  Document.Paragraphs.Item(1).Range.Bold := 1;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  Document, Paragraph: Variant;
begin
  // activate if not running
  if not (OleContainer1.State = osRunning) then
    OleContainer1.Run;

  // get the document
  Document := OleContainer1.OleObject;
  // add paragraphs, getting the last one
  Document.Paragraphs.Add;
  Paragraph := Document.Paragraphs.Add;
  // add text to the paragraph, using random font size
  Paragraph.Range.Font.Size := 10 + Random (20);
  Paragraph.Range.Text := 'New text (' +
    IntToStr (Paragraph.Range.Font.Size) + ')'#13;
end;
```

You can see the effect of this code in Figure 16.13. The code is not terribly powerful, but it does show how you can merge the usage of OLE Containers and OLE Automation techniques.

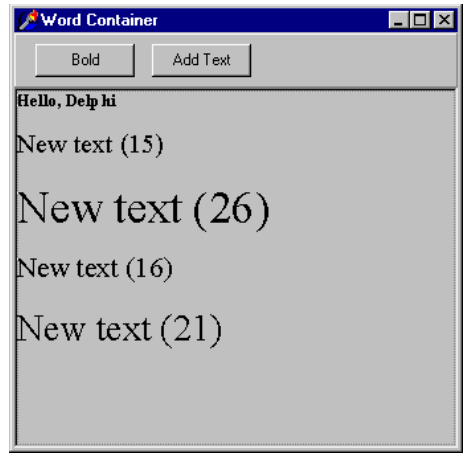


Figure 16.13: The WordCont example shows how to use OLE Automation with an embedded object. Image from the original book.

Introducing ActiveX Controls

Microsoft's Visual Basic was the first program development environment to introduce the idea of supplying software components to the mass market. Actually, the concept of reusable software components is older than Visual Basic—it's well rooted in the theories of object-oriented programming (OOP). But OOP languages never delivered the reusability they promised, probably more because of marketing and standardization problems than for any other reason. Although Visual Basic does not fully exploit object-oriented programming, it applies the component concept through its standard way of building and distributing new controls that developers can integrate into the environment.

The first technical standard promoted by Visual Basic was *VBX*, a 16-bit specification that was fully available in the 16-bit version of Delphi. In moving to the 32-bit platforms, Microsoft replaced the VBX standard with the more powerful and more open *ActiveX* controls.

note ActiveX controls used to be called OLE Controls (or OCX). The name change reflects a new marketing strategy from Microsoft rather than a technical innovation. Technically, ActiveX can be considered a minor extension to the OCX technology. Not surprisingly, then, ActiveX controls are usually saved in files with the `.ocx` extension.

From a general perspective, an ActiveX control is not very different from a Windows, Delphi, or Visual Basic control. A control in any of these languages is always a window, with its associated code defining its behavior. The key difference between various families of controls is in the *interface* of the control, the interaction between the control and the rest of the application. Typical Windows controls use a message-based interface, VBX controls use properties and events, OLE Automation objects use properties and methods, and ActiveX controls use properties, methods, and events. These three elements of properties, methods, and events are also found in Delphi's own components.

Using OLE jargon, an ActiveX control is a "compound document object which is implemented as an in-process server DLL and supports OLE Automation, visual editing, and inside-out activation." Perfectly clear, right? Let's see what this definition actually means.

An ActiveX control uses the same approach as OLE server objects, which are the objects you can insert into an OLE Document, as we saw in the last chapter. The difference between a generic OLE server and an ActiveX control is that, whereas ActiveX controls can only be implemented in one way, OLE servers can be implemented in three different ways:

- As stand-alone applications (for example, Microsoft Excel)
- As out-of-process servers—that is, executable files that cannot be run by themselves and can only be invoked by a server (for example, Microsoft Graph and similar applications)
- As in-process servers, such as DLLs loaded into the same memory space as the program using them

ActiveX controls can only be implemented using the last technique, which is also the fastest: as in-process servers. Furthermore, ActiveX controls are OLE Automation servers (also discussed in the last chapter). This means you can access properties of these objects and call their methods.

You can see an ActiveX control in the application that is using it and interact with it directly in the container application window. This is the meaning of the term *visual editing,* or *in-place activation*. A single click activates the control rather than the double-click used by OLE Documents, and the control is active whenever it is visible (which is what the term *inside-out activation* means), without having to double-click on it.

As I've mentioned before, an ActiveX control has properties, methods, and events. Properties can identify states, but they can also activate methods. (This is particularly true for ActiveX controls that are *updated* VBX controls, because in a VBX

there *was* no other way to activate a method than by setting a property.) Properties can refer to aggregate values, arrays, subobjects, and so on. Properties can also be dynamic (or read-only, to use the Delphi term).

In an ActiveX control, properties are divided into different groups: stock properties that most controls need to implement; ambient properties that offer information about the container (similar to the `ParentColor` or `ParentFont` properties in Delphi); extended properties managed by the container, such as the position of the object; and custom properties, which can be anything.

Events and methods are, well, events and methods. *Events* relate to a mouse click, a key press, the activation of a component, and other specific user actions. *Methods* are functions and procedures related to the control. There is no major difference between the ActiveX and Delphi concepts of events and methods.

ActiveX Controls versus Delphi Components

Before I show you how to use and write ActiveX controls in Delphi, let's go over some of the technical differences between the two kinds of controls. ActiveX controls are DLL-based. This means that when you use them, you need to distribute their code (the OCX file) along with the application using them. In Delphi, the code of the components can be statically linked to the executable file or dynamically linked to it using a run-time package, so you can always choose.

Having a separate file allows you to share code among different applications, as DLLs usually do. If two applications use the same control (or run-time package), you need only one copy of it on the hard disk and a single copy in memory. The drawback, however, is that if the two programs have to use two different versions (or builds) of the ActiveX control, some compatibility problems might arise. An advantage of having a self-contained executable file is that you will also have fewer installation problems.

Now, what is the drawback of using Delphi components? The real problem is not that there are fewer Delphi components than ActiveX controls, but that if you buy a Delphi component, you'll only be able to use it in Delphi and Borland C++Builder. If you buy an ActiveX control, on the other hand, you'll be able to use it in multiple development environments from multiple vendors. Even so, if you develop mainly in Delphi and find two similar components based on the two technologies, I suggest you buy the Delphi one—it will be more integrated with your environment, and therefore easier for you to use. Also, the native Delphi component will probably be better documented (from the Pascal perspective), and it will take advantage of Del-

phi and Object Pascal features not available in the general ActiveX interface, which is traditionally based on C and C++.

Using ActiveX Controls in Delphi

Delphi comes with some preinstalled ActiveX controls, and you can buy and install more third-party ActiveX controls easily. After this description of how ActiveX controls work in general, I'll demonstrate one in a simple example.

The Delphi installation process is very simple. Select Component ➢ Import ActiveX Control in the Delphi menu. This opens the Import ActiveX dialog box, where you can see the list of ActiveX control libraries registered in Windows. If you choose one, Delphi will read its Type Library, list its controls, and suggest a filename for its unit. If the information is correct, simply click on the Create Unit button to view the Pascal source code file created by Delphi as a *wrapper* for the ActiveX control. Click on the Install button to add this new unit to a Delphi package and to the Component Palette.

Using the WebBrowser Control

To build my example, I've used a preinstalled ActiveX control available in Delphi. Unlike the third-party controls, this is not available in the ActiveX page of the palette, but in the Internet page. The control, called WebBrowser, is simply a wrapper around Microsoft's Internet Explorer engine. The example is a very simple Web browser[361].

The WebBrows program has a `TWebBrowser` ActiveX control covering its client area and a control bar at the top and a status bar at the bottom. To move to a given Web page, a user can simply type in the combo box of the toolbar, select one of the visited URLs (saved in the combo box), or click on the Open File button to select a local file.

---

361 For many years Delphi VCL library made this WebBrowser component available, wrapping the Internet Explorer ActiveX. For some time this was also used in the Delphi IDE for the Welcome page. While the WebBrowser component is still there, it now offers a dual interface to the Internet Explorer ActiveX or to the Chromium-based Edge browsers, surfaced via the Windows platform WebView 2 control. The Welcome Page has been rewritten as a native VCL-based surface.

The actual implementation of the code used to select a Web or local HTML file is in the `GotoPage` method:

```
procedure TForm1.GotoPage(ReqUrl: string);
begin
  WebBrowser1.Navigate (ReqUrl, EmptyParam, EmptyParam,
    EmptyParam, EmptyParam);
end;
```

`EmptyParam` is a predefined OleVariant you can use whenever you want to pass a default value as a reference parameter. This is a handy shortcut you can use to avoid creating an empty OleVariant each time you need a similar parameter. This method is called for by a file, when the user clicks on the Enter key in the combo box, or by selecting the Go button:

```
procedure TForm1.BtnOpenClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
      GotoPage (OpenDialog1.FileName);
end;

procedure TForm1.ComboURLKeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then
    GotoPage (ComboUrl.Text);
end;

procedure TForm1.BtnGoClick(Sender: TObject);
begin
  GotoPage (ComboUrl.Text);
end;
```

Actually there is a fourth use of the `GotoPage` method. When the program starts, it loads a greeting HTML from the current directory, with the effect you can see in Figure 16.14:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  GotoPage (ExtractFilePath (Application.ExeName) +
    'greeting.htm');
end;
```

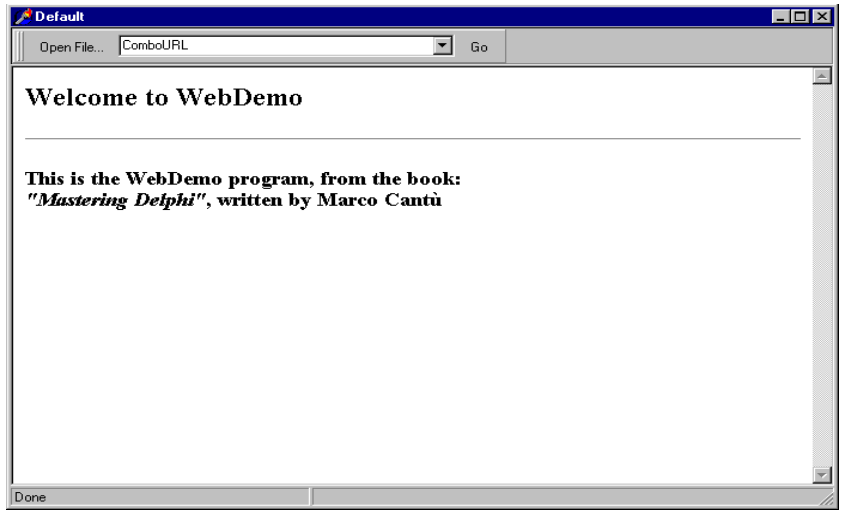


Figure 16.14: The WebDemo program at start-up. As you use it, you'll notice that the program fully supports graphics and all other Web extensions, as it is based on Microsoft Internet Explorer engine. Image from the original book.

The program also handles four events of the WebBrowser control. When the download operations start and end, the program updates the text of the status bar and also the drop-down list of the combo box:

```
procedure TForm1.WebBrowser1DownloadBegin(Sender: TObject);
begin
  StatusBar1.Panels[0].Text := 'Downloading ' +
    WebBrowser1.LocationURL + '...';
end;

procedure TForm1.WebBrowser1DownloadComplete(Sender: TObject);
var
  NewUrl: string;
begin
  StatusBar1.Panels[0].Text := 'Done';
  // add URL to combobox
  NewUrl := WebBrowser1.LocationURL;
  if (NewUrl <> '') and
      (ComboURL.Items.IndexOf (NewUrl) < 0) then
    ComboURL.Items.Add (NewUrl);
end;
```

Two other useful events are the `OnTitleChange`, used to update the caption with the title of the HTML document, and the `OnStatusTextChange` event, used to update the second part of the status bar. This code basically duplicates the information displayed in the first part of the status bar by the previous two event handlers:

```
procedure TForm1.WebBrowser1TitleChange(Sender: TObject;
  const Text: WideString);
```

```
begin
  Caption := Text;
end;

procedure TForm1.WebBrowser1StatusTextChange(Sender: TObject;
  const Text: WideString);
begin
  StatusBar1.Panels[1].Text := Text;
end;
```

Writing ActiveX Controls

Besides using existing ActiveX controls in Delphi, you can easily develop new ones. Although you can write the code of a new ActiveX control yourself, implementing all the required OLE interfaces (and there are many), it's much easier to use one of the techniques directly supported by Delphi:

- You can use the ActiveX Control Wizard to turn a VCL control into an ActiveX control. You start from an existing VCL component, which must be a `TWinControl` descendant, and Delphi wraps an ActiveX around it. During this step Delphi adds a Type Library to the control. (Wrapping an ActiveX control around a Delphi component is exactly the opposite of what we did to use an ActiveX inside Delphi.)
- You can create an ActiveForm, place several controls inside it, and ship the entire form (without borders) as an ActiveX control. This second technique is the same one used by Visual Basic and is generally aimed at building Internet applications. However, it is also a very good alternative for the construction of an ActiveX control based on multiple Delphi controls or on Delphi components that do not descend from `TWinControl`.

An optional step you can take in both cases is to prepare a property page for the control, to use as a sort of property editor for setting the initial value of the properties of the control in any development environment. It is a sort of alternative to the Object Inspector in Delphi. Because most development environments allow only limited editing, it is more important to write a property page than it is to write a component or a property editor for a Delphi control.

Building an ActiveX Arrow

As an example of the development of an ActiveX control, I've decided to take the Arrow component we developed in Chapter 13 and turn it into an ActiveX. Actually, we cannot use that component directly, because it was a graphical control, a subclass of `TGraphicControl`. However, turning a graphical control into a window-based control is usually a straightforward operation.

In this case, I've just changed the base class name to `TCustomControl` (and changed the name of the class of the control, as well, to avoid a name clash):

```
type
  TMdWArrow = class(TCustomControl)
  ...
```

The `TWinControl` class has very minimal support for graphical output. Its `TCustomControl` subclass, however, has basically the same capabilities as the `TGraphicControl` class. The key difference is that a `TCustomControl` object has a window handle.

After installing this new component in Delphi, we are ready to start developing the new example. To create a new ActiveX library, simply select File ➢ New, move to the ActiveX page, and choose ActiveX library. Delphi creates the bare skeleton of a DLL, as we saw at the beginning of this chapter. I've saved this library as `XArrow`, in a directory with the same name, as usual.

Now it is time to use the ActiveX Control Wizard, available in the ActiveX page of the Object Repository—Delphi's New dialog box. In this wizard (shown in Figure 16.15), you simply select the VCL class you are interested in, customize the names shown in the edit boxes, and click OK; Delphi then builds the complete source code of an ActiveX control for you.

The use of the three check boxes at the bottom of the ActiveX Control Wizard window may not be obvious. If you include design-time license support, the user of the control won't be able to use it in a design environment without the proper *license key* for the control. The second check box allows you to include version information for the ActiveX, in the OCX file. (Version information is discussed in Chapter 19.) If the third check box is selected, the ActiveX Control Wizard automatically adds an About box to the control.

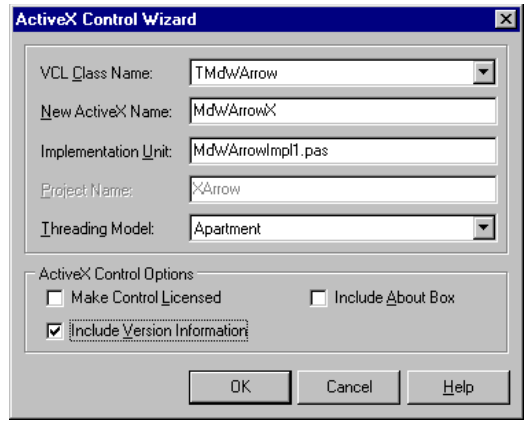


Figure 16.15: Delphi's ActiveX Control Wizard. Image from the original book.

Take a look at the code the ActiveX Control Wizard generates. The key element of this wizard is the generation of a Type Library. You can see the library generated for our arrow control in Delphi's Type Library editor in Figure 16.16. From the Type Library information, the Wizard also generates an import file with the definition of an interface, the `dispinterface`, and other types and constants.

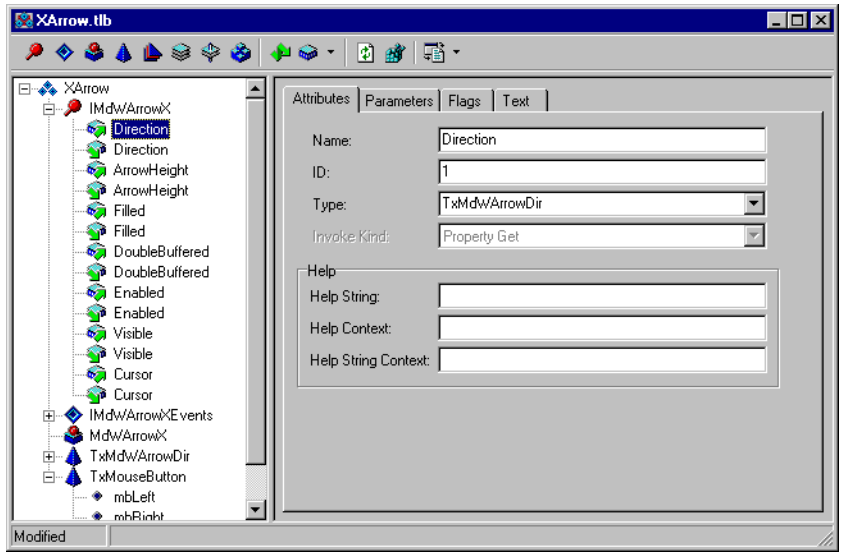


Figure 16.16: The Type Library editor with the Type Library of the demo ActiveX control I've created. Image from the original book.

In this example, the import file is named `XArrow_TLB.PAS`. The first part of this file includes a couple of GUIDs, one for the library as a whole and one for the control

and other constants for the definition of values corresponding to the OLE enumerated types used by properties of the Delphi control, for example:

```
type
  TxMdWArrowDir = TOleEnum;

const
  adUp = $00000000;
  adLeft = $00000001;
  adDown = $00000002;
  adRight = $00000003;
```

The real meat is the declaration of the `IMdWArrowX` interface, which I suggest you look at in the source code. Notice that the final part of the import unit includes the declaration of the `TMdWArrowX` class. This is a `TOleControl`-derived class you can use to install the control in Delphi, as we've seen in the first part of this chapter. You don't need this class to build the ActiveX control. You need it to install the ActiveX control in Delphi. The class used by the ActiveX server has the same class name but a different implementation.

The rest of the code, and the code you'll customize, is in the main unit, which in my example is called MdWArrowImpl1. This unit has the declaration of the ActiveX server object, `TMdWArrowX`, which inherits from `TActiveXControl` and implements the specific `IMdWArrowX` interface:

```
type
  TMdWArrowX = class(TActiveXControl, IMdWArrowX)
  ...
```

note The `TActiveXControl` class does most of the work for providing ActiveX support in Delphi. This class implements a number of interfaces required by every ActiveX control: `IConnectionPointContainer`, `IDataObject`, `IObjectSafety`, `IOleControl`, `IOleInPlaceActiveObject`, `IOleInPlaceObject`, `IOleObject`, `IPerPropertyBrowsing`, `IPersistPropertyBag`, `IPersistStorage`, `IPersistStreamInit`, `IQuickActivate`, `ISimpleFrameSite`, `ISpecifyPropertyPages`, `IViewObject`, and `IViewObject2`. Just the declaration of the `TActiveXControl` class takes more than 250 lines of code, and its implementation code is responsible for a good part of the 4,000 lines of code of the `AxCtrls` unit.

Before we customize this control in any way, let's see how it works. You should first compile the ActiveX library and then register it using Delphi's Run ➢ Register ActiveX Server menu command. Now you can install the ActiveX control as we've done in the past, except you have to specify a different name for the new class to avoid a name clash. If you use this control, it doesn't look much different from the

original VCL control, but the advantage is that the same component can now be installed also in other development environments.

Adding New Properties

Once you've created an ActiveX control, adding new properties, events, or methods to it is—surprisingly—simpler than doing the same operation for a VCL component. Delphi, in fact, provides specific visual support for the former, not for the latter.

You can simply open the Pascal unit with the implementation of the ActiveX control, and choose Edit ➢ Add To Interface. As an alternative you can use the same command from the local menu of the editor. Delphi opens the Add to Interface dialog box (see Figure 16.17). In the combo box of this dialog box, you can choose between a new property, method, or event. In this example, the first selection will affect the `IMdWArrowX` interface and the second the `IMdWArrowXEvents` interface.

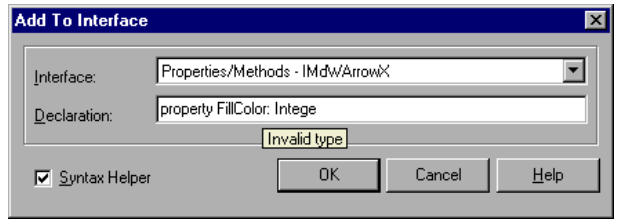


Figure 16.17: The Add to Interface dialog box, with the syntax helper in action. Image from the original book.

In the edit box you can then type the declaration of this new interface element. If the Syntax Helper check box is activated, you'll get hints describing what you should type next and highlighting any errors. You can see the syntax helper in action in Figure 16.17. When you define a new ActiveX interface element, keep in mind that you are restricted to OLE data types. In the XArrow example, I've added two properties to the ActiveX control. Because the `Pen` and the `Brush` properties of the original Delphi components are not accessible, I've made available their color. These are examples of what you can write in the edit box of the Add to Interface dialog (executing it twice):

```
property FillColor: Integer;
property PenColor: Integer;
```

note Since a `TColor` is a specific Delphi definition, it is not legal to use it. `TColor` is an Integer subrange that defaults to Integer size, so I've used the standard Integer type directly.

The declarations you enter in the Add to Interface dialog box are automatically added to the control's Type Library (TLB) file, to its import library unit, and to its implementation unit:

```
type
  IMdWArrowX = interface(IDispatch)
    function Get_FillColor: Integer; safecall;
    procedure Set_FillColor(Value: Integer); safecall;
    function Get_PenColor: Integer; safecall;
    procedure Set_PenColor(Value: Integer); safecall;
    ...
    property FillColor: Integer
      read Get_FillColor write Set_FillColor;
    property PenColor: Integer
      read Get_PenColor write Set_PenColor;
```

All you have to do to finish the ActiveX control is to fill in the `Get` and `Set` methods of the implementation. Here is the code of the first property:

```
function TMdWArrowX.Get_FillColor: Integer;
begin
  Result := ColorToRGB (FDelphiControl.Brush.Color);
end;

procedure TMdWArrowX.Set_FillColor(Value: Integer);
begin
  FDelphiControl.Brush.Color := Value;
end;
```

If you now install this ActiveX control in Delphi once more, the two new properties will appear. The only problem with this property is that Delphi uses a plain integer editor, making it quite difficult to enter the value of a new color by hand. A program, by contrast, can easily use the `RGB` function to create the proper color value.

Adding a Property Page

As it stands, other development environments can do very little with our component, because we've prepared no property page—no property editor. A property page is fundamental so that programmers using the control can edit its attributes. However, adding a property page is not as simple as adding a form with a few controls. The property page, in fact, will integrate with the host development environment. The property page for our control will show up inside a property page dialog of the host environment, which will provide the OK, Cancel, and Apply buttons, and the tabs for showing multiple property pages (some of which might be provided by the host development environment).

The nice thing is that support for property pages is built into Delphi, so adding one is quite simple. You simply open an ActiveX project, then open the usual New Items dialog box, move to the ActiveX page, and choose Property Page. What you get is not very different from a form. In fact, the `TPropertyPage1` class (created by default) inherits from the `TPropertyPage` class of the VCL, which in turn inherits from `TCustomForm`.

note Delphi provides four built-in property pages for colors, fonts, pictures, and strings. The GUIDs of these classes are indicated by the constants `Class_DColorPropPage`, `Class_DFontPropPage`, `Class_DPicturePropPage`, and `Class_DStringPropPage` in the AxCtrls unit.

In the property page, you can add controls as in a normal Delphi form and you can write code to let the controls interact. I've added to the property page a combo box with the possible values of the `Direction` property, a check box for the `Filled` property, an edit box with an UpDown control to set the `ArrowHeight` property, and two shapes with corresponding buttons for the colors. The only code added to the form relates to the two buttons used to change the color of the two shape components, which offer a preview of the colors of the actual ActiveX control. The `OnClick` event of the button uses a ColorDialog component, as usual:

```
procedure TPropertyPage1.ButtonPenClick(Sender: TObject);
begin
  with ColorDialog1 do
  begin
    Color := ShapePen.Brush.Color;
    if Execute then
    begin
      ShapePen.Brush.Color := Color;
      Modified; // enable Apply button!
    end;
  end;
end;
```

What is important to notice in this code is the call to the `Modified` method of the `TPropertyPage` class. This call is required to let the property page dialog box know we've modified one of the values and to enable the Apply button. When a user interacts with one of the other controls of this form, this call is made automatically. For the two buttons, however, we need to add this line ourselves.

note Another tip relates to the `Caption` of the property page form. This will be used in the property dialog box of the host environment as the caption of the tab corresponding to the property page.

The next step is to associate the controls of the property page with the actual properties of the ActiveX control. The property page class automatically has two methods for this: `UpdateOleObject` and `UpdatePropertyPage`. As their names suggest, these two methods copy data from the property page to the ActiveX control and vice versa. Here is the code for my example:

```
procedure TPropertyPage1.UpdatePropertyPage;
begin
  { Update your controls from the OleObject }
  ComboDir.ItemIndex := OleObject.Direction;
  CheckFilled.Checked := OleObject.Filled;
  EditHeight.Text := IntToStr (OleObject.ArrowHeight);
  ShapePen.Brush.Color := OleObject.PenColor;
  ShapePoint.Brush.Color := OleObject.FillColor;
end;

procedure TPropertyPage1.UpdateObject;
begin
  { Update the OleObject from your controls }
  OleObject.Direction := ComboDir.ItemIndex;
  OleObject.Filled := CheckFilled.Checked;
  OleObject.ArrowHeight := UpDownHeight.Position;
  OleObject.PenColor := ColorToRGB (ShapePen.Brush.Color);
  OleObject.FillColor := ColorToRGB (ShapePoint.Brush.Color);
end;
```

The final step is to connect the property page itself to the ActiveX control. When the control was created, the Delphi ActiveX Control Wizard automatically added a declaration for the `DefinePropertyPages` method to the implementation unit. In this method, we simply call the `DefinePropertyPage` method (this time the method name is singular) for each property page we want to add to the ActiveX. This method has as its parameter the GUID of the property page, something you can find in the corresponding unit. (Of course, you'll need to add a `uses` statement referring to that unit.) Here is the code of my example:

```
procedure TMdWArrowX.DefinePropertyPages(
  DefinePropertyPage: TDefinePropertyPage);
begin
  DefinePropertyPage(Class_PropertyPage1);
end;
```

note The connection between the ActiveX control and its property page takes place using a GUID. This is possible because the property page object can be created through a class factory, and its GUID is stored in the Windows Registry when you register the ActiveX control library. To see what's going on, look at the `initialization` section of the property page unit, which calls `TActiveXPropertyPageFactory.Create`.

Now that we've finished developing the property page, and after recompiling and reregistering the ActiveX library, we can install the ActiveX control inside a host development environment (including Delphi itself) and see how it looks. Figure 16.18 shows an example. (If you've already installed the ActiveX control in Delphi, you should uninstall it prior to rebuilding it. This process might also require closing and reopening Delphi itself.)

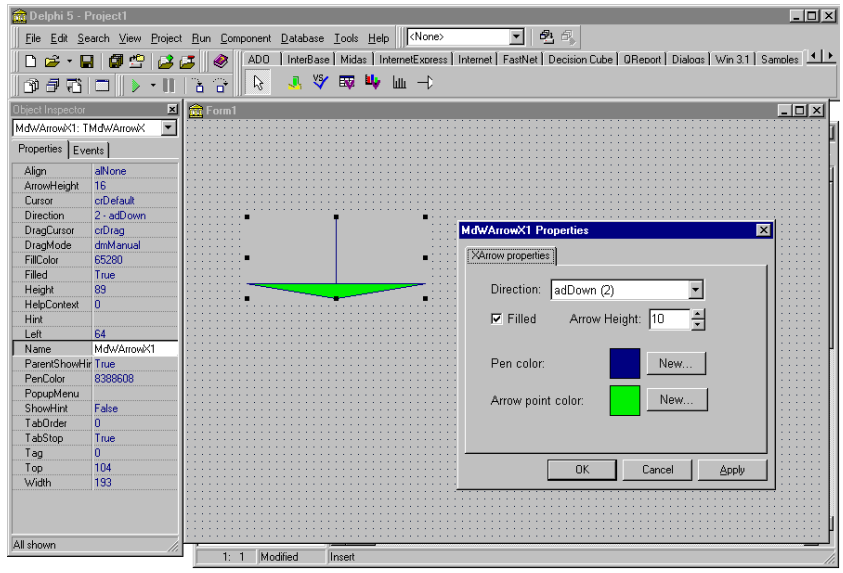


Figure 16.18: The XArrow ActiveX control and its property page, hosted by the Delphi environment. Image from the original book.

ActiveForms

As I've mentioned before, Delphi provides an alternative to the use of the ActiveX Control Wizard to generate an ActiveX control. You can use an ActiveForm, which is an ActiveX control that is based on a form and can host one or more Delphi components. This is exactly the technique used in Visual Basic to build new controls, and it makes sense when you want to create a compound component.

For example, to create an ActiveX clock we can simply place on an ActiveForm a label (which is a graphic control that cannot be used as a starting point for an ActiveX control) and a timer, and connect the two with a little code. The form/con-

trol becomes basically a container of other controls, which makes it very easy to build compound components (easier than for a VCL compound component).

To build such a control, simply close the current project, and select the ActiveForm icon in the ActiveX page of the File ➢ New dialog box. Delphi asks you some information in the following ActiveForm Wizard dialog box, similar to the ActiveX Control Wizard dialog box.

ActiveForm Internals

Before we continue with the example, let's look at the code generated by the ActiveForm Wizard. The key difference from a plain Delphi form is in the declaration of the new form class, which inherits from the `TActiveForm` class and implements a specific ActiveForm interface:

```
type
  TAXForm1 = class(TActiveForm, IAXForm1)
```

As usual, the `IAXForm` interface is declared in the Type Library and in a corresponding Pascal file generated by Delphi. Here is a small excerpt of the `IAXForm1` interface from the `XF1Lib.pas` file, with some comments I've added:

```
type
  IAXForm1 = interface(IDispatch)
    ['{51661AA1-9468-11D0-98D0-444553540000}']
    // Get and Set methods for TForm properties
    function Get_Caption: WideString; safecall;
    procedure Set_Caption(const Value: WideString); safecall;
    ...
    // TForm methods redeclared
    procedure Close; safecall;
    ...
    // TForm properties
    property Caption: WideString
      read Get_Caption write Set_Caption;
```

The code generated for the `TAXForm1` class implements all the `Set` and `Get` methods, which simply change or return the corresponding properties of the form, and it implements the events, which again are the events of the form. Here is a small excerpt:

```
  private
    procedure ActivateEvent(Sender: TObject);
  protected
    procedure Initialize; override;
    function Get_Caption: WideString; safecall;
```

```
    procedure Close; safecall;
    procedure Set_Caption(const Value: WideString); safecall;
```

Let's look at the implementation of properties first:

```
function TAXForm1.Get_Caption: WideString;
begin
  Result := WideString(Caption);
end;

procedure TAXForm1.Set_Caption(const Value: WideString);
begin
  Caption := TCaption(Value);
end;
```

The `TForm` events are set to the internal methods when the form is created:

```
procedure TAXForm1.Initialize;
begin
  OnActivate := ActivateEvent;
  ...
end;
```

Each event then maps itself to the external ActiveX event, as in the following two methods:

```
procedure TAXForm1.ActivateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnActivate;
end;
```

Because of this mapping you should not handle the events of the form directly. Instead, you can either add some code to these default handlers or simply override the `TForm` methods that end up calling the events. (This is exactly the approach you use when building a Delphi component.) Keep in mind that the interface properties of an ActiveForm are meant for developers using it as a control, not for final users of the ActiveForm on the Web. This mapping problem refers only to the events of the form itself, not to the events of the components of the form. You can continue to handle the events of the components as usual.

The XClock ActiveX Control

Now that we've looked at the code generated by Delphi, we can return to the development of the XClock example. Simply place the components on the form, and set their properties as described below:

```
object XClock: TXClock
  AxBorderStyle = afbSunken
  Caption = 'XClock'
  Color = clBtnFace
  object Label1: TLabel
    Align = alClient
    Alignment = taCenter
    Font.Height = -27
    Font.Name = 'Arial'
    Font.Style = [fsBold]
    Layout = tlCenter
  end
  object Timer1: TTimer
    OnTimer = Timer1Timer
  end
end
```

The last step is to write an event handler for the `OnTimer` event of the timer itself, so that the control updates the output of the label with the current time every second:

```
procedure TXClock.Timer1Timer(Sender: TObject);
begin
  Label1.Caption := TimeToStr (Time);
end;
```

Now simply compile this library, register it, and install it in a package to test it in the Delphi environment. You can see an example of its use in Figure 16.19. Notice in this figure the effect of the sunken border. This is controlled by the `AxBorderStyle` property of the active form, one of the few properties of active forms that is not available for a plain form.

ActiveForms are usually considered as a technique to deploy a Delphi application via the Internet. However, the ActiveX and ActiveForm support provided by Delphi simply represent to different ways to build ActiveX controls, which can be used both on a Web page and in another development environment.

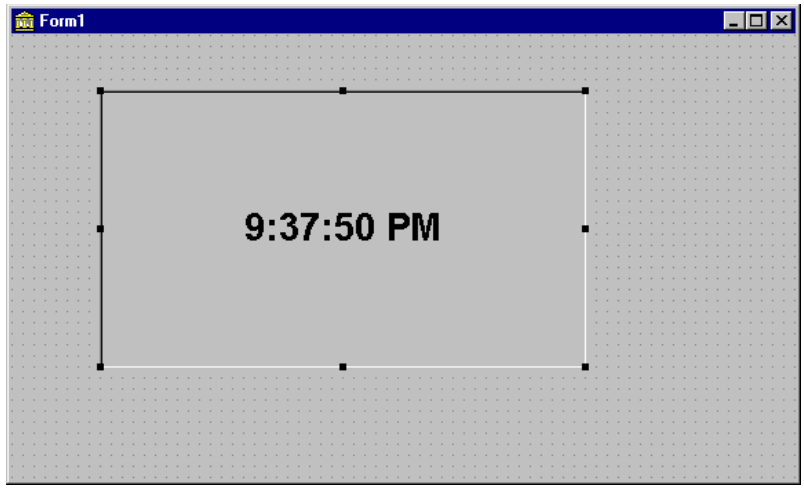


Figure 16.19: The ActiveX timer installed in a Delphi package. Image from the original book.

What's Next?

In this chapter, I have discussed applications of the Microsoft's COM technology, covering automation, documents, and controls. We've seen how Delphi makes the development of Automation servers and clients, and ActiveX controls very simple. Delphi 5 has improved this support by enabling us to wrap simple components around Automation server, such as Word and Excel.

We'll get back to other elements related to COM when discussing Internet and distributed applications, in Chapters 20 and 21. For the moment, however, most of Part V of the book will focus on other interesting programming tasks, such as multithreading, debugging, and printing. The following chapters offer you some building blocks for the development of *real-world* applications. In other words, these chapters try to offer answers for common everyday programming problems.

Chapter 17: Multitasking, Multithreading, And Synchronization

The Windows 32 platform allows the execution of many simultaneous programs and the activation of multiple concurrent threads of execution for a single program. Although there are differences between Windows NT (now Windows 2000) and the

Windows 95/98 operating systems, most of the key elements are common to the entire Win32 platform[362].

In this chapter we'll discuss threads, mutexes, and synchronization objects. We'll also take a second look at messages and some `Application` events, which in some cases offer a simpler solution to the background computing needs of a program.

Events, Messages, and Multitasking in Windows

To understand how Windows applications work internally, we need to spend a minute discussing how multitasking is supported in this environment. We also need to understand the role of timers (and the Timer component) and of background (or *idle*) computing.

In short, we need to delve deeper into the event-driven structure of Windows and its multitasking support. Because this is a book about *Delphi* programming, I won't discuss this topic in detail, but I will provide an overview for readers who have limited experience with Windows API programming.

Event-Driven Programming

The basic idea behind event-driven programming is that specific events determine the control flow of the application. A program spends most of its time waiting for these events and provides code to respond to them. For example, when a user clicks one of the mouse buttons, an event occurs. A message describing this event is sent to the window currently under the mouse cursor. The program code that responds to events for that window will receive the event, process it, and respond accordingly. When the program has finished responding to the event, it returns to a waiting or "idle" state.

As this explanation shows, events are serialized; each event is handled only after the previous one is completed. When an application is executing event-handling code (that is, when it is not waiting for an event), other events for that application have to

362 More recent versions of Windows, starting from Windows 7 up to Windows 11, used the Windows NT core architecture, offering a much more powerful threading model.

wait in a message queue reserved for that application (unless the application uses multiple threads, each with its own message queue). When an application had responded to a message and returned to a waiting state, it became the last in the list of programs waiting to handle additional messages. In 16-bit Windows, there was no way to stop an application from executing a complex event handler, and other applications simply had to wait.

Event handling and the message queues are still the core of Win32, but in Windows 9x and NT, after a fixed amount of time has elapsed, the system interrupts the current application and immediately gives control to the next one in the list. The first program is resumed only after each application has had a turn. This is called preemptive multitasking.

Because of the limited nonpreemptive multitasking, Win16 applications used many different techniques to try to divide an algorithm into smaller chunks and execute them one at a time. These techniques included using timers and performing *idle* computing, and they are still useful to obtain the execution of a background task without the extra effort required by the use of threads. Therefore I'll describe them in the following sections.

In Win32, if an application has responded to its events and is waiting for its turn to process messages, it has no chance to regain control until it receives another message (unless it uses multithreading). This is one reason timers continue to be used. One final note—when you think about events, remember that input events (using the mouse or the keyboard) account for only a small percentage of the total message flow in a Windows application. Most of the messages are the system's internal messages or messages exchanged between different controls and windows. Even a familiar input operation such as clicking a mouse button can result in a huge number of messages, most of which are internal Windows messages.

You can test this yourself by using the WinSight utility included in Delphi[363]. In WinSight, choose to view the Message Trace, and select the messages for all of the windows. Select Start, and then perform some normal operations with the mouse. You'll see hundreds of messages in a few seconds (as in Figure 17.1). Of course, WinSight causes Windows to run much slower than usual because of its monitoring. At normal speed, the flow of messages is much faster than the already high rate you'll see when you run WinSight.

---

363 As mentioned earlier in the book, this utility is no longer available.

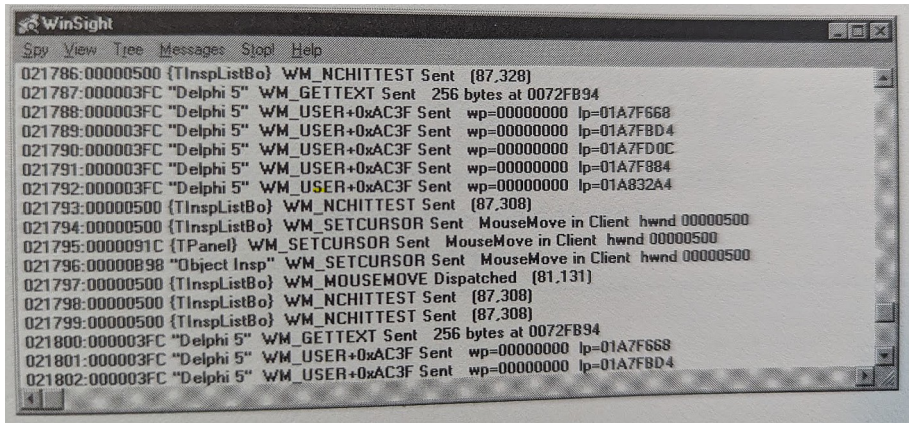


Figure 17.1: The Windows message tracing performed by the WinSight tool included in Delphi. Image based on a picture of the original printed book.

Windows Message Delivery

Before looking at some real examples, we need to consider another key element of message handling. Windows has two different ways to send a message to a window:

- The `PostMessage` API function is used to place a message in the application's message queue. The message will be handled only when the application has a chance to access its message queue (that is, when it receives control from the system), and only after earlier messages have been processed. This is an asynchronous call, since you do not know when the message will actually be received.
- The `SendMessage` API function is used to execute message-handler code immediately. `SendMessage` bypasses the application's message queue and sends the message directly to a target window or control. This is a synchronous call. This function even has a return value, which is passed back by the message-handling code. Calling `SendMessage` is no different than directly calling another method or function of the program.

The difference between these two ways of sending messages is similar to that between mailing a letter, which will reach its destination sooner or later, and sending a fax, which goes immediately to the recipient. Although you will rarely need to use these low-level functions in Delphi, this description should help you determine which one to use if you do need to write this type of code.

Background Processing and Multitasking

Suppose that you need to implement a time-consuming algorithm. If you write the algorithm as a response to an event, your application will be stopped completely during all the time it takes to process that algorithm. To let the user know that something is being processed, you can display the hourglass cursor, but this is not a user-friendly solution. Win32 allows other programs to continue their execution, but the program in question will freeze; it won't even update its own user interface if a repaint is requested. In fact, while the algorithm is executing, the application won't be able to receive and process any other messages, including the paint messages.

The simplest solution to this problem is to call the `ProcessMessages` method of the `Application` object many times within the algorithm, usually inside an internal loop. This call stops the execution, allows the program to receive and handle a message, and then resumes execution. This technique has been used in earlier chapters (such as the Credits example in Chapter 10 and the Callback example in Chapter 10), so I won't demonstrate it here. The problem with this approach, however, is that while the program is paused to accept messages, the user is free to do any operation and might again click the button or press the keystrokes that started the algorithm. To fix this, you can disable the buttons and commands you don't want the user to select, and you can display the hourglass cursor (which technically doesn't prevent a mouse click event, but it does suggest that the user should wait before doing any other operation). An alternative solution is to split the algorithm into smaller pieces and execute each of them in turn, letting the application respond to pending messages in between processing the pieces. We have seen in many examples in past chapters (including the MdEdit5 example in Chapter 7, the LockTest example in Chapter 10, and the MdClock component in Chapter 13) that we can use a timer to let the system notify us once a time interval has elapsed. Although you can use timers to implement some form of background computing, this is far from a good solution. A slightly better technique would be to execute each step of the program when the `Application` object receives the `OnIdle` event.

The difference between calling `ProcessMessages` and using the `OnIdle` events is that by calling `ProcessMessages`, you will give your code more processing time than with the `OnIdle` approach. Calling `ProcessMessages` is a way to let the system perform other operations while your program is computing; using the `OnIdle` event is a way to let your application perform background tasks when it doesn't have pending requests from the user.

A third way to implement background processing—less common and more complex—is to have the application post a user-defined message to itself at the end of each step in the background process:

```
PostMessage (Handle, wm_User, 0, 0);
```

The application will get this message after a while, so it can execute the next step and then post another message to itself, continuing until the background processing is done.

note All these techniques for background computing were necessary in 16-bit Windows days. In Win32, multitasking between applications works better and the system provides threads exactly to let you implement background processing. However, techniques such as using timers, processing the `OnIdle` event, and posting custom message are still very common[364].

Checking for a Previous Instance of an Application

One form of multitasking is the execution of two or more instances of the same application. Any application can generally be executed by a user in more than one instance, and it needs to be able to check for a previous instance already running, in order to disable this default behavior and allow for one instance at most. This section demonstrates several ways of implementing such a check, allowing me to discuss a number of interesting Windows programming techniques.

note In 16-bit Windows applications, you could test the value of the `HPrevInstance` system parameter to see if a previous instance of the application was running. Unfortunately, in 32-bit Windows, this parameter is always 0.

364 They were very common by the time the book was written, not today. With the advent of multi-core CPUs, the issue has shifted from keeping the UI responsive to take more advantage of the computing power by letting multiple operations happen in parallel. The use of threads has become more common because it lets you address both use cases.

Looking for a Copy of the Main Window

To find a copy of the main window of a previous instance, use the `FindWindow` API function and pass it the name of the window class (the name used to register the form's window type, or `WNDCLASS`, in the system) and the caption of the window for which you are looking. In a Delphi application, the name of the `WNDCLASS` window class is the same as the Object Pascal name for the form's class (for example, *TForm1*). The result of the `FindWindow` function is either a handle to the window or zero (if no matching window was found).

The main code of your Delphi application should be written so that it will execute only if the `FindWindow` result is zero:

```
var
  Hwnd: THandle;
begin
  Hwnd := FindWindow ('TForm1', nil);
  if Hwnd = 0 then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end
  else
    SetForegroundWindow (Hwnd)
end.
```

To activate the window of the previous instance of the application, you can use the `SetForegroundWindow` function, which works for windows owned by other processes. This call produces its effect only if the window passed as parameter hasn't been minimized. When the main form of a Delphi application is minimized, in fact, it is hidden and for this reason the activation code has no effect.

Unfortunately, if you run a program that uses the `FindWindow` call just shown from within the Delphi IDE, a window with that caption and class may already exist: the design-time form. Thus, the program won't start even once. However, it will run if you close the form and its corresponding source code file (closing only the form, in fact, simply hides the window), or if you close the project and run the program from the Windows Explorer.

Using a Mutex

A completely different approach is to use a *mutex*, or mutual exclusion object. This is a typical Win32 approach, commonly used for synchronizing threads, as we'll see

later in this chapter. Here we are going to use a mutex for synchronizing two different applications, or (to be more precise) two instances of the same application.

Once an application has created a mutex with a given name, it can test whether this object is already owned by another application, calling the `WaitForSingleObject` Windows API function. If the mutex has no owner, the application calling this function becomes the owner. If the mutex is already owned, the application waits until the time-out (the second parameter of the function) elapses. It then returns an error code.

To implement this technique, you can use the following project source code, which you'll find in the OneCopy example:

```
var
  hMutex: THandle;
begin
  HMutex := CreateMutex (nil, False, 'OneCopyMutex');
  if WaitForSingleObject (hMutex, 0) <> wait_TimeOut then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end;
end.
```

If you run this example twice, you'll see that it creates a new, temporary copy of the application (the icon appears in the Taskbar) and then destroys it when the time-out elapses. This approach is certainly more robust than the previous one, but it lacks a feature: how do we enable the existing instance of the application? We still need to find its form, but we can use a better approach.

Searching the Window List

When you want to search for a specific main window in the system, you can use the `EnumWindows` API functions. Enumeration functions are quite peculiar in Windows, because they usually require another function as a parameter. These enumeration functions require a pointer to a function (often described as a *callback* function) as parameter. The idea is that this function is applied to each element of the list (in this case, the list of main windows), until the list ends or the function returns `False`. Here is the enumeration function from the OneCopy example:

```
function EnumWndProc (hwnd: THandle;
  Param: Cardinal): Bool; stdcall;
var
  ClassName, WinModuleName: string;
```

```
  WinInstance: THandle;
begin
  Result := True;
  SetLength (ClassName, 100);
  GetClassName (hwnd, PChar (ClassName), Length (ClassName));
  ClassName := PChar (ClassName);
  if ClassName = TForm1.ClassName then
  begin
    // get the module name of the target window
    SetLength (WinModuleName, 200);
    WinInstance := GetWindowLong (hwnd, GWL_HINSTANCE);
    GetModuleFileName (WinInstance,
      PChar (WinModuleName), Length (WinModuleName));
    WinModuleName := PChar(WinModuleName); // adjust length

    // compare module names
    if WinModuleName = ModuleName then
    begin
      FoundWnd := Hwnd;
      Result := False; // stop enumeration
    end;
  end;
end;
```

This function, called for each nonchild window of the system, checks the name of each window's class, looking for the name of the `TForm1` class. When it finds a window with this string in its class name, it uses `GetModuleFilename` to extract the name of the executable file of the application that owns the matching form. If the module name matches that of the current program (which was extracted previously with similar code), you can be quite sure that you have found a previous instance of the same program. Here is how you can call the enumerated function:

```
var
  FoundWnd: THandle;
  ModuleName: string;

begin
  if WaitForSingleObject (hMutex, 0) <> wait_TimeOut then
    ...
  else
  begin
    // get the current module name
    SetLength (ModuleName, 200);
    GetModuleFileName (HInstance,
      PChar (ModuleName), Length (ModuleName));
    ModuleName := PChar (ModuleName); // adjust length

    // find window of previous instance
    EnumWindows (@EnumWndProc, 0);
```

Handling User-Defined Window Messages

I've mentioned earlier that the `SetForegroundWindow` call doesn't work if the main form of the program has been minimized. Now we can solve this problem. You can ask the form of another application, the previous instance of the same program in this case, to restore its main form by sending it a user-defined window message. You can then test whether the form is minimized and post a new user-defined message to the old window. Here is the code; in the OneCopy program, it follows the last fragment shown in the previous section:

```
    if FoundWnd <> 0 then
    begin
      // show the window, eventually
      if not IsWindowVisible (FoundWnd) then
        PostMessage (FoundWnd, wm_User, 0, 0);
      SetForegroundWindow (FoundWnd);
    end;
```

Again, the `PostMessage` API function sends a message to the message queue of the application that owns the destination window. In the code of the form, you can add a special function to handle this message:

```
public
  procedure WMUser (var msg: TMessage);
    message wm_User;
```

Now you can write the code of this method, which is simple:

```
procedure TForm1.WMUser (var msg: TMessage);
begin
  Application.Restore;
end;
```

Multithreading in Delphi

Win32 allows us to let two procedures or methods execute at the same time and let our program control them. Before we look at the implementation of multithreading, we should ask ourselves why we might want to have several threads of execution inside a given program. First, consider some of the *disadvantages* of multithreading:

- Multithreading makes a program run slower, unless you have multiple CPUs and the operating system can split the threads among processors[365].
- A poorly written multithreaded application can run slower on a multiprocessor system than on a single processor system. Synchronization between threads is much more expensive on multiprocessor systems than on single.
- Multithreading programs must synchronize access to shared resources and memory, which makes them much more complex to write, as we'll see in many cases.

Fortunately, multithreading also has some advantages. For example, you can run a thread in the background, letting the user continue to operate the program. You can make one thread run faster than others by adjusting its priority, regulate the resource access of different threads, assign local storage to each thread, and spawn multiple threads of the same type. But the key advantage is that you can simply write your algorithm inside a thread, without having to worry about splitting it, letting the system refresh the user interface, or anything else.

The best operation to consider putting in a background thread is something that takes considerable CPU time to complete (threads are expensive to create for the operating system, so you must make sure it's worth the trouble) and is fairly isolated in terms of data access (no synchronization to access shared resources). If an operation is quick to finish, or heavily dependent on external shared data, threads aren't worth the trouble. Similarly, using threads to split up a multistep process that is inherently sequential in nature is a waste of time. (My own example programs don't always stick to these rules; but they are, after all, only examples designed to demonstrate specific techniques.)

The TThread Class

Windows provides a series of API calls to control threads (the key one is `CreateThread`), but I won't discuss them here because Delphi provides a `TThread` class that will let us control threads well enough[366].

---

365 While multi core CPUs where uncommon for PCs at the time this book was written, now they are everywhere, including in your phone. Therefore, as already mentioned, the relevance of multi-threading has grown significantly.

366 While additional features have been added to the `TThread` class, the core concepts remain the same today. Delphi introduced also newer concepts, like parallel for loops, tasks and futures, but those fall way beyond the scope of the notes to this book.

The first thing to know about the `TThread` class is that you never use it directly, because it is an abstract class—a class with a virtual abstract method. To use threads, you always subclass `TThread` and use the features of this base class. The `TThread` class has a constructor with a single parameter that lets you choose whether to start the thread immediately or suspend it until later:

```
constructor Create(CreateSuspended: Boolean);
```

There are also some public synchronization methods:

```
procedure Resume;
procedure Suspend;
function Terminate: Integer;
function WaitFor: Integer;
```

The published properties include `Priority`, `Suspended`, and two read-only, low-level values: `Handle` and `ThreadID`. The class also provides a protected interface, which includes two key methods for your thread subclasses:

```
procedure Execute; virtual; abstract;
procedure Synchronize(Method: TThreadMethod);
```

The `Execute` method, declared as a virtual abstract procedure, must be redefined by each thread class. It contains the main code of the thread, the code you would typically place in a *thread function* when using the Windows API. The `Synchronize` method is used to avoid concurrent access to VCL components. The VCL code runs inside the main thread of the program, and you need to synchronize access to the VCL to avoid reentrancy problems (errors from reentering a function before a previous call is completed). The only parameter of `Synchronize` is a method that accepts no parameters, typically a method of the same thread class[367].

General VCL controls, by contrast, are not thread-safe. For example, you should not create VCL controls in the context of a background thread: window handles only receive messages on the message queue of the thread in which the handle was created. If you create a control in a background thread, the `Application` object in the main thread will not dispatch messages to that control's window handle. Microsoft strongly advises against using multiple threads for user interface creation.

What VCL does support, to a limited degree, is manipulation of main-thread VCL controls from background threads. The most common case is a background thread wanting to draw on the form. For example, the `TCanvas` class has two locking methods, `Lock` and `UnLock`, that allow a background thread to paint directly on the `Canvas`

367 There is now an overloaded version of the `Synchronize` method, which takes an anonymous method as parameter.

of the main form. `TBitmap` and `TJPEGImage` classes can be use for image processing in background threads. Another helpful class is `TThreadList`, which allow different threads to access the same `TList` safely and concurrently.

note Delphi 5 has improved the thread safety of parts of the run-time library (RTL). String reference counting now works across threads, and the reference counting of COM objects has also been improved, to better support the COM apartment-threading model.

A First Example

As a first simple example, I've built a program that uses the `Synchronize` method. The program, called ThOld, uses a thread to paint on the surface of a form. The thread class, `TPainterThread`, overrides the `Execute` method and defines a custom `Paint` method. The `Paint` method is used to access VCL objects, so it is called only from within the `Synchronize` method. Since the `Paint` method cannot accept parameters directly and still be a compatible argument for the `Synchronize` method, the class requires some private data. Here is the thread class declaration:

```
type
  TPainterThread = class(TThread)
  private
    X, Y: Integer;
  protected
    procedure Execute; override;
    procedure Paint;
end;
```

The `Paint` method marks pixels of the form in red:

```
procedure TPainterThread.Paint;
begin
  Form1.Canvas.Pixels [X, Y] := clRed;
end;
```

The main function of the thread, `Execute`, randomly updates a pixel by passing the `Paint` method as the parameter of the `Synchronize` method:

```
procedure TPainterThread.Execute;
begin
  Randomize;
  repeat
    X := Random (300);
    Y := Random (Form1.ClientHeight);
    Synchronize (Paint);
  until Terminated;
```

```
end;
```

Figure 17.2 shows the result of this code. As you can see in the listing above, the thread runs until it is terminated. On the main form, there is a button used to start the thread. Because we pass `False` to the `Create` constructor (in the `Button1Click` method), the thread starts immediately:

```
PT := TPainterThread.Create (False);  // start immediately
```

`PT` is a private `TPainterThread` field of the form class. A second button frees the thread object.

Figure 17.2: The output of the ThOld example. The colored pixels are painted by a background thread. Picture captured after rebuilding the program in Delphi 12.

The main form also handles mouse clicks; when a mouse button is pressed on the form, the red pixels within a circular portion of the form around the click position are wiped out, repainting the form with its background color (as you can guess by looking at Figure 17.2):

```
procedure TForm1.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Canvas.Pen.Color := Color; // of the form
  Canvas.Brush.Color := Color;
  Canvas.Ellipse (x - 30, y - 30, x + 30, y + 30);
end;
```

A Locking Example

We can now rewrite the previous example using the native support for thread synchronization provided by the `TCanvas` class. In this case, to access the form's canvas

we can simply lock it and avoid the call to `Synchronize`, simplifying the code and making its execution much faster. The thread class in the ThLock example has just one method:

```
type
  TPainterThread = class(TThread)
  protected
    procedure Execute; override;
  end;
```

The code of `Execute` now does everything, including displaying the output, after locking the canvas of the form:

```
procedure TPainterThread.Execute;
var
  X, Y: Integer;
begin
  Randomize;
  repeat
    X := Random (300);
    Y := Random (Form1.ClientHeight);
    with Form1.Canvas do
    begin
      Lock;
      try
        Pixels [X, Y] := clRed;
      finally
        Unlock;
      end;
    end;
  until Terminated;
end;
```

If you then want to access the `Canvas` in an event handler other than `OnPaint` (which handles locking automatically) you should call the `Lock` method again:

```
procedure TForm1.FormMouseDown (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Canvas.Lock;
  try
    Canvas.Pen.Color := clYellow;
    Canvas.Brush.Color := clYellow;
    Canvas.Ellipse (x - 30, y - 30, x + 30, y + 30);
  finally
    Canvas.Unlock;
  end;
end;
```

Synchronization Alternatives

The use of the `Lock` method is limited to the few Delphi objects that have this capability. The alternative is to continue using the `Synchronize` protected method of the `TThread` class when you expect several threads to need access to a component's properties at the same time. Generally you will use secondary threads for background operations, such as file transfer or number crunching, with little or no need to update the user interface. If you need limited updates of the user interface, there are alternative approaches to the two techniques just mentioned.

First, a thread can update some data structure (such as a queue or a circular buffer), which the main thread scans from time to time. You have to take care to avoid read/write collisions between different threads. As a second alternative, you can use traditional Windows-based multitasking; the thread can post a message to the main window, asking for an update. Keep in mind that you cannot always use `SendMessage` (which is synchronous, similar to a direct function call) to do this; instead, you should use only `PostMessage` (which is asynchronous and uses the message queue). These alternative techniques are not used frequently anyway, compared to the first two I've shown you, so there is no example in the book implementing them.

Thread Priorities

Our third example of threads is an extension of the previous one. This time we use several threads at the same time, and the program allows users to change their priorities with some track bars. Here is the new version of the `TPainterThread` class:

```
type
  TPainterThread = class(TThread)
  private
    Color: Integer;
  protected
    procedure Execute; override;
  public
    constructor Create (Col: TColor);
  end;
```

I've added a constructor to the class to pass an initial color value to the thread. As an alternative, I could have made the `Color` a public field of the thread class to allow the program to manipulate them directly. Here is the code of the constructor:

```
constructor TPainterThread.Create(Col: TColor);
begin
```

```
  Color:= Col;
  inherited Create (True);
end;
```

The constructor initializes the private data and then calls the constructor of the base class, creating the thread in a suspended state. The `Execute` method of the thread simply scans each screen line, setting each pixel to the given color:

```
procedure TPainterThread.Execute;
var
  X, Y, X1: Integer;
begin
  X := 0;
  Y := 0;
  repeat
    // scan the lines...
    X1 := X + 1;
    X := X1 mod 250;
    Y := Y + X1 div 250;
    Y := Y mod Form1.ClientHeight;
    Form1.Canvas.Lock;
    try
      Form1.Canvas.Pixels [X, Y] := Color;
    finally
      Form1.Canvas.UnLock;
    end;
  until Terminated;
end;
```

The main form has four check boxes and four track bars (as you can see in Fig-ure 17.3). The form has some local data, too, an array to hold the four thread objects:

```
private
  PT: array [1..4] of TPainterThread;
```

This array is initialized when the form is created:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  PT [1] := TPainterThread.Create (clRed);
  PT [2] := TPainterThread.Create (clBlue);
  PT [3] := TPainterThread.Create (clGreen);
  PT [4] := TPainterThread.Create (ClBlack);
end;
```

Figure 17.3: The output of the ThPrior example, with four threads updating the user interface concurrently. Image recaptured by rebuilding the app in Delphi 12.

Notice that the program creates the four threads as suspended. They are started when the corresponding check box is checked and suspended again when the check box is cleared:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if (Sender as TCheckbox).Checked then
    PT [(Sender as TCheckbox).Tag].Resume
  else
    PT [(Sender as TCheckbox).Tag].Suspend;
end;
```

To use the same event handler for all check boxes, I've set the value of the `Tag` property of each one to the number of the corresponding thread. Sometimes when you disable one of the threads by selecting the corresponding check box, all the threads stop at once. If you happen to stop a thread while it has locked the canvas, all the other threads will have to wait for the canvas to be unlocked, but this cannot happen until you resume the threads. This problem can be solved in various ways, using the `TryLock` and `LockCount` methods of the Canvas. In the final version of the ThPrior example I've replaced the call to `Suspend` shown above with a call to another function I've added to the thread:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if (Sender as TCheckbox).Checked then
    PT [(Sender as TCheckbox).Tag].Resume
  else
    PT [(Sender as TCheckbox).Tag].DelayedSuspend;
end;
```

The `DelayedSuspend` method of the thread simply sets the `SuspendRequest` Boolean field of the thread. This Boolean value is checked at the end of the cycle inside the thread execution code, which eventually calls `Suspend` on itself. This is the code added to `TPainterThread.Execute` (added to the listing shown earlier, just before the end of the `repeat-until` statement):

```
    if SuspendRequest then
    begin
      Suspend;
      SuspendRequest := False;
    end;
```

With this technique we know for sure that `Suspend` is called only when the thread is not locking the canvas.

The `Tag` property is also used with the track bars, which set the current priority of the thread:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  PT [(Sender as TTrackBar).Tag].Priority :=
    TThreadPriority ((Sender as TTrackBar).Position);
end;
```

To set the priority, I simply cast the current `Position` of the track bar to the corresponding `TThreadPriority` enumeration value. Then I use the resulting value to set the priority of the corresponding thread, as determined by the `Tag` property. This program is quite instructive, because you can alter thread priorities and see the effect on the screen.

Running it, you'll notice that a nonblocking thread running at a higher priority (even just one notch above normal) will consume almost all the CPU time and starve all lower-priority threads in the process. This means you should only raise the priority of a thread that spends most of its time blocked, waiting for an event to signal; and even then, you should question the need to tinker with priority. What's more common in Windows applications is to lower the priority of a thread to below normal, so that it executes only when everything else is quiet and the application remains responsive to user input.

Synchronizing Threads

We have seen that there are two common approaches for synchronizing a thread with the rest of the application: using the `Synchronize` method of a thread object

and using the `Lock` method of a VCL class that provides it, such as `TCanvas`. In an earlier example we also used the `Lock` method of the `Canvas` of the main form to synchronize four threads that were painting on the screen. But that was an atypical case. Generally you'll need to use other techniques to synchronize two threads, including low-level techniques available in the Windows API.

In the next section I'll discuss a simple case, waiting for a thread to terminate. Then, in following sections, we will see more complex examples.

Waiting for a Thread

When a thread should wait until another thread is done, it can simply call the `WaitFor` method of the object corresponding to the thread that should terminate. Here is a portion of an example, in which a program starts a thread and then waits for its result:

```
Comp := TMyThread.Create (True);
// initialize the thread...
Comp.Resume;
Comp.WaitFor;
// look for final values...
Comp.Free;
```

note If you create a thread in the suspended state and then destroy it before resuming it, you will leak memory. This is not a specific Delphi problem; Microsoft's tools warn against this as well. The technical reason is that the creation of the thread handle involves some temporary memory allocation to pass context information into the procedure associated with the thread. As soon as the thread begins execution, the temporary memory is freed; but if the thread is destroyed without ever executing, the temporary memory is lost. The amount of memory lost is quite small (only 12 bytes), but even this can turn into a fatal situation in a server application intended to run continuously for months or years.

This code is quite simple to write, but remember that you cannot write this code as part of the main thread (for example, in a normal message-response function) if the secondary thread has to synchronize with it. If the thread connected to the main form is waiting for another thread to finish, and the secondary thread is waiting to access the user interface (hence waiting for the main thread to finish its current job), the program will enter a deadlock!

To avoid this problem, you can use `WaitFor` to synchronize two threads. A first thread creates a secondary thread, and waits for it to end, without interfering with the main form's thread.

To show you an example of synchronization with multiple threads, I've built a character-counting program, called ThWait. The program computes how many copies of the four characters specified in an edit box are present in the text of a Memo component (as an alternative, you can load the text from any file, as long as you do so before starting the computation). The program looks for each of the four characters at the same time, using multiple threads spawned by the main thread. To improve the output, each thread shows its status—that is, how far through the text it has searched—in a progress bar, as you can see in Figure 17.4.

Because memo lines are fetched using a `SendMessage` to the control's window handle, and a window's messages are processed in the context of the main thread, in practice there is no advantage to using four threads to scan through the text of the same memo control. Still, the example demonstrates some useful techniques.

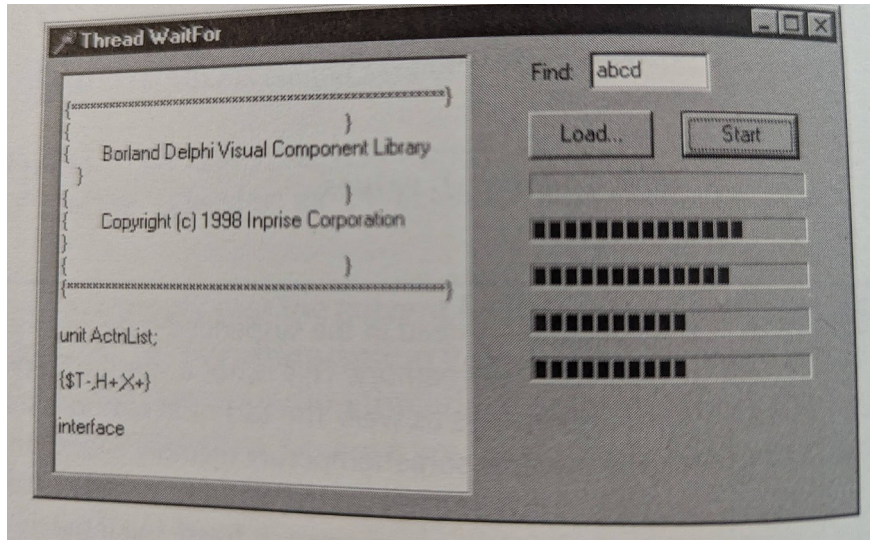


Figure 17.4: In the ThWait example, each thread outputs its status in a progress bar. Image based on a picture of the original printed book.

The actual engine of the program is the `TFindThread` class, which contains the `LookFor` field to hold the character we are searching for and a `Progress` field to store the value of the progress bar and update its status. The result of the computation is placed in the public `Found` field:

```
type
  TFindThread = class(TThread)
  protected
    Progr: Integer;
    procedure UpdateProgress;
    procedure Execute; override;
```

```
  public
    Found: Integer;
    LookFor: Char;
    Progress: TProgressBar;
  end;
```

As usual, the core of the thread is in its `Execute` method, which scans the lines of the memo, looking for the given character. Notice that we can freely access the properties of the memo without synchronization, since this operation is a nondestructive read—it doesn't affect the status of the Memo component:

```
procedure TFindThread.Execute;
var
  I, J: Integer;
  Line: string;
begin
  Found := 0;
  with Form1.Memo1 do
    for I := 0 to Lines.Count - 1 do
    begin
      Line := Lines [I];
      for J := 1 to Length (Line) do
        if Line [J] = LookFor then
          Inc (Found);
      Progr := I + 1;
      Synchronize (UpdateProgress);
    end;
end;
```

The `UpdateProgress` method simply updates the status of the progress bar using the value of the field `Progr`:

```
procedure TFindThread.UpdateProgress;
begin
  Progress.Position := Progr;
end;
```

Four copies of this thread are activated by the primary thread, an object of the `TMultiFind` class. Here is the declaration of this class:

```
type
  TMultiFind = class(TThread)
  protected
    Progr: Integer;
    procedure UpdateProgress;
    procedure Execute; override;
    procedure Show;
  public
    LookFor, Output: String;
    Progresses: array [1..5] of TProgressBar;
  end;
```

This thread class looks for the characters of the `LookFor` string (there must be four characters for the program to work correctly), using four `TFindThread` objects:

```
procedure TMultiFind.Execute;
var
  Finders: array [1..4] of TFindThread;
  I: Integer;
begin
  // set up the four threads
  for I := 1 to 4 do
  begin
    Finders[I] := TFindThread.Create (True);
    Finders[I].LookFor := LookFor[I];
    Finders[I].Progress := Progresses [I+1];
    Finders[I].Resume;
  end;

  // wait for the threads to end...
  for I := 1 to 4 do
  begin
    Finders[I].WaitFor;
    Progr := I;
    Synchronize (UpdateProgress);
  end;

  // show the result
  Output := 'Found: ';
  for I := 1 to 4 do
    Output := Output + Format ('%d %s, ',
      [Finders[I].Found, LookFor[I]]);
  Synchronize (Show);

  // delete threads
  for I := 1 to 4 do
    Finders[I].Free;
end;
```

Notice in particular the `for` loop with the `WaitFor` call. At the end, the `Execute` method shows the result in a synchronized method, `Show`. The program I've written to test these threads is quite simple. As you saw in Figure 17.4, it has a memo where you can load a file and an edit box containing four characters. The number of characters is checked when the user exits the edit box:

```
procedure TForm1.Edit1Exit(Sender: TObject);
begin
  if Length (Edit1.Text) <> 4 then
  begin
    Edit1.SetFocus;
    ShowMessage ('The edit box requires four characters');
  end;
end;
```

The Start button starts the thread, which in turn immediately spawns the secondary threads:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  if Assigned (MainThread) then
    MainThread.Free;
  MainThread := TMultiFind.Create (True);
  MainThread.Progresses [1] := ProgressBar1;
  MainThread.Progresses [2] := ProgressBar2;
  ...
  MainThread.Progresses [1].Max := 4;
  for I := 2 to 5 do
    MainThread.Progresses[I].Max := Memo1.Lines.Count;
  for I := 1 to 5 do
    MainThread.Progresses[I].Position := 0;
  MainThread.LookFor := Edit1.Text;
  MainThread.Resume;
end;
```

Notice that we cannot delete the thread at the end of the method, because we cannot call `WaitFor` on it without creating a deadlock. Take care when writing multi-threaded applications, because such a deadlock can freeze the whole system, leaving you with no option but to press Ctrl+Alt+Del on Windows 95/98.

At the same time, to keep the operating system stable, we must remember to delete the thread, either before creating a second one (as at the beginning of the code above) or when the program terminates. This is also the reason we need to declare the thread object as a private field of the form and not as a local variable of the method starting it.

Windows Synchronization Techniques

The Win32 API functions offer many further synchronization techniques[368]:

- *Critical sections* are portions of source code that cannot be executed by two threads at the same time. By using a critical section, you can serialize the execution of specific portions of the source code. Critical sections can be used only within a single process, a single application.

368 Most of the core platform synchronization objects are now available as part of the Delphi RTL, along with a few additional one, like a Multi-Read-Single-Write-Syncronizer, and many other related features. Threading is an area that saw many changes since Delphi 5.

- *Mutexes* are global objects you can use to serialize access to a resource. You first set a mutex, then access the resource, and finally release the mutex, as we saw in the OneCopy example. While the mutex is set, if another thread (or process) tries to set the same mutex, it is stopped until the mutex is released by the previous thread (or process). A mutex can be shared by different applications.
- *Semaphores* are similar to mutexes but they are counted: you could allow, for example, three and no more than three accesses to a given resource at the same time. A mutex is equivalent to a semaphore with a maximum count of 1. We'll see an example of the use of the semaphore in the section "Threaded Database Access," later in this chapter.
- *Events* can be used as a mean of synchronizing a thread with system events, such as user file operations. The `WaitFor` method of the Delphi `TThread` class uses an event. Events can also be used to *awaken* several threads at the same time.

Building an Example

To demonstrate all of these different techniques I've built the ThSynch example. Suppose we have two threads operating on a string, both of them using its value in some way and then updating the string as a result of the operation. Suppose also that the current value of the string is shared by the two threads. In the example the string initially contains 20 *A* characters, then is updated to contain 20 *B* characters, and so on. Each thread simply computes the next value of the string and then sends it to its own list box.

note In real applications, of course, threads should generally avoid using global variables. Delphi helps in this direction by wrapping threads in classes and by allowing you to define thread variables, with the `threadvar` keyword. The ThSynch program was built specifically to demonstrate synchronization problems; it should not be considered a good example for sharing data among threads!

The ThSynch example presents a main form with four buttons, each of which displays a secondary form demonstrating one of the synchronization techniques. Each secondary form consists of two list boxes, where the text is displayed in a nonproportional Courier font, and a button to start the related thread. So we end up with four different versions of basically the same form and the same thread class. In each of these forms the Start button simply creates two instances of a thread, associating a list box with the `LBox` public field of each:

```
procedure TForm2.BtnStartClick(Sender: TObject);
begin
```

```
  ListBox1.Clear;
  ListBox2.Clear;
  Th1 := TListThread.Create (True);
  Th2 := TListThread.Create (True);
  Th1.FreeOnTerminate := True;
  Th2.FreeOnTerminate := True;
  Th1.LBox := Listbox1;
  Th2.LBox := Listbox2;
  Th1.Resume;
  Th2.Resume;
end;
```

Notice that the `FreeOnTerminate` property is set to `True`, so that the thread object will automatically free itself when its execution is completed. The thread classes are declared in each of the units defining the form, to avoid having too many files. This same unit contains the string variable used by the two thread objects:

```
var
  Letters: string = 'AAAAAAAAAAAAAAAAAAAA';
```

This is far from elegant, but in this program we are looking for trouble on purpose, so forget the coding style.

The Plain Thread

Here is the thread class and its `Execute` method in a first simple version:

```
type
  TListThread = class (TThread)
  private
    Str: String;
  protected
    procedure AddToList;
    procedure Execute; override;
  public
    LBox: TListBox;
  end;

procedure TListThread.Execute;
var
  I, J, K: Integer;
begin
  for I := 0 to 50 do
  begin
    for J := 1 to 20 do
      for K := 1 to 2601 do // useless repetition...
        if Letters [J] <> 'Z' then
          Letters [J] := Succ (Letters [J])
        else
          Letters [J] := 'A';
```

```
    Str := Letters;
    Synchronize (AddToList);
  end;
end;
```

The `AddToList` method simply adds the `Str` string to the list box connected with the thread. I've made each computation artificially long, by increasing each letter `2601` times instead of once: The effect is the same, but there are more chances of having a conflict between the two threads. You can see this effect in Figure 25.5. Even better, you can press the Start button two or three times in a row, starting several threads at once, and increasing the chance of errors.

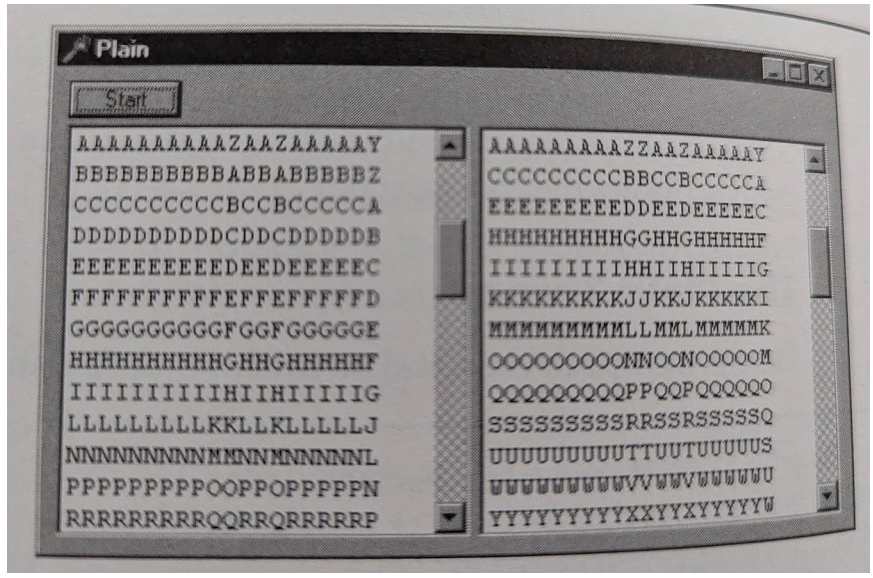


Figure 17.5: The first secondary form of the ThSynch example shows some errors in the values of two list boxes. Image based on a picture of the original printed book.

Using Critical Sections

If you want to serialize the operations of the two threads, and have more control over what the threads do, you can use one of the Windows synchronization techniques I've discussed before. In this second version I'll use critical sections. To do this you should add the declaration of another global variable (or a form class field) for the critical section:

```
var
```

```
  Critical1: TRTLCriticalSection;
```

This variable is initialized when the form is created, and it is destroyed at the end, with two API calls:

```
procedure TForm3.FormCreate(Sender: TObject);
begin
  InitializeCriticalSection (Critical1);
end;

procedure TForm3.FormDestroy(Sender: TObject);
begin
  DeleteCriticalSection (Critical1);
end;
```

note When using critical sections, always make sure the threads that depend on a critical section are terminated before you delete that synchronization object.

You can use critical sections to serialize specific portions of the code, such as the code that updates each letter of the string. Here is how I've updated the code of the `Execute` method of the thread object:

```
procedure TListThread.Execute;
var
  I, J, K: Integer;
begin
  for I := 0 to 50 do
  begin
    EnterCriticalSection (Critical1);
    try
      for J := 1 to 20 do
        ... // same code as above
      Str := Letters;
    finally
      LeaveCriticalSection (Critical1);
    end;
    Synchronize (AddToList);
  end;
end;
```

The effect of this code is that while one thread is computing a new string the other thread will wait before doing the same, so all the output strings will always contain 20 copies of the same character.

Using a Mutex

Now we can write the same code, but using a mutex instead of a critical section. The effect will be the same, but I'd like to show you this technique as well. We simply

need to declare the `hMutex` variable of type `THandle` and then replace the four API calls of the previous example with the following four:

```
// in FormCreate
hMutex := CreateMutex (nil, false, nil);
// in FormDestroy
CloseHandle (hMutex);
// in the for loop
WaitForSingleObject (hMutex, INFINITE);
...
ReleaseMutex (hMutex);
```

Using a TCriticalSection VCL Object

TheEnterprise Edition of Delphi offers one further alternative. The SyncObjs unit defines VCL classes for some synchronization objects: events and critical sections. So we can build a fourth version of the example, very similar to the second one based on a critical section and Windows API calls. The difference is that now we declare the object as:

```
var
  Critical1: TCriticalSection;
```

and use it as follows:

```
// in FormCreate
Critical1 := TCriticalSection.Create;
// in FormDestroy
Critical1.Free;
// in the for loop
Critical1.Enter;
...
Critical1.Leave;
```

There is very little difference, but the code is a little easier to write. The SyncObjs unit declares the `TCriticalSection` class along with the `THandleObject` class, the `TEvent` class and the `TSingleEvent` class.

To summarize the ideas described with this long example, a mutex, a critical section, or a semaphore is required whenever two threads access a shared resource or data. Otherwise, the system may move control from thread to thread before any of them has completed an intermediate operation. In this intermediate state, the data might be invalid, but the other threads will use it anyway. You can access these synchronization objects using simple Windows API calls, or even simpler VCL objects.

Threaded Database Access

At times it is handy to perform two different operations on a database at once, starting a separate request (with a table or query component) before the first one is completed. In order to do that, you need to use threads. If you have two queries, for example, you might want to execute the second before the first has returned its result, to speed up operations on your (possibly) fast server. In other cases, you might want to execute a complex query or scan a large table within a thread to avoid interrupting user input. In fact, when an event starts a query in the main thread or operates on a dataset, the user cannot operate on the program until the operation is completed. Using a secondary thread for computing anything that's not immediately necessary will make the application more responsive.

The BDE supports multiple simultaneous requests from a single application connection, through the use of multiple sessions. In other words we need to use a `TSession` object to make multiple connections with the BDE from the same program (actually one connection from the main program and one from each secondary thread)[369]. The ThreadDB example demonstrates how you can use a thread and a separate `Session` object to perform background processing on a database table you're currently viewing. The program uses a data module (visible in Figure 17.6), which hosts three database-access components: a database, a session, and a query. Here is the relevant code:

```
object Session1: TSession
  Active = True
  AutoSessionName = True
end
object Database1: TDatabase
  AliasName = 'DBDEMOS'
  Connected = True
  DatabaseName = 'mydb'
  SessionName = 'Session1_1'
end
object Query1: TQuery
  DatabaseName = 'mydb'
  SessionName = 'Session1_2'
  SQL.Strings = (
    'select count (*) '
    'from orders'
    'where CustNo = :Cust;')
  ParamData = <
    item
```

369 By contrast, FireDAC requires one connection for each thread, but it includes a powerful connection pooling architecture to avoid creating a new connection for each thread.

```
      DataType = ftInteger
      Name = 'Cust'
      ParamType = ptUnknown
    end>
  object Query1COUNT: TIntegerField
    FieldName = 'COUNT(*)'
  end
end
```

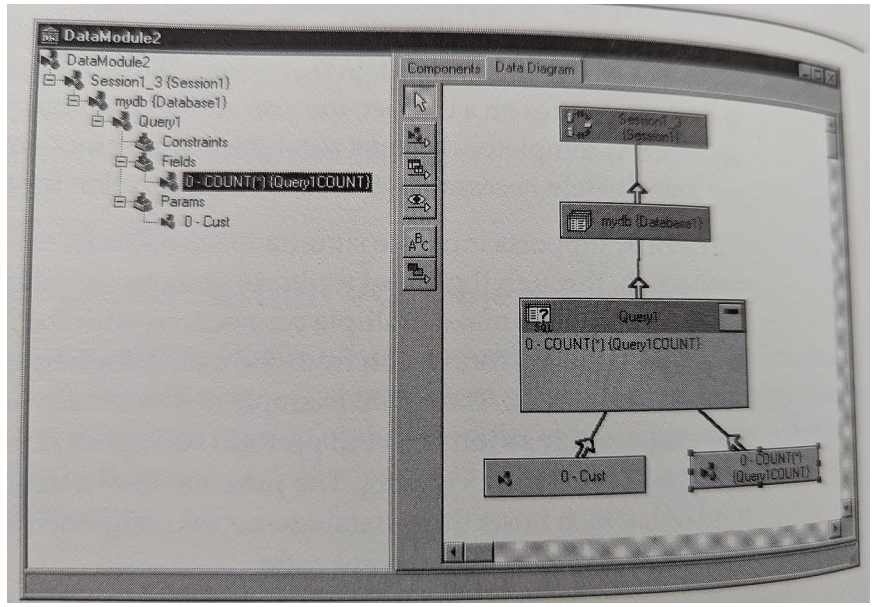


Figure 17.6: The tree and data diagram views of the data module of the ThreadDB example. Image based on a picture of the original printed book.

What is the aim of the ThreadDB application? The main form of this example shows a list of customers, and the query in the data module can be used to compute the number of orders made by the current customer. Whenever the current record in the customer table changes, you might want to recalculate its number of orders. But if you simply execute this query every time the record within the main table changes, the program will slow down. A user pressing the down-arrow key to move quickly through the records will have to wait until the query is computed and its result is displayed before reaching the next record.

As an alternative, the ThreadDB program executes the query in a background thread, letting the user scroll down to a new record before the query has returned its value. By rapidly scrolling the current record, you'll be able to start multiple queries at the same time, using multiple instances of the data module. The main program,

in fact, creates a thread every time the current record of the Customers table changes:

```
procedure TForm1.Table1AfterScroll(DataSet: TDataSet);
var
  Th1: TDatabaseThread;
begin
  // create and start a new thread
  Th1 := TDatabaseThread.Create (True);
  Th1.Priority := tpLower;
  Th1.FreeOnTerminate := True;
  Th1.CustNo := Table1CustNo.AsInteger;
  Th1.Resume;
end;
```

Notice that the thread has low priority (to maximize the responsiveness of the program to user input) and receives the ID of the customer for the current record. As soon as the thread starts, it creates a new data module, sets the customer-number parameter of the query, and opens it:

```
procedure TDatabaseThread.Execute;
begin
  with TDataModule2.Create (nil) do
  begin
    try
      Query1.ParamByName('Cust').AsInteger := CustNo;
      Query1.Open;
      NewCaption := 'Number of Orders ' +
        Query1Count.AsString;
    finally
      Synchronize (UpdateCaption);
      Query1.Close;
      Free; // the data module
    end;
  end;
end;
```

This program works acceptably; it allows the user to change the current record before the background thread has completed executing the query. However, if you simply keep the down arrow key pressed for a while, it will start more simultaneous threads than Windows and the BDE can handle. Of course, it is unreasonable to start an unlimited number of queries, an operation that might adversely affect the performance of any database server. It is equally unreasonable to start an unlimited number of threads, as the Win32 kernel starts to bog down with as few as 15 or 16 threads in a process.

For this reason we can let a few threads start and let the query simply fail to execute the others (or keep them in a pending state) We can easily implement this idea by using a semaphore.

To do that, I've declared an `hSemaphore` variable of type `THandle` in the unit of the thread class, and added this code in the initialization section of the unit to create a semaphore with a limit of 10:

```
initialization
  hSemaphore := CreateSemaphore (
    nil, 10, 10, 'ThDB_MD_Semaphore');
```

The main program uses this semaphore before starting the thread, with a call to the `WaitForSingleObject` API. In this case the function decreases the semaphore counter if it is more than zero. If the semaphore counter is zero, it waits for another thread to release the semaphore before acquiring it. Waiting in the main program can lead to a deadlock since the threads will need to synchronize with it (as discussed in the FindTh example). For this reason, if a thread is not available the program simply skips the user request:

```
  // procedure TForm1.Table1AfterScroll
  if WaitForSingleobject (hSemaphore, 0) = Wait_Object_0 then
  begin
    // create and start a new thread
    Th1 := TDatabaseThread.Create (True);
    // continue as before
```

Of course, the thread must call `ReleaseSemaphore` before terminating. Here is the complete code, which also logs the current status of the threads in a list box of the main form:

```
procedure TDatabaseThread.Execute;
begin
  // log
  Inc (thcount);
  LogText := Format ('Thread %d started (%d active)',
    [CustNo, thcount]);
  Synchronize (AddToLog);

  with TDataModule2.Create (nil) do
  begin
    try
      Query1.ParamByName('Cust').AsInteger := CustNo;
      Query1.Open;
      NewCaption := 'Number of Orders ' +
        Query1Count.AsString;
    finally
      Synchronize (UpdateCaption);
      Query1.Close;
      Free; // the data module
      // log
      Dec (thcount);
      LogText := Format ('Thread %d completed (%d active)',
        [CustNo, thcount]);
```

```
      Synchronize (AddToLog);
    end;
  end;

  // thread is done, release semaphore
  ReleaseSemaphore (hSemaphore, 1, nil);
end;
```

You can now start only 10 threads at a time, while the other user requests will be dismissed. The program doesn't keep these requests in a list, but simply forgets them. In any case the user has probably already moved to another record. You can see an example of the output of this program in Figure 17.7.

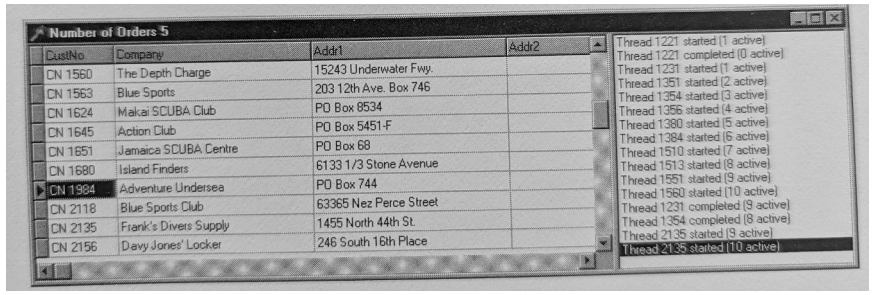


Figure 17.7: In the ThDB example, multiple background threads update the form's caption (see the log on the side). Image based on a picture of the original printed book.

What's Next?

In this chapter, we've discussed multitasking, multithreading, and synchronization issues between threads and separate processes. These examples have demonstrated that you can use Delphi to delve into the intricacies of Windows programming, while using very low-level features of the system. To learn more, you can refer to Windows API programming books and then apply the information to Delphi programming.

The next chapter moves into another technique you need to master to become an expert Delphi programmer: the use of the debugger and debugging techniques in general. The chapter will give you more insight into the inner workings of a Delphi application, and also into the Windows message flow.

Chapter 18: Debugging Delphi Programs

Once you compile a program in Delphi and run it, you may think you're finished, but not all of your problems may be solved. Programs can have run-time errors, or they may not work as you planned. When this happens, you will need to discover what has gone wrong and how to correct it. Fortunately, many options and tools are available for exploring the behavior of a Windows application.

Delphi includes an integrated debugger and several other tools to let you monitor the result of a compilation process in different ways. This chapter provides an overview of all these topics, demonstrating the key ideas with simple examples. The first part of the chapter covers Delphi's integrated debugger and various features Delphi provides for run-time debugging. Then, I'll describe some other debugging techniques and discuss how you can monitor the flow of messages in your application.

The final section describes how you can examine the status of the memory used by a program.

Using the Integrated Debugger

As I've mentioned before, when you run a program from within the Delphi environment, the internal debugger actually executes the program. (You can change this behavior by disabling the Integrated Debugger option in the Debugger Options dialog box.) Most of the Run commands relate to the debugger. Some of these commands are available also in the Debug submenu of the Editor's shortcut menu.

When a program is running in the debugger, clicking on the Pause button on the SpeedBar suspends execution. Once a program is suspended, clicking on the Step Over button executes the program step by step. You can also run a program step by step from the beginning, by pressing the Step Over button while in design mode. Consider, however, that Windows applications are message-driven, so there is really no way to execute an application step by step all the way, as you can do with a DOS application. For this reason, the most common way to debug a Delphi application (or any other Windows application) is to set some *breakpoints* in the portions of the code you want to debug.

When a program has been stopped in the debugger, you can continue executing it by using the Run command. This will stop the program on the next breakpoint. As an alternative, you can monitor the execution more closely by tracing the program. You can use the Step Over command (the F8 key) to execute the next line of code, or the Trace Into command (F7) to delve into the source code of a function or method (that is, to execute the code of the subroutines step by step and to execute the code of the subroutines called by the subroutines, and so on). Delphi highlights the line that is about to be executed with a different color and a small arrow-shaped icon, so that you can see what your program is doing.

A third option, Trace to Next Source Line (Shift + F7), will move control to the next line of the source code of your program being executed, regardless of the control flow. This source code line might be the following line of the source code (as in a Step Over command), a line inside a function called by your code (as in a Trace Into command), or a line of code inside an event handler or a callback function of the program activated by the system. If you want to monitor the effect of the execution of a given line of code, you can also move to that position and call Run to Cursor (F4). The program will run until it reaches that line, so this is similar to setting a

temporary breakpoint. Finally, the new Delphi 5 command Run until Return (Shift + F8) executes the method or function until it terminates. This is handy to use when you accidentally trace into a function you are not interested in debugging.

Debugging Libraries (and ActiveX Controls)

You can also use the integrated debugger to debug a DLL or any other kind of library (such as an ActiveX control). Simply open the library's Delphi project, choose the Run ➢ Parameters menu command, and enter the name of the Host Application. (This option is available only if the current project's target is a library.) Now when you press the Run button (or the F9 key), Delphi will start the main executable file, which will then load the library. If you set a breakpoint within the library source code, execution will stop in the library code, as expected.

Similarly, you can use this capability to debug an ActiveForm. Simply enter the full path name of the Web browser as the Host application, for example `C:\Program Files\Microsoft Internet\Iexplore.exe`; then enter the full path name of the test HTML file as the Run parameter. To make this work, you should also use the Run ➢ Register ActiveX Server menu command. Once the ActiveX is registered, the Web browser will use that version and not another version available in its OCX cache.

Debug Information

To debug a Delphi program, you must add debug information to your compiled code. Delphi does this by default, but if it has been turned off, you can turn it back on through the Project Options dialog box. As shown in Figure 18.1, the Compiler page includes a Debugging section with four check boxes:

- **Debug Information** adds to each unit a map of executable addresses and corresponding source code line numbers. This increases the size of the DCU file but does not affect the size or speed of the executable program. (The linker doesn't include this information when it builds the EXE file, unless you explicitly request TD32 debug information, which is technically in a different format.)
- **Local Symbols** adds debug information about all the local identifiers, the names and types of symbols in the implementation portion of the unit.
- **Reference Info** adds reference information about the symbols defined in a module to allow the Project Inspector (or Object Browser) to display them. If the

sub-option **Definitions Only** is not checked, the reference information tracks where each symbol is used, enabling the cross-references in the Project Explorer.

- **Assertions** allows you to add assertions, code that will stop your program if a specific test condition fails. Unlike exceptions or other error-detecting code, assertions can be removed from your program automatically; just deselect this option. I'll discuss assertions later in this chapter. After changing this setting, you have to rebuild your project to add or remove the assertion code from your application.
- **Use Debug DCUs** to link the debug version of the DCU files of the VCL in your program. In practice, this option adds the Debug DCU path (specified in General page of the Debugger Options) to the Search path (specified in Directories/Conditionals page of the Project Options dialog box).

The integrated debugger uses this information while debugging a program. Debug information is not attached to the executable file unless you set the Include TD32 Debug Info option in the Linker page of the Project Options dialog box. You should add debug information to your executable file only if you plan to use an external debugger, such as Borland's Turbo Debugger for Windows (TD32). Do not include it if you plan to use only the integrated debugger, and remember to remove it from the executable file that you ship.

Remote Debugging

A feature first introduced in Delphi 4 is remote debugging[370]. This technique allows you to debug a program that is running on a different computer, typically a server. To activate remote debugging, you must first install the remote debugger client on the target machine. Then you should start the remote debugging client, with the command `borrdg.exe -listen`, eventually starting it as a service on Windows NT.

370 The idea of Remote Debugging has been largely expanded in recent versions of Delphi as it's the foundation for debugging on non-Windows platforms, using the "Platform Assistant Server" (or PAServer) for the target platform. You run this engine on the target computer and the Delphi IDE connects to it to debug an app running on the device.

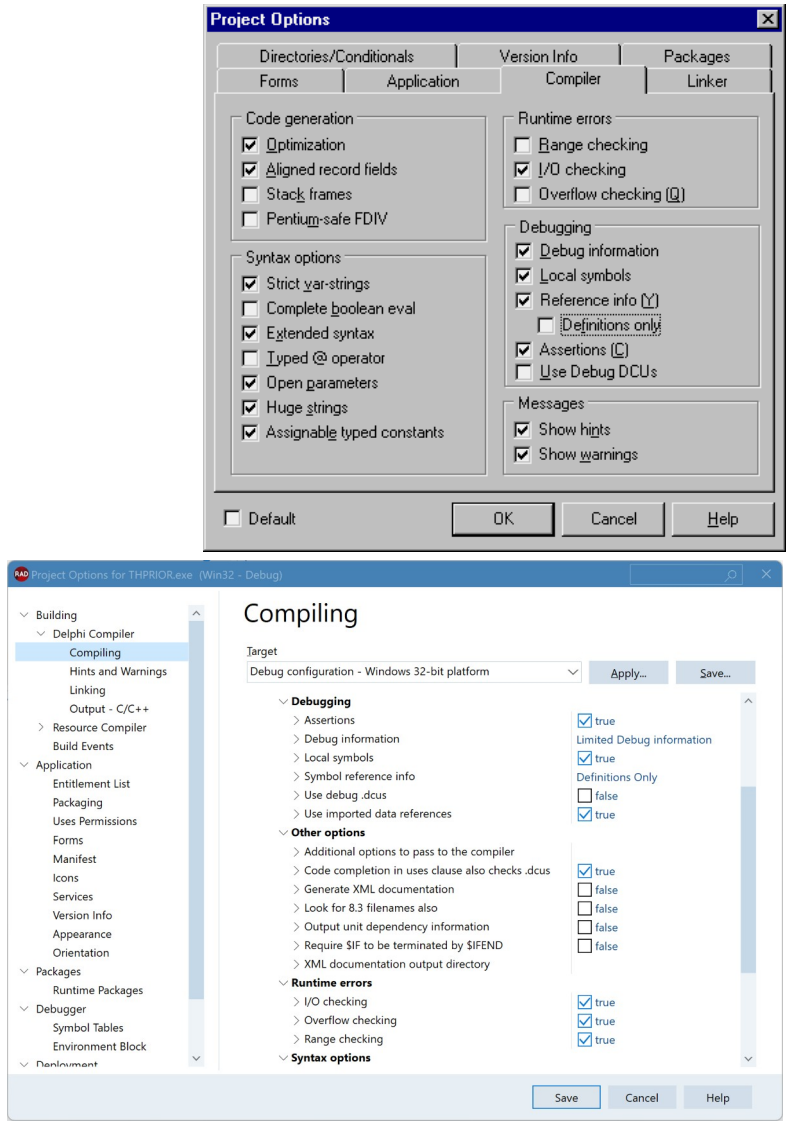


Figure 18.1: Use the Compiler page of the Project Options dialog box to include debug information in a compiled unit. Image from the original edition of the book and Delphi 12.

Now you should compile your program including remote debug symbols in the Linker page of the Project Options dialog box. You can also set the Output directory to the remote machine in the Directories Conditional page of the same dialog, so that you don't have to manually copy the program and the companion RMS file on the remote computer each time you recompile the program. Finally, set the remote

path and project in Run ➢ Parameters, filling the Remote box with the machine name or its IP address.

When everything is properly set up, you'll be able to use the Delphi integrated debugger to debug the program running on the remote computer. You'll be able to set breakpoints and perform all of the standard debugging operations as usual.

Attach to Process

New in Delphi 5 is the Attach to Process feature, available via the Run command. This feature allows you to start debugging a program that is already running in the system. For example, you might want to attach the debugger to a process that has just displayed an exception to understand what has gone wrong.

When you select the Attach to Process command, you'll see a list of running processes. If you choose a Delphi program for which you have the source code, you'll be back to the traditional debugging situation. If you choose another program for which you don't have the source code, you'll only be able to trace its execution at the assembly level, using the CPU window but not the source code editor.

Using Breakpoints

There are several breakpoint types in Delphi:

- **Source breakpoints** and **address breakpoints** are similar, as they halt execution when the processor is about to execute the instructions at a specific address.
- **Data breakpoints** halt execution when the value at a given location changes.
- **Module load breakpoints** halt execution when a specific code module loads.

As the name implies, a breakpoint, when reached, is supposed to stop the program execution. In Delphi 5, breakpoints can do more than just stop—each breakpoint can have any of several actions associated with it. These actions can be the traditional break action, the display of a fixed string or a calculated expression in the message log, or the activation or deactivation of other groups of breakpoints. The Breakpoint List window (see Figure 18.2) shows this extra information, along with the description of the breakpoints of the entire program. The figure is taken from a

simple example, BreakP, I'm using to illustrate some of the features of breakpoints in Delphi.

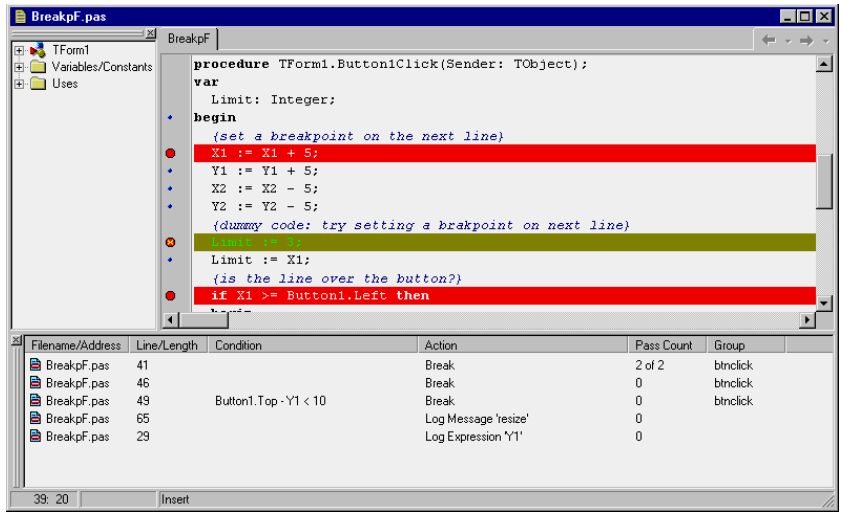


Figure 18.2: The Breakpoint List window, with a conditional breakpoint, docked on the bottom of the editor. Image from the original edition of the book.

Another new feature is that you can assign breakpoints to groups. You can then enable or disable all the breakpoints of a group at once, either using a direct command (in the shortcut menu of the Breakpoint List window) or as the automatic effect of a breakpoint action.

Source Breakpoints

If you want to break upon execution of statements in your source code, you'll use a source breakpoint, which is the most common type of breakpoint. You can create a source breakpoint by clicking in the gutter region of the code editor window, by right-clicking on a specific line in a source file and choosing the Toggle Breakpoint command from the shortcut menu, or by using the Add Source Breakpoint dialog box. (You can display this dialog box by choosing the Run ➢ Add Breakpoint ➢ Source Breakpoint menu command, by choosing Add ➢ Source Breakpoint from the Breakpoint List window's local menu, or by pressing F5.)

When you create a new source breakpoint (using any of these techniques), an icon will appear in the margin to the left of the code, and the source line will display in a different color. However, you can't set a valid breakpoint on just any line of the source code. A source breakpoint is valid only if Delphi generated executable code for that source line. This means you can't specify a breakpoint for a comment, decla-

ration, compiler directive line, or any other statement that's not executable. If you've compiled a program at least once (with debugging information enabled), small dots in the frame to the left of the gutter area indicate source lines where you can place a valid breakpoint. Although you can set a breakpoint at an invalid location, Delphi will alert you when you run the program, and it will mark the invalid breakpoint with a different icon and color.

Also note that because Delphi uses an optimizing compiler, it will not generate any executable code for *unreachable* lines of source code in your program nor for any other lines that have no effect on the program logic. If you create an invalid source breakpoint and then execute the program step by step, the debugger will skip the line, because that code doesn't exist in the optimized version of the compiled program. There is an example of an invalid breakpoint in the BreakpF unit of the BreakP program.

Once you've set a valid source code breakpoint, you can change some of its properties. The Source Breakpoint Properties dialog box is available from the Breakpoint list window by right-clicking on the icon in the editor gutter. In this window (see Figure 18.3), you can set a condition for the breakpoint, indicate its pass count, and assign it to a group. You can also click the Advanced button to display an extended version of the window, offering the breakpoint action features introduced in Delphi 5, which I'll discuss in the next section.

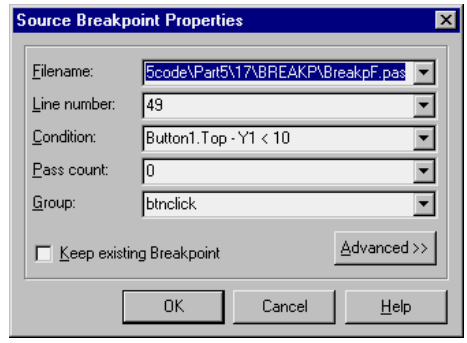


Figure 18.3: The standard portion of the Source Breakpoint Properties dialog box. Image from the original edition of the book.

In Figure 18.3, you can see the definition of a *conditional* breakpoint, used to stop the program only when a given expression is true. For example, we might want to stop the `Button1Click` method's execution of the BreakP example only when the lines (indicated by the Y1 variable) have moved near the button, using this condition:

```
Button1.Top - Y1 < 10
```

The condition is also added to the Breakpoint List window, as you can see in Figure 8.2. Now you can run the program and click the button a number of times. The breakpoint is ignored until the condition is met (when `Button1.Top - Y1` is less than 10). Only after you click several times will the program actually stop in the debugger.

If you know how many times a line of code should be allowed to execute before the debugger should stop, you can set a value for the Pass Count. As soon as the debugger reaches the breakpoint, it will increase the count until the specified number of passes has been reached. The Breakpoint List window will indicate the status (see the first line of Figure 18.2), displaying "2 of 5" if the program has executed the line only twice after you've set a pass count of 5.

The Keep Existing Breakpoint check box of the Breakpoint Properties window is used when you want to duplicate an existing point by moving the source code line it refers to. If the check box is not selected, the existing breakpoint is moved; if it is selected, a new breakpoint is created. This provides a sort of cloning mechanism if you want to create a new breakpoint based on an existing one.

Finally, notice that in Delphi 5 all of the information displayed by the Breakpoint List window is available also in a fly-by hint displayed as you move the mouse over the breakpoint icon in the editor gutter, as shown in Figure 18.4.

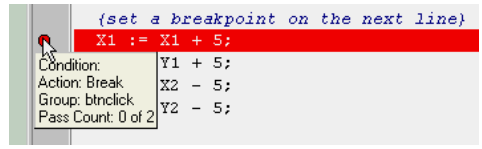


Figure 18.4: The new fly-by hint for breakpoints. Image from the original edition of the book.

note The breakpoints you add to a program are saved in the project desktop file (`*.dsk`) if Desktop Saving is enabled. By saving this information, you'll be able to reopen the project and restart your debugging session. You can also keep your existing breakpoints, at least the complex ones, for future use: simply disable them, and save the project desktop settings instead of removing the breakpoints.

Breakpoint Actions

As I've already mentioned, Delphi 5 makes breakpoints a little more flexible. Besides simply breaking the program execution, they can perform other actions, as

shown in the extended version of the Breakpoint Properties window (accessible by pressing the Advanced button), illustrated in Figure 18.5.

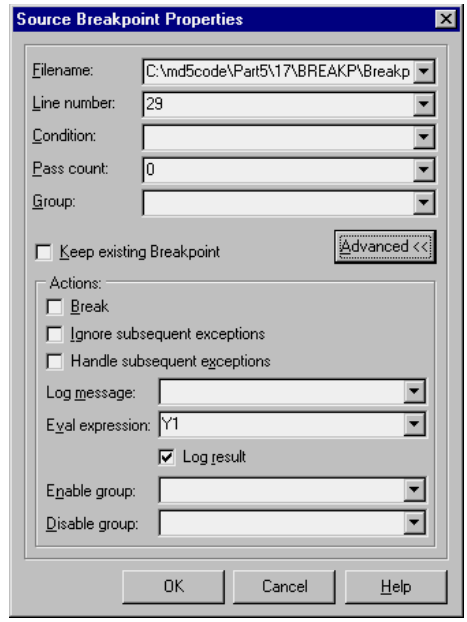


Figure 18.5: The advanced version of the Breakpoint Properties window. Image from the original edition of the book.

If you disable the Break check box, the program won't stop when the debugger reaches that line of code. You can ask for an interruption the following time the breakpoint is reached or simply send a message or the result of an expression to the Event log windows (described later), as shown in the illustration. You can also enable or disable a group of breakpoints only when a conditional breakpoint is met.

You might wonder when logging a message is better than effectively breaking the program. There are some interesting cases, as the BreakP program demonstrates. One of these message log options is used in the handler of the `OnResize` event of the form. If you drag the form border, in fact, this event will fire multiple times in a row, stopping the debugger each time.

This problem is even more obvious with the `OnPaint` event handler. If the editor window and the form overlap, you'll enter an endless series of breakpoints. Each time the form is repainted, the breakpoint stops the program, moving the editor window in front of the form and causing the form to be repainted again, which stops the program on the same breakpoint—over and over again. You could try to position the editor window and the form of your program so that they do not overlap. A more comprehensive solution is to use a conditional breakpoint, set a pass count, or log a

message with the value you are interested in, such as the `Y1` variable in the case depicted in Figure 18.5.

Also Windows focus change messages are very difficult to debug with a breakpoint, because switching to the debugger for the breakpoint will force the debugged program to lose focus. You can't work around this by repositioning windows to not overlap, as for an `OnPaint` event; your choices are to use message logging or remote debugging. Remote debugging is an excellent tool for "nonintrusive" debugging of tricky state problems like painting or focus change messages.

Address Breakpoints

If you don't have the source code for a given function or procedure, you'll want to create an address breakpoint to halt execution at that point. However, without the source code for Delphi to use in calculating the address where you wish to pause, you'll need to use some technique to determine the address of the code in question. You can create an address breakpoint either directly in the CPU view (which we'll discuss later in this chapter) or indirectly (once you have the address) using the Add Address Breakpoint dialog box.

To create an address breakpoint from the CPU view, you'll simply click in the gutter region of the Disassembly Pane, next to the instruction where you want to pause, or right-click on an instruction and select Toggle Breakpoint from the local menu. Once you've created an address breakpoint in the source code editor, you can modify its properties by right-clicking on the breakpoint icon (not on the instruction) and choosing Breakpoint Properties from the local menu.

To create an address breakpoint indirectly, you'll need to determine the address of the instruction where you want to pause execution. If you're going to pause execution of the program, but the source code isn't part of your project (as with standard VCL methods), you need to capture the address of a function or procedure that's already been compiled.

One way to determine the addresses of an object's methods is to use the Debug Inspector for that object at run time. We'll examine the Debug Inspector in more detail later in this chapter. For now, we'll just consider how you can obtain the address of a method for which you don't have the source code. For example, if you want to break when the user clicks on a button, but before entering the button's event handler code, you can locate the `TButton.Click` method. To do this, right-click on the button's declaration in the form's class declaration, and choose Debug ➢ Inspect. In the Debug Inspector window for the button, click on the Methods tab, scroll to the very end of the method list, and locate the `StdCtrls.TButton.Click` method.

Next to the name of the method, you'll see a hexadecimal address in parentheses. Copy this address (including the "$" prefix) and then create a new address breakpoint using that value. When you resume running the program and click the button, the CPU View window will appear, displaying the disassembled instructions for the `TButton.Click` method.

note Unless you have the Standard edition of Delphi, you have VCL source code. You should know that one of its key uses is in debugging your applications. You can include the library source code in your program and use the debugger to trace its execution. Of course, you need to be bold enough and have enough free time to delve into the intricacies of the VCL source code. But when nothing else seems to work, this might be the only solution. To include the library source code, simply add the name of the directory with the VCL source code (by default, `Source\VCL` under your Delphi directory) to the Search Path combo box of the Directories/Conditional page of the Project Options dialog box. An alternative is to link the Debug DCUs, as described earlier in this chapter. Then rebuild the whole program and start debugging. When you reach statements containing method or property calls, you can trace the program and see the VCL code executed line after line. Naturally, you can do any common debugging operation with the VCL code that you can do with your own code.

Data Breakpoints

Data breakpoints are dramatically different from source and address breakpoints in that they monitor a memory address for changes in value. It doesn't matter where the code resides that changes the data at that memory location; the debugger will pause immediately and display the execution point. The display will be either in the source code editor window, if the source for that code is part of the project, or the CPU View window if the source isn't available.

You can set data breakpoints two ways. The first technique involves pausing execution using a source breakpoint at a point in the source code where the identifier you want to monitor is in scope. With the program paused, you can enter the name of the identifier directly into the Address field of the Add Data Breakpoint dialog box, and Delphi will calculate the variable's address for you.

The second way to set a data breakpoint is to first create a watch for the variable, as discussed later in this chapter. Then, you'll pause execution by creating a source breakpoint at a location where the identifier is in scope. When you display the Watch List window, right-click on the watch for this identifier, and choose Break when Changed from the local menu. With either of these methods, any change in the variable's value will pause the program and display the current execution point.

Module Load Breakpoints

If you want to break when a specific code module loads, you'll create a module load breakpoint. You can create a module load breakpoint by choosing Run ➢ Add Breakpoint ➢ Module Load Breakpoint and then selecting the EXE or DLL you wish to monitor. When that module loads, Delphi will suspend the program's execution and highlight the execution point, either in the source editor window or in the CPU View window.

An easier way to achieve the same effect is to open the Modules window, run the program to a source breakpoint that occurs after the loading of all the modules, and then select the modules whose loading you wish break on. You can set a module load breakpoint for a given module by either right-clicking on the module and selecting Break on Load from the local menu or clicking in the gutter region of the module list in the Modules window.

Debugger Views

While you are debugging a program, there are many windows (or views) you can open to monitor the program execution and its status. Most of these windows are quite intuitive, so I'll just introduce them briefly, suggesting a few hints and tips. To activate them, use the Debug submenu of the View menu of the Delphi IDE.

The Call Stack

While you're tracing a program, you can see the sequence of subroutine calls currently on the stack. Call stack information is particularly useful when you have many nested calls or when you use the Debug DCUs of the VCL. This second example is shown in Figure 18.6 for a simple `OnClick` event handler, which is called after many internal VCL calls.

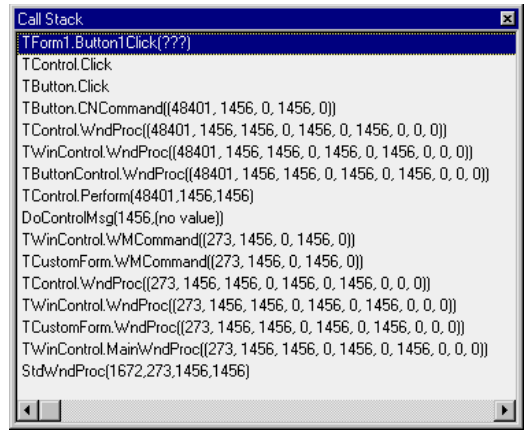


Figure 18.6: The Call stack window when a button is clicked (with Debug DCUs enabled). Image from the original edition of the book.

The Call Stack window shows the names of the methods on the stack and the parameters passed to each function call. The top of the window lists the last function called by your program, followed by the function that called it, and so on. In the figure, you can see that the `Button1Click` event handler of `TForm1` is called by the `Click` method of the `TControl` class, which is called by the same method of the derived class `TButton`, which in turn is activated by the `WndProc` message handling method. There are multiple calls to this `WndProc` method because it is redefined in many VCL classes.

note If you are interested, you can find a complete technical description of the steps from a Windows message to a Delphi event handler in the *Delphi Developer's Handbook* (Sybex, 1998).

Inspecting Values

When a program is stopped in the debugger, you can inspect the value of any identifier (variables, objects, components, properties, and so on) that's accessible from the current execution point (that is, only the identifiers visible in the current scope). There are many ways to accomplish this: using the fly-by debugger hints, using the Evaluate/Modify dialog box, adding a watch to the Watch List, or using the Local Variables window or the Debug Inspector.

The Fly-By Debugger Hints

When Delphi 3 introduced *fly-by evaluation hints,* this feature immediately became one of the most common ways to inspect values at run time. While a program is stopped in the debugger, you can simply move the mouse over a variable, object, property, field, or any other identifier indicating a value, and you'll immediately get a hint window showing the current value of the identifier, as you can see in Figure 18.7.

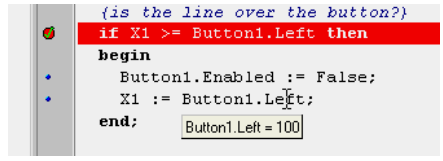


Figure 18.7: One of the more useful debugging features: fly-by evaluation hints. Image from the original edition of the book.

For simple variables, such as `X1` or `Y1` in the BreakP example, and for object properties (as in the figure), fly-by hints simply show the corresponding value, which is easy to understand. But what is the value of an object, such as `Form1` or `Button1`? Past versions of Delphi used a minimalist approach, displaying only the list of its private fields. Delphi 5 shows the entire set of properties of the object, as you can see in Figure 18.8. This is an improvement, but I think that using a Debug Inspector (see following sections) makes the status of the object more readable[371].

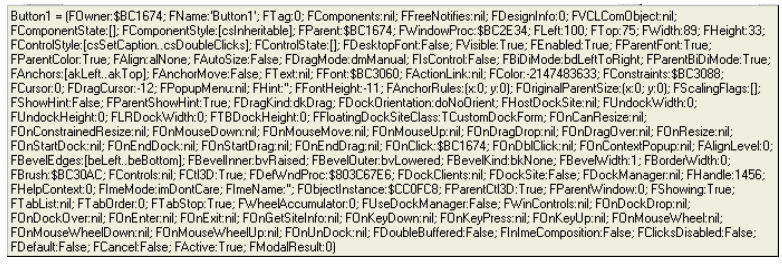


Figure 18.8: The fly-by evaluation hint for an object in Delphi 5. Image from the original edition of the book.

Remember that you can see the value of a variable when the program is stopped in the debugger but not when it is running. Additionally, you can inspect only the vari-

371 Debugger hints offer now way more structure and make it easier to locate the specific information. This technology improved a bit over the years.

ables that are visible in the current scope, because they must exist in order for you to see them!

The Evaluate/Modify Window

The Evaluate/Modify dialog box can still be used to see the value of a complex expression and to modify the value of a variable or property. The easiest way to open this dialog box is to select the variable in the code editor and then choose Evaluate/Modify from the editor's shortcut menu (or press Ctrl+F7). Long selections are not automatically used, so to select a long expression, it's best to copy it from the editor and paste it into the dialog box. In Delphi 5, you can now also drag a variable or an entire expression from the source code editor to the Evaluate/Modify dialog box (see Figure 18.9).

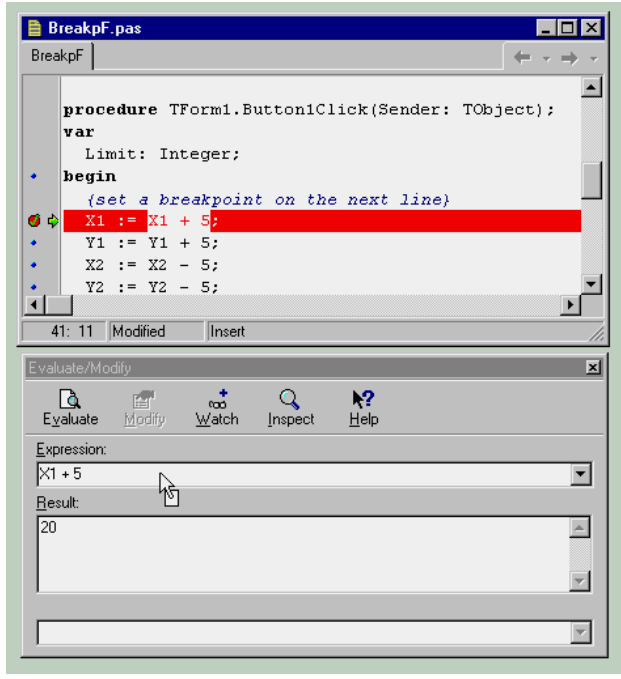


Figure 18.9: The Evaluate/Modify dialog box can be used to inspect—and change—the value of a variable. You can now also drag an expression from the editor into this window. Image from the original edition of the book.

The Watch List Window

When you want to test the value of a group of variables over and over, using the fly-by hints can become a little tedious. As an alternative, you can set some *watches*

(entries in a list of variables you're interested in monitoring) for variables, properties, or components. For example, you might set a watch for each of the values used in the BreakP example's `Button1Click` method, which is called each time the user clicks on the button. I've added a number of watches to see the values of the most relevant variables and properties involved in this method, as you can see in Figure 18.10. As mentioned earlier, you can also use this window as a starting point to set data breakpoints.

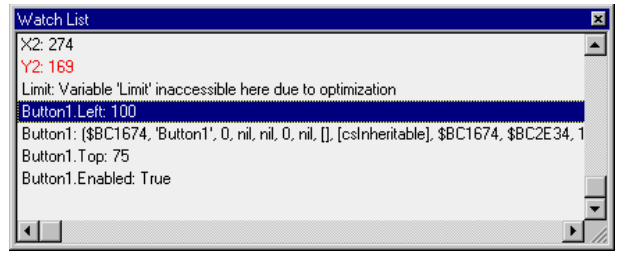


Figure 18.10: Using the Watch List window. Image from the original edition of the book.

You can set watches by using the Add Watch at Cursor command on the editor's local menu (or just press Ctrl+F5), but the faster technique in Delphi 5 is to drag a variable or expression from the source code editor to the Watch List window. When you add a watch, you'll need to choose the proper output format, and you may need to enter some text for more complex expressions. This is accomplished by double-clicking on the watch in the list, which opens the Watch Properties dialog box, or with the equivalent Edit Watch command of the shortcut menu.

note Keep in mind that this window, like many other debugging windows, can be kept in view by docking it to the editor or by toggling its Stay on Top option.

The Local Variables Window

Another useful feature of Delphi is the Local Variables window. This window automatically displays the name and value of any local variables in the current procedure or function when you're paused at a breakpoint. For methods, you'll also see the implicit `Self` variable's private data. The Local Variables window is very similar to the Watch List window, but you don't have to set up its content, because it is automatically updated as you trace into a new function or method or stop on a breakpoint in a different one.

For any object references that appear in the Local Variables window (or the Watch List window) as well as displaying its detailed value on a single line, you can also

open Debug Inspector windows. To do so, either double-click on the variable in the Local Variables window or use the Inspect command of the shortcut menu in the Watch List window.

The Debug Inspector

Debug Inspector windows allow you to view data, methods, and properties of an object or component at run time, with a user interface similar to the design-time Object Inspector (as you can see in Figure 18.11). The main difference is that a Debug Inspector doesn't show just the published properties but the entire list of properties, methods, and local data fields of the object, including private ones. As already described, to activate a similar Inspector at debug time, you can select a visible identifier in the editor, activate the local menu, and select Debug ➢ Inspect.

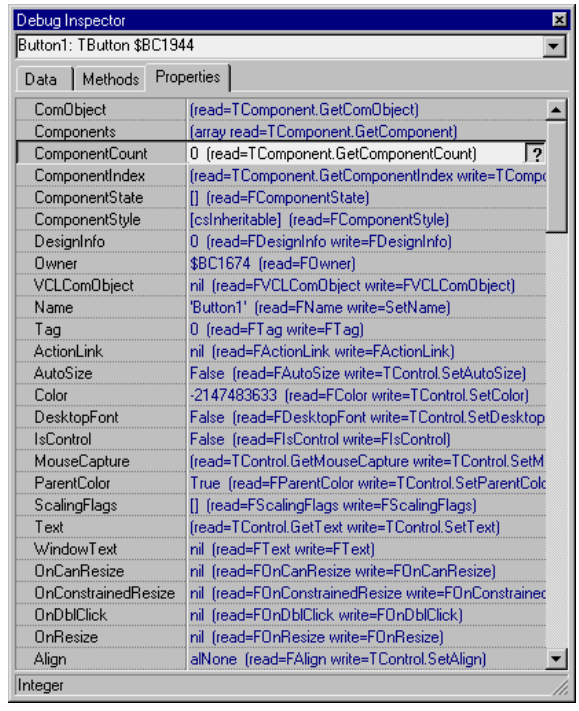


Figure 18.11: A Debug Inspector window showing the properties of a button. Image from the original edition of the book.

note The Debug Inspector is similar to the Object Debugger component I've written for the *Delphi Developer's Handbook* and available on my Web site (`www.marcocantu.com`). This component allows you to get a full list of the values of the published properties of a component at run time.

You'll see that the Debug Inspector shows the definition of the properties and not their values. To activate this, you have to select the property and click the small question mark button on the right. This computes the value, if available. You can also modify the object's data or property value.

If you are inspecting a `Sender` parameter, you can also cast the entire object to a different type, so that you'll be able to see its specific properties (without a cast you get information only about the generic `TObject` structure). When you are working on a component, you can drill down and inspect a sub-object, such as a font. You can use multiple Debug Inspector windows or use a single one and move back to the items you've inspected in the past. A drop-down list box at the top of the inspector maintains a history list of expressions for the current Debug Inspector.

Exploring Modules and Threads

Another important area of the debugger is related to exploring the overall structure of an application. The Modules window (see Figure 18.12) displays all the executable modules for the current application (usually the main executable file as well as the DLLs it uses). The right pane shows a list of the different procedure or function entry points of each module. The bottom pane displays a list of Pascal units that the module contains, if that information is known.

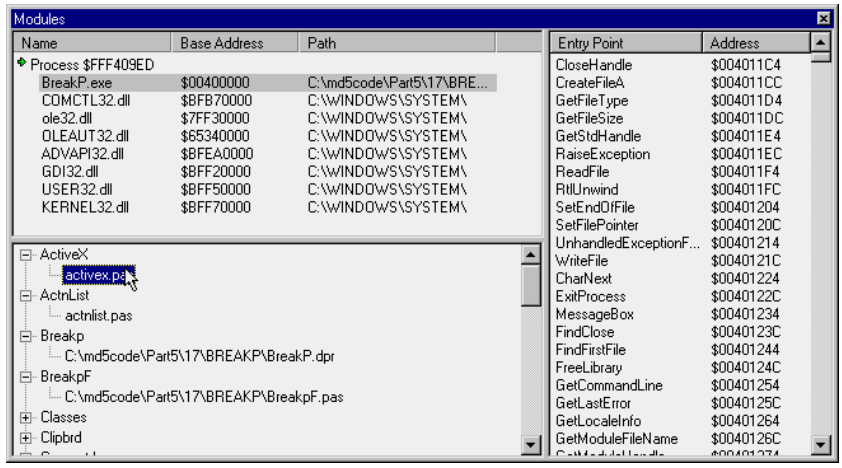


Figure 18.12: The Modules window. Image from the original edition of the book.

The Module window can be used to check the system DLLs and Delphi run-time packages required by a program, and it lets you explore how they relate in terms of

exported and imported functions. While tools as TDump.exe or the Executable QuickView included in Windows can perform a static analysis of the EXE file to determine the DLLs it requires to run, the Modules window shows the current libraries in use, even if it's one you've dynamically loaded (see Chapter 14 for examples of dynamically loaded libraries). Remember that you can use the Module window to set a module breakpoint, one that will fire when the module is loaded by the system.

Another related window is the Thread Status window, which shows details about each thread of a multithreaded program. Figure 18.13 shows an example of this window. In this window, you can change the active thread as well as operate on the main process. Notice that this ability is particularly interesting when you are debugging two processes at the same time, a feature available only on the Windows NT platform.

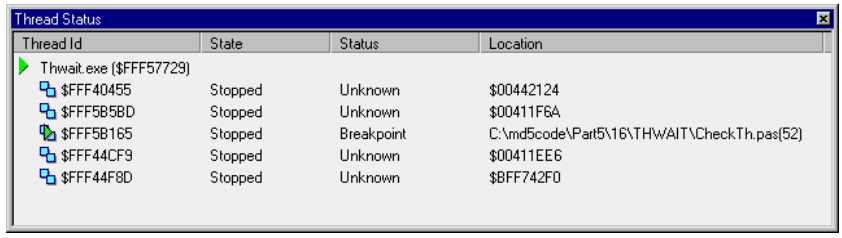


Figure 18.13: The Thread Status window. Image from the original edition of the book.

The Event Log

Another handy debugger window, first introduced in Delphi 4, is the Event Log, which lets you monitor a number of system events: modules loading, breakpoints and their log messages, Windows messages, and custom messages sent by the application. Tracking program flow with a log can be invaluable. As an example, consider debugging an application where timing is important, so that it would be affected by stopping the program in a debugger. To avoid stopping the program, you can log debug information for later review.

To generate a direct log, you can embed calls to the `OutputDebugString` Windows API procedure in your program. This procedure accepts a string pointer and sends that string to a valid debug logging device. The Event Log window will capture and display the text you pass to the `OutputDebugString` procedure. Figure 18.14 shows an example of a debugging session made with the Event Log window. (By the way, calls to the `OutputDebugString` procedure appear in the log as text with the ODS

prefix.) In the same figure you can also see the effect of some breakpoints logging the value of a variable, `J`. This output is taken from the OdsDemo example and was generated by the following code:

```
OutputDebugString (
  PChar ('Button2Click - I=' + IntToStr (I)));
```

note Although the effects of calling `OutputDebugString` and logging a message as a result of a breakpoint might seem similar, there is a big difference. The breakpoints are external to the program (they're part of the debugger support); while the direct strings must be added to the source code of the program, potentially changing it and introducing bugs. You might also want to remove this code for the final build, although you can use conditional compilation for this, as described later in the section "Using Conditional Compilation for Debug and Release Versions."

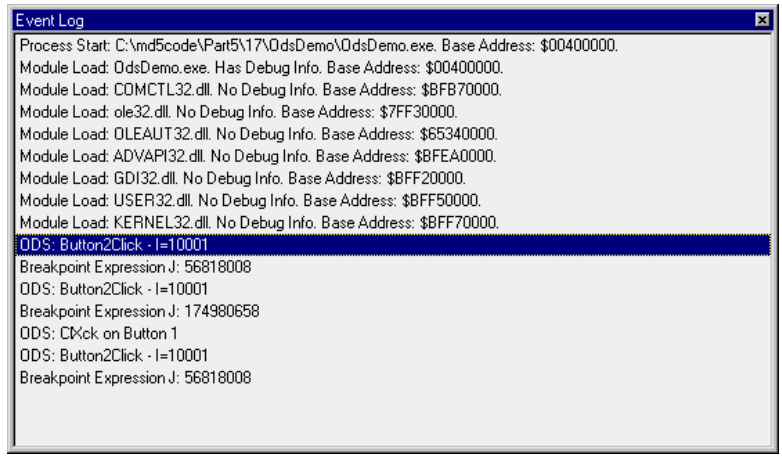


Figure 18.14: The output of calls to the `OutputDebugString` debugging procedure and the information logged by a breakpoint in the Event Log window. Image from the original edition of the book.

To capture the image in Figure 18.14, I've disabled the default breakpoint logging, which indicates when the program has stopped or restarted for a breakpoint. (As noted earlier, you can also specify logging as one of the breakpoint actions; disabling default breakpoint logging does not affect this type of logging. I've also disabled the process information (a new Delphi 5 option "Display Process Info with Event"), which is not particularly useful when debugging a single process.

You can configure the Event Log with this and other options in the Debugger Event Log Properties dialog box, shown in Figure 18.15. In addition to logging the text from calls to the `OutputDebugString` procedure, the breakpoint data, and the process information, the Event Log window can also capture all the Windows mes-

sages reaching the application. This provides an alternative to the WinSight program described later in this chapter.

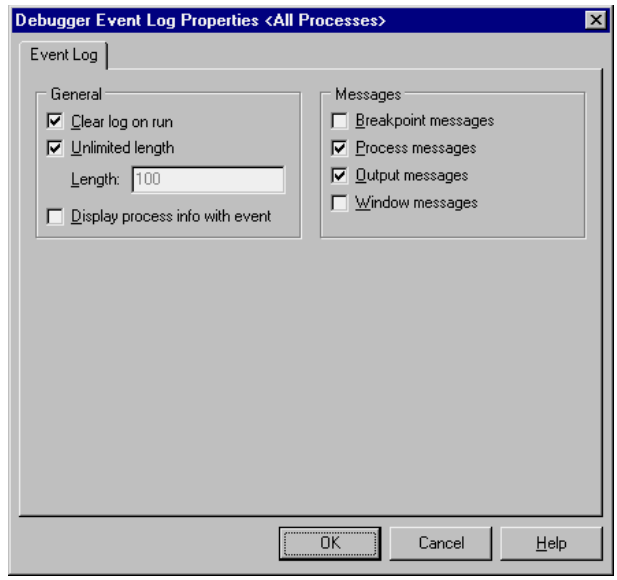


Figure 18.15: You determine which events you wish to track using the Debugger Event Log Properties dialog box. Image from the original edition of the book.

Down to the Metal: CPU and FPU views

There are two more debugger windows that are not for the faint-hearted. The CPU view and the new Delphi 5 FPU view show you what's going on inside the Central Processing Unit and the Floating Point Unit of the computer.

Using the CPU view during debugging lets you see a lot of system information: the values of the CPU registers, including special flags and a disassembly of the program (eventually with the Pascal source code included in comments). Similarly, the FPU view opens up more registers and status information regarding the floating point and MMX support of the newer Pentium class chips.

If you have a basic knowledge of assembly language, you can use this information to understand in detail how the Delphi compiler has translated your source code into the executable and see the effect of the Delphi optimizing compiler on the final code. Although it seems bare at first, the CPU window provides programmers with a lot of power. By right-clicking and using the shortcut menus, you can even change the value of CPU registers directly!

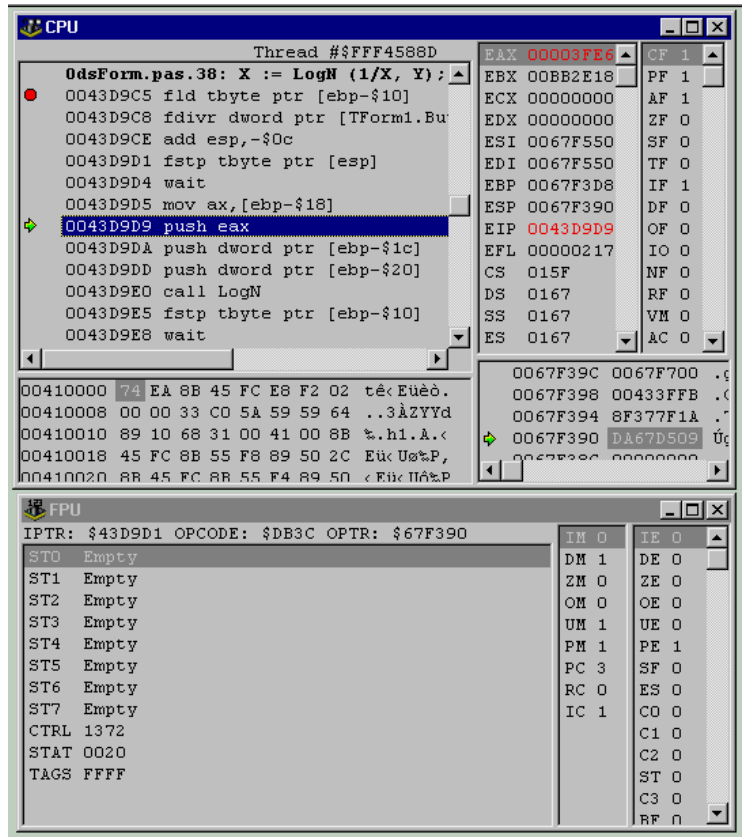


Figure 18.16: An example of the CPU and FPU views. Image from the original edition of the book.

Other Debugging Techniques

One common use of breakpoints is to find out when a program has reached a certain stage, but there are other ways to get this information. A common technique is to show simple messages (using the `ShowMessage` procedure) on specific lines, just for debugging purposes. There are many other manual techniques, such as changing the text of a label in a special form, writing to a log file, or adding a line in a list box or a memo field.

All of these alternatives serve one of two basic purposes: either they let you know that a certain statement of the code has been executed or they let you watch some

values, in both cases without actually stopping the program. Conditional compilation, assertions, and message-flow spying are some of the techniques you can use to complement the features offered by the debugger.

Using Conditional Compilation for Debug and Release Versions

Adding debugging code to an application is certainly interesting, as the OdsDemo example demonstrates, but this approach has a serious flaw. In the final version of the program, the one you give to your customers, you need to disable the debugging output, and you may need to remove all of the debugging code to reduce the size of the program and improve its speed. If you are a C/C++ programmer, however, you may have some ideas on how to remove program code automatically. The solution to this problem lies in a typical C technique known as *conditional compilation*. The idea is simple: you write some lines of code that you want to compile only in certain circumstances and skip on other occasions.

In Delphi, you can use some conditional compiler directives: `$IFDEF`, `$IFNDEF`, `$IFOPT`, `$ELSE`, and `$ENDIF`. For example, in the `Button2Click` event handler of the OdsDemo example, you'll find the following code:

```
{$IFDEF DEBUG}
  OutputDebugString (
    PChar ('Button2Click - I=' + IntToStr (I)));
{$ENDIF}
```

This code is included in the compilation only if there is a `DEBUG` symbol defined before the line or if the `DEBUG` symbol has been defined in the Project Options dialog box. Later on, you can remove the symbol definition, choose the Build All command from Delphi's Compile menu, and run it again. The size of the executable file will probably change slightly between the two versions because some source code is removed. Note that each time you change the symbol definitions in the Project Options dialog box, you need to rebuild the whole program. If you simply run it, the older version will be executed because the executable file seems up-to-date compared with the source files.

note You should use conditional compilation for debugging purposes with extreme care. In fact, if you debug a program with this technique and later change its code (when removing the `DEBUG` definition), you might introduce new bugs or expose bugs that were hidden by the debug process. For this reason, it is generally better to debug the final version of your application carefully, without making any more changes to the source code. A widespread use of `IFDEF` also destroys long-term code maintainability and readability.

Using Assertions

Assertions are a technique you can use in Delphi for custom debugging. An assertion is basically an expression that should invariably be true, because it is part of the logic of the program. For example, I might assert that the number of users of my program should always be at least one because my program cannot run without any user. When an assertion is false, that means there is a flaw in the program code (in the *code,* not in the execution).

The only parameter of the `Assert` procedure is the Boolean condition you want to test (or *assert*). If the condition is met, the program can continue as usual; if the condition is not met (the assertion fails), the program raises an `EAssertionFailed` exception. Here is an example, from the Assert program:

```
procedure TForm1.BtnIncClick(Sender: TObject);
begin
  if Number < 100 then
    Inc (Number);
  ProgressBar1.Position := Number;
  // test the condition
  Assert ((Number > 0) and (Number <= 100));
end;
```

Another button on the Assert example's form generates code that is partially wrong, so that the assertion might actually fail, with the effect shown in Figure 18.17. Keep in mind that assertions are a debugging tool. They should be used to verify that the code of the program is correct. Users should never see assertions failing, no matter what happens in the program and which data they input, because if an assertion fails, there is probably an error in your code. To test for special error conditions, you should use exceptions, not assertions.

Figure 18.17: The error message of an assertion (from the Assert example). Image from the original edition of the book.

Assertions are so closely linked to debugging and testing that you'll generally want to remove them using the `$ASSERTIONS` or `$C` compiler directive. Simply add the line `{$C-}` somewhere in a source code file, and that unit will be compiled without assertions. This doesn't just disable assertions, it actually removes the corresponding code from the program. You can also disable assertions from the Compiler page of the Project Options dialog box.

note Don't forget to rebuild your project after changing the assertions compiler settings, or the setting won't have effect.

Exploring the Message Flow

The integrated debugger provides common ways to explore the source code of a program. In Windows, however, this is often not enough. When you want to understand the details of the interaction between your program and the environment, you'll often need a tool to track the messages the system sends to your application. You can do this in the integrated debugger by using the Event Log and enabling Windows messages in the Event Log Properties dialog box. Note that this type of logging is not enabled by default.

The Event Log, however, is not very flexible, as you cannot choose the message categories or the destination windows: you get a full log of all of the messages for windows of your program, which is often a huge list. An alternative tool for tracing Windows messages is WinSight, a multipurpose tool included in Delphi. Other similar tools are available from many sources, including shareware, books, and magazine articles. I've built my own version, which you'll see shortly.

To become an expert Delphi programmer, you must learn to study the message flow following an input action by a user. As you know, Delphi programs (like Windows applications in general) are event-driven. Code is executed in response to an event. Windows messages are the key element behind Delphi events, although there isn't a one-to-one correspondence between the two. In Windows, there are many more messages than there are events in Delphi, but some Delphi events occur at a higher

level than Windows messages. For example, Windows provides only a limited amount of support for mouse dragging, while Delphi components offer a full set of mouse-dragging events.

Using WinSight

WinSight is a Borland tool available in the Delphi/Bin directory[372]. It can be used to build a hierarchical graph of existing windows and to display detailed information about the message flow. Of course, WinSight knows nothing about Delphi events, so you'll have to figure out the correspondence between many events and messages by yourself (or study the VCL source code, if you have it). WinSight can show you, in a readable format, all of the Windows messages that reach a window, indicating the destination window, its title or class, and its parameters. You can use the Options command from WinSight's Messages menu to filter out some of the messages and see only the groups you are interested in.

Usually, for Delphi programmers, spying the message flow can be useful when you are faced with some bugs related to the order of window activation and deactivation or to receiving and losing the input focus (`OnEnter` and `OnExit` events), particularly when message boxes or other modal windows are involved. This is quite a common problem area, and you can often see why things went wrong by looking at the message flow. You might also want to see the message flow when you are handling a Windows message directly (instead of using event handlers). Using WinSight, you can get more information about when that message arrives and the parameters it carries.

A Look at Posted Messages

Another way to see the message flow is to trap some Windows messages directly in a Delphi application. If you limit this analysis to posted messages (delivered with `PostMessage`) and exclude sent messages (delivered with `SendMessage`), it becomes almost trivial, since we can use the `OnMessage` event of the `TApplication` class and the `TApplicationEvents` component. This event is intended to give the application a chance to filter the messages it receives and to handle certain messages in special ways. For example, you can use it to handle the messages for the window connected with the `Application` object itself, which has no specific event handlers, as we've done in the SysMenu2 example of Chapter 6.

In the MsgFlow example, however, we'll take a look at all of the messages extracted from the message queue of the application (that is, the posted messages). A descrip-

372 As already mentioned, this tool is no longer available with Delphi.

tion of each message is added to a list box covering the form of the example. For this list box, I've chosen Courier font because it is a nonproportional, or *monospaced*, font so that the output will be formatted with the fields aligned correctly in the list box. The speed buttons in the toolbar can be used to turn message viewing on and off, empty the list box, and skip consecutive, repeated messages. For example, if you move the mouse you get many consecutive `wm_MouseMove` messages, which can be skipped without losing much information.

To let you make some real tests, the program has a second form (launched by the fourth speed button), filled with various kinds of components (chosen at random). You can use this form to see the message flow of a standard Delphi window. Figure 18.18 shows an example of the output of the MsgFlow program when the second form is visible. The (lengthy) source code of the program is not described in the text, but it is fully available along with the examples of the book.

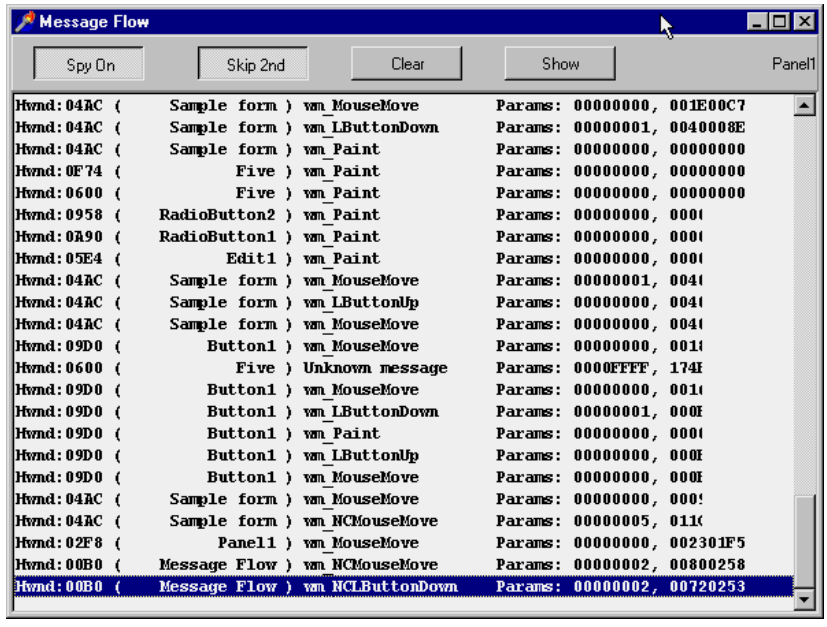


Figure 18.18: The MsgFlow program at run time, with a copy of the second form. Image from the original edition of the book.

Memory Problems

One of the biggest problems when debugging a Delphi program is to check what happens with the memory of the application and of the system. Two of the most common memory problems are *leaks* (not releasing unused memory, so that the program will use much more memory than it actually needs) and memory overruns (using memory already in use or referencing an object that has already been deleted).

There are multiple approaches you can use to detect and solve these memory problems in Delphi, but there isn't much help you can receive from the integrated debugger. To detect these problems you can use the techniques described later in this section or the some of the third-party tools discussed at the end of this chapter. What I want to focus on is an overview of the different memory areas, so that you can better understand when these memory problems will surface and use a preventive approach to avoid them altogether.

Processes and Memory

It's not easy to review memory management for Delphi applications exhaustively, because there's so much information to consider. First, there's Windows memory management, which on Win32 platforms is fairly simple and robust for applications but a bit more complex for DLLs. On the application level, there's Delphi's own internal memory management[373].

In Win32 every application sees its local memory as a single large segment of 4GB, regardless of the amount of physical memory available. This is possible because the operating system maps the virtual memory addresses of each application into physical RAM addresses and swaps these blocks of memory to disk as necessary (automatically loading the proper page of the swap file in memory). This single, huge memory segment is managed by the operating system in chunks of 4KB each, called *pages*.

---

373 Years ago, the Delphi's RTL adopted FastMM4, an efficient external memory manager now part of the core RTL. It offers, among other features, memory leaks tracking. It's also possible to install the official version to instrument an application with additional memory safeguards for tracking overruns and other complex scenarios. See https://github.com/pleriche/FastMM4 for more information.

note Every process has its own private address space, totally separate from the others. This makes the operating system more robust than in the days of 16-bit Windows, when all applications shared a single address space. The drawback is that it is more difficult to pass data between applications.

In fact, in both 95/98 and NT, an application can directly manage only about half of its address space (2GB), while the other half is reserved for the operating system. Fortunately, 2GB is usually more than enough.

Another important element of Win32 memory management is virtual memory allocation. Besides allocating memory, a process can simply reserve memory for future use (using a low-level operation called virtual allocation). For example, in a Delphi application, you can use the `SetLength` procedure to reserve space for a string. Delphi does the same thing transparently when you create a huge array. This memory won't be allocated—just reserved for future use. In practice, this means that the memory subsystem won't use addresses in that range for other memory allocations.

Fortunately, most of the memory management, both at the application level and at the system level, is completely transparent to programmers. For this reason, you don't typically need to know the details of how memory pages work, and we won't explore that topic further here. Instead, we'll explore the status of a region of memory, something you might find very useful while writing and debugging an application.

Global Data, Stack, and Heap

The memory used by a specific Delphi application can be divided into two areas: code and data. Portions of the executable file of a program, of its resources (bitmaps and DFM files), and of the libraries used by the program are loaded in its memory space. These memory blocks are read-only, and they can be shared among multiple processes.

It is more interesting to look at the data portion. The data of a Delphi program is stored in three clearly distinct areas: the global memory, the stack, and the heap.

Global Memory

When the Delphi compiler generates the executable file, it determines the space required to store variables that exist for the entire lifetime of the program. Global variables declared in the interface or in the implementation portions of a unit fall into this category. Note that if a global variable is of a class type, only a 4-byte object reference is stored in the global memory.

You can determine the size of the global memory by using the Project ➢ Information menu item after compiling the program and looking at the value for data size. Figure 18.19, shows a usage of almost 6K of global data, which is not much considering it includes global data of the VCL and of your program.

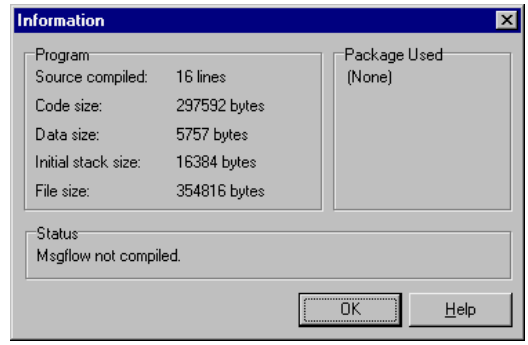


Figure 18.19: The information about a compiled program shown by Delphi. Image from the original edition of the book.

Stack

The *stack* is a dynamic memory area, which is allocated and deallocated following the LIFO order: Last In, First Out. This means that the last memory object you've allocated will be the first to be deleted.

Stack memory is typically used by routines (procedure, function, and method calls) for passing parameters and their return values and for the variables you declare within a routine. Once a routine call is terminated, its memory area on the stack is released. Remember, anyway, that using Delphi's default register-calling convention, the parameters are passed in CPU registers instead of the stack.

Windows application can reserve a large amount of memory for the stack. In Delphi you set this parameter in the linker page of the Project options. However, the default is generally OK. If you receive a stack full of error messages, this is probably because you have a function recursively calling itself forever, not because the stack space is too limited. The initial stack size is another piece of information provided by the Project ➢ Information menu item.

Heap

The *heap* is the area in which the allocation and deallocation of memory happens in random order. This means that if you allocate three blocks of memory in sequence, they can be destroyed later on in any order. The heap manager takes care of all the

details, so you simply ask for new memory with `GetMem` or by calling a constructor to create an object, and Delphi will return a new memory block for you (possibly reusing memory blocks already discarded). Delphi uses the heap for allocating the memory of each and every object, the text of the strings, for dynamic arrays, and for other specific requests for dynamic memory.

Because it is dynamic, the heap is the memory area where programs generally have the most problems. Delphi uses numerous techniques to handle memory, including reference counting (for strings, dynamic arrays, or interface-type object variables) and ownership (for VCL components). Understanding these techniques and applying them properly is the foundation for a correct management of dynamic memory.

To check whether everything is working properly and to understand what is going wrong, Delphi's debugger is of little help. Both Windows and Delphi show the status of the memory (from two different perspectives), which lets you examine the current situation. Delphi also exposes its internal memory manager, so that you can hook into it or even replace it altogether. You can even change the memory handling for a specific class, by overriding its memory allocation and deallocation methods.

Tracking Memory

The Windows API includes a few functions that let us inspect the status of memory. The most powerful of these functions are part of the so-called ToolHelp API (nothing to do with Delphi's own ToolsAPI) and are platform-specific: They are available either on Windows 98 or on Windows NT—but not both.

If we want to remain on common ground we can use `GlobalMemoryStatus`, a function that allows us to inspect the status of the memory in the entire operating system. This function returns system information on the physical RAM, the page file (or swap file), and the global address space. To demonstrate the use of this function, I've built the MemIcon program, which is described in Chapter 19 because its key element is to display the use of tray icons.

Generally, the information related to Windows memory status is of little interest to Delphi programmers, particularly if you compare it to the detailed information about Delphi's own memory manager returned by the `GetHeapStatus` VCL function. This function is defined by the System unit (or the ShareMem unit) and returns a structure with quite a bit of information: the address space that is virtually allocated; the space that's committed or uncommitted, physically allocated, free (distinguishing large and small memory blocks), and unused; and the total overhead of the memory manager.

You can see the data returned by this function in the Memory Status window, which the VclMem example displays as its About box. This window is visible in Figure 18.20. Its form hosts a string grid component, which is automatically updated when the program starts and by a timer (so you can keep this window open and see the information periodically updated). Aside from the memory information displayed, the program is not particularly interesting; you should add this form to one of your complex applications, so that you can test its memory status.

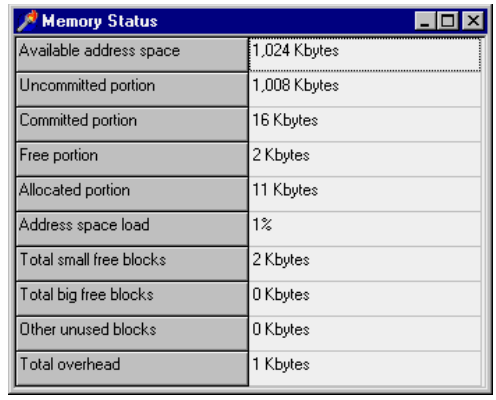


Figure 18.20: The Memory Status window of the simple VclMem example. You should add this window to your own programs to check how much memory they use over time. Image from the original edition of the book.

note This program is a reduced version of a test example from *Delphi Developer's Handbook* (Sybex, 1998). That book goes into more depth, discussing not only some cases in which the memory manager might cause problems but also how you can write a custom memory manager. You might do that to replace Delphi's memory management scheme with your own or to hook into the memory manager; for example, to count the number of memory blocks allocated and deallocated or checking for memory leaks.

Third-Party Tools

Delphi's integrated debugger, the stand-alone Turbo Debugger[374], and the remote debugger are great for helping you to trace source-code errors, but they won't help you much with memory problems, and they are still quite limited in some areas. Besides the techniques I've just discussed here, there are a few third-party tools that can be extremely helpful in tracing and solving memory problems and other debug-

374 The Turbo Debugger as a stand-alone product doesn't exist any more.

ging issues. In this section, I'll just list a few of them and highlight their features in short, to give you an idea of what's available.

These notes are not intended as a full review, and I have no interest in endorsing any of these programs. I have simply noticed that their use can save you a lot of debugging time, and I would like to share this information[375].

Memory Sleuth

Memory Sleuth[376] is produced by Turbo Power Software Company, Inc. (`http://www.turbopower.com`). It was originally developed for Delphi 1 by Per Larsen as MemMonD32. After compiling your Delphi program, you simply run it through Memory Sleuth (loading and running it from this environment instead of from the Delphi IDE).

The tool detects memory and Windows resource leaks, providing a detailed output with the source code lines causing problems. This tool hooks into the Delphi memory manager and monitors the allocation of Delphi objects but also checks the Windows memory status. Besides giving you a report about the problems, the program can also detect peak memory and resource usage, and even draw some nice graphs. A recent enhanced version adds profiling capabilities to the tool.

CodeSite

CodeSite[377] is produced by Raize Software Solutions, Inc. (`http://www.raize.com`). The author is Ray Konopka (who also produced the Raize Components). In the words of its author, "CodeSite is an advanced Delphi debugging tool based on the time-honored approach of sending messages from an application to a message viewer. However, unlike all of its predecessors, CodeSite handles much more than simple strings."

In fact, you can send properties and entire objects to the debug window, which makes the entire operation very fast without being intrusive into the program code. Instead of using a standard debug window, you have to use the one provided by CodeSite, which stores extended information, and allows you, for example, to compare two snapshots of the same object done at different times.

---

375 One of the most interesting solutions in this space, these days, is Nexus Quality Suite (see https://www.nexusdb.com/).

376 While other TurboPower tools were transitioned to open source (see http://turbopack.net/), Memory Sleuth was abandoned when TurboPower closed.

377 CodeSite is currently bundled with Delphi and it's available in the GetIt Package Manager.

BoundsChecker

BoundsChecker[378] is a well-known Windows error detection tool from NuMega Technologies, Inc. (`http://www.numega.com`). The program has a long tradition with Microsoft C++ programmers and has been available for Delphi for a long time, as well. BoundsChecker monitors all Windows API calls made by your program or by the VCL, tracking wrong parameters, resource leaks, stack and heap memory errors, and more. The tool validates the latest Windows APIs including Win32, ActiveX, DirectX, COM, Winsock, and Internet APIs.

You simply run the BoundsChecker program, load your executable file (which must be compiled with debug information and stack frames), and run it. Every time an error is detected, all the details are logged, so that you can trace the problems at the end of the debug session. You can also use BoundsChecker to test for compliance of your program with the different flavors of the Win32 API, if you think the problem is likely to have problems under Windows 98 or Windows NT.

What's Next?

In this chapter, you have seen that there are a number of tools you can use to debug a Delphi application, both by itself and in relation to the Windows system. Windows applications do not live in a world by themselves. They have a strong relationship with the system and, usually less directly, with the other applications that are running. The presence of other running Windows applications can affect the performance of your programs as well as their stability.

In the next chapter, I'll discuss a number of techniques related to printing, using resources, manipulating files, accessing the Clipboard, using Windows INI files, using the Registry, creating and linking help files, and creating an installation program, as well as new Delphi 5 features as TeamSource and the Integrated Translation Environment. Each technique represents a useful approach to solving a specific programming problem.

---

378 BoundsChecker changed ownership a few times to land under Micro Focus. It looks like it's development stopped years ago.

Chapter 19:More Delphi Techniques

In previous chapters, we've presented discussions about major Delphi features. In this chapter, we'll shift the focus slightly to concentrate on a series of real-world tasks you'll often have to consider in your work. We will look at the corresponding Delphi techniques to implement each of those tasks, and we'll demonstrate them with examples.

Because there are many topics to explore in this area, this chapter will move quickly and present examples with a minimum of elaboration. As usual, you can examine the downloaded source code of the example files in more detail.

Managing Windows Resources

Windows resources play an important role from the perspective of memory use. Resources are stored in separate blocks in the executable file of an application, so

that these blocks are loaded in memory on demand, that they can be discarded, and that resources can be considered read-only data.

Before looking at special uses of resources in Delphi, let's look at the tools you can use to prepare resources. Delphi includes an Image Editor for manipulating bitmaps, icons, and cursors, but in some cases, you might still prefer to use a full resource editor, such as Borland's Resource Workshop (which is now included with Delphi) or one of the shareware resource editors available.

Using Resource Editors

You can activate Delphi's Image Editor[379] by choosing it from the Tools menu. Image Editor lets you manipulate four kinds of files. Three of them are file types that contain specific resource types (ICO, CUR, and BMP), and the last is a file format for compiled resource files (RES), which can contain all three kinds of graphical resources. Individual RES files can contain one or more resources of any type (including the graphical resource types).

In the Image Editor, you can prepare any kind of icon, cursor, or bitmap. A single icon resource can contain multiple bitmaps with different sizes and colors. An icon usually has a standard 32 x 32 image and a 16 x 16 image (the small image). In Figure 19.1, you can see an example of an icon with multiple images defined (only one is visible), inside Delphi's Image Editor. A cursor, instead, has a single image but you must set its *hot-spot* position, to designate which point in the image is the active one.

379 As already mentioned, this tool is no longer available with Delphi. There are many graphic design tools offering a much better and complete experience. This makes significant portions of this section no longer applicable.

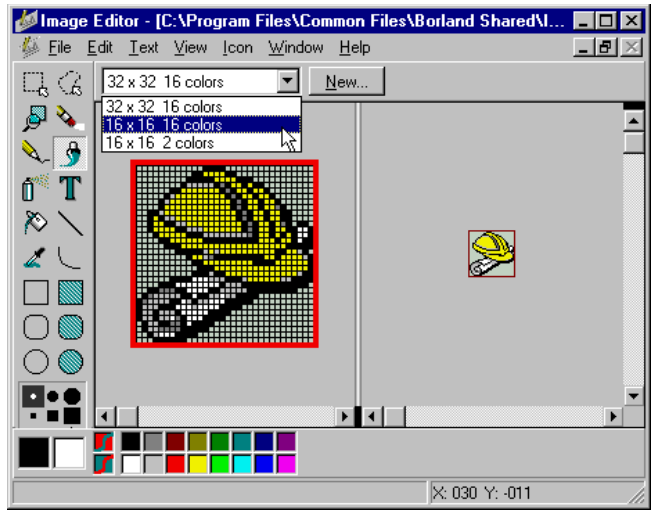


Figure 19.1: Delphi's Image Editor, with the different kinds of images you can define for an icon resource. Image from the original book.

There are basically two ways to use Image Editor:

- Prepare specific files (particularly bitmaps and icons) to be loaded in the Delphi environment at design time (using properties) or at run time (using some of the `LoadFromFile` methods in your code).
- Prepare a resource file with multiple resources and load the resources at run time using Windows API calls, as described in the next section. When you work with resource files in Image Editor, a Tree view lets you see a list of elements of each group.

The Image Editor is a useful tool, but its capabilities are somewhat limited. When you need a full-fledged resource editor, you can use Borland's Resource Workshop[380] (available on the Delphi 5 installation CD). The Resource Workshop lets you open and edit any resource file. You can also use this tool to extract the resources from a compiled program, a DLL, or any other executable file (which doesn't mean that it is always legal to do so; be sure to determine the copyright status of any image you plan to use).

If you open a Delphi application with the Resource Workshop, you will discover that it actually contains a series of resources, in addition to the icon present in its main RES file. By default, a Delphi application's executable file contains a string table with system messages, captions, and other generic strings (such as the names of the

380 Also this tool is no longer available.

months), binary data in custom resources (describing forms in the custom RCDATA format), some cursors, and one icon. Figure 19.2 shows the list of resources for the VclMem example from the last chapter, which is a relatively simple program. You can see the complete list, with an active cursor, a portion of the string table, and the binary data of a form. More complex programs may have many more resources, depending on the number of forms, the VCL units you include, the icons and bitmaps you add to the project, and so on.

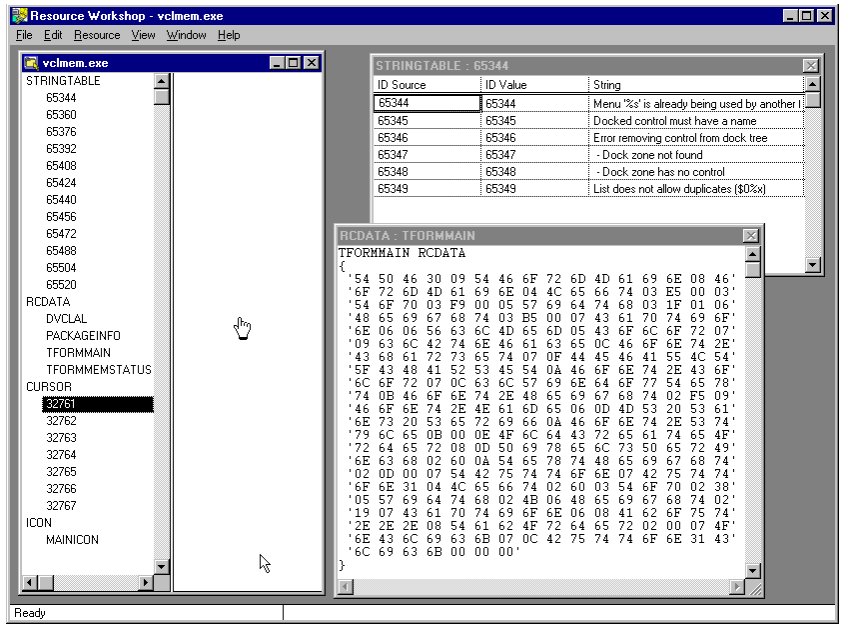


Figure 19.2: The list of resources of a compiled Delphi application in the Resource Workshop. Image from the original book.

note Delphi includes an interesting sample program, named Resource Explorer, that lets you open an executable file (EXE or DLL) and see most of its resources, just as Resource Workshop does. Resource Explorer has no integrated resource editors, but with it you can easily copy a resource, which you can then paste into a resource file for your Delphi application using another resource editor.

Loading Resources

The simplest way to access a graphical file is to load it in a property. Currently, however, the only resources that you can load using properties are icons and bitmaps and sometimes metafiles. For example, you can place an Image component as the

background of a dialog box and load a bitmap into it. In this case, the bitmap is an embedded resource of the application, which means that you do not need to ship the original BMP file (something you have to do if the bitmap is loaded into the image at run time, by calling the `LoadFromFile` method of the image). However, the image is not added to the executable file as a stand-alone bitmap resource. It is included in the binary resource representing the form.

This approach makes it difficult for the unscrupulous to use a resource editor to steal your bitmap. However, keep in mind that it is possible to extract a custom Delphi form resource using a tool such as Resource Workshop, save that resource to a RES-format file (but with a DFM extension), and load it back as a DFM file in the Delphi editor. To do this, someone simply needs to know that your application was created using Delphi. By contrast, form and application icons are placed in the compiled file in a standard resource format, to let applications such as the Explorer or the Windows 95 shell extract them and use them as a hint for the user.

The second technique you can use to access resources in Delphi is the manual approach. For this method, you must first define a separate resource file with the resources you need. The second step is to include the resource file in the project, using the `$R` compiler directive. In fact, contrary to the typical C/C++ approach, Delphi projects can have a number of resource files.

note Don't customize the default resource file—the one that has the same name as the project—because sometimes Delphi changes that file, and you might lose your customizations. Simply add other RES files to the current directory, and add a `$R` compiler directive anywhere in the source code to load them. In Delphi 5, you can now also add an RC file to a project, using the Project Manager. This file will be automatically compiled to a RES file and linked into the program executable, even if it not referenced in a `$R` directive.

Once you have defined some resources and included them in your application, you can use the following Windows API functions to load resources: `LoadAccelerators`, `LoadBitmap`, `LoadCursor`, `LoadIcon`, `LoadMenu`, `LoadResource`, and `LoadString`. Each of these functions loads a specific resource type, except for the `LoadResource` function, which is used for custom resources. The first parameter of these functions is the handle of the application instance, which in Delphi is stored in the `HInstance` global variable. The second parameter is the name of the resource you want to load. Of course, each application can have a number of icons, bitmaps, and other resources, which you can access by name.

The `LoadString` function has some other parameters to specify the buffer in which to copy the string and the size of this buffer. The other loading functions simply return a handle to the loaded resource. You can assign this handle directly to the

`Handle` property of the corresponding VCL object property, or (even better) use the specific methods of the VCL objects as in the following code:

```
var
  Bmp: TBitmap;
begin
  Bmp := TBitmap.Create;
  Bmp.LoadFromResourceName(HInstance, 'MyBitmap');
```

Check out the Mines example in the Chapter 22, which shows the complete use of bitmaps in resource files.

The Icons for Applications and Forms

In Delphi, each application and each form has its own `Icon` property. When you don't set this property for a form, the program simply uses the value of the `Application` object's `Icon` property. You can see and change this default application icon via the Application page of the Project Options dialog box. This icon is also used by the Windows Taskbar, because the window that appears in the Taskbar for a Delphi application is the hidden window of the global `Application` object.

All these scenarios are demonstrated by a sample program I've written called Icons. The form of this example is divided into two parts: on the left are a label, an image component, and two buttons referring to the icon of the form; on the right there are similar components referring to the icon of the application. There's also an OpenDialog component that we'll use to browse for icon resource files. Notice that I've defined the `Icon` property of the form (using the `bb.ico` file) as well as that of the application (using the `aa.ico` file). Each time you click one of the Change buttons, a new icon is loaded from an external file:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with OpenDialog1 do
    if Execute then
    begin
      Application.Icon.LoadFromFile (Filename);
      Image1.Picture.LoadFromFile (Filename);
    end;
end;
```

When you click one of the two Remove buttons, the corresponding icon is simply removed:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
```

```
  Application.Icon := nil;
  Image1.Picture := nil;
end;
```

You can use this program to see how the two icon properties affect the icon of the minimized application. Some examples are shown in Figure 19.3. When you launch the Icons application in the Explorer, Explorer will automatically use the application icon when you minimize the form and not the icon of the main form. This is because the icon of the form is hidden inside the custom resources describing the form (the DFM file), while the icon of the application is stored in an icon resource that's bound to the executable file in the traditional manner.

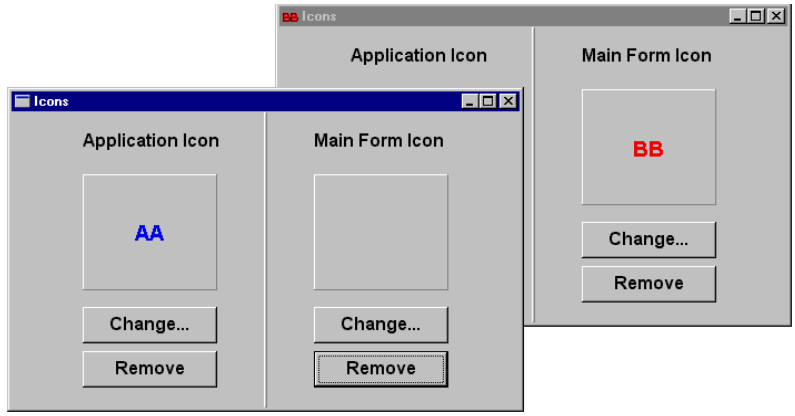


Figure 19.3: Some of the effects of the Icons application. Image from the original book.

Using the Icon Tray of the Taskbar

Windows 95 introduced a new way to display system information: the use of the tray area of the Taskbar. In fact, Windows allows for programs that run only as tray icons. In the lower-right corner of the screen, close to the clock, there is some space (the Taskbar tray) you can use to show your programs or utilities.

There is just one API function involved, `Shell_NotifyIcon`. This function is very simple. It has two parameters: a pointer to a `TNotifyIconData` structure and a flag indicating whether you want to add, remove, or modify the icon. The fields of the data structure include its size (`cbSize`, actually used to determine the version of the structure), the handle of the window to which the tray icon should send notifications (`hWnd`), the number of the notification message (`uCallbackMessage`), an

identifier of the icon (`uID`), some flags indicating which fields are provided, the icon to display (`hIcon`), and its Tooltip message (`szTip`).

When the user interacts with the tray icon, Windows sends back to the given window a message defined by the program, passing as parameters the action performed by the user on the icon (typically a mouse message) and the ID given to the icon. I've used this information about tray icons in the Mem3 example, visible in action in Figure 19.4. This example program is actually useful for displaying the memory status; it is based on the `GlobalMemoryStatus` API function described in Chapter 18. The program uses three icons to highlight the current memory status: green when you still have some free RAM, yellow when the RAM is full but there is plenty of space in the swap file, and red when even the swap file is full.

Besides looking at the icon color, you can move the mouse over it to see a hint with some details; you can also click the tray icon to open a detail window with a lot of extra information, as shown in Figure 19.4. The program uses some advanced techniques to avoid displaying the main window at start-up and to regularly update the icon. If you are interested in this type of program, you should study the source code with care.

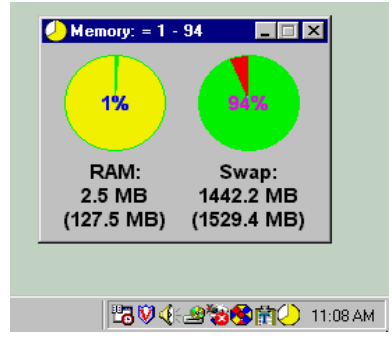


Figure 19.4: The Mem3 example uses two labels to display details about the memory status and adds an icon to the tray. Image from the original book.

Using the Cursor in Delphi

Delphi's support for cursors is extensive, so it takes much less work to customize cursors than it does to customize icons. For example, Delphi includes a number of predefined cursors. Some of these are Windows default cursors, but others are added by Delphi. The use of cursors in Delphi is straightforward; you simply use the Object Inspector to select the corresponding value for the `Cursor` property or `DragCursor` property of a component. In Delphi 5, you can even see the shape of the

cursors in the drop-down list of the Object Inspector, which makes their selection even more intuitive.

If you need to set a global cursor for the whole application for a certain amount of time, you can use the `Cursor` property of the global `Screen` component. The following code fragment demonstrates a common way to display the wait cursor (the hourglass) for the application while a long task is executing:

```
Screen.Cursor := crHourglass;
try
  {time-consuming code would appear here}
finally
  Screen.Cursor := crDefault;
end;
```

This code uses exception handling to ensure that even if something goes wrong in the execution, the default cursor is restored anyway.

The `Cursor` property of the form and other components and the `Cursor` property of the `Screen` object are both of type `TCursor`. If you look up the definition of this data type in Delphi's help, you are in for a surprise. `TCursor` is not a class but rather a numeric type. Technically, a `TCursor` is an integer value that references an array of cursor handles, stored in the `Cursors` property (notice the final *s*) of the `Screen` object. This array can also be used to load a new cursor from an application's resources.

Using String Table Resources

The third kind of resource we will explore in this chapter is the *string table*. String tables have an important role in Delphi, and specific support for them is built into Object Pascal. When you need a string constant in a program, instead of declaring it in a `const` section you can simply declare it in a `resourcestring` section:

```
resourcestring
  Text1 = 'This is some text';
```

When you write this declaration, Delphi automatically adds a new entry to the string table of the application, which will be included in the executable file as a resource. This means that every time you use the `Text1` string, Delphi will automatically add the code to load the string from the string table resource. We'll see this code in a while. The effect of this approach is a different memory layout of the code and data of the program, which is generally beneficial since resources are handled in a very efficient way by the operating system.

A second reason for using resources in a Windows program is to simplify localization, or translation of a program into another language (such as from English to German). To localize a traditional program, you would have to search through each source file, locate the embedded text, translate the text, and then recompile the entire program. In contrast, when you localize a Windows application that uses string tables, you simply translate the text included in the resources, recompile only the resources (a very simple process), and then bind the new resources to the previously compiled EXE code.

note We'll discuss the localization process in Delphi in a later section of this chapter that is fully devoted to the new ITE, the Integrated Translation Environment[381].

Version Information

The last type of resource I'll focus on is version information. Simply open the project options, go into the Version Info page of the dialog box, and enter the proper values for the version number, product name, copyright, and other information. You can see an example of this dialog box in Figure 19.5. This version information of a Delphi project is added to the standard RES file (the one with the same name as the project), generally referenced from the project source code and included in the executable file. Don't remove this reference from the source code of a DLL to add version information to it.

381 The tool is currently an unsupported add-on, but the concepts in terms of how you can use resources for localization still apply and many third-party translation tools rely on the same core foundation.

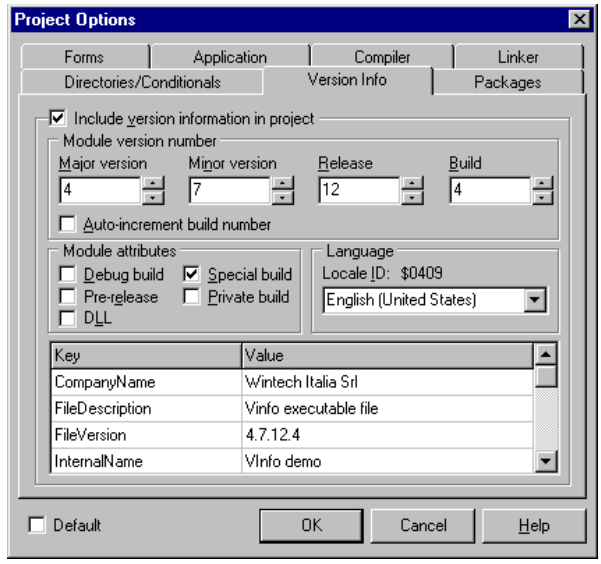


Figure 19.5: The Version Info page of the Project Options dialog box. Image from the original book.

Version information is required by DLLs and OLE servers (including ActiveX controls) so that installation programs can determine whether you already have the most recent version of a DLL. Without this technique, you risk installing an older DLL or control over a newer version. Knowing the details of version information for DLLs is particularly important if you want to make effective use of an installation program (or write your own, comparing the version information of files already present on the hard disk with that of the files you are installing).

Because you can find a good description of the role of version information in Microsoft documentation, I'll skip repeating that. What I want to do instead is to show you a simple use of version information inside an executable program. I've just added the version information of Figure 19.5 to the VInfo program and written some code that extracts some of this information to a memo component when the user clicks the form's Read Version Info button. You can see the result in Figure 19.6. The problem, as you'll see, is that the API for accessing version information is far from simple.

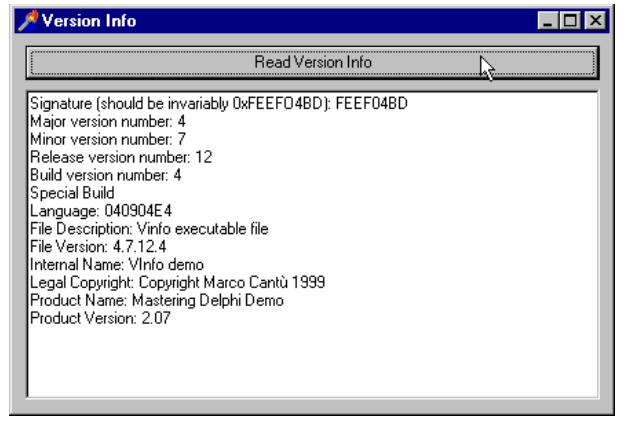


Figure 19.6: The version information extracted by the VInfo example. Image from the original book.

The central API call is `GetFileVersionInfo`. This function requires as parameters the filename, a pointer to a block of memory to store the data, and the size of this block of memory. To allocate a memory block of the proper size, a program can call the `GetFileVersionInfoSize` API function first. Here is the first part of the button's `OnClick` event handler:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  VInfoSize, DetSize: DWord;
  pVInfo, pDetail: Pointer;
begin
  Memo1.Lines.Clear;
  VInfoSize := GetFileVersionInfoSize (
    PChar (ParamStr (0)), DetSize);
  if VInfoSize > 0 then
  begin
    GetMem (pVInfo, VInfoSize);
    try
      GetFileVersionInfo (PChar (ParamStr (0)), 0,
         VInfoSize, pVInfo);
        ...
    finally
      FreeMem (pVInfo);
    end;
  end;
end;
```

The final part of the code, inside the `finally` block, deletes the memory block. In between, the program uses the `pVInfo` pointer to access version information. We access data by calling the `VerQueryValue` API function, which requires as parameters the pointer to the data, a string with a path indicating the requested

information, and a pointer. The function sets this pointer to the requested string or data structure.

The first part of the actual code accesses the fixed portion of the file information, a group of flags and numbers defined by the `TVSFixedFileInfo` structure. Here is the code to access some of this data (in the downloaded files you'll find a longer version):

```
// show the fixed information
VerQueryValue (pVInfo, '\', pDetail, DetSize);
with TVSFixedFileInfo (pDetail^) do
begin
  Memo1.Lines.Add (
    'Signature (should be invariably 0xFEEF04BD): '
    + IntToHex (dwSignature, 8));
  Memo1.Lines.Add ('Major version number: ' +
    IntToStr (HiWord (dwFileVersionMS)));
  if (dwFileFlagsMask and dwFileFlags
      and VS_FF_DEBUG) <> 0 then
    Memo1.Lines.Add ('Debug info included');
```

The second part of this code reads some of the strings included in the version information of the program. Each string should be separately accessed, again using the `VerQueryValue` API function. Here is how you can write one of these calls:

```
VerQueryValue(pVInfo,
  '\StringFileInfo\040904E4\FileDescription', pDetail, DetSize);
Memo1.Lines.Add ('File Description: ' + PChar (pt2));
```

If you write the code this way, however, you end up embedding the locale and character set information (`040904E4`) into the code. Although you can set this language information in the Project Options page, your code should be able to read the current language and not use a hard-coded value. To accomplish this, you should first read the language from the version information (not a terribly easy pointer operation) and then use it for further processing, as in the following code (part of the `Button1Click` method of the VInfo example):

```
type
  TLangInfoBuffer = array [1..4] of SmallInt;

var
  pLangInfo: ^TLangInfoBuffer;
  strLangId: string;
begin
  // get the first language
  VerQueryValue(pVInfo,
    '\VarFileInfo\Translation',
    Pointer(pLangInfo), DetSize);
  strLangId := IntToHex (SmallInt (pLangInfo^ [1]), 4) +
```

```
  IntToHex (SmallInt (pLangInfo^ [2]), 4);
Memo1.Lines.Add ('Language: ' + strLangId);

// show some of the strings
strLangId := '\StringFileInfo\' + strLangId;
VerQueryValue(pVInfo, PChar(strLangId + '\FileDescription'),
  pDetail, DetSize);
Memo1.Lines.Add ('File Description: ' +
  PChar (pDetail));
```

The Integrated Translation Environment[382]

One of the brand-new features of Delphi 5 is the Integrated Translation Environment, or ITE for short. This book does not have enough space to discuss this complex tool in detail. As usual, then, I'm simply going to guide you through some of its features and show you an example.

To start the translation process, you can use the Resource DLL Wizard located in the File ➢ New dialog box or use the Project ➢ Language ➢ Add menu command. Both methods open a wizard where you select one of the projects of the active group, choose the language (in my example, I selected Italian standard; see Figure 19.7), select whether this is a new translation or an update of an existing one, and click the Finish button. If this is a new project, you'll obtain a new directory and some statistics such as those shown in Figure 19.8.

382 This tool is still currently available as a separate download, but it's no longer officially part of Delphi and supported. I will not point out to specific difference in this section, as it's completely obsolete at this time. I recommend looking for an alternative solution.

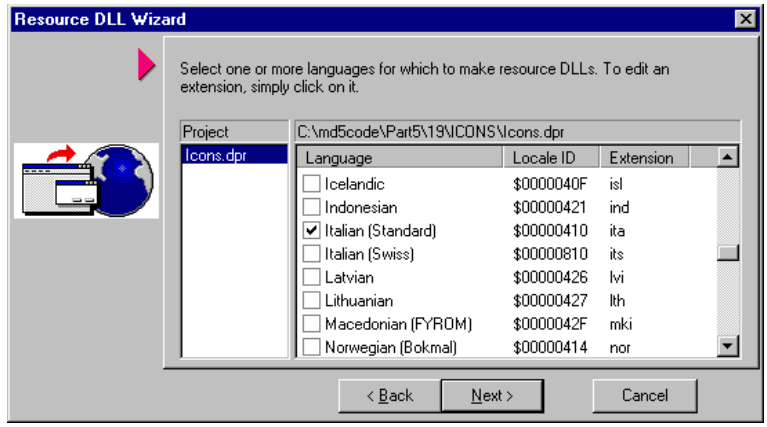


Figure 19.7: Selecting a language in the Resource DLL Wizard. Image from the original book.

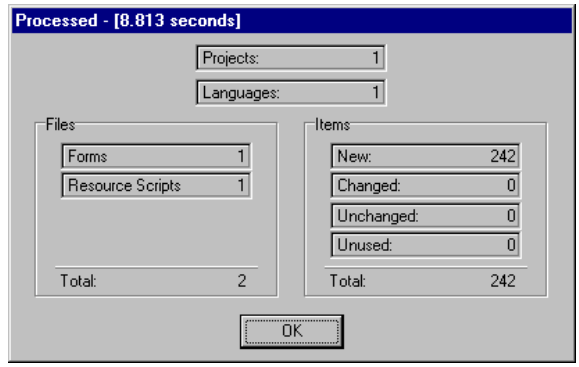


Figure 19.8: The final statistics displayed by the Resource DLL Wizard after adding a new language to a project. Image from the original book.

Once everything is properly set up, you can start working with the Translation Manager. This is a dialog box where the ITE shows a list of resources, including forms and strings, which can be modified for the translation. This window lists the elements you can modify in the translation, provides the original text as well as the translated text, and tracks past versions and changes dates. You can filter this table and choose the columns you want to see, using its shortcut menu. In Figure 19.9, you can see the Translation Manager with a form selected (taken from my Italian translation of the Icons example).

The Translation Manager works in conjunction with another tool, the Translation Repository, where you can store the standard translation of a frequent term. You can update the repository manually (use the Tools ➢ Translation Repository command to open it) or use the Repository ➢ Add strings to Repository command from

the Translation Manager's shortcut menu. You can use another command from the same shortcut submenu (Get Strings from Repository) to automatically translate all of the terms available in the repository. Notice that the repository can handle multiple translations for the same word and other advanced elements.

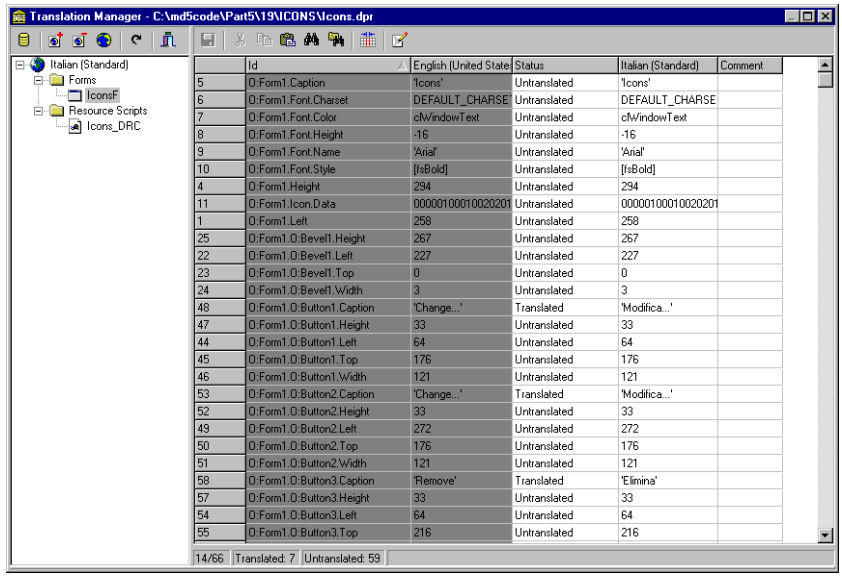


Figure 19.9: The new Delphi 5 Translation Manager, part of the ITE. Image from the original book.

In Figure 19.9, you can see that you can modify the text of the strings, as I've done, but you can also change other parameters, such as positions and fonts. This might be required when the different length of the translated string affects the user interface. Of course, it is not easy to determine whether the size and position of the controls is correct by looking at these numbers. What you can do is close the Translation Manager, select the new project the ITE has added to the current project group, and move to the form, opening it at design time. This is what I've done to produce Figure 19.10, where you can see that the translation into Italian has created a problem with the length of the labels. The interesting feature is that you can freely edit the translated form (provided you don't add components to it) by moving the controls, changing their fonts, and so on.

In fact, every time you reopen the Translation Manager (by selecting the main project and issuing the nonobvious Project ➤ Languages ➤ Update Resource DLL command), this will read the current values from the DFM files (the original and the translated ones) and refresh the Translation Manager's structure accordingly. This way, if you update the original form, you don't have to translate it again, just pro-

vide the translation for the new elements. At the same time, you can modify the translated DFM file and see the changes reflected in the translation system. I've done this in my example.

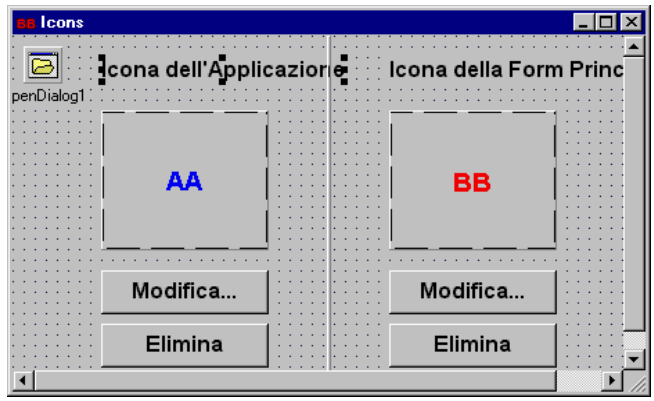


Figure 19.10: When translating captions, some might become larger, leading to problems with the program. Image from the original book.

Once you've compiled the translated project, which is technically a DLL, you can use the Project ➤ Languages ➤ Set Active command to activate it, so that running the project in the debugger will load the active language extension. This is only a test; usually you'll simply need to run the main executable file, and it will automatically use the translated DLL corresponding to the current regional locale selected on the computer (via the Control Panel's Regional Settings applet).

How does this work and what happens behind the scenes? The Resource DLL Wizard creates a DLL project that includes the translated DFM and strings. The DLL includes only the resources, not a copy of the compiled code, and has an extension that corresponds to the three-letter code identifying the locale, ITA in the example. This structure is visible in the project source code:

```
library Icons;
{ITE} {$R 'IconsF.dfm' Form1:TForm}
  {DFMFileType} {IconsF.dfm}
{ITE} {$R 'Icons_DRC.res' 'Icons_DRC.rc'}
  {RCFileType} {Icons_DRC.rc}
{$E ita}
begin
end.
```

The two `$R` directives determine the resources to include in the project, and they are followed by comments required by the ITE (you should not modify them). The `$E` directive determines the extension of the executable file. The ITE creates a subdirectory for each locale, so that the filenames won't conflict.

As you compile the resource DLL, its output is placed in the parent directory, the same one that hosts the project, so that it will become immediately usable. As you start this executable file, in fact, the VCL will load the resources either from the main EXE file or from the DLL matching the current regional settings.

note To activate the translated program, you'll need to close the current one, change the regional settings, and reexecute it. Technically, it is possible also to change the language on the fly, reloading the forms without stopping the program. However, the current user input will be lost, as forms have to be re-created from scratch. If you want to explore this dynamic approach refer to the RichEdit example included in the Delphi demos, particularly the global functions of the `ReInit.pas` unit. By adding this unit to your programs you'll be able to obtain the dynamic language change demonstrated by that program.

Printing

Delphi supports printing in a number of ways. Forms can print their graphical output (see the `Print` method and the `PrintScale` property of `TForm`), and some components have direct support for printing, such a the RichEdit control. For all but the simplest operations, you'll use the global `Printer` variable to manipulate a printer from a Delphi program. Actually, `Printer` is the name of a global function; it returns an object of class `TPrinter`, which is defined in the Printers unit.

You can use the object returned by the `Printer` function to access some global properties related to the printer, such as a list of installed drivers or printer fonts. However, its key property is its `Canvas`. You can use the canvas of a printer the same way that you use the canvas of a form; that is, you can print text, graphics, and everything else. To use this canvas, you need to call the printer's `BeginDoc` method to start the printing job, use the canvas methods to produce the output, and then call the `EndDoc` method to send the output to the printer. As alternatives, you can call the `Abort` method to discard the print job or call the `NewPage` method to send the output to the printer and start working on a new page.

A Print Preview of Graphics

Our first example with the global printer object (using the `Printer` function) is a simple application you can use to print bitmaps. Basically, this is an extension of the TabOnly example presented in Chapter 8. That example used a TabControl compo-

nent to let the user browse through a series of bitmaps. As shown in Fig-ure 19.11, the PrintBmp example displays a preview form that has a toolbar with four buttons at the top and a ScrollBox component that contains an Image component. If the image is bigger than the form, you can use the ScrollBox component to scroll the image without affecting the toolbar.

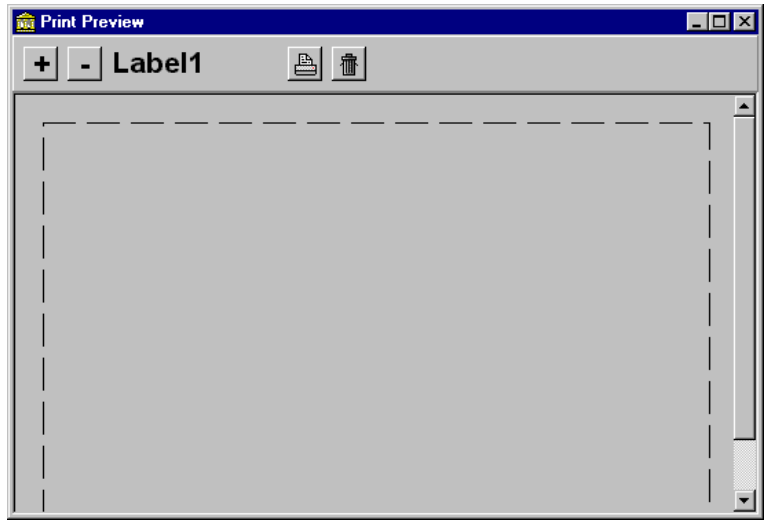


Figure 19.11: The Print preview form of the PrintBmp example at design time. Image from the original book.

This preview dialog box is opened from the application's File ➢ Print command. The preview form lets you compare the size of the resulting bitmap with the printed page (indicated by the size of the image component) and scale the bitmap as neces-sary, to increase its size.

note Changing the image size affects both the screen output in the preview form and the printed out-put. The reason for scaling a bitmap prior to printing is that bitmaps printed at their standard pixel-per-inch ratio tend to appear quite small on the printed page.

The code is based on the `StretchDraw` method of the `TCanvas` class, which I've used to generate both the preview bitmap and the output bitmap. The `StretchDraw` method has two parameters: a rectangle indicating the output region and a graphic object (the source image). The result is an image stretched to fit the output rectan-gle. Now let's review some of the code. The main form responds to the Print command by initializing and running the Preview form:

```
procedure TForm1.Print1Click(Sender: TObject);
begin
```

```
  {double-check whether an image is selected}
  if Image1.Picture.Graphic <> nil then
  begin
    {set a default scale, and start the preview}
    PreviewForm.Scale := 2;
    PreviewForm.SetPage;
    PreviewForm.DrawPreview
    PreviewForm.ShowModal;
  end;
end;
```

The test at the beginning could have been omitted, since the Print menu item is disabled until an image file is selected, but it ensures that a file is selected in any case. This code sets a public field of the `PreviewForm` object (`Scale`), calls two methods of this form (`SetPage` and `DrawPreview`), and finally displays it as a modal form. Figure 19.12 shows an example of the Print Preview form at run time.

The `SetPage` method sets the size of the Print Preview form's Image component, using the size of the printed page:

```
procedure TPreviewForm.SetPage;
begin
  Image1.Width := Printer.PageWidth div 5;
  Image1.Height := Printer.PageHeight div 5;
  {output the scale to the toolbar}
  Label1.Caption := IntToStr (Scale);
end;
```

The size of the page is divided by five to make it fit into a reasonable area of the screen. You might use a parameter instead of this fixed value to add a zooming feature on the preview page. However, it seemed confusing to have one button to increase the size of the printed image and another button to increase it only in the preview, so I decided to skip the zooming capability.

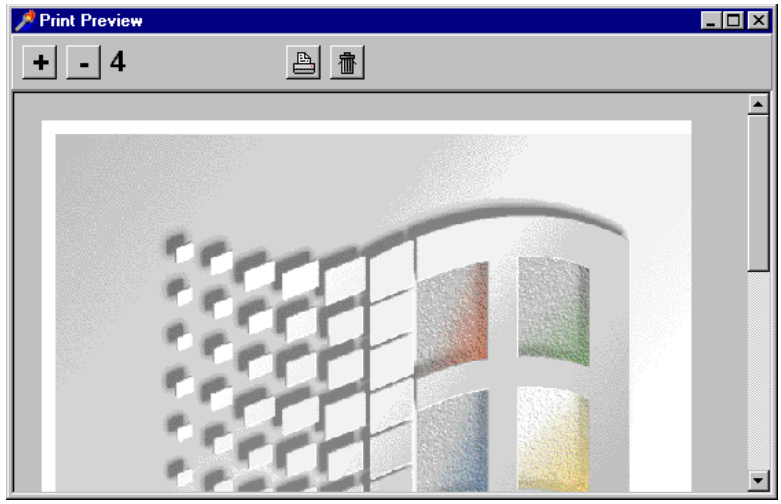


Figure 19.12: The PrintBmp example's Print Preview form, with the program's main form in the background. Image from the original book.

The heart of the preview form's code is in the `DrawPreview` method, which has three sections. At the beginning, it computes the destination rectangle, leaving a 10-pixel margin, scaling the image, and using the fixed zoom factor of 5 (as you can see in the following listing). The second step erases the old image that is still on the screen by drawing a white rectangle over it. The third step calls the `StretchDraw` method of the canvas, using the rectangle calculated before and the current image from the image component of the main form (`Form1.Image1.Picture.Graphic`). To access this information, we need to add a `uses` clause in the `implementation` portion of the code, referring to the `Viewer` unit (which declares the `TForm1` class). Here is the code for the `DrawPreview` method:

```
procedure TPreviewForm.DrawPreview;
var
  Rect: TRect;
begin
  {compute the rectangle for the bitmap preview}
  Rect.Top := 10;
  Rect.Left := 10;
  Rect.Right := 10 +
    (Form1.Image1.Picture.Graphic.Width * Scale) div 5;
  Rect.Bottom := 10 +
   (Form1.Image1.Picture.Graphic.Height * Scale) div 5;
  {remove the current image}
  Image1.Canvas.Pen.Mode := pmWhite;
  Image1.Canvas.Rectangle (0, 0,
    Image1.Width, Image1.Height);
  {stretch the bitmap into the rectangle}
  Image1.Canvas.StretchDraw (Rect,
```

```
    Form1.Image1.Picture.Graphic);
end;
```

All of this code is executed just to initialize the form. The start-up code is split into two methods but only because `DrawPreview` will be called again later. When initialization is complete and the modal form is visible, the user can click the four toolbar buttons to resize the image, print it, or skip it.

The two resize methods are simple, because they just set the new scale value and call the `DrawPreview` procedure to update the image:

```
procedure TPreviewForm.ScalePlusButtonClick(
  Sender: TObject);
begin
  Scale := Scale * 2;
  Label1.Caption := IntToStr (Scale);
  DrawPreview;
end;
```

The `PrintButtonClick` method of the preview form is almost a clone of `DrawPreview`. The only differences are that the destination rectangle is not zoomed, and the bitmap is sent to the printer in a new document (a new page):

```
procedure TPreviewForm.PrintButtonClick(Sender: TObject);
var
  Rect: TRect;
begin
  {compute the rectangle for the printer}
  Rect.Top := 10;
  Rect.Left := 10;
  Rect.Right := 10 +
    (Form1.Image1.Picture.Graphic.Width * Scale);
  Rect.Bottom := 10 +
    (Form1.Image1.Picture.Graphic.Height * Scale);
  {print the bitmap}
  Printer.BeginDoc;
  try
    Printer.Canvas.StretchDraw (Rect,
      Form1.Image1.Picture.Graphic);
    Printer.EndDoc;
  except
    Printer.AbortDoc;
    raise;
  end;
end;
```

note When your program paints on the form canvas, it can be adapted to produce the same output directly on the printer canvas. The same code can target a generic canvas, possibly after the coordinate system is changed. This is demonstrated by the Shapes example in the Chapter 22.

Printing Text

There are times you need to print some text directly and as fast as possible, without the need of extra fancy graphics. When this happens, you can rely on the direct printing support offered by text files. Once you've created a file storing text, you can assign that file to a printer and write the text to it. Take a look at this code, which is part of the QrNav example:

```
procedure TNavigator.PrintButtonClick(Sender: TObject);
var
  PrintFile: TextFile;
begin
  {assigning the printer to a file}
  AssignPrn (PrintFile);
  Rewrite (PrintFile);
  try
    {set the font of the form, and output each element}
    Printer.Canvas.Font := Font;
    Writeln (PrintFile, Label1.Caption,
      ' ', DBEdit1.Text);
    Writeln (PrintFile, Label2.Caption,
      ' ', DBEdit2.Text);
    Writeln (PrintFile, Label3.Caption,
      ' ', DBEdit3.Text);
  finally
    {close the printing process}
    System.CloseFile (PrintFile);
  end;
end;
```

This event handler prints the text of some edit boxes and the related labels. As you can see in the source code, the program uses the same technique to print the entire content of the database table.

---

note This type of printing support still relies on Windows printer drivers, which can generate only graphical output. If you want to be really fast, you can use the `Escape` API to drive your printer directly with Escape commands. Some dot matrix printers can run very fast with continuous paper, and this can be a handy way to get all of their speed.

The QuickReport Components[383]

The professional edition of Delphi includes QuickReport, a collection of reporting components tightly integrated with Delphi and licensed to Borland by QSD AS, Norway. Similar Delphi components are available from other third-party companies, but I'll focus on this one simply because it will be readily available to most Delphi developers.

note Alternatives to the QuickReport reporting component set include ReportPrinter, ReportBuilder (formerly known as Piparti), ACE Reporter, and many others.

QuickReport uses a form to visually build a report in a way very similar to the way you build normal forms. However, you'll use this report form only to develop the report; it is never actually shown on screen at run time. To print or display the report, you can call the `Print` or `Preview` methods of the QuickReport component, which you place on each report form (placing a QuickReport component on a form turns it into a report form).

Using QuickReport, a report is constructed from *bands,* or horizontal regions of information. You can use a band to output data, to provide a header and footer in each printed page, or to include totals and other special information. To build a report, you simply place the QuickReport component in a secondary form (not the application's main form), add one or more bands, and then place on those bands some of the QuickReport data-aware reporting components, which connect to a Delphi data source in the usual way. The data can come from one or more tables or queries, as in the standard data-access components.

The use of the QuickReport components is demonstrated in the QrNav example mentioned in the last section. The secondary form of the program, the report form, has a QuickReport component and three QRBand components, as shown in Figure 19.13.

383 The QuickReport components no longer ship with Delphi. Again, this entire section is now obsolete. There is now a light version of FastReport bundled with the product in the GetIt package manager. Recent news indicate that the work on QuickReport has resumed and moving to a recent version might offer the best compatibility, if you used it in the past. There are also tool helping with migration to a different reporting engine.

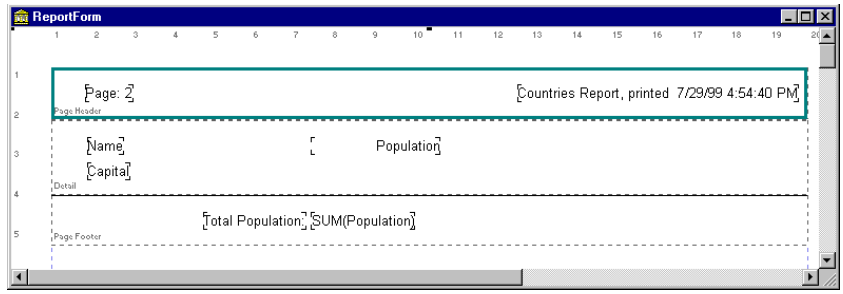


Figure 19.13: The form used to build the report of the QrNav example at design time. This form is used by the report component it contains, but it is never displayed at run time. Image from the original book.

One of the key properties of the QRBand component is `BandType`, used to indicate the role of the band in the report. In this example, the first band is of type `rbPageHeader`, the second of type `rbDetail`, and the third of type `rbPageFooter`. Other types of bands you can use include `rbTitle`, included before or after the first page's header; `rbSummary`, printed only at the end of the report; `rbGroupHeader` and `rbGroupFooter` for groups defined with the specific QRGroup component; `rbColumnHeader` for multicolumn reports; and a few others.

In the example, I've placed two QRSysData components in the first band (the page header). These components display the page number, date, and time, and their `Text` property contains a description. You can print many other types of system information using this component, as indicated by its `Data` property. I've also set a border for the enclosing band, using its `Frame` property.

The second band contains the real data from the database. Detail bands are replicated on the report for each record that appears in the data source, so you have to specify the dataset for each report component. In this case, I've used the `Table1` from the program's main form (after choosing File ➢ Use Unit). This same dataset is also connected to the three QRDBText components placed in the second band.

note In addition to the `DataField` property they share with the standard Delphi data-aware components, the report components also have some formatting capabilities. If you try setting the value *'###,###,###'* for the `Mask` property, the numbers will be output with thousands separators.

In the last band, I've added a QRExpr component to display the total population of the countries in the report (actually in all the records up to the current page). This component can do complex calculations. The simplest approach is to use its `Expression` property to indicate the kind of operation (such as sum, min, max, average, or count) and the related field (such as Population). The `Expression` property has a special editor you can use to create the expression, instead of typing it. Remember, however, to set the `Master` property to the report component,

`QuickRep1`, because this is the only way to connect the calculated value with the proper dataset.

Having designed the report, you can test it by simply double-clicking the report component. This displays the print preview form, which you can use directly to print the report without even compiling the program. You can obtain the same print preview at run time (see Figure 19.14) by including the following in the main form's code:

```
procedure TNavigator.ReportButtonClick(Sender: TObject);
begin
  ReportForm.QuickReport1.Preview;
end;
```

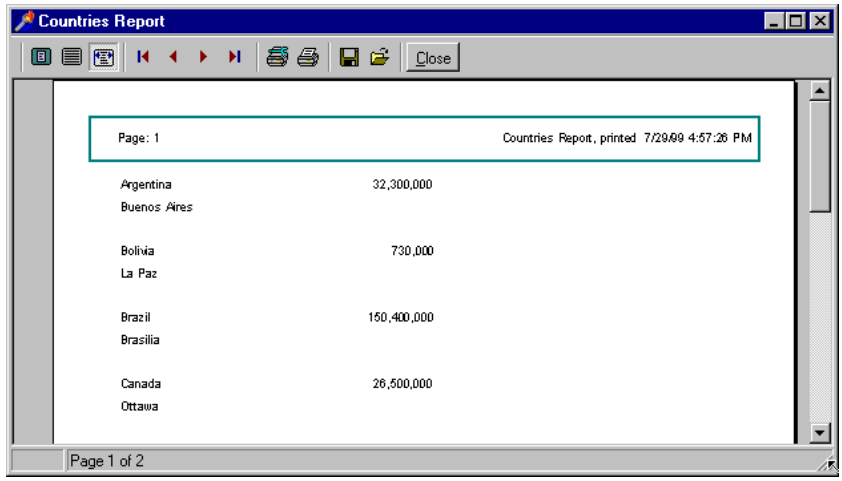


Figure 19.14: The print preview form of the QrNav example, which is based on the QuickReport component. Image from the original book.

Manipulating Files

One of the peculiarities of Pascal compared with other programming languages is its built-in support for files. The language has a `file` keyword, which is a type specifier, like `array` or `record`. You use `file` to define a new type, and then you can use the new data type to declare new variables:

```
type
  IntFile: file of Integers;
```

```
var
  IntFile1: IntFile;
```

It is also possible to use the `file` keyword without indicating a data type, to specify an untyped file. Alternatively, you can use the `TextFile` type, defined in the System units, to declare files of ASCII characters. Each kind of file has its own predefined routines[384].

Once you have declared a file variable, you can assign it to a real file in the file system using the `AssignFile` method. The next step is usually to call `Reset` to open the file for reading at the beginning, `Rewrite` to create a new file, or `Append` (which only applies to files of type TextFile) to add new items to the end of the file without removing the older items. Once the input or output operations are done, you should call `CloseFile`. This operation should typically be done inside a `finally` block, to avoid leaving the file open in case the file-handling code generates an exception.

File Support in Delphi Components

Besides the standard Pascal language file support, Delphi includes a number of other options for manipulating files. Several components have methods to save or load their contents from a file (such as a text or a bitmap file), and there are other specific classes to handle files. Many component classes have `SaveToFile` and `LoadFromFile` methods. In this book, we have used these methods for `TBitmap`, `TPicture`, and `TStrings` classes (used in `TMemo`, `TListBox`, and many other component classes). They are also available for some data-aware components (`TBlobField`, `TMemoField`, and `TGraphicField`), for other graphic formats (`TGraphic`, `TIcon`, and `TMetaFile`), for OLE (Object Linking and Embedding) containers, and for the TreeView and other Windows common controls.

Similar methods are available in the `TMediaPlayer` class. These methods are named `Open` and `Save`, and they have a slightly different syntax and meaning than their `LoadFromFile` and `SaveToFile` counterparts. Another file-related class we'll use later in this chapter is `TIniFile`. This class implements management of Windows initialization files or any custom file using the same format. The new topics we will cover in the following sections are file-system components, streaming components, and the components that implement object persistency.

---

384 While still available, using the file keyword is not recommended at all these days. There are many other Delphi RTL features for file management, including those listed later in this section and many other added over time.

File System Components

The Delphi file system components are located in the System page of the Components Palette: `TDirectoryListBox`, `TDriveComboBox`, `TFileListBox`, and `TFilterComboBox`. These components sport the old-fashioned Windows 3.1 user interface, and for this reason they are not terribly useful. However, the same FileCtrl unit that defines these components also contains three interesting routines:

- `DirectoryExists`, which is used to check whether a directory exists.
- `ForceDirectories`, which can create several directories at once.
- `SelectDirectory`, which displays a standard Delphi dialog box for directory selection.

The Dirs example demonstrates the use of these little-known routines. You can see the related calls in the source code. Actually, the Windows shell has a number of similar but more advanced routines, and it offers a number of default dialogs, including one to select a folder. As a starting point, you might want to check the Windows API under the group Shell's Namespace Functions. There are many goodies available there and even more undocumented!

In the Dirs example, I've done only a simple step in this direction. A button calls the `ShBrowseForFolder` API, which displays the system dialog used to find a folder (or a printer or a computer, depending on the parameters). Here is the code:

```
uses
  ShlObj;

procedure TForm1.btnBrowseClick(Sender: TObject);
var
  bi: TBrowseInfo;
  pidl: pItemIdList;
  strpath: string;
begin
  bi.hwndOwner := Handle;
  bi.pidlRoot := nil;
  bi.pszDisplayName := '';
  bi.lpszTitle := 'Select a folder';
  bi.ulFlags := bif_StatusText;
  bi.lpfn := nil;
  bi.lParam := 0;

  pidl := ShBrowseForFolder (bi);

  SetLength (strPath, 100);
  ShGetPathFromIdList (pidl, PChar(strPath));
  Edit1.Text := strPath;
```

```
end;
```

You can see the effect of this code in Figure 19.15, side by side with Delphi's own dialog box generated by the `SelectDirectory` call.

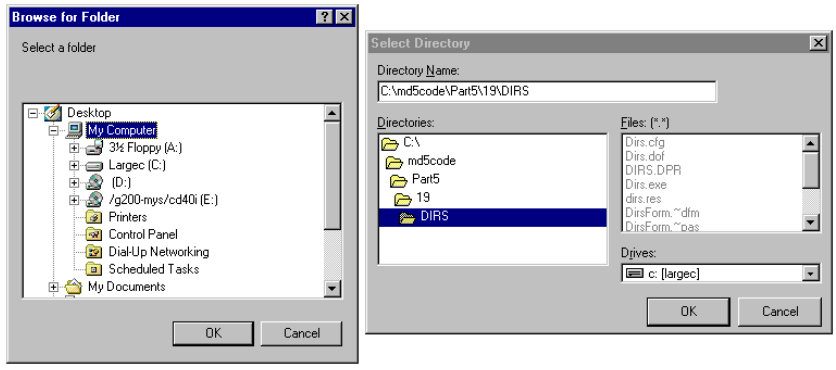


Figure 19.15: Windows Browse for Folder dialog box and Delphi's Select Directory dialog box, both activated by the Dirs example. Image from the original book.

note When you have to work with files and directories, you might want to check the new `TMask` class introduced in Delphi 5. This new class is declared in the Masks unit (not to be confused with the older Mask unit, which defines the edit masks). The `TMask` class allows you to perform pattern matching using the wildcards (the standard `*` and `?`), compare values with a set of masks, and examine the ranges. See Delphi's help file under `TMask.Create` for more information.

Streaming Data

Another topic worth exploring is Delphi's support for file streams. The VCL defines the abstract `TStream` class and a number of subclasses[385]. The parent class, `TStream`, has just a few properties, but it also has an interesting list of methods you can use to save or load data.

Creating a `TStream` instance makes no sense, because this class is abstract and provides no direct support for saving data. Instead, you can use one of the derived classes to load data from or store it to an actual file, a BLOB field, a socket, or a memory block. Use `TFileStream` when you want to work with a file, passing the filename and some options to the `Create` method. Use `TMemoryStream` to manipulate a stream in memory and not an actual file. However, this class has special methods to

385 Using `TStream` classes is still fundamental today, although there are many extensions and new classes like readers and writers that extend the TStream classes.

copy its contents to or from another stream, which can be a file stream. Creating and using a file stream is as simple as creating a variable of a type that descends from `TStream`:

```
var
  S: TFileStream;
begin
  if OpenDialog1.Execute then
  begin
    S := TFileStream.Create (OpenDialog1.FileName,
      fmOpenRead);
    try
      {use the stream S ...}
    finally
      S.Free;
    end;
  end;
end;
```

As you can see in this code, the `Create` method for file streams has two parameters: the name of the file and a flag indicating the requested access mode. In this case, we want to read the file, so we used the `fmOpenRead` flag (other available flags are documented in the Delphi help). Streams can be used instead of traditional Pascal files, although they might be less intuitive to use at first. A big advantage of streams is that they're very interchangeable, so you can work with memory streams and then save them to a file, or you can perform the opposite operations. This might be a way to improve the speed of a file-intensive program. Here is a snippet of code, a file-copying function, to give you an idea of how you can use streams:

```
procedure CopyFile (SourceName, TargetName: String);
var
  Stream1, Stream2: TFileStream;
begin
  Stream1 := TFileStream.Create (SourceName, fmOpenRead);
  try
    Stream2 := TFileStream.Create (TargetName,
      fmOpenWrite or fmCreate);
    try
      Stream2.CopyFrom (Stream1, Stream1.Size);
    finally
      Stream2.Free;
    end
  finally
    Stream1.Free;
  end
end;
```

Another important use of streams (both file streams and memory streams) is to handle database BLOB fields or other large fields directly. In fact, you can export

such data to a stream or read it from one by simply calling the `SaveToStream` and `LoadFromStream` methods of the `TBlobField` class.

The Clipboard

In Delphi, Clipboard support comes in two forms:

- Some components have specific Clipboard-related methods. For example, `TMemo`, `TEdit`, and `TDBImage`, among other components, have the `CopyToClipboard`, `CutToClipboard`, and `PasteFromClipboard` methods.
- There is a global `Clipboard` object of the `TClipboard` class, which has a number of specific Clipboard features. For full Clipboard support, the use of the `Clipboard` object, defined in the `ClipBrd` unit, is required.

A program can use the `Clipboard` object to see if the Clipboard currently holds data of the requested format, such as text or bitmap, using the `HasFormat` method. As the Clipboard can also hold multiple versions of the same data, at times it is useful to list all the available formats. Finally, you can use the global object to place data in the Clipboard, when this function isn't handled directly by other components. The `Clipboard` object can also be used to open the Clipboard and copy data in different formats. This is the only case in which you need to open and close the Clipboard in Delphi—something that is also required when using the Windows API directly.

Copying and Pasting Text

We've already seen an example of the use of the Clipboard in Chapter 7. The Actions example used an ActionList component to implement the typical Cut, Copy, and Paste operations on the text of a Memo control. In that case the interaction with the Clipboard and the user interface updates were handled by predefined actions. In Chapter 13, the ListText example demonstrated how to write similar actions for list boxes, using the `Clipboard` global object.

The Delphi Help file documents five different formats for the `HasFormat` method: `CF_TEXT`, `CF_PICTURE`, `CF_BITMAP`, `CF_OBJECT`, and `CF_METAFILE`. These are the formats typically used by Delphi and by VCL components. The Windows API, however, defines many more formats, including the following:

```
CF_BITMAP              CF_DSPMETAFILEPICT
```

CF_OWNERDISPLAY	CF_SYLK
CF_DIB	CF_DSPTEXT
CF_PALETTE	CF_TEXT
CF_DIF	CF_METAFILEPICT
CF_PENDATA	CF_TIFF
CF_DSPBITMAP	CF_OEMTEXT
CF_RIFF	CF_WAVE

You can use these Windows formats without any particular problems, although Delphi has no specific support to retrieve these types of data. In this example, we have used two methods of the TMemo class to perform the Clipboard operations, but we could have accomplished the same effect with some of the text-related features of the TClipboard class. For instance, we can use the AsText property (used to copy or paste strings) and the SetTextBuf and GetTextBuf methods (used to handle PChar strings). The TClipboard class has specific support only for text. When you want to work with other elements, you need to use its Assign method or work with handles.

Copying and Pasting Bitmaps

The most common technique for copying or pasting a bitmap in Delphi is to use the `Assign` method of the `TClipboard` and `TBitmap` classes. As a slightly more advanced example of the use of the Clipboard, I've made a new version of the PrintBmp example shown earlier, called ClipBmp. This program can show bitmaps from a selected file or from the Clipboard, if the format is available. The structure of the form is always the same, with a TabControl component covering the whole form and an Image component inside it. The menu, however, is slightly more complex, because it now includes the commands from the Edit pull-down menu.

When you select the Edit ➢ Paste command of the ClipBmp example, a new tab named Clipboard is added to the tab set (unless it is already present), as you can see in Figure 19.16. Then the number of the new tab is used to change the active tab:

```
procedure TForm1.Paste1Click(Sender: TObject);
var
  TabNum: Integer;
begin
  {try to locate the page}
  TabNum := TabControl1.Tabs.IndexOf ('Clipboard');
  if TabNum < 0 then
    {create a new page for the Clipboard}
    TabNum := TabControl1.Tabs.Add ('Clipboard');
```

```
  {go to the Clipboard page and force repaint}
  TabControl1.TabIndex := TabNum;
  TabControl1Change (Self);
end;
```

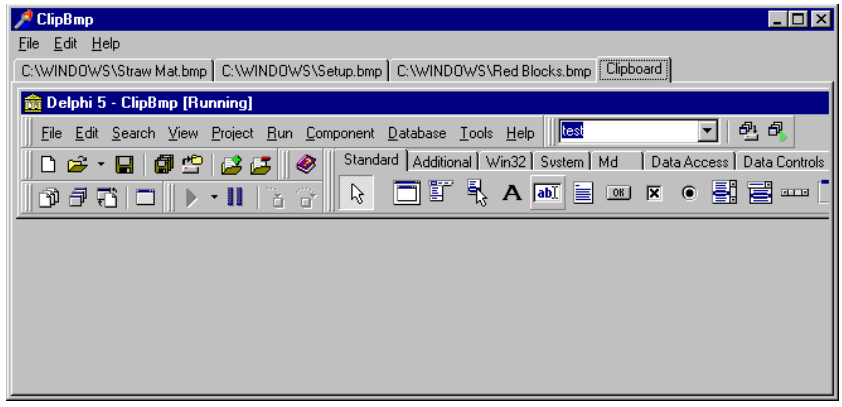


Figure 19.16: The Clipboard page of the ClipBmp example tab set shows the current contents of the Clipboard if it is a bitmap (in this case a bitmap of the Delphi main window). Image from the original book.

At the end of the `Paste1Click` method, the program calls `TabControl1Change`, the event handler associated with the selection of a new tab, which can load the bitmap from the current file or paste it from the Clipboard:

```
procedure TForm1.TabControl1Change(Sender: TObject);
var
  TabText: string;
begin
  Image1.Visible := True;
  TabText := TabControl1.Tabs [TabControl1.TabIndex];
  if TabText <> 'Clipboard' then
    {load the file indicated in the tab}
    Image1.Picture.LoadFromFile (TabText)
  else if Clipboard.HasFormat (cf_Bitmap) then
    {if the tab is 'Clipboard' and a bitmap
    is available in the Clipboard}
    Image1.Picture.Assign (Clipboard)
  else
  begin
    {else remove the Clipboard tab}
    TabControl1.Tabs.Delete (TabControl1.TabIndex);
    if TabControl1.Tabs.Count = 0 then
      Image1.Visible := False;
  end;
end;
```

Notice that if the Picture property of the Image component is still not initialized, you must create the bitmap before calling the `Assign` method. If you forget to create the

new bitmap and no graphic is associated with the picture, the `Assign` operation will fail (raising an exception). This is because the `Assign` method isn't a constructor; it is a method of an object; and if the object has not been created, you'll get an access violation exception when you try to call one of its methods.

note The `Assign` method doesn't make a copy of the actual bitmap. Its effect is to let two `TBitmap` objects refer to the same bitmap memory image and the same bitmap handle.

This program pastes the bitmap from the Clipboard each time you change the tab. The program stores only one image at a time, and it has no way to store the Clipboard bitmap. However, if the Clipboard content changes and the bitmap format is no longer available, the Clipboard tab is automatically deleted (as you can see in the listing above). If no more tabs are left, the Image component is hidden.

An image can also be removed using either of two menu commands: Cut or Delete. Cut removes the tab after making a copy of the bitmap to the Clipboard. In practice, the `Cut1Click` method does nothing besides calling the `Copy1Click` and `Delete1Click` methods. The `Copy1Click` method is responsible for copying the current image to the Clipboard, `Delete1Click` simply removes the current tab. Here is their code:

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign (Image1.Picture.Graphic);
end;

procedure TForm1.Delete1Click(Sender: TObject);
begin
  with TabControl1 do
  begin
    if TabIndex >= 0 then
      Tabs.Delete (TabIndex);
    if Tabs.Count = 0 then
      Image1.Visible := False;
  end;
end;
```

Saving the Status: INI and Registry

If you want to save information about the status of an application in order to restore it the next time the program is executed, you can use the explicit support that Windows provides for storing this kind of information. In previous versions of

Windows, the standard approach was to create an initialization (INI) file. In Windows 95, 98, and NT you can still use INI files, but Microsoft recommends using the system Registry instead. In this section, we'll review both methods of storing status information[386].

Using Windows INI Files

Delphi provides a class you can use to manipulate INI files, `TIniFile`. Once you have created an object of this class and connected it to a file, you can read and write information to it. To create the object, you need to call the constructor, passing a filename to it, as in the following code:

```
var
  IniFile: TIniFile;
begin
  IniFile := TIniFile.Create ('inione.ini');
```

There are two choices for the location of the INI file. The code just listed will store the file in the Windows directory (unless an IniOne.ini file already exists in the application directory). To be on the safe side, it is better to provide a full path to the `TIniFile.Create` constructor. We can easily extract the path from the program name, as I'll do in the IniOne example.

The format of INI files requires some explanation. These files are divided into sections, each indicated by a name enclosed in square brackets. Each section can contain a number of items of three possible kinds: strings, integers, or Booleans. If you are not familiar with the structure of an INI file, you should look at one, using any text editor, such as Windows Notepad.

The `TIniFile` class has three Read methods, one for each kind of data: `ReadBool`, `ReadInteger`, and `ReadString`. There are also three corresponding methods to write the data: `WriteBool`, `WriteInteger`, and `WriteString`. Other methods allow you to read or erase a whole section. In the Read methods, you can also specify a default value to be used if the corresponding entry doesn't exist in the INI file.

Our example, called IniOne, uses an INI file to store the location, the size, and the status (normal, maximized, or minimized) of the main form. The only real problem in this example is that the value of the state property is not always updated properly by the VCL, so we need to introduce an additional test to confirm whether the form has been minimized. The main form of the IniOne example is just a blank form,

---

386 Both techniques, INI files and the Registry, remain commonly used today on Windows.

without any components. The program handles two events: `OnCreate`, to create or open the INI file and read the initial values, and `OnClose`, to save the status after confirmation by the user. Here is the code of the first method, `FormClose`, which saves the data to be retrieved the next time the program is run:

```
procedure TForm1.FormClose(Sender: TObject;
  var Action: TCloseAction);
var
  Status: Integer;
begin
  if MessageDlg ('Save the current status of the form?',
    mtConfirmation, [mbYes, mbNo], 0) = IdYes then
  begin
    case WindowState of
      wsNormal: begin
        {save position and size, only if the state is normal}
        IniFile.WriteInteger ('MainForm', 'Top', Top);
        IniFile.WriteInteger ('MainForm', 'Left', Left);
        IniFile.WriteInteger ('MainForm', 'Width', Width);
        IniFile.WriteInteger ('MainForm', 'Height', Height);
        Status := 1;
      end;
      wsMinimized: Status := 2;
        {useless: this value is never set by VCL for the main form!}
      wsMaximized: Status := 3;
    end;
    {check if the window is minimized, that is,
    if the form is hidden and not active}
    if not Active then
      Status := 2;
    {write status information}
    IniFile.WriteInteger ('MainForm', 'Status', Status);
  end;
  {in any case destroy the IniFile object}
  IniFile.Free;
end;
```

note The main form of a VCL-based application is never minimized—when the main form receives a minimize message, it is forwarded to the application window which minimizes the entire application. `WindowState` does return `wsMinimize` value for secondary forms in the application that are minimized but not for the main form. To know whether the application is minimized, we can check if the main form is active, as in the code above.

The other method is `FormCreate`, which simply reads the saved data and restores the previous situation. Notice the initialization code, which looks for the INI file in the directory of the program, taking its name and changing the extension. Here is the complete code:

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

```
var
  Status: Integer;
begin
  IniFile := TIniFile.Create (ChangeFileExt (
    Application.ExeName, '.ini'));
  {try to read a value and test if it exists}
  Status := IniFile.ReadInteger ('MainForm', 'Status', 0);
  if Status <> 0 then
  begin
    {read position and size using current values as default}
    Top := IniFile.ReadInteger ('MainForm', 'Top', Top);
    Left := IniFile.ReadInteger ('MainForm', 'Left', Left);
    Width := IniFile.ReadInteger ('MainForm', 'Width', Width);
    Height := IniFile.ReadInteger ('MainForm', 'Height', Height);
    {set the minimized or maximized status}
    case Status of
      // 1: WindowState := wsNormal;
        // this is already the default
      2: WindowState := wsMinimized;
      3: WindowState := wsMaximized;
    end;
  end;
end;
```

This code uses a field named `IniFile`, of type `TIniFile`, which I've added to the private section to the `TForm1` class. I didn't include a figure showing the program's output, because showing you an empty form or an icon is not particularly helpful. Instead, you should simply run the program a number of times to investigate its behavior, resizing and repositioning its window each time. What I will provide is an example of an INI file generated by the program:

```
[MainForm]
Top=359
Left=567
Width=217
Height=201
Status=1
```

note Delphi uses INI files quite often, but they are disguised with different names. For example, the desktop (`.dsk`) and options (`.dof`) files are structured as INI files[387].

387 These files are no longer in use.

Using the Registry

Now we can write a similar program using the system Registry instead of plain INI files. Before we do so, I want to briefly discuss the role and the structure of the Registry. Essentially, the Registry is a hierarchical database of information about the computer, the software configuration, and the user preferences. Windows has a set of API functions to interact with the Registry; you basically open a key (or folder) and then work with subkeys (or subfolders) and with values (or items), but you must be aware of the structure and the details of the Registry. The Windows 95/98 Registry is based on six top-level keys.

Refer to specific Microsoft documentation (such as the Resource Kit) to get the details of the organization of the Registry and information about where to add your own keys. The importance of the Registry should not be underestimated. The Registry holds crucial information about the system hardware configuration, Control Panel settings, and OLE servers, and it even contains statistics about the machine. To study the structure of the Registry and examine the current values of the keys, you can use the RegEdit program.

note You can also use RegEdit to edit the values in the Registry, but you'd do better to avoid changing anything unless you are sure about what you are doing.

Delphi provides basically two approaches to the use of the Registry and supports each one with a VCL class: `TRegistry` and `TRegIniFile`. The first class provides a generic encapsulation of the Registry API, while the latter provides the interface (methods and properties) of the `TIniFile` class but saves the data in the Registry instead of using the files. This class is the natural choice for the Registry version of our last program example; by using it we won't have to make too many changes in the source code (a real advantage whenever you have existing code based on INI files). Here are the three changes you have to make to the `IniOne` program:

1. Use `TRegIniFile` instead of `TIniFile` as the class of the `IniFile` object.
2. Create a new `TRegIniFile` object, instead of a `TIniFile` object in the `FormCreate` method:

```
IniFile := TRegIniFile.Create ('IniOne.ini');
```

3. In the `uses` statement of the interface portion of the unit, replace the IniFiles unit with the Registry unit.

Easy, isn't it? With these simple changes, I've built the Registr example, which has exactly the same capabilities as the previous one but saves its data to the Registry instead of an INI file. Actually, when using the `TRegIniFile` class, Delphi adds a

new subkey with the name of the INI file under the `HKEY_CURRENT_USER` key. Instead of adding your entries directly under this root key, you should place them under the `Software` subkey and perhaps add one more level for your software company. This is the actual code of the Registr example:

```
IniFile := TRegIniFile.Create (
  'Software\Mastering Delphi\Registr');
```

We can see the effect of this code by exploring the Registry with the RegEdit application, as shown in Figure 19.17.

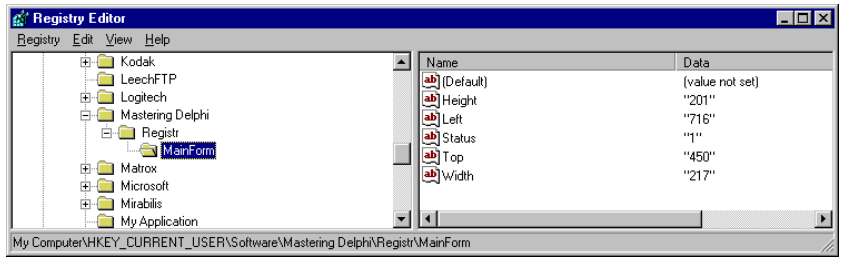


Figure 19.17: Using the Registr program, you add new entries to the registration database, as you can see by viewing the Registry with RegEdit. Image from the original book.

The `TRegIniFile` class is actually a subclass of the more generic `TRegistry` class, which has a number of methods very similar to the functions in the Registry API. These functions are not very simple to use, so I suggest that you stay with the simpler `TRegIniFile` for most cases. To use the `TRegistry` class, you need to open a key first and then access its data, including its values and its subkeys.

To show you the basic capabilities of the `TRegistry` class, I've built a very simple Registry viewer application. This program can show the structure of the Registry and list the values of the keys and items, but it doesn't display the actual data connected with the keys and items. It has only a subset of the capabilities of the RegEdit application, but I think it is an interesting example anyway.

The RegView program is based on a form with two combo boxes and two list boxes. When the application starts, a `TRegistry` object is created:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Reg := TRegistry.Create;
  Reg.OpenKey ('\', False);
  UpdateAll;
  // select the current root
  ComboKey.ItemIndex := 1;
  ComboLast.Items.Add ('\');
  ComboLast.ItemIndex := 0;
```

```
end;
```

This code opens the default root key (indicated by the backslash character) and then updates the user interface. At the end, it selects the default root key in the first combo box (more on this shortly) and adds the current element to the `ComboLast` combo box. The `UpdateAll` method simply copies the current path to the caption of the form and fills the two list boxes with the subkeys and the values of the current key (as you can see in Figure 19.18):

```
procedure TForm1.UpdateAll;
begin
  Caption := Reg.CurrentPath;
  if Reg.HasSubKeys then
    Reg.GetKeyNames(ListSub.Items)
  else
    ListSub.Clear;
  Reg.GetValueNames(ListValues.Items);
end;
```

When you select an item from the first list box (`ListSub`), the program jumps to the selected subkey. To accomplish this, simply write the following code:

```
procedure TForm1.ListSubClick(Sender: TObject);
var
  NewKey: string;
begin
  NewKey := ListSub.Items [ListSub.ItemIndex];
  Reg.OpenKey (NewKey, False);
  UpdateAll;
end;
```

This is enough to navigate the full tree. The two combo boxes add some more capabilities to the program. The first lists the possible root keys for Windows 95 and 98. When the selection changes, the corresponding contents are selected as root key of the `TRegistry` object:

```
procedure TForm1.ComboKeyChange(Sender: TObject);
begin
  case ComboKey.ItemIndex of
    0: Reg.RootKey := HKEY_CLASSES_ROOT;
    1: Reg.RootKey := HKEY_CURRENT_USER;
    2: Reg.RootKey := HKEY_LOCAL_MACHINE;
    3: Reg.RootKey := HKEY_USERS;
    4: Reg.RootKey := HKEY_CURRENT_CONFIG;
    5: Reg.RootKey := HKEY_DYN_DATA;
  end;
  Reg.OpenKey ('\', False);
  UpdateAll;
  ComboLast.Items.Clear;
end;
```

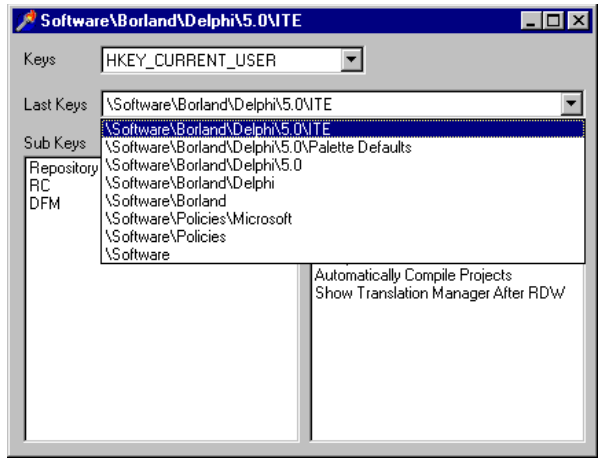


Figure 19.18: The output of the RegView example, showing keys and values of the Registry (the section with Delphi's own settings). The combo box displays a list of recently used Registry keys. Image from the original book.

After setting the new root key, the program opens its root item, updates the user interface, and empties the second combo box, Last Keys. This box stores a log, a history list of the previous selections, and can be used to navigate the tree without having to start from the root item each time. This is the code:

```
procedure TForm1.ComboLastChange(Sender: TObject);
begin
  Reg.OpenKey (ComboLast.Text, False);
  UpdateAll;
end;
```

This code is simple, but the code needed to update the list of the items of this combo box is quite complicated. Besides checking whether a path is already present, we have to add a new backslash in front of any path that doesn't have it at the beginning. The problem here is that the method just listed doesn't set the `CurrentPath` property of the `Reg` object properly, even though it produces the correct effect. To make further selections from that path (once it is added to the list), we need to correct it first. Considering all these issues, here is the final version of the `ListSubClick` method:

```
procedure TForm1.ListSubClick(Sender: TObject);
var
  NewKey, Path: string;
  nItem: Integer;
begin
  // get the selection
  NewKey := ListSub.Items [ListSub.ItemIndex];
  Reg.OpenKey (NewKey, False);
  // save the current path (eventually adding a \)
```

```
  // only if the it is not already listed
  Path := Reg.CurrentPath;
  if Path < '\' then
    Path := '\' + Path;
  nItem := ComboLast.Items.IndexOf (Path);
  if nItem < 0 then
  begin
    ComboLast.Items.Insert (0, Path);
    ComboLast.ItemIndex := 0;
  end
  else
    ComboLast.ItemIndex := nItem;
  UpdateAll;
end;
```

Accessing Properties by Name

As you know, the Object Inspector displays a list of an object's published properties, even for components you've written. To do this, it relies on the RTTI information generated for published properties. Using some advanced techniques, an application can retrieve a list of the published properties of an object and use them.

Although this capability is not very well known, in Delphi it is possible to access properties by name simply by using the string with the name of the property and then retrieving its value. Access to the RTTI information of properties is provided through a group of undocumented subroutines, part of the TypInfo unit[388].

note The reason these subroutines are not documented is that Borland wants to be free to change them in future versions of Delphi. In the past (from Delphi 1 to Delphi 4) they've changed just a little. Delphi 5's TypInfo unit provides many more goodies, and this can cause a few incompatibilities. If you use anything from TypInfo, be aware that you might need to update your code for future versions of Delphi.

Rather than explore the entire TypInfo unit here, we will look at only the minimal code required to access properties by name. Prior to Delphi 5 it was necessary to use the `GetPropInfo` function to retrieve a pointer to some internal property information and then apply one of the access functions, such as GetStrProp, to this pointer. You had also to check for the existence and the type of the property.

388 Later Delphi added a much more sophisticated, comprehensive, and easy to use RTI system. You can find the classes and interfaces in the `System.RTTI` unit. The core, traditional RTTI remains the foundation for properties and streaming and is still applicable, even if less used.

Delphi 5 introduces a new set of TypInfo routines, including the handy `GetPropValue`, which returns a variant with the value of the property, or `NULL` if the property doesn't exists. You simply pass to this function the object and a string with the property name. A further optional parameter allows you to choose the format for returning values of properties of the set type.

For example, we can call

```
ShowMessage (GetPropValue (Button1, 'Caption'));
```

This call has the same effect as calling `ShowMessage`, passing as parameter `Button1.Caption`. The only real difference is that this version of the code is much slower, since the compiler generally resolves normal access to properties in a more efficient way. The advantage of the run-time access is that you can make it very flexible, as in the following RunProp example.

This program displays in a list box the value of a property of any type for each component of a form. The name of the property we are looking for is provided in an edit box. This makes the program very flexible. Besides the edit box and the list box, the form has a button to generate the output and some other components added only to test their properties. When you press the button the following code is executed:

```
uses
  TypInfo;

procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  Value: Variant;
begin
  ListBox1.Clear;
  for I := 0 to ComponentCount -1 do
  begin
    Value := GetPropValue (Components[I], Edit1.Text);
    if Value <> NULL then
      ListBox1.Items.Add (Components[I].Name + '.' +
        Edit1.Text + ' = ' + string (Value))
    else
      ListBox1.Items.Add ('No ' + Components[I].Name + '.' +
        Edit1.Text);
  end;
end;
```

You can see the effect of pressing the Fill List button while using the default *Caption* value in the edit box in Figure 19.19. You can try with any other property name. Numbers will be converted to strings by the variant conversion. Objects (such as the value of the `Font` property) will be displayed as memory addresses.

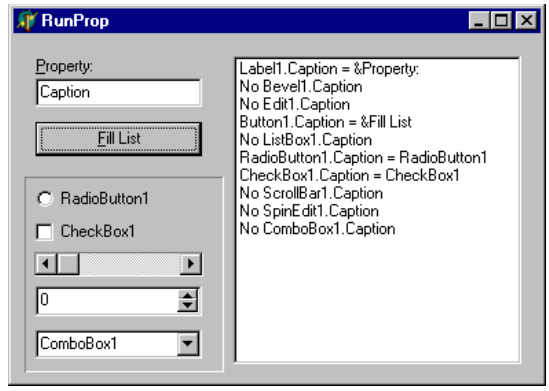


Figure 19.19: The output of the RunProp example, which accesses properties by name at run time. Image from the original book.

Do not use regularly the TypInfo unit instead of other property-access techniques. Use base-class property access first, or use the safe `as` typecast when required, and reserve RTTI access to properties as a very last resort. Using TypInfo techniques makes your code slower, more complex, and more error prone; it skips the compile-time type-checking; and it makes the code less portable to future versions of Delphi.

Building Online Help[389]

Unless your applications are very simple, you'll usually want to include some form of online help to answer questions that arise while users are working with your software. Windows provides support for online help at the operating-system level, but you must still do a fair amount of work to provide help for your applications. To simplify this task, you can use several third-party products, such as ForeHelp, RoboHelp, and several others[390]. However, you can also create online help using the tools that ship with Delphi, along with a word processor that can generate RTF-format text files. We'll review the basic steps required for creating online help this way, but keep in mind that the third-party tools generate the same files and do the same things. They also automate many of the repetitive and error-prone elements of the process.

389 Not much of this section still applies as the help system format Microsoft and Delphi use has changed, and most of the tools of the old time no longer exist

390 I doubt these tools for building help files still exist today.

There are three main files that you'll need to generate before you can create an online help (HLP) file: a project file, a contents file, and the help text file itself. Once you have these three files, you'll then use Microsoft's Help Compiler to bring everything together as an HLP file. The Microsoft Help Workshop[391], which ships with Delphi, can be used to create the project and contents files if you're not going to use a third-party tool. Using Help Workshop and MS Word (or another RTF-generating word processor), you're ready to create online help files. Let's consider this process in four stages: creating the help text file(s), creating the Contents file, creating the Project file, and mapping/linking the help topics to the elements of your Delphi application.

As I mentioned above, to create the help text files, you'll need a word processor that can generate RTF-format text files, such as WordPerfect or MS Word. When you begin creating the text, you can put the text in multiple files (menu help in `Menu.rtf`, dialog box help in `Dialog.rtf`, and so on), or you can put it all in a single file. Below are my suggestions for the steps to follow in creating the help text:

- Create a help page for each form and dialog box. Title each page with the name of the form or dialog box.
- Create a help page for each menu item that doesn't display a dialog box. Title these pages with the name of the menu, followed by the name of the menu item, separated by the "pipe" character (as in Edit | Copy).
- Create "pop up" help pages for terms that may be unfamiliar to the user. These pages don't need a title, because you'll present the text to the user in a pop-up window adjacent to the unfamiliar term.
- Create "pop up" help pages that contain links to related topics, such as command-line options.
- Create help pages for common tasks, and create links to the pages for the menu items, forms, and dialog boxes those tasks require.
- Create tutorial pages as appropriate.
- Create a help page that summarizes the product's use and main features.
- If the product is a new release of an existing product, create a help page that summarizes the new features.

---

391 This is no longer available, and Delphi moved to a different help file format (still from Microsoft)

Now let's consider how to format a basic help page. Most help pages contain the following elements (consult the `HCW.HLP` file for more detailed information on formatting help pages):

- The topic title text, typically boldface, sometimes a different color than the body text, and usually 14-point type or larger.
- The context string for the page, entered as a # custom footnote to the topic title. This is the string that the help system uses to uniquely identify this help page. It must be unique, and it cannot contain spaces or punctuation characters.
- The search text string, entered as a $ custom footnote to the topic title. This is the string the user will see in the Search dialog, the History list, and the Bookmark menu. You'll usually want this text to match the topic title text.
- The keyword strings for this topic, entered as a K custom footnote to the topic title. These are the strings that will appear in the Index tab. You can enter more than one keyword string for each topic, but you must separate the strings with a semicolon. To display various subheadings in the index under a given main heading, enter the heading as the first word of the string and the subheading as the second, and separate them with a comma.
- Links to other help pages, entered as text that's double-underlined and followed immediately by a context string, formatted as hidden text.
- Links to pop-up help pages, entered as text that's single-underlined and followed immediately by a context string, formatted as hidden text.
- A forced page break. All help pages must end this way.

In addition, you may want to be aware of the following as you format your help pages:

- To create a nonscrolling region at the top of a help page, use the Keep With Next paragraph style for the title and any links (such as a "Related Topics" pop-up link) that you want to display in the nonscrolling region.
- You'll generally want to use Arial for most help text, as it's easily readable in most font sizes and is a standard font that will be on every system.
- You can link to topics in other HLP files. If you know the context strings in the other HLP file, you can link to those topics using a standard link, but you must format the context string differently. Instead of just entering the context string as the hidden text, you'll need to enter the HLP filename, an asterisk, and then the context string (no spaces in between), all as hidden text. If you don't have access

to the context strings, you'll need to use the JumpKeyword macro, documented in the Help Workshop's online help.

At this point, you should have created all your help text, formatted as individual help pages. Figure 19.20 shows a sample RTF-formatted help text file, with several brief help pages.

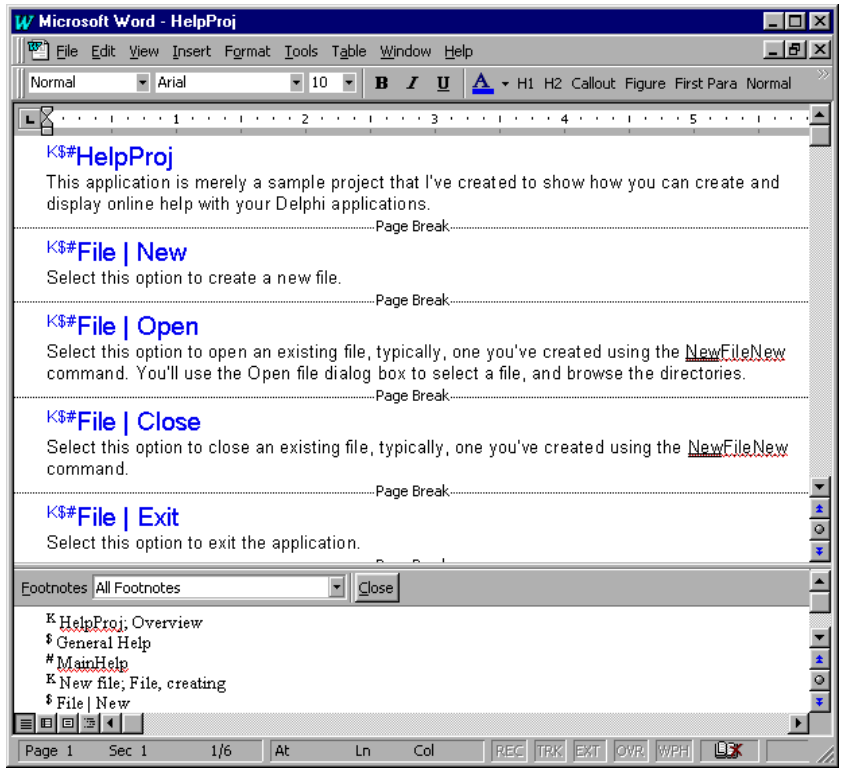


Figure 19.20: A simple, RTF-formatted help text file. Image from the original book.

Now let's consider how you generate the Help Contents file. The Contents and Project files aren't binary files, they're text files. Even so, it's much easier to use the Help Workshop to format and maintain them.

In Help Workshop, you'll create a new Contents file, which requires that you specify the HLP filename, the name of the main help window type, and the default title that you want to appear at the top of the help window. Then, you create the headings and topics that will appear in the Contents tab of the Contents/Index/Search window. For each topic entry, you'll specify first the topic text and then the context string of

the help page you want to associate with that topic. You can see an example of a Contents file being edited in Help Workshop in Figure 19.21.

Figure 19.21: A Contents file, as it appears in Help Workshop. Image from the original book.

When you're ready to create the Project file, you can create it from within Help Workshop, as well. In the Project file, you'll specify the following critical information:

- The RTF filenames for the files that contain the help text pages
- The filename of the Contents file
- The name of the resulting HLP file
- Compression techniques to use, if any
- A map of corresponding context strings and context ID
- The size, position, color, and appearance of the help window

Now let's consider the last two pieces of information in a bit more detail.

A context ID number map is necessary for displaying help information for menu items, forms, or components. When you create the context string and this number

map, you'll want to use a standard technique for assigning these numbers. I suggest you use the following numbering system:

- Map the context string for the summary or overview help page to both ID 0 and 1.
- Map the context string for the new features help page (if one exists) to ID 2.
- Use numbers from 3–99 as ID numbers for pop-up, related topic, common task, and tutorial help pages.
- Use 100–199 as ID numbers for the first menu, its menu items, and the resulting dialog boxes. For example, ID 100 would describe the menu and provide links to the help for each menu item. ID 101 would describe the first menu item, 102 the second, and so on. You'd then use 130–199 to describe the dialog boxes that appear for the first menu's items.
- Use 200–299 as ID numbers for the second menu, its menu items, and the resulting dialog boxes. Number the remaining menus, menu items, and dialog boxes accordingly.

Using this technique, you can quickly assign the ID numbers to the context strings and keep the help pages organized more easily.

As you may have noticed, different applications display the online help file in different initial locations. You may want to do this to make sure that the online help doesn't obscure an important form or window, or you may want to make the help window as large as possible. In addition to specifying the window's size and position in the Project file, you can also determine the background color for the text (as well as for nonscrolling regions, if you create them). To enforce these settings on the default help window, create a new window style named "Main" and apply the changes there.

Once you've created the Project file and specified the appropriate parameters, you can compile and test the HLP file. Help Workshop displays error messages and a detailed report on this process, informing you of the number of topics, links, keywords, and bitmaps your HLP file contains. You can also run WinHelp directly from Help Workshop, which allows you to test the HLP file before you proceed to the last step, linking the help topics to specific elements in your Delphi application.

Linking your Delphi application to the HLP file is actually quite simple; it consists of just a few steps:

- Specify the HLP filename in the application's Project Options dialog box.
- Set the `HelpContext` property of the main form to 1.

- Set the `HelpContext` property of other components to the appropriate values, based on the ID mapping assignments you made in the Project file.
- If you haven't created a help page for a particular component, menu, or menu item, leave the `HelpContext` property at 0, because this value is mapped to the product summary topic.
- Use the standard actions related to the help (`THelpContenst`, `THelpTopicSearch`, and `THelpOnHelp`) to implement a Help menu in a few seconds. These help actions are new to Delphi 5.

That's it! Once you've tested the application, written the online help, and tested the help system (and then retested the application several times), you're ready to create an installation program.

InstallShield Express[392]

Almost every commercial application uses some form of setup program to manage the installation and configuration of the application before its first use. Among the many good utilities on the market for creating setup programs, one of the most popular is InstallShield. Delphi ships with a feature-reduced version of this tool, known as InstallShield Express. Even though complete treatment of this topic would require a small book, you can create simple setup programs quite easily using InstallShield.

There are many advantages to using a third-party tool like InstallShield to create your setup program, and as the complexity of your application increases, so does the value of this type of tool. For example, if your application requires the BDE, or any of the BDE drivers, you simply pick and choose which of these components you need, and InstallShield handles the messy details. Similarly, if you're developing a new version of a complex application that spreads code across several DLLs, InstallShield can create a setup program that detects version information in an existing installation and updates the files only as necessary. Need to install one set of files for Windows 95 or 98 and another for Windows NT? You can do that, too. Let's quickly review how you use InstallShield Express to create a setup program.

---

392 This tool no longer ships with Delphi, while it still exists. Most of this section is no longer applicable. I haven't use the tools since the time, and have no idea if any of the feature described here are still actual today.

When you use InstallShield Express to create a new setup program, you'll begin by specifying three important pieces of information: the project's name, the sub-directory where you'll store the project's files, and whether or not you want to specify a custom setup in your installation. You'll enter this information in the New Project dialog box. In particular, you should note the Include a Custom Setup Type check box, as making the correct choice is important. If you don't select this now, you won't be able to add a custom setup later.

Once you create a new InstallShield Express project, you'll see a Setup Checklist, visually hosted in a legal notepad, as shown in Figure 19.22.

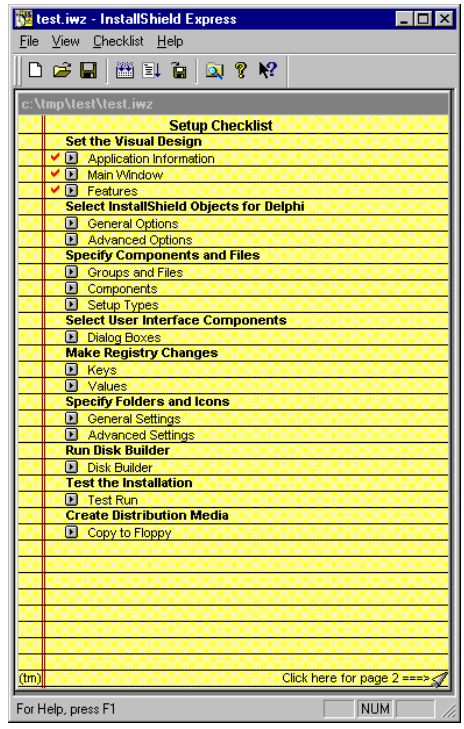


Figure 19.22: The user interface of InstallShield Express is based on a notepad metaphor. Image from the original book.

Although it's natural to specify these items in the order they appear, you can generally perform them in any order (except for building, testing, and deploying the setup application, which you must naturally perform last). Each section contains one or more checklist items, each of which corresponds to a tab in a dialog box for that section. Briefly, let's examine each of the main sections of the checklist, and review some of the actions you'll need to take.

The first section, Visual Design, contains three items, and the Set the Visual Design dialog box will display three corresponding tabs. In the App Info tab, you'll specify the name of your setup application, the executable's path and filename, the version, and the name of your company. On the Main Window tab, you'll specify the title of the setup program, a bitmap logo, and the background color of the setup program's main window. On the Features tab, you'll specify whether InstallShield Express should create an uninstall option for you. (I recommend that you use this option, because there's almost no penalty for doing so.)

After you've entered the appropriate data in this dialog box, click OK to save the data. When the Setup Checklist window reappears, you'll notice a check mark beside each item in the Visual Design section. These check marks simply remind you which items you've entered.

The second section in the Setup Checklist is titled Select InstallShield Objects for Delphi; it includes sections such as General and Advanced. You'll use the resulting dialog box to select which of the optional Delphi components or accessories you'd like to install. For example, if you're creating a setup program for a database application, you'll no doubt want to install the Borland Database Engine (BDE), which can be somewhat complex, even for otherwise simple applications. In this dialog box, you can not only choose whether or not to install the BDE, you can also pick and choose which portions of the BDE you'd like to install.

The third section, Specify Components and Files, contains three items, which once again means three sections in the corresponding dialog box. In this dialog box, you'll specify Groups and Files, Components, and Setup Types. Because the different meanings of these terms may not be obvious, I'll explain them briefly:

- A *File Group* is a logical set of files that the setup program must install into a specific directory. For instance, you'll typically want to place all your online help files in the same directory. While this is generally the same directory as the application, it may not be. You'll separate these files from the others in the Program Files group, because there may be situations where you don't want to install them. When in doubt, create a new file group for one or more files when installing them is optional.
- A *Component* is a set of one or more file groups. Where the file groups were distinguished based on the destination directory, you'll distinguish components based on their logical function. For example, if you're not going to install some optional online help files, you probably won't want to install the related tutorial files that you've prepared. By creating a component that contains the supplemental help file group and a tutorial file group, you can specify that you don't want to

install them as a set. However, even though they're logically grouped as a component, if you install them, the setup program will respect the destination directories you specified for the file groups.

- A *Setup Type* is a logical group of components. By default, the three standard setup types will install all components, but you can customize this to eliminate a component (that is, a set of file groups) from a specific setup type.

The next section in the Setup Checklist window is titled Select User Interface Components. By selecting the Dialog Box item, you can specify which of the standard setup program dialog boxes you'd like to display to the user. Here's a list of the dialog boxes you can choose from:

Welcome Bitmap: Although it's not used by default, you can display an opening bitmap image when your setup program starts.

Welcome Message: Choose this to display a standard text greeting.

Software License Agreement: Select this option to display the text from your own license agreement.

Readme Information: Same as the license agreement, but here you'll display "readme" information.

User Information: User enters name, address, and optionally, a product serial number.

Choose Destination Information: User specifies a target disk drive and/ or directory.

Setup Type: User selects between Typical, Compact, or Custom (the Custom option will be available only if you specified for InstallShield to generate a custom setup).

Custom Setup: If Custom was selected in the Setup Type option, this dialog box allows the user to pick which components they'd like to install.

Select Program Folder: Determines the default folder name that will contain the application and its support files.

Start Copying Files: Begins the installation process.

Progress Indicator: Displays a visual image that updates the user on the status of the setup process.

Billboards: Use this option to display one or more BMP files while the user is running the setup program.

Setup Complete: Here, you'll select the nature of the message that appears when the user has finished setting up the application.

You can choose to display all of these dialog boxes, none of them, or any combination you like. The setup program will present them to the user in the order that they appear in the dialog box. Figure 19.23 shows the Select User Interface Components dialog box.

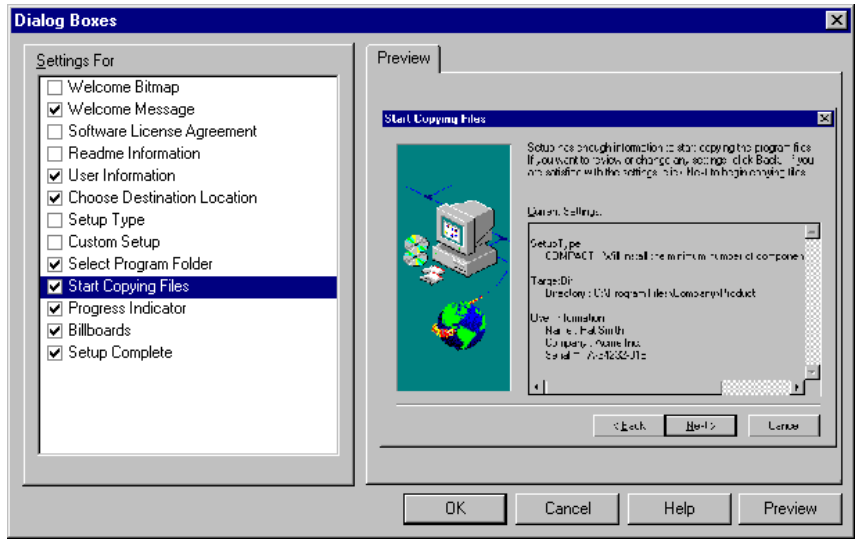


Figure 19.23: The Select User Interface Components dialog box. Image from the original book.

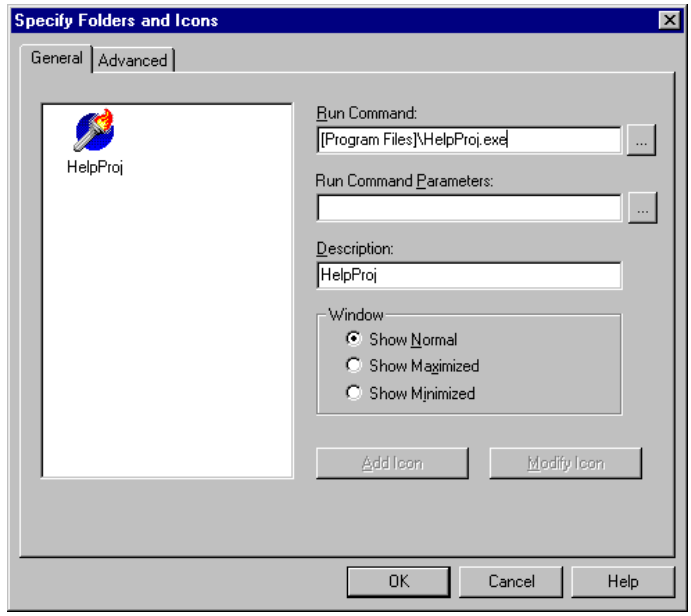
The next section in the Setup Checklist is titled Make Registry Changes. This section can be simple to use, but its effects are far-reaching, because you're making changes directly to the System Registry. (The most common reason for adding Registry entries is to create associations between various file extensions and your application.) Therefore, use caution when adding Registry data or values. By default, InstallShield creates several Registry entries for you, primarily to register the full application path for correct launching from the Start menu.

The last section of the Setup Checklist before you begin generating the setup program is titled Specify Folders and Icons. You'll use this section to specify the command-line parameters for the application (either the EXE path and filename, as well as command-line parameters, or the path and filename of a default document), its appearance in the Start menu, and the other properties of its Start menu icon (such as if the application should launch in a minimized, maximized, or normal window). You can see the Specify Folders and Icons dialog box in Figure 19.24.

Now you're ready to build your setup program. The next section of the Setup Checklist, called Run Disk Builder, brings together all the settings and configurations from the previous checklist items, merges them into compressed data files, and then

generates one or more disk images (files that represent the actual contents of the installation disks). This is where you actually begin building your setup program. In the Disk Builder dialog box, you choose the distribution medium for your setup program, and then click the Build button. From that point, you can sit back and monitor the process, noting any error messages or warnings that InstallShield displays. If errors occur, you can make changes to the checklist items and then rebuild the setup program. Figure 19.25 shows the Disk Builder dialog box after building a setup program that contains several errors.

Figure 19.24: The Specify Folders and Icons dialog box. Image from the original book.



Once you've built your setup program and eliminated any errors that the Disk Builder dialog box reported, you're ready to begin testing the setup by actually executing the setup program. Fortunately, InstallShield allows you to do this *before* you generate the actual disks. When you click the Test Run item in the checklist, InstallShield will launch the program, and you can begin testing various options in your installation. If you wish to test different setup types at this stage (before you've actually created the installation disks), you'll want to uninstall the software, via the Add/Remove Programs tool in the Windows Control Panel. Figure 19.26 shows the new setup program in action.

After you've tested your setup program and determined that it's working correctly, you're ready to build the deployment disks. This is the last item on the Setup Check-

list, and as you'd expect, it's the last task you'll perform in preparing the setup program prior to final testing and duplication.

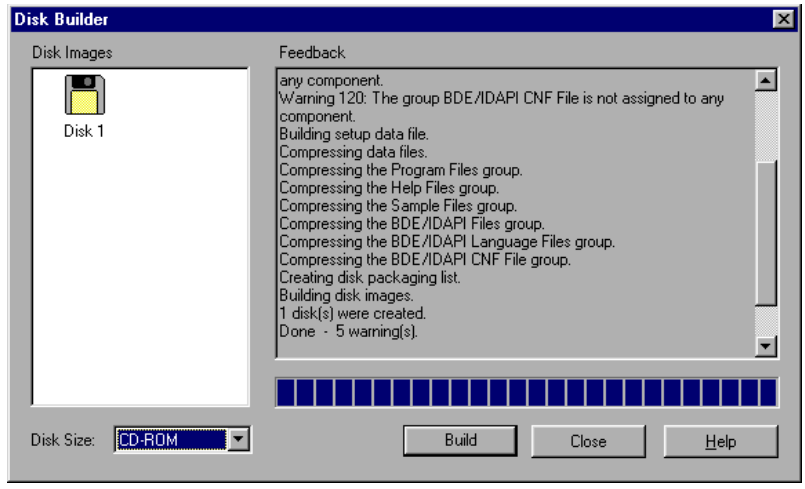


Figure 19.25: The Disk Builder dialog box. Image from the original book.

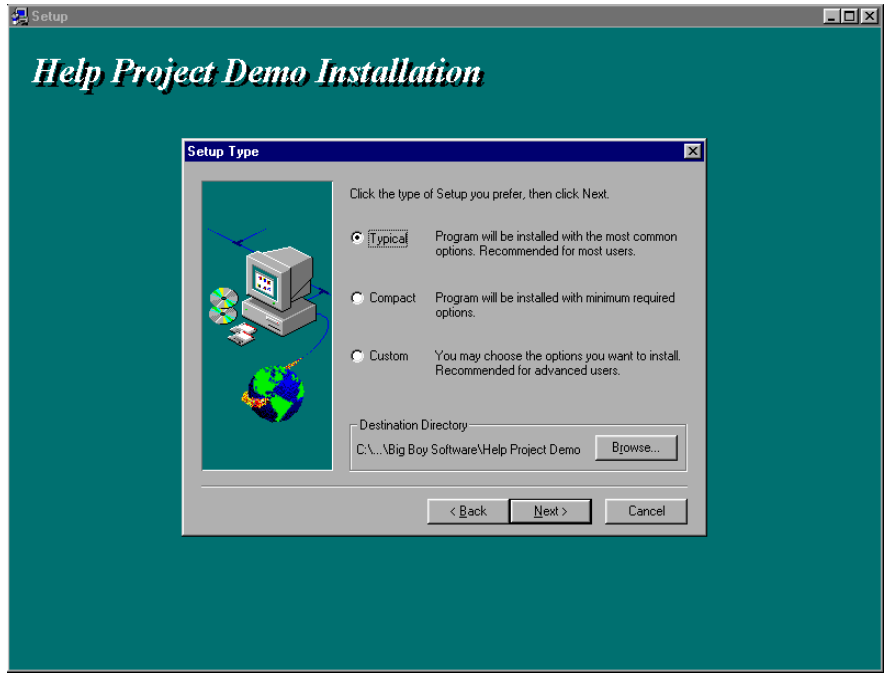


Figure 19.26: The setup program for the Help Project example. Image from the original book.

Managing Source Code[393]

When working on a large project or in a team of developers sharing source code, do not work on the source code files and simply replace older versions with new ones. This standard procedure, in fact, can produce problems when you want to revert to a previous version of the source code or when different programmers need to work on the same files or need to know what other programmers have changed.

Delphi 5 provides a built-in solution to solve these types of problems as well as version tracking in general[394]. The tool is called TeamSource, and it must be installed separately from the Delphi environment (note that TeamSource is available only in

393 The concept of managing source and, source code control and related has grown in relevance. Over the years Subversion emerged at the top tool, later replaced by Git. Whichever tool you use, it's fundamental to use one, even if you are coding as a single developer.

394 Today the Delphi IDE offer integration for Subversion and Git.

the Enterprise version of Delphi or as a separate buy). Borland defines TeamSource as[395] "a workflow management tool," which is actually a good description of what the tool can do for you (even in a single user situation). TeamSource uses a version control system behind the scenes for storing and retrieving shared files. You can use any version control system, called *controller,* including the simple `Borland.zlib`, included in the tool.

note You can use TeamSource for your Delphi development and also for any other programming language or set of ASCII-based files. (Files must be at least text-based if you want to be able to compare the differences and resolve conflicts). For example, you can use TeamSource to manage the HTML files of a (large) Web site.

TeamSource is a rather complex tool, with many options and features, so I will simply give you a summary of its capabilities. The basic idea is that each user will have his or her own version of the source code, can make changes, and then can reconcile the changes back to the shared *remote* repository. This is called *parallel version control,* because many programmers can modify the source code files at the same time. Each programmer works on a local copy, as though he or she were the only developer on the project. Once in a while the user can reconcile the local files with the remote image, upload the files the user changed, and get a new copy of the files other programmers have worked on.

This process, however, is not simply a file replacement. TeamSource keeps track of the file differences, stores all the past versions of the source code files, keeps a log of the messages every user has added to the system, forwards files and messages by e-mail, and more. There are some operations, such as reconciling files, that can be done only by one user at a time. When you perform an operation like that, TeamSource will automatically place a lock on the entire project (two users cannot reconcile their own differences at the same time). These automatic locks expire after a short time, unless you extend them.

TeamSource's main window uses an Outlook-type menu bar, where you can select the various areas: remote, local (shown in Figure 19.27), project history, and general settings. The local section is where you operate on files on the local computer, which are kept up to date by the reconciliation process. TeamSource compares local information with remote shared versions and then displays in this view the changes, suggesting how to reconcile them (for example, uploading a change to the shared repository or getting an update that another developer has done). From the local view, you can see the difference between your version and the one that was last checked in (choosing one of the available versions).

395 Borland TeamSource no longer exists.

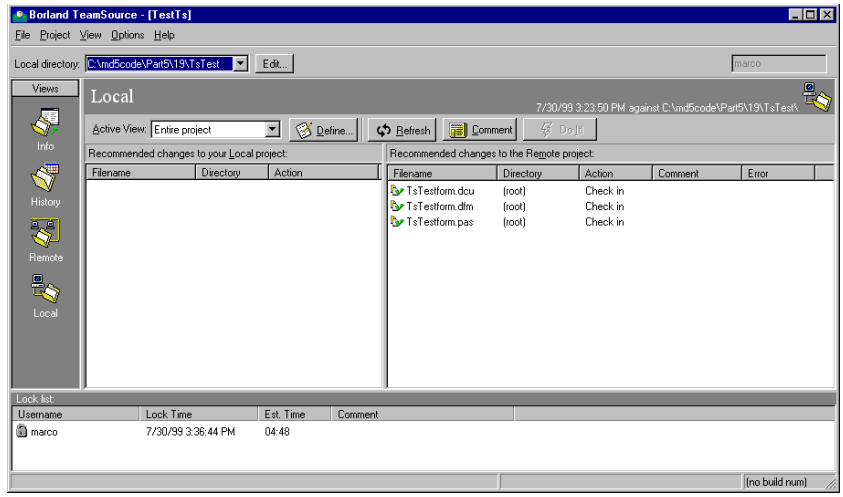


Figure 19.27: TeamSource main window, with the Local view. Image from the original book.

Contrary to other version control systems, TeamSource allows you to check in your files but not check them out. You don't need to let the system know that you are working on a file (that is, lock the shared file to prevent others from modifying it), because when you check in, TeamSource will figure out what changes you made and how to merge those changes into the shared repository on the server. TeamSource is able to perform this magic by using a "three-way reconciliation" algorithm that compares the source file at the time you last copied it from the server, the modified source file now on your local machine, and the source file currently on the server. With these three data points, it can figure out not only what changes you have made but also what changes have been made to the shared file and how to merge those together without loss of information. The only time you have to manually merge changes in TeamSource is when you and another developer modify the same source line in the same source file at the same time. In this case, the first person to check in will merge without incident, but the second person to check in will be informed of the "collision" and prompted to merge the changes by hand. As with optimistic record locking in databases, these check-in collisions become less and less likely as the size of the project increases.

The best part about TeamSource is that this magic lets many developers work on the same project without getting in each other's way. You simply work on the local copy of the files with Delphi, in the standard way. This makes TeamSource much less intrusive than many other version-control systems.

The remote view gives you the state of the project. In this view, you can see all the changes to the project, do revisions, and compare files in the history list (that is, the

files that have been checked in by the various developers over time). The file comparison is based on the source code and displays the differences, as you can see in Figure 19.28. You can see the history list and all the comments logged by the various users, including the reasons for changes and for acquiring locks. Another important concept is that of *productions*. A production indicates to the system that a given file is the result of the compilation of another one. For example, a DCU file is the production of a PAS file, and a RES file is the production of an RC file.

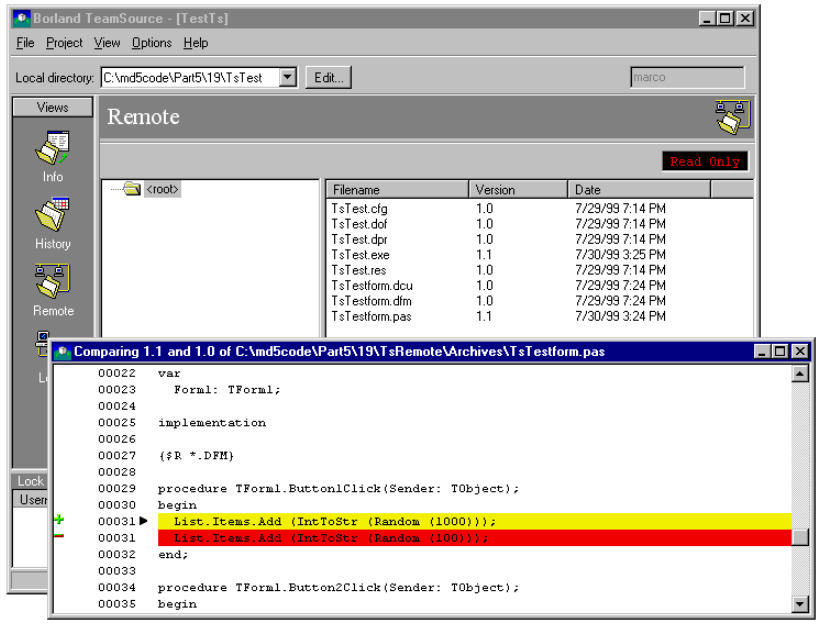


Figure 19.28: The source code comparison window of TeamSource; on the background, the Remote view Image from the original book.

When you need to create a snapshot of the project for later use, you can create a bookmark. Afterward, you'll be able to make a local copy of the project (pull-in) of any of the bookmarks. Creating a bookmark is a way to save the state of your project at a specific time and date. When pulling project files, you can tell TeamSource to pull the project files in the state they were in at the time and date of a particular bookmark. By the way, watch out when you do a pull, as the local changes you have not checked might get lost. There can be local bookmarks, visible only to the developer who defined them, and global ones, set up by the administrator.

Another very common notion of version control systems is that of a branch. Consider the case of a development process between two different programmers or subteams. One team might be fixing bugs for a release, while the other might be adding features that are not going to be immediately available to the users. This

would not be possible if everyone checks the same code base. However, TeamSource allows you to replicate the source code in two different directories, locally and remotely. In this case, every user will update the local changes to one of the two directories, containing the different branches. After some time, when the two groups need to resynchronize their separate efforts, they'll use TeamSource to synchronize between the branch directory and the original one.

There is much more to TeamSource than this short introduction suggests. You can produce autonomous local copies for experiments (pulling in the project file you need), rollback to previously saved versions, and so many other operations. The only suggestion I can give you is to try for yourself. But its value is worth the effort.

What's Next?

In this chapter, we have examined some details about the role, definition, and use of Windows resources in traditional applications and demonstrated how to use them in Delphi programming and for localizing applications. Then we examined some techniques related to printing, manipulating files, and using the Clipboard. Next, we reviewed Delphi's support for both INI files and the system Registry, and we tried out some related techniques. Finally, we discussed Delphi's online help, installation tools, and TeamSource, giving a brief overview of their use.

This chapter ends our discussion of real-world Delphi techniques. With the next chapter, we start exploring another advanced topic that has plenty of specific support in Delphi: Internet and distributed programming.

Chapter 20: Internet Programming

In this chapter, I'll provide an introduction to Internet and Web programming in Delphi, using some of the components available in the IDE. With the advent of the Internet era, writing programs for the World Wide Web has become commonplace[396].

We'll start by looking at HTML files and building a couple of HTML generators. The next step will include the coverage of ActiveForms. Then we'll go on with the use of Delphi socket components, other Internet components, and techniques you can use for automated e-mail processing.

---

396 While this indication was in the right direction, the growth of web technologies in general has been so extensive and pervasive it would have been difficult to anticipate in its full extension.

Finally, we'll focus on the server side, particularly the development of server extensions based on the Common Gateway Interface (CGI), ISAPI, and Active Server pages (ASP). Of course, I'll try to focus on database publishing on the Web using the specific components and tools provided in Delphi. Notice that most of these components are part of the WebBroker technology, available only in Delphi Enterprise or as a separate add-on.

note To test some of the examples in this chapter, you'll need access to a Web server. The best test bed is probably the use of a server under Windows NT or Windows 2000, but you can try the examples with Microsoft's Personal Web Server, as well, which is included in Windows 98[397].

HyperText Markup Language (HTML)

The HyperText Markup Language, better known by its acronym HTML, is a very widespread format for hypertext on the Web. HTML is the format Web browsers typically read. HTML is a standard defined by the W3C, the World Wide Web Consortium, which is one of the bodies controlling the Internet. The current standard is represented by HTML 4, although not all of the browsers fully support it[398]. When building a Web site, you always need to choose a lowest common denominator approach to support most of the browsers in use, that is, unless you are targeting a specific group of users who you ask to adopt a specific browser (as happens in Intranet situations). If you don't know much about the tags included in HTML files, you may want to read the sidebar "The Format of HTML Files" for a fast introduction.

On the client side of the Web, the main activity is browsing—reading HTML files. We've already seen in Chapter 16 how you can write a simple customized browser by embedding Microsoft Internet Controls into your application (that is, using the WebBrowser component available in the Internet page of Delphi's Components palette).

397 Nowadays, Microsoft IIS is available in all editions of Windows, but you can also install a local copy of the Apache server or build a stand along HTTP server in Delphi, using Indy components.

398 The current situation is very different, with most browsers adopting a single engine (Chromium) and all of the others trying to conform to it. While browser differences exists, they are not the issue they used to be at the time I wrote this book.

You can also activate directly the browser installed on the computer of the user, for example, opening an HTML page by calling the `ShellExecute` method (defined in the ShellApi unit):

```
ShellExecute (Handle, 'open',
 FileName, '', '', sw_ShowNormal);
```

Using `ShellExecute` we can simply execute a document, such as a file. Windows will start the program associated with the HTM extension[399], using the action passed as the parameter (in this case, *open*). You can use a similar call to view a Web site, by simply using a string like *'http://www.borland.com'* instead of a filename[400]. In this case, the system recognizes the *http* section of the request as requiring a Web browser, and launches it.

On the server side, you generate and make available the HTML pages. At times, it may be enough to have a way to produce static pages, occasionally extracting new data from a database table to update the HTML files as needed. In other cases, you'll need to generate pages dynamically based on a request from a user. I'm going to write a couple of examples covering the first case in this chapter, but I'll defer the dynamic generation to the next chapter.

The Format of HTML Files

If you have a little familiarity with HTML but don't work with it often enough to have all the basic elements "down cold," here's a quick summary.

HTML files are basically ASCII text files. Besides plain text, an HTML file contains many tags, which might determine the style of the font, the type of paragraph, or a link to another HTML file or an image, among other things.

Most tags are paired as opening tags and closing tags (the closing tag is usually the same as the opening tag but is preceded by a / slash symbol) to indicate where the style begins and ends. For example, you write `<b>important</b>` to set the word *important* in bold, and you write `<title>Document title</title>` to set the title of a document.

Some tags, however, have no closing version (or "termination"). The `<p>` tag, used to separate paragraphs, is one of these. The <p> tag is a particularly important one, because the line spaces and new-line characters in an HTML file are totally ignored. Only by using a `<p>` or `<li>` tag, or by starting a new heading, will you move the following text to a new line.

---

399 Or the HTML extension, more common now.

400 Beside favoring https over http, notice that the domain listed here currently doesn't respond any more. Today, I'd have used as a demo *'https://www.embarcadero.com'*.

An HTML document begins with the `<html>` tag and is divided into two parts, marked as `<head>` and `<body>`. Each of these three tags requires the corresponding terminator. In the head portion of the HTML file, you'll generally write the title (often displayed in the title bar of the browser) and a few other generic elements.

In the body, you write the contents of the file, generally starting with its visible title. You can have several headings with different levels, marked with the `<hX>` tag, where you'd replace `X` with a number from 1 to 6. These are followed by plain paragraphs (`<p>`), preformatted paragraphs (`<pre>`, a style generally used for program listings), various types of lists, and many other elements. The text will often have links to other pages or other parts of the current page, using the `<a>` tag.

Another relevant element of HTML is tables. The `<table>` and `</table>` tags indicates the beginning and the end of the table, and its optional `border` attribute displays borders with a given width. The `<tr>` and `</tr>` tags introduce and close each row, and the tags `<th>`, `</th>`, `<td>`, and `</td>` indicate a table header cell and a table data cell, respectively. The number of columns depends on the items in each row. Different rows, in fact, can have a different number of items.

HTML is the subject of many books (from Sybex and other publishers), and you can find dozens of HTML tutorials just by browsing the Web.

Delphi's HTML Producer Components

If your version of Delphi includes the HTML producer components (available on the Internet page of the Components Palette), you can use them to generate the HTML files and particularly to turn a database table into an HTML table[401]. Many developers believe that the use of these components makes sense only when writing a Web server extension. Although they were introduced for this purpose and are part of the WebBroker technology, you can still use three out of the four producer components in any application in which you have to generate a static HTML file.

Before looking at the HtmlProd example, which demonstrates the use of these HTML producer components, let me summarize their role:

- The simplest of the HTML producer components is the PageProducer, which manipulates an HTML file in which you've embedded special tags[402]. The advan-

401 These components have long been available in each version of Delphi, but they are fairly useless by today's HTML standards.

402 Recently (in RAD Studio 12.2) Embarcadero introduced an HTML template engine called WebStencils, which is the first full replacement for PageProducer and it compatible with the original produce interface (so it's a plug-in replacement). The scripting features are extensive.

tage of this approach is that you can generate such a file using the HTML editor you prefer. At run time, the PageProducer converts the special tags to actual HTML code, giving you a straightforward method for modifying sections of an HTML document. The special tags have the basic format `<#tagname>`, but you can also supply named parameters within the tag. You'll process the tags in the `OnTag` event handler of the PageProducer.

- The DataSetPageProducer extends the PageProducer by automatically replacing tags corresponding to field names of a connected data source.
- The DataSetTableProducer component is generally useful for displaying the contents of a table, query, or other dataset. The idea is to produce an HTML table from a dataset, in a simple yet flexible way. The component has a very nice preview, so you can see how the HTML output will look in a browser directly at design time[403].
- The QueryTableProducer is similar to the previous one (it is actually a subclass), but it's specifically tailored for building parametric queries based on input from an HTML search form. For this reason, I'll delay the coverage of this component until I cover server-side programming.

Producing HTML Pages[404]

A very simple example of using tags is creating an HTML file that displays fields with the current date or a date computed relative to the current date, such as an expiration date. If you examine the HtmlProd example, you'll find the following component in the main form:

```
object PageProducer1: TPageProducer
  HTMLDoc.Strings = (...)
  OnHTMLTag = PageProducer1HTMLTag
end
```

The source HTML can be specified using an external file (with the advantage that you can edit it without having to recompile the application using it) or a string list, stored in the `HTMLDoc` property. This is a plain HTML file that might contain a few special tags introduced by the `#` symbol:

---

403 While there isn't a direct repalcement, WebStencils is focused on helping generating HTML files mapped to datasets.

404 This section (and the following ones) make sense only in a historical perspective, as these components are no longer recommended.

```
<HTML><HEAD>
<TITLE>Producer Demo</TITLE>
</HEAD><BODY>
<H1>Producer Demo</H1>
This is a demo of the page produced by the <b><#appname></b>
application on <b><#date></b>.<p>
<hr>
The prices in this catalog are valid until <b>
<#expiration days=21></b>.<p>
</BODY></HTML>
```

note If you prepare this file with an HTML editor (something I suggest you do), it might automatically place quotes around tag parameters, as in `days="21"`, because this is required by HTML 4. The PageProducer component in Delphi 5 has a new `StripParamQuotes` property, which can be activated to remove those extra quotes when the component parses the code (before calling the `OnHTMLTag` event handler).

The Demo Page button simply copies the PageProducer component's output to the `Text` of a Memo with the statement

```
Memo1.Text := PageProducer1.Content;
```

As you call the `Content` function of the PageProducer component, it reads the input HTML code, parses it, and triggers the `OnTag` event handler for every special tag. In this method, we check the value of the tag (passed in the `TagString` parameter) and return a different HTML text (in the `ReplaceText` reference parameter), producing the output of Figure 20.1.

```
procedure TFormProd.PageProducer1HTMLTag(Sender: TObject;
  Tag: TTag; const TagString: String; TagParams: TStrings;
  var ReplaceText: String);
var
  nDays: Integer;
begin
  if TagString = 'date' then
    ReplaceText := DateToStr (Now)
  else if TagString = 'appname' then
    ReplaceText := ExtractFilename (Forms.Application.Exename)
  else if TagString = 'expiration' then
  begin
    nDays := StrToIntDef (TagParams.Values['days'], 0);
    if nDays <> 0 then
      ReplaceText := DateToStr (Now + nDays)
    else
      ReplaceText := '<I>{expiration tag error}</I>';
  end;
end;
```

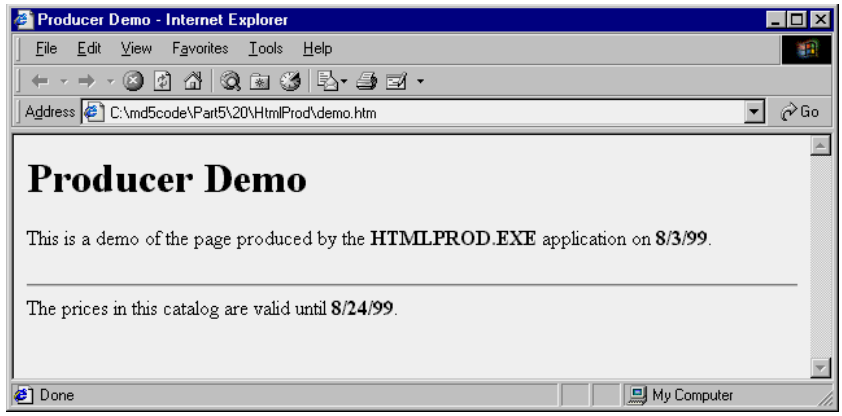


Figure 20.1: The output of the HtmlProd example, a simple demonstration of the PageProducer component, when the user presses the Demo Page button. Image from the original book.

Notice in particular the code we've written to convert the last tag, *expiration,* which requires a parameter. The PageProducer places the entire text of the tag parameter (in this case, *days=21*) in a string that's part of the `TagParams` list. To extract the value portion of this string (the portion after the equal sign), you can use the `Values` property of the `TagParams` string list and search for the proper entry at the same time. If it can't locate the parameter or if its value isn't an integer, the DLL displays an error message.

note The PageProducer component supports user-defined tags, which can be any string you like, but you should first review the special tags defined by the `TTags` enumeration. The possible values include `tgLink` (for the LINK tag), `tgImage` (for the IMAGE tag), `tgTable` (for the TABLE tag), and a few others. If you create a custom tag, as in the PageProd example, the value of the `Tag` parameter to the HTMLTag handler will be `tgCustom`.

Producing Pages of Data

The HtmlProd example has also a DataSetPageProducer component, with the following settings and HTML source code:

```
object DataSetPageProducer1: TDataSetPageProducer
  HTMLDoc.Strings = (
    '<HTML><HEAD>'
    '<TITLE>Data for <#name></TITLE>'
    '</HEAD><BODY>'
    '<H1><CENTER>Data for <#name></CENTER></H1>'
    'Capital: <#capital><p>'
    'Continent: <#continent><p>'
```

```
    'Area: <#area><p>'
    'Population: <#population><p>'
    '<HR>'
    'Last updated on <#date><br>'
    'HTML file produced by the program <#program>'
    '</BODY></HTML>')
  OnHTMLTag = DataSetPageProducer1HTMLTag
  DataSet = Table1
end
```

Simply by using tags with the names of the fields of the connected dataset (the usual Country.DB database table), the program automatically gets the value of the fields of the current record and replaces it automatically, producing the output of Figure 20.2. In the source code of the program related to this component, in fact, there is no reference to the database data:

```
procedure TFormProd.BtnLineClick(Sender: TObject);
begin
  Memo1.Clear;
  Memo1.Text := DataSetPageProducer1.Content;
  BtnSave.Enabled := True;
end;

procedure TFormProd.DataSetPageProducer1HTMLTag(
  Sender: TObject; Tag: TTag; const TagString: String;
  TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'program' then
    ReplaceText := ExtractFilename (Forms.Application.Exename)
  else if TagString = 'date' then
    ReplaceText := DateToStr (Date);
end;
```

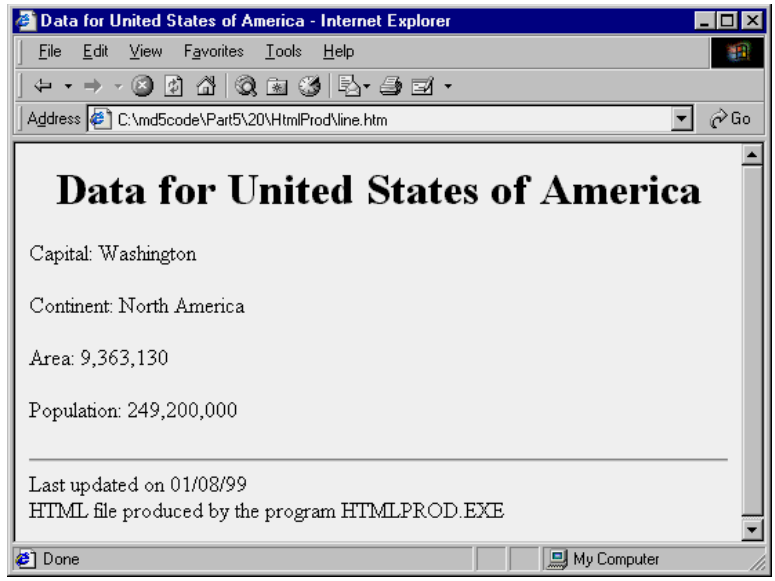


Figure 20.2: The output of the HtmlProd example for the Print Line button. Image from the original book.

Producing HTML Tables

The last button of the HtmlProd example is Print Table. This button is connected to a DataSetTableProducer component[405], again calling its `Content` function and copying its result to the `Text` of the Memo. By simply connecting the `DataSet` property of the DataSetTableProducer to `Table1`, you can produce a standard HTML table. Actually, the component by default generates only 20 rows, as indicated by the `MaxRows` property. If you want to get all the records of the table you can set this property to `-1`, a simple but undocumented setting.

note The DataSetTableProducer component starts from the current record rather than from the first one. This means that the second time you press the Print Table button, you'll see no records in the output. Adding a call to the `First` method of the table before calling the `Content` method of the producer component fixes the problem.

To make the output of this producer component more complete, you can do two different operations. The first is to provide some `Header` and `Footer` information, to generate the HTML heading and closing elements, and add a `Caption` to the HTML

405 Again, this works today, but it's not recommended.

table. The second is to customize the table itself, by using the setting specified by the `RowAttributes`, `TableAttributes`, and `Columns` properties. The property editor of the columns, which is also the default component editor, allows you to set most of these properties, providing at the same time a very nice preview of the output, as you can see in Figure 20.3. Before using this editor, you can set up properties for fields of the table, using the Fields Editor. This is how, for example, you can format the output of the population and area fields to use thousand separators.

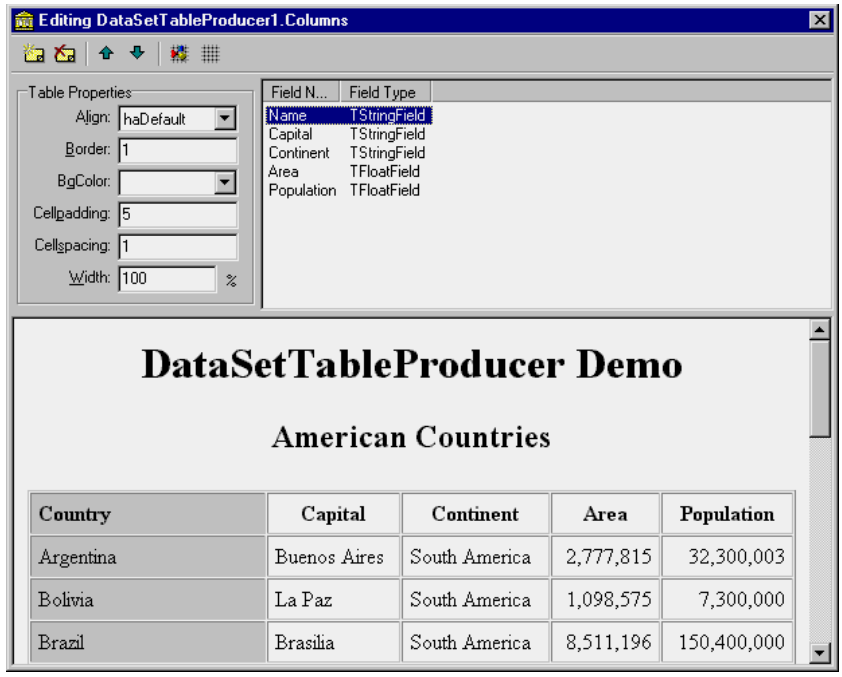


Figure 20.3: The editor of the Columns property of the DataSetTableProducer component provides you with a preview of the final HTML table (if the database table is active). Image from the original book.

There are three techniques you can use to customize the HTML table, and it's worth reviewing each of them:

- You can use the table producer component's `Column` property to set properties, such as the text and color of the title, or the color and the alignment for the cells in the rest of the column. You can see the values for the example in the listing above.
- You can use the `TField` properties, particularly those related to output. In the example, I've set the `DisplayFormat` property of the `Table1Continent` field object to `###,###,###`. This is the approach to use if you want to determine the actual

output of each field. You might go even further and embed HTML tags in the output of a field.

- You can handle the DataSetTableProducer component's `OnFormatCell` event to customize the output further. In this event, you can set the various column attributes uniquely for a given cell, but you can also customize the output string (stored in the `CellData` parameter) and embed HTML tags. This is something you can't do using the `Columns` property.

In the example I've used a handler for this event to turn the text of the Population and Area columns to bold font and to a red background for large values (unless it is the header row). Here is the code:

```
procedure TFormProd.DataSetTableProducer1FormatCell(
  Sender: TObject; CellRow, CellColumn: Integer;
  var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellRow > 0) and
    (((CellColumn = 3) and (Length (CellData) > 8)) or
    ((CellColumn = 4) and (Length (CellData) > 9))) then
  begin
    BgColor := 'red';
    CellData := '<b>' + CellData + '</b>';
  end;
end;
```

The rest of the code is summarized by the settings of the table producer component:

```
object DataSetTableProducer1: TDataSetTableProducer
  Caption = '<h2>American Countries</h2>'
  Columns = <
    item
      BgColor = 'Silver'
      FieldName = 'Name'
      Title.Align = haLeft
      Title.BgColor = 'Silver'
      Title.Caption = 'Country'
    end
    item
      FieldName = 'Capital'
    end
    item
      FieldName = 'Continent'
    end
    item
      Align = haRight
      FieldName = 'Area'
    end
    item
      Align = haRight
```

```
      FieldName = 'Population'
    end>
  Footer.Strings = (
    '<hr><i>Produced by EmplProd</i>'
    '</body></html>')
  Header.Strings = (
    '<html><head>'
    '<title>DataSetTableProducer Demo</title>'
    '</head><body>'
    '<h1><center>DataSetTableProducer Demo</center></h1>')
  MaxRows = -1
  DataSet = Table1
  TableAttributes.Border = 1
  TableAttributes.CellPadding = 5
  OnFormatCell = DataSetTableProducer1FormatCell
end
```

You can see the output of this program in Figure 20.4. I suggest you study the source code of the HTML file this program generates so that you can see the richness of its output and therefore the advantage of using this component.

Using Style Sheets

The latest incarnations of HTML include a very powerful mechanism for separating content from presentation: cascading style sheets (CSS)[406]. Using a style sheet you can separate the formatting of the HTML (colors, fonts, font sizes, and so on) from the actual text displayed (the content of the page). This approach makes your code more flexible and your Web site easier to update. In addition, you can separate the task of making the site graphically appealing (the work of a Web designer) from automatic content generation (the work of a programmer). Style sheets are a rather complex technique, in which you give formatting values to the main types of HTML sections and to special "classes" (which have nothing to do with OOP). Again, see an HTML reference for the details.

406 While this is a core feature today, rather than a new one, using a CSS remains fundamental in today's Web development.

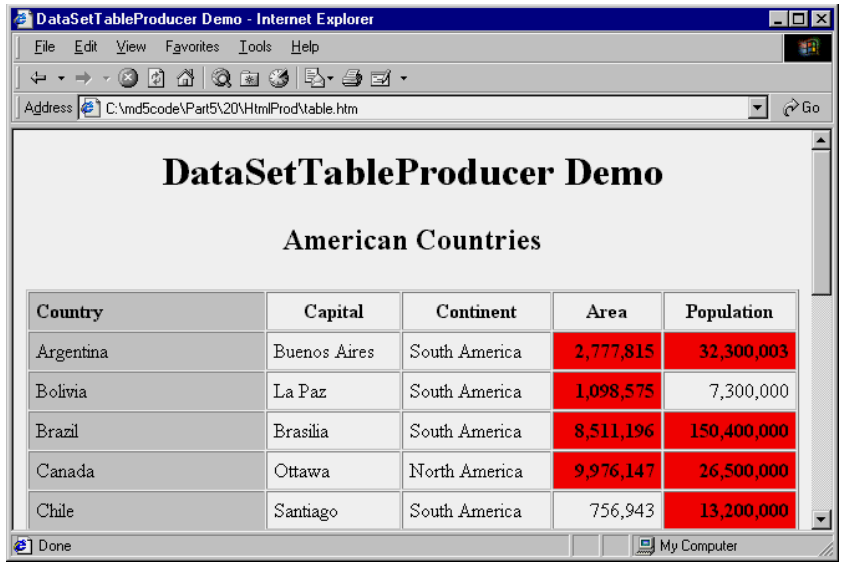


Figure 20.4: The output of the Print All button of the HtmlProd example, which is based on the DataSetTableProducer component. Image from the original book.

How can we update table generation in the HtmlProd example to include style sheets? Simply enough, we can provide a link to the style sheet to use in the `Header` property of a second DataSetTableProducer component, with the line

```
<link rel="stylesheet" type="text/css" href="test.css">
```

We can then update the code of the `OnFormatCell` event handler with the following action (instead of the two lines changing the color and adding the bold font tag):

```
CustomAttrs := 'class="highlight"';
```

The style sheet I've provided (TEST.CSS, available in the source code of the example) defines a *highlight* style, which has exactly the bold font and red background that were hard-coded in the code of the first DataSetTableProducer component.

The advantage of this approach is that now a graphic artist can modify the CSS file and give our table a nicer look without touching its code. When you want to provide many formatting elements, using a style sheet can also reduce the total size of the HTML file. This is an important element that can reduce download time.

Publishing Static Databases on the Web

Once you know how to produce files, you can simply add links from one to another and produce a series of cross-linked HTML files, representing a portion of a Web site. There are circumstances in which writing a program that examines a database and produces files is the best approach for publishing database data on a Web site[407]. You can use a similar technique if the following conditions apply:

- **If the data doesn't change very often:** A catalogue updated monthly or weekly is a good example. Even if you can update the site automatically every night, this is still a possible technique. (For real-time information, of course, this is certainly not a good approach!)
- **If the amount of data is limited and smaller than your available space on the Web site:** This seems obvious, but the formatted HTML output might take much more space than the original database files. If you use a server-side program (like those I'll be discussing in the next chapter) to generate the HTML from the database data on the fly, you might need less disk space on the Web site. Keep in mind that preparing all the HTML files beforehand usually results in much better performance (faster server response time to Web requests, and lower memory overhead to process the requests) than generating the data on the fly.
- **If the number of ways to navigate is limited:** If there are three or four obvious paths of navigation (a main one and two or three cross-references) you can generate all of them statically. Otherwise, the cross-referencing HTML files will be much larger than the files with the actual data, and the time required to generate them may become excessive.

Even if only parts of these conditions apply to your specific needs, you can consider using a mixed approach. You can have a portion of the data and of the navigational files generated periodically and have a CGI and ISAPI application on the site, as well as let users do free searches and follow other less frequent paths. We'll see how later in this chapter. For the moment, though, I'll focus instead on another totally different technology: publishing ActiveX controls and ActiveForms on a Web site.

---

407 I'd say generating static HTML for a complex web site is less and less common these days, but some of the advice in this chapter should be taken into consideration anyway.

ActiveForms in Web Pages

In Chapter 16, you learned how you can use Delphi's ActiveForm technology to create a new ActiveX control. In that chapter, I mentioned that an ActiveForm is simply an ActiveX control based on a form. Borland documentation often implies that ActiveForms should be used in HTML pages, but you can use any ActiveX control on a Web page[408].

Basically, each time you create an ActiveX library, Delphi enables the Project ➢ Web Deployment Options and Project ➢ Web Deploy menu items. The first allows you to specify how and where to deliver the proper files. As shown in Figure 20.5, in this dialog box you can set the server directory for deploying the ActiveX component, the URL of this directory, and the server directory for deploying the HTML file (which will have a reference to the ActiveX library using the URL you provide).

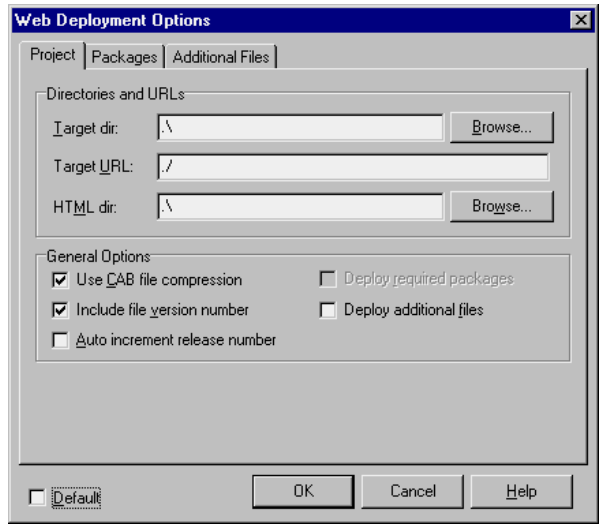


Figure 20.5: The Web Deployment options dialog box. Image from the original book.

You can also specify the use of a compressed CAB file, which can store the OCX file and other auxiliary files, such as packages, making it easier and faster to deliver the application to the user. A compressed file, in fact, means a faster download. Using

408 The entire idea of using ActiveX controls and ActiveForms in the browser had severe security issues and was strictly tied to the Windows platform and the use of the Internet Explorer browser. Needless to say, these elements make the feature totally obsolete – although you might still be able to make it work.

the options shown in Figure 20.5, Delphi generates the HTML file and CAB file for the XClock project (built in Chapter 16) in the same directory. Opening this HTML file in Internet Explorer (remember, Netscape has no support for ActiveX controls) produces the output shown in Figure 20.6.

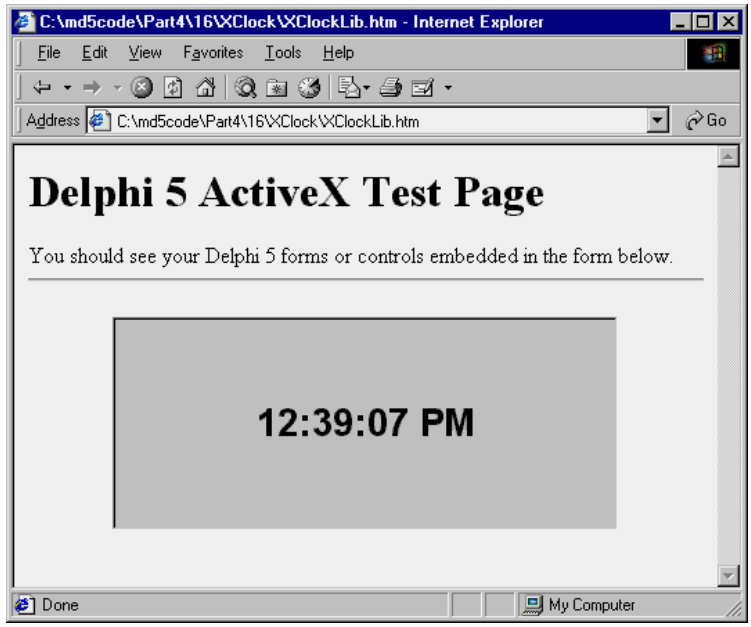


Figure 20.6: The XClock control in the sample HTML page. Image from the original book.

note At times, when you load an HTML page referring to an ActiveX, all you get is a red *X* marker indicating a failure to download the control. There are various possible explanations for this problem. First, Internet Explorer must be set up properly, allowing the download of controls and (if the control is not signed) lowering the security level. Second, other problems might arise when the control requires a DLL or a package that is not part of the downloaded CAB file. Third, you might get the red slash marker when there is a mismatch in the version number—*or* you might see an older version of the control in action. That's because even when you've rebuilt the control, Internet Explorer might decide to use the cached version instead (stored in either the `windows/occache` or the `windows/downloaded program files` directory). You can use version information and other related techniques to avoid this third problem. It's not very professional, but as a last resort you can simply remove the copy of the file from the cache when everything else fails. Due to bugs in Internet Explorer 3 and 4, if an older version of the ActiveX control is still loaded in the browser's in-memory cache, the browser will not download or install a newer version of the control even when the HTML object tag reference makes it clear that it is a newer version number. Also, the file of the cached control might be locked, so in order to be able to deploy the file correctly, you need to shut down Internet Explorer and then deploy the file.

Besides showing you how to deploy the XClock control on a Web page, I've created the XForm1 example to demonstrate the problems with event handlers of Active Forms mentioned at the end of Chapter 16, in the section "ActiveForm Internals." Because the form events are exported as events of the control, you should not handle the events of the form directly but add some code to the default handlers provided by the Active Form. For example, if you add a handler for the `OnPaint` event of the form and write the following code, it will never be executed:

```
procedure TFormX1.FormPaint(Sender: TObject);
begin
  Canvas.Brush.Color := clYellow;
  Canvas.Ellipse(0, 0, Width, Height);
end;
```

If you want to paint something on the form's background, instead, you have to modify the corresponding handler installed by the ActiveForm Wizard:

```
procedure TFormX1.PaintEvent(Sender: TObject);
begin
  Canvas.Brush.Color := clBlue;
  Canvas.Rectangle (20, 20,
    ClientWidth - 20, ClientHeight - 20);
  if FEvents <> nil then FEvents.OnPaint;
end;
```

As an alternative, you can place a frame, a panel, or another component on the surface of the form, and handle *its* events. In the XForm1 example I've simply added a PaintBox component, with a bevel component behind it to make the area of the PaintBox visible.

The Role of an ActiveX Form on a Web Page

Before we look at another example, it is important to stop for a second to consider the role of an ActiveX form placed inside a Web page. Basically, placing a form in a Web page corresponds to letting a user download and execute a custom Windows application. There is little else happening. You download an executable file and start it. (This is one of the reasons the ActiveX technology raises so many concerns about security.[409])

409 As I mentioned earlier, these concerns for security ended up killing the use of ActiveX controls in the browser.

A simple example can highlight the situation. For the following example, I generated a new ActiveForm, added a button and a label to it, and then wrote the following code for the `OnClick` event of the button:

```
procedure TXFormUser.Button1Click(Sender: TObject);
var
  UserName: string;
  Size: Cardinal;
begin
  Size := 128;
  SetLength (UserName, Size);
  GetUserName (PChar(UserName), Size);
  Label1.Caption := UserName;
end;
```

This method simply calls the `GetUserName` Windows API function, and its effect is certainly not astonishing, as the name of the user will simply be displayed in a label. However, this example highlights a couple of important points (which apply both to ActiveForms and ActiveX controls in general):

- In an ActiveX control or form, you can call any Windows API function (which means the user viewing the Web page must have Windows on his or her computer) or certain Windows API-compatible libraries.
- An ActiveX can access the system information of the computer, such as the user name, the directory structure, and so on. This is why, before downloading an ActiveX, Web browsers check whether the ActiveX has a proper authentication, or signature. (You should note that this signature simply identifies the author of the control and that the module has not been corrupted or tampered with since the author published it; it doesn't prove in any way that the control is safe.)

Well, I could continue, but I think my point is clear. ActiveX controls and ActiveForms are great tools, particularly in an intranet. On the Internet, however, some users might not like having to use ActiveX controls.

A Multipage ActiveForm

As a final example, I'll take an existing program and turn it into an ActiveForm. We have seen that there are three standard approaches in the development of a complex program: MDI, multiple modal or modeless forms, and multipage forms. The last approach is the one best suited for the development of a complex ActiveForm.

If you want to turn an existing form into an ActiveX form, there are several approaches you can follow. The simplest is probably to select all of the components

in the original program, build a component template out of them (so that you copy the component properties and their event handlers), and then paste them into a new ActiveForm. An alternative to this approach is to take the existing form and place it inside an ActiveForm. This is not a particularly complex task, and it provides the great advantage that there is nothing you have to change in the original source code, not even the methods handling events of the form. (Because the source is not the ActiveForm, its events are not connected to the outside world.)

To show you this approach in practice, I've taken the WizardUI example of Chapter 8, added Web connectivity to it, and then hosted it inside an ActiveForm. The new example is called XWebWiz. If you remember the original example, it showed information about Web sites. I've taken the labels and list boxes referring to Web sites and set their properties, as in the following case:

```
object Label2: TLabel
  Cursor = crHandPoint
  Caption = 'Main site: www.inprise.com'
  Font.Color = clBlue
  Font.Style = [fsUnderline]
  OnClick = LabelLinkClick
end
```

The blue underlined text will look like a typical link inside a browser, and the hand cursor completes the picture. To activate these links, we can extract the Web site's URL from the label caption or list-box item and call the `ShellExecute` API function to browse the URL. To turn the program into an ActiveForm, I've simply created a new ActiveForm, added the unit of the original form to the project, and written the following `OnCreate` handler for the active form:

```
uses
  WizForm;

procedure TXWizForm.FormCreate(Sender: TObject);
begin
  WizardForm := TWizardForm.Create (Self);
  WizardForm.Parent := Self;
  WizardForm.Align := alClient;
  WizardForm.BorderStyle := bsNone;
  WizardForm.Show;
end;
```

By setting the `Parent` property to the active form, aligning it, and removing the border, the existing form will cover the entire surface of the active form, making the two indistinguishable. You can see an example of the output in Figure 20.7.

note Of course, a multipage ActiveForm can be based on frames, as we've already seen in Chapter 8 for multipage forms.

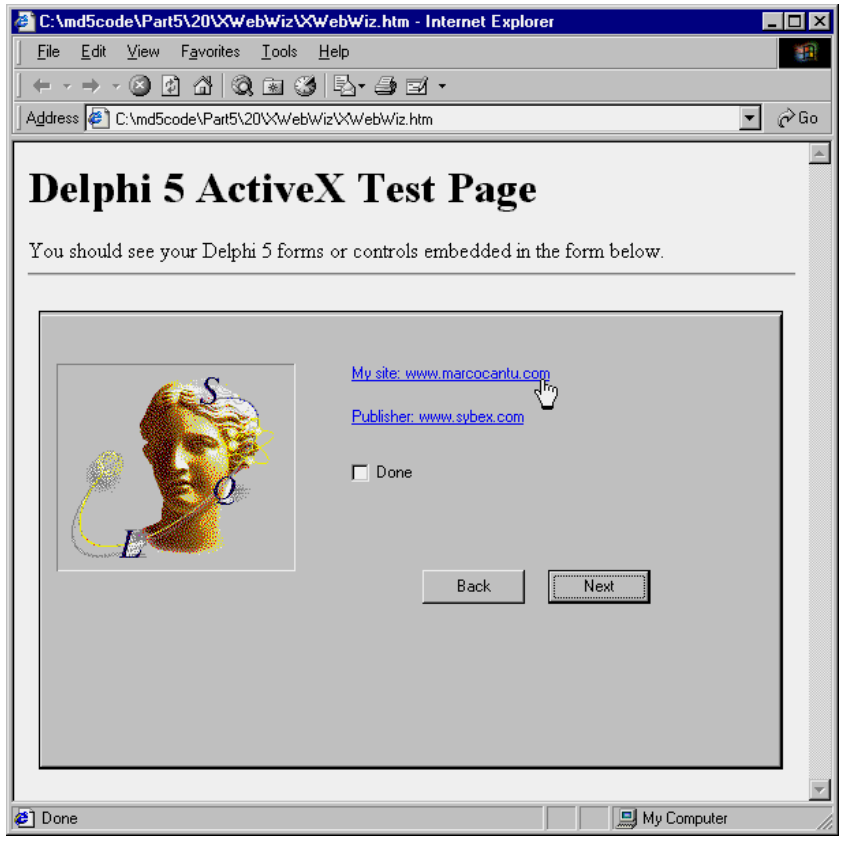


Figure 20.7: The output of the XWebWiz control in an HTML page. By selecting the links, you can jump to the corresponding Web site. Image from the original book.

Setting Properties for the XArrow

An ActiveForm has a few properties you can set when you use it inside a development environment, and a plain ActiveX control has even more. For example, if you want to set properties in the HTML file hosting the control, you can use a special `Param` tag, but the control must support a special interface known as `IPersistPropertyBag`.

Starting with Delphi 4, the `IPersistPropertyBag` support is built in, providing support for all of the properties of the ActiveX control or ActiveForm. As an example, I've used the Web Deploy options on the XArrow control built in Chapter 16. Then, I've modified the automatically generated HTML file with three `Param` tags:

```
<OBJECT
  classid="clsid:5551EB27-0AC6-11D2-B9F1-004845400FAA"
  codebase="./XArrow.cab"#version=1,0,0,0
  width=350
  height=250
  align=center
  hspace=0
  vspace=0
>
<Param Name="ArrowHeight" Value="100">
<Param Name="Filled" Value="-1">
<Param Name="FillColor" Value="111829">
</OBJECT>
```

You can compare the default and customized output of the control in Figure 20.8.

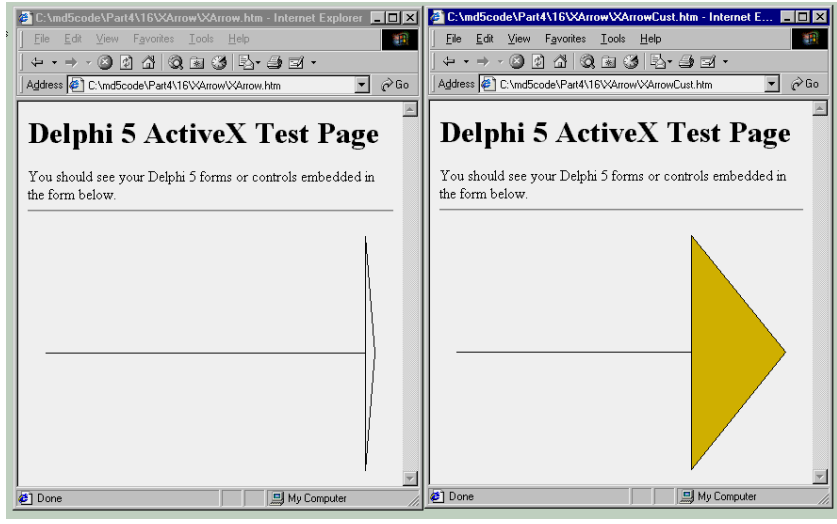


Figure 20.8: By using the Param tag, we can set values for the properties of an ActiveX control in the HTML file hosting it. The two copies of the program show the default and the customized output. Images from the original book.

Socket Programming with Delphi

Up to now in this chapter we've seen how to publish static HTML files on a Web site and how to make the HTML pages richer by inserting Windows programs (in the form of ActiveX controls) into them.

Now I'm going to focus more on Internet programming, specifically the use of the connectivity provided by Delphi socket components, which are based on TCP/IP and the low-level Windows Sockets. Before we look into the foundations of sockets, let me list a number of alternative approaches you can use for Internet programming, which I'll cover in more detail in the next few sections:

- The Delphi Socket Components provide a good interface for a direct use of the Windows sockets API, implementing some custom protocol of your own. Using Delphi higher-level socket components is generally much easier than using the low-level API[410].
- For standard protocols, you can also use the FastNet Tools VCL components (from NetMasters), included in Delphi, or look for similar controls from other third-parties[411].
- The WinInet (Windows Internet) library is a collection of higher-level services provided by Microsoft. These services make the development of HTTP and FTP programs very simple[412].

---

410 The core socket Delphi components still exists,, but they are not recommended.

411 What Delphi has long been shipping is a different set of open source components, called Internet Direct (or Indy, for short). Indy offers low level TCP/IP components but also high end protocols, both on the server and the client side. Despite a few issues, Indy remains by far the most use Internet library by Delphi develoeprs.

412 As an alternative, Delphi now includes an HTTP client library (and also a REST client library based on it). This library encapsulates the platform HTTP library for each operating system, rather than offering a custom implementation like Indy does. For HTTP clients, the HTTP client library is the recommended approach, as it offers HTTPS support without having to deploy an SLL library.

Foundations of Socket Programming

To understand the description of the Socket components in the Delphi Help file, and also to read along the description of the examples in the book, you need to be confident with a number of terms related to the Internet in general and with sockets in particular.

The heart of the Internet is the Transmission Control Protocol/Internet Protocol (TCP/IP for short), a combination of two separate protocols which work together to provide connection over the Internet (and can also provide connection over a private intranet). In brief, IP is responsible for defining and routing the *datagrams* (Internet transmission units) and specifying the addressing scheme. TCP is responsible for higher-level transport services. Beside TCP there is another, less-known protocol: UDP (User Datagram Protocol).

Configuring a Local Network: IP Addresses[413]

If you have a local network available, you'll be able to test the following programs; otherwise, you can simply use the same computer as client and server. In this case, as I've done in the examples, you can use the address `127.0.0.1` (or *localhost*), which is invariably the address of the current computer. If your network is complex, ask your network administrator to set up proper IP addresses for you. If you want to set up a simple network with a couple of spare computers, you can simply set up the IP address yourself, a 32-bit number usually represented with each of its four components (called octets) separated by a dot. These numbers have a complex logic underneath them, with the first octet indicating the class of the address.

There are actually specific IP addresses reserved for unregistered internal networks. Internet routers will ignore these address ranges, so you can freely do your tests without interfering with an actual network. The "free" IP address range `192.168.0.0` through `192.168.255.0` can be used for experiments on a network of fewer than 255 machines.

Local Domain Names

How does the IP address map to a name? On the Internet, the client program looks up the values on a domain name server. But it is also possible to have a local *hosts* file, a text file that you can easily edit to provide nice local mappings. You can take a

413 While all of these is likely very well known by most developers today, it wasn't the same when the book had been written. At the time, many developers had little information about the foundations of ntwworking.

look at the Hosts.SAM[414] file (installed in the `Windows` directory) to see a sample and then eventually rename the file as HOSTS, without the extension, to activate local host mapping.

Should you use an IP or a host name in your programs? Host names are easier to remember and won't require a change if the IP address changes (for whatever reason). On the other hand, IP addresses don't require any resolution, while host names must be resolved (a time-consuming operation if the lookup takes place on the Web).

TCP Ports

Each TCP connection take place though a *port*. A port is represented by a 16-bit number. The IP address and the TCP port together specifies an Internet connection, or a *socket* (to use a more precise term). Different processes running on the same machine cannot use the same socket—the same port.

Some TCP ports have a standard usage for specific high-level protocols and services. In other words, you should use those port numbers when implementing those services and stay away from them in any other case. Here is a short list:

Protocol	Port
HTTP (Hypertext Transfer Protocol)	80
FTP (File Transfer Protocol)	21
SMTP (Simple Mail Transfer Protocol)	25
POP3 (Post Office Protocol, version 3)	110
Telnet	23

The Services file (another text file similar to the Hosts file) lists the standard ports used by services. You can add your own entry to the list, giving your service a name of your own choosing. Client sockets always specify the port number or the service name of the server socket to which they want to connect.

High-Level Protocols

I've used the term *protocol* many times now, but what does it mean exactly? A protocol is a set of rules the client and the server agree upon to determine the communication flow. The low-level Internet protocols, such as TCP/IP, are usually implemented by an operating system. But the term *protocol* is also used for high-

414 This is now the `hosts` file in the `c:\Windows\System32\Drivers\etc` folder.

level Internet Standard Protocols (such as HTTP, FTP, or SMTP). These protocols are defined in standard documents available on the Web on the `http://www.internic.net` site.

If you want to implement a custom communication, you can define your own (possibly simple) protocol, a set of rules determining which request the client can send to the server and how the server can respond to the various possible requests. We'll see an example of a custom protocol later on. Transfer protocols are at a higher level than transmission protocols, because they abstract from the transport mechanism provided by TCP/IP. This makes the protocols independent not only from the operating system and the hardware but also from the physical network.

Socket Connections

How do you start the communication through a socket? The server program starts running first, but it simply waits for a request from a client. The client program requests a connection indicating the server it wishes to connect to. When the client sends the request, the server can accept the connection, starting a specific server-side socket, which connects to the client-side socket.

To support this model, there are three different types of socket connections:

- *Client connections* are initiated by the client and connect a local client socket with a remote server socket. Client sockets must describe the server they want to connect to, by providing either its host name or IP address and its port.
- *Listening connections* are passive server sockets waiting for a client. Once a client makes a new request, the server spawns a new socket devoted to that specific connection and then gets back to listening. Listening server sockets must indicate the port that represents the service they provide. (In fact, the client is going to connect through that port.)
- *Server connections* are the connections activated by servers, as they accept a request from a client.

These different types of connections are important only for establishing the link from the client to the server. Once the link is established, both sides are free to make requests and to send data to the other side.

Delphi Socket Components[415]

Delphi has both a client-socket component and a server-socket component. The aim of the socket components is to make it very simple to read and write information over a TCP/IP connection.

On the Components Palette there are just two socket components, `TServerSocket` and `TClientSocket`. Both classes inherit from the base class `TAbstractSocket`, an abstract base class for all socket components, defined in the `ScktComp` unit. The properties of `TAbstractSocket` describe the IP address of the socket and the service it provides or seeks. Not all descendants of `TAbstractSocket` use all of these properties. For example, server sockets do not make the IP address available, because it is read implicitly from the system running the application.

Using a socket component you can determine a host and a service by using any of the following:

- The `Host` property, indicating the domain name and service of a particular system (used by client sockets)
- The `Address` property, a string with the four numbers in the standard Internet dot notation (used by client sockets)
- The `Port` property, a number indicating the port
- The `Service` property, a string indicating the service name

However, these are not the only classes providing socket support. For interfacing with Windows sockets, there is a `TCustomWinSocket` class with three subclasses: `TServerWinSocket`, `TClientWinSocket`, and `TServerClientWinSocket`. These classes wrap the handle of the Windows socket connection, and they are used by the main socket components to manage the Windows socket API calls and to store information about a socket communication link. The `TCustomWinSocket` class refers to the socket handle, indicated by the `SocketHandle` property, and also to the handle of a hidden window used to receive socket messages, indicated by the `Handle` property.

Descendants of `TCustomWinSocket` represent different types of connections: `TClientWinSocket` represents a client connection, `TServerWinSocket` represents the listening connection, and `TServerClientWinSocket` represents the server connection. Finally, there is a specific `TStream` subclass, `TWinSocketStream`, and a specific `TThread` subclass, `TServerClientThread`.

---

415 As mentioned, these components still work, but they are not recommended, compared to using the Indy socket components.

Using Sockets

After all that theory, let us take a look at a couple of simple examples. The first is stored in the `Sock1` directory and is made of the Server1 and Client1 applications. The server has a form with the following component:

```
object ServerSocket1: TServerSocket
  Active = True
  Port = 50
  ServerType = stNonBlocking
  OnClientConnect = ServerSocket1ClientConnect
  OnClientDisconnect = ServerSocket1ClientDisconnect
  OnClientRead = ServerSocket1ClientRead
end
```

All the code of the application relates to the events of this component, as the program provides no specific interaction with the user. However, the server has three list boxes for outputting the status, the messages sent from the client, and a log of the events. For example, as a client connects, the server adds the client address to the log:

```
procedure TForm1.ServerSocket1ClientConnect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  lbLog.Items.Add ('Connected: ' +
    Socket.RemoteHost + ' (' +
    Socket.RemoteAddress + ')' );
  PostMessage (Handle, wm_RefreshClients, 0, 0);
end;
```

Notice that the `OnClientConnect` event indicates the first occasion for the server to know about the connected client. Using the `Socket` property, which refers to the low-level `TCustomWinSocket`, the server can track who is trying to connect. At the end of this and other events I want to update the list of the connections, using the `ActiveConnections` property of the server. However, in the `OnClientConnect` event handler this list is still not updated, so I post a message to the form to delay the operation:

```
const
  wm_RefreshClients = wm_User;

procedure TForm1.RefreshClients; // message wm_RefreshClients
var
  I: Integer;
begin
  lbClients.Clear;
  for I := 0 to ServerSocket1.Socket.ActiveConnections - 1 do
    with ServerSocket1.Socket.Connections [I] do
```

```
      lbClients.Items.Add (
        RemoteAddress + ' (' + RemoteHost + ')');
end;
```

Similar code is executed as the client disconnects from the server:

```
procedure TForm1.ServerSocket1ClientDisconnect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  lbLog.Items.Add ('Disconnected: ' +
    Socket.RemoteHost + ' (' +
    Socket.RemoteAddress + ')' );
  PostMessage (Handle, wm_RefreshClients, 0, 0);
end;
```

Finally, as the client sends some information to the server (writes to the socket) the server can read the message by calling the `ReceiveText` function. You should do this read operation only when there is some data available—that is, when the `OnClientRead` event is fired. Notice also that this is a *destructive* read: the information extracted from the stream is removed from it. Here is the code:

```
procedure TForm1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  // read from the client
  lbMsg.Items.Add (Socket.RemoteHost + ': ' +
    Socket.ReceiveText);
end;
```

Now we can move to the client side of the application, which has a form hosting a client-socket component with the following properties:

```
object ClientSocket1: TClientSocket
  Active = False
  Address = '127.0.0.1'
  ClientType = ctNonBlocking
  Port = 50
  OnConnect = ClientSocket1Connect
  OnDisconnect = ClientSocket1Disconnect
end
```

The client form is more interactive. It has two edit boxes and a check box. In the first edit box, you can type the address of the server you want to connect to (to replace the default value listed above), using the check box to activate or deactivate the socket connection:

```
procedure TForm1.cbActivateClick(Sender: TObject);
begin
  if not ClientSocket1.Active then
    ClientSocket1.Address := EditServer.Text;
```

```
  ClientSocket1.Active := cbActivate.Checked;
end;
```

As you connect or disconnect, the program simply updates the caption of the form. In the second edit box, you can type a message to send to the server and a button you can press to send the message:

```
procedure TForm1.btnSendClick(Sender: TObject);
begin
  ClientSocket1.Socket.SendText (EditMsg.Text);
end;
```

Notice that this example program doesn't check whether the connection is active before using it, which can result in errors. In Figure 20.9, you can see an example of the client and the server. As the server indicates, there is a second copy of the client application running on another computer and connected to it.

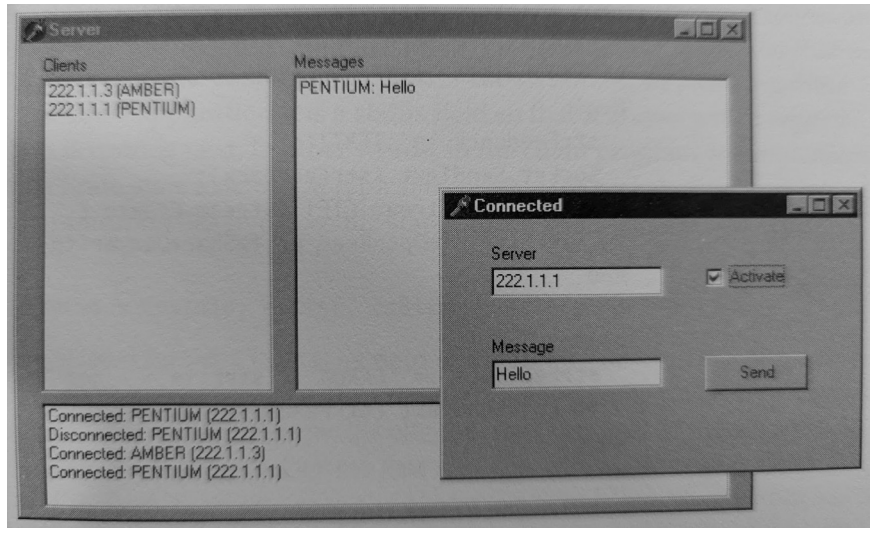


Figure 20.9: The client and server applications of the `Sock1` example, demonstrating the use of the socket components. Image based on a picture of the original printed book.

Using Sockets with a Custom Protocol

Unless you want to send and receive only simple text messages, you might want to define some communication rules between the client and the server. A set of communication rules is generally indicated as a protocol. Basically, the server can receive different requests, and depending on the type of request and whether it can be accomplished or not, reply to the client.

The server program of the Sock2 example accepts four types of requests: the listing of a directory, a bitmap file, a text file, and the execution of a program on the server. When the server sends back a file, its reply should indicate both what it is going to send back and the actual information. The only method modified from the Sock1 example is the `ServerSocket1ClientRead` procedure, which starts by extracting the five initial characters of the text received by the client that host the command:

```
strCommand := Socket.ReceiveText;
lbLog.Items.Add ('Client: ' + Socket.RemoteAddress +
  ': ' + strCommand);
// extract the file name (all commands have 5 characters)
strFile := Copy (strCommand, 6, Length (strCommand) - 5);
```

The actual code depends on the initial command defined by the protocol (in this case either *EXEC!* to execute a file on the server, *TEXT!* to return a text file, *BITM!* to retrieve a bitmap file, or *LIST!* to return a directory listing). Here is the code for two out of these four alternatives:

```
// send back a text file
if Pos ('TEXT!', strCommand) = 1 then
begin
  if FileExists (strFile) then
  begin
    strFeedback := 'TEXT!';
    Socket.SendText (strFeedback);
    Socket.SendStream (TFileStream.Create (
      strFile, fmOpenRead or fmShareDenyWrite));
  end
  else
  begin
    strFeedback := 'ERROR' + strFile + ' not found';
    Socket.SendText (strFeedback);
  end;
end
// send back a directory listing
else if Pos ('LIST!', strCommand) = 1 then
begin
  if DirectoryExists (strFile) then
  begin
    strFeedback := 'LIST!';
```

```
      Socket.SendText (strFeedback);
      FileListBox1.Directory := strFile;
      Socket.SendText (FileListBox1.Items.Text);
    end
    else
    begin
      strFeedback := 'ERROR' + strFile + ' not found';
      Socket.SendText (strFeedback);
    end;
  end
  else
  begin
    strFeedback := 'ERROR' + 'Undefined command: ' + strCommand;
    Socket.SendText (strFeedback);
  end;
```

For the directory listings, I've used an invisible FileListBox component. For sending back the text file, I've used the `SendStream` method, creating a new stream on the fly. The advantage is that there is no need to destroy the temporary stream, as the `SendStream` method becomes the owner of the stream and destroys it when it is done.

The program sends back multiple pieces of information one after the other. This will create a few problems on the client side, as all the information is received in a single stream. However, the server responds with a five-character header that we can use to determine the content of the rest of the stream. After receiving these headers, the client application sets a status field so that it knows which type of information is coming next. In other words, in the client program we implement a very simple finite-state machine, a typical technique for socket programming. The client application has five possible states, listed in an enumerated type:

```
type
  TCliStatus = (csIdle, csList, csBitmap, csText, csError);
```

This type is used for the `CliStatus` field of the form. The form has two edit boxes referring to a directory or a file a user can request from the server. When the user presses the Get Dir button, the client program passes to the server the name of the directory indicated by the first edit box. The server will return a list of files that the client program saves in a list box. At this point, the user can select one of the files from the list box, and the client program will copy it, along with the complete path, into the second edit box. The text of this second edit box is used by the other three buttons, Exec, Bitmap, and Text, which send further requests to the server. In Figure 20.10, you can see an example of the main form of the client program after a directory has been retrieved.

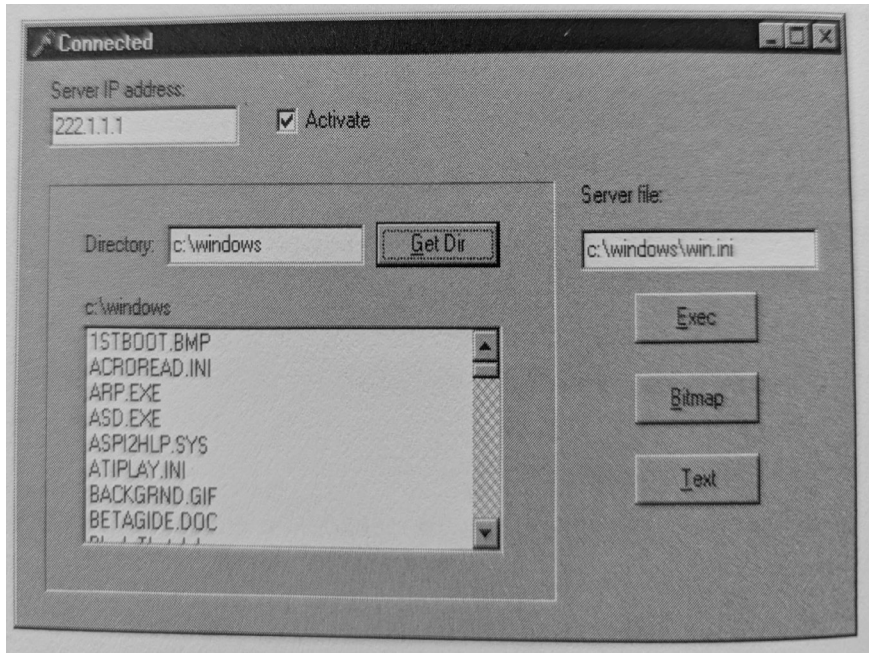


Figure 20.10: The form of the Client2 program after the server has returned the list of the files of a directory. Image based on a picture of the original printed book.

The core of the program is in the `ClientSocket1Read` method, triggered by the socket when there is data to read. The method is first used to get the header indicating which type of data is reaching the program and to set the client program to the proper status:

```
case CliStatus of
  // look for data to receive
  csIdle:
  begin
    Socket.ReceiveBuf (Buffer, 5);
    strIn := Copy (Buffer, 1, 5);
    if strIn = 'TEXT!' then
      CliStatus := csText
    else if strIn = 'BITM!' then
      CliStatus := csBitmap
    // .. and so on
```

Since we don't retrieve all the data, the event is triggered again soon afterwards, and this time we are ready to get the actual data. Here are two more branches of the `case` statement:

```
    // get a directory listing
    csList:
```

```
begin
  ListFiles.Items.Text := Socket.ReceiveText;
  cliStatus := csIdle;
end;
// read a bitmap file
csBitmap:
  with TFormBmp.Create (Application) do
  begin
    Stream := TMemoryStream.Create;
    Screen.Cursor := crHourglass;
    try
      while True do
      begin
        nReceived := Socket.ReceiveBuf (Buffer, sizeof (Buffer));
        if nReceived <= 0 then
          Break
        else
          Stream.Write (Buffer, nReceived);
        // delay (200 milliseconds)
        Sleep (200);
      end;
      // reset and load the temporary file
      Stream.Position := 0;
      Image1.Picture.Bitmap.LoadFromStream (Stream);
    finally
      Stream.Free;
      Screen.Cursor := crDefault;
    end;
    Show;
    cliStatus := csIdle;
  end;
```

For loading the bitmap, I simply move the data to a `Buffer` (declared as `array [0..9999] of Char`) and then from the buffer to a memory stream, which is later loaded in the Image component of the secondary form. Because the data flow can slow down, the program has a hard-coded delay of 200 milliseconds every time some data is read. Unlike file-reading operations, the loop doesn't stop when the data read is less than the data requested, but only when *no* data is read. (In case of error, the value returned by the `ReceiveBuff` method is –1.)

Blocking, Nonblocking, and Multithreaded Connections

Reading and writing over sockets can occur asynchronously, so that it does not block the execution of other code in your network application. This is called a *non-blocking connection* and is what we have done up to now, leaving the `ClientType`

property and the `ServerType` property of the two socket components to the default value, `ctNonBlocking`. Nonblocking connections read and write asynchronously: the `OnRead` and `OnWrite` events of the client and `OnClientRead` or `OnClientWrite` events of servers inform your socket when the other end of the connection tries to read or write some data.

As an alternative to this asynchronous approach, you can also use *blocking connections,* where your application waits for the reading or writing to be completed before executing the next line of code[416]. In this case, you have to write the code in sequence on both sides, because otherwise the events won't be triggered. When using a blocking connection, you must use a thread on the server, and you'll generally use a thread also on the client. On the server the alternative value for the `ServerType` property is `stThreadBlocking`.

As mentioned before, when writing threaded code that is working on a blocking connection, you can use `TWinSocketStream` class to do the actual reading and writing operations. You can use the `WaitForData` method of the `TWinSocketStream` class to wait until the socket on the other end is ready to write. You can also create the socket stream class and specify a timeout value, so that if the connection is lost, it won't hang forever.

Sending Database Data over a Socket Connection

Using the techniques we've seen so far, we can write an application that moves database records over a socket. The idea will be to write a front end for data input and a back end for data storage. The client application will have a simple data entry form and use a database table with string fields for Company, Address, State, Country, Email, and Contact, and a floating-point field for the company ID (called CompID).

note Moving database records over a socket is exactly what you can do with MIDAS and a sockets connection component. This is discussed in the next chapter.

The client program I've come up with works on a table with this structure saved in the current directory. (You can see the related code in the `OnCreate` event handler.) The core method on the client side is the handler of the `OnClick` event of the Send

416 Indy uses blocking connections and requires using threads. This is true for most socket client libraries. Servers need to use threading, to offer connections to multiple clients at the same time.

All button, which sends all the new records to the server. The new records are determined by looking to see if the record has a valid value for the CompID field. This field, in fact, is not set up by the user but is determined by the server application when the data is sent.

For all new records, the client program packages the field information in a string list, using the structure *FieldName=FieldValue,* obtained using the *Values* property of the string list. The string corresponding to the entire list is then sent to the server. At this point, the program stops in an apparently infinite loop:

```
// save database data in a string list
Data := TStringList.Create;
table1.First;
while not Table1.Eof do
begin
  // if the record is still not logged
  if Table1CompID.IsNull or (Table1CompId.AsInteger = 0) then
  begin
    lbLog.Items.Add ('Sending ' + Table1Company.AsString);
    Data.Clear;
    // create strings with structure "FieldName=Value"
    for I := 0 to Table1.FieldCount - 1 do
      Data.Values [Table1.Fields[I].FieldName] :=
        Table1.Fields [I].AsString;
    // send the record
    ClientSocket1.Socket.SendText (Data.Text);
    // wait for response
    fWaiting := True;
    while fWaiting do
      Application.ProcessMessages;
  end;
  Table1.Next;
end;
```

The program waits forever...or until the handler of another message sets the `fWaiting` field of the form to `False`. This happens when the server sends some feedback indicating that the record was received or when the user presses the Stop button. The `btnSendAllClick` method automatically connects to the server at the beginning and disconnects at the end.

Now let us look at the server. This program has a database table, again stored in the local directory, with two new fields added to the client application's table: LoggedBy, a string field; and LoggedOn, a data field. The values of the two extra fields are determined automatically by the server as it receives data, along with the value of the CompID field. All these operations are done in the `ServerSocket1ClientRead` method after unpacking the data received by the client:

```
  // read from the client
```

```
  strCommand := Socket.ReceiveText;
  // reassemble the data
  Data := TStringList.Create;
  try
    Data.Text := strCommand;
    // new record
    Table1.Insert;
    // set the fields using the strings
    for I := 0 to Table1.FieldCount - 1 do
      Table1.Fields [I].AsString :=
        Data.Values [Table1.Fields[I].FieldName];
    // complete with random ID, sender, and date
    Table1CompID.AsInteger := GetTickCount;
    Table1LoggedBy.AsString := Socket.RemoteAddress;
    Table1LoggetOn.AsDateTime := Date;
    Table1.Post;
    // get the value to return
    strFeedback := Table1CompID.AsString;
    // send results back
    lbLog.Items.Add (strFeedback);
    Socket.SendText (strFeedback);
  finally
    Data.Free;
  end;
```

Except for the fact that some data might be lost, there is no problem when fields have a different order and if they do not match, because the data is stored in the *FieldName=FieldValue* structure. After receiving all the data and posting it to the local table, the server sends back the company ID to the client. The client program, after sending the record, goes into *waiting mode,* a situation modified by receiving feedback from the server:

```
procedure TForm1.ClientSocket1Read(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  if fWaiting then
  begin
    Table1.Edit;
    Table1CompId.AsString := Socket.ReceiveText;
    Table1.Post;
    lbLog.Items.Add (Table1Company.AsString +
      ' logged as ' + Table1CompId.AsString);
    fWaiting := False;
  end;
end;
```

When receiving feedback, the client program saves the company ID, which marks the record as sent. If the user modifies the record, there is no way to send an update to the server. To accomplish this, you might add a modified field to the client database table and make the server check to see if it is receiving a new field or a

modified field. With a modified field, the server should not add a new record but update the existing one.

This is one of the many additions you can make to the program, to make it usable in a real-world environment. The existing code of the program and the previous examples on sockets should provide all you need to complete a similar task. I've limited myself to this version of the application, as shown in Figure 20.11. Notice that the server program has two pages, one with the usual log and the other with a DBGrid showing the current data of the server database table.

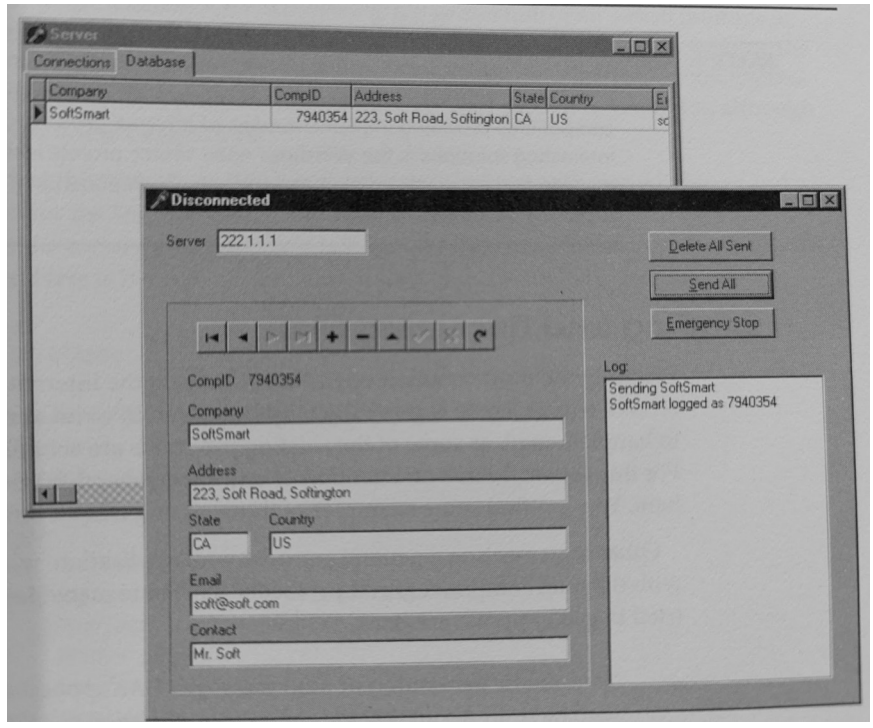


Figure 20.11: The client and server programs of the database socket example (DbSock). Image based on a picture of the original printed book.

Internet Protocols

After discussing the generation of HTML files, the use of the ActiveX technology on Web sites, and the low-level socket components, we are ready to delve in the final

topic of this chapter, the use of higher-level Internet protocols. This is actually the simplest part of the chapter, as the high-level protocols we'll cover can be programmed using high-level components or APIs.

As already mentioned, Delphi ships with a collection of Internet components by NetMasters. These components provide a complete solution, using an alternative approach to Delphi's own socket components. The more interesting components of the FastNet Tools series are those implementing specific protocols, NMFinger, NMNNPT, NMFTP, NNHTTP, NMPOP3, NMSMTP. These components are generally used on client applications to connect to specific servers. Delphi ships with ready-to-use examples for most of the NetMasters components[417].

note These third-party components are already installed in the Delphi IDE, but they are not the only possible solution. There are many free, shareware, and retail Delphi components providing implementations of Internet protocols. One of the most interesting solutions is the Winshoes open source project lead by Chad Hower[418]. You can find more information and the actual components to download on the site `www.pbe.com/Winshoes`.

Sending and Receiving Mail

Probably the most common operation you do on the Internet is to send and receive e-mail. There is generally very little need to write a complete application to handle e-mail, as some of the existing programs are actually rather complete. For this reason, I have no intention of writing a general-purpose mail program here. You can find some examples of those among Delphi Internet demos.

Other than creating a general-purpose mail application, what else can one do with the mail components and protocols? There are many possibilities, which I've tried to group in two areas:

Automatic generation of mail messages. An application you've written can have an About box for sending a registration message back to your marketing department or a specific menu item for sending a request to your tech support. You might even decide to enable a tech-support connection whenever an exception occurs. Another related task would be automating the dispatching of a message to a list of people or generating an automatic message from your Web site, an example I'll show you toward the end of this chapter.

417 This isn't true any more. For a long time, Delphi ahs included the Indy components, instead.

418 This library later morphed into what is known as Indy. I doubt that site exists. Today the main location to look for is https://github.com/IndySockets/Indy.

Usage of mail protocols for communication with users who are only occasionally online. When you have to move some data between users who are not always online, you can write an application on a server to synchronize among them, and you can give each user a specialized client application for interacting with the server. An alternative is to use an existing server application, such as a mail server, and write the two specialized programs based on the mail protocols. The data sent over this connection will generally be formatted in special ways, so you'll want to use a specific e-mail addresses for these messages (not your primary e-mail address).

Sending Messages to the Mail Program

The simplest technique for automating the generation of an e-mail message is to use your existing mail application, adding a message to its outbox. Using the `ShellExecute` API function, you can easily send a message to Outlook Express (or whichever mail program is registered as your default in Windows, although there are a few exceptions).

To test this technique, I've prepared a simple form with two edit boxes and a memo for the input. Pressing a button creates a string with all the information about the message and then sends it, simply executing the string with the *mailto:* prefix. Here is the code of the Send button of the MailGen example:

```
uses
  ShellApi;

procedure TForm1.BtnSendClick(Sender: TObject);
var
  strMsg: string;
  I: Integer;
begin
  // set the basic information
  strMsg := 'mailto:' + EditAddress.Text +
    '?Subject=' + EditSubject.Text +
    '&Body=';
  // add first line
  if Memo1.Lines.Count > 0 then
    strMsg := strMsg + Memo1.Lines [0];
  // add subsequent lines separated by the newline symbol
  for I := 1 to Memo1.Lines.Count - 1 do
    strMsg := strMsg + '%0D%0A' + Memo1.Lines [I];
  // send the message
  ShellExecute (Handle, 'open', pChar (strMsg),
    '', '', SW_SHOW);
end;
```

To show the body of the message on multiple lines, you can separate each line with the carriage return and line feed characters (usually indicated in Delphi as `#13` and `#10`). These values should be explicitly added to the string in hexadecimal format and prefixed by the % sign, as required by a URL. You can actually obtain this encoding automatically by using the NMURL component.

The WinInet API[419]

When you need to use the FTP and HTTP protocols, as alternatives to using specific VCL components, you can use a simple and high-level API provided by Microsoft in the WinInet DLL. This library is part of the core operating system and is available on the Microsoft Web site for download. It basically implements the FTP and HTTP protocols on top of the Windows sockets API.

Simply with three calls—`InternetOpen`, `InternetOpenURL`, and `InternetReadFile`—you can retrieve a file corresponding to any URL and store a local copy or analyze it. Other simple methods can be used for FTP: I suggest you look for the source code of the Delphi unit, listing all the functions, and for the specific Help file for the DLL, which is not part of the SDK Help shipping with Delphi.

As an example of the use of the HTTP protocols, I've decided to write a very specific search application. The program simply hooks onto the Yahoo Web site, searches for a keyword, and retrieves the first hundred sites found. Instead of showing the resulting HTML file, the program parses it to extract only the URLs of the related sites. So the program demonstrates two techniques at once: retrieving a Web page and parsing its HTML code.

After a little testing, I've noticed that the WinInet functions take a lot of time to execute, as they have to gather information from the Web. For this reason, I've decided to implement the program using a background thread for the actual processing. This approach also gives the advantage of being able to start multiple searches at once. The thread class used by the WebFind application receives as input a URL to look for, `strUrl`:

```
type
  TFindWebThread = class(TThread)
  protected
    strAddr, strStatus: string;
    procedure Execute; override;
    procedure AddToList;
    procedure ShowStatus;
```

419 The successor of this API is now embedded in Delphi's the HTTP client library.

```
  public
    strUrl: string;
  end;
```

The class has two output procedures, `AddToList` and `ShowStatus`, to be called inside the `Synchronize` method. (Refer to Chapter 17 for the details on threading.) The code of these two methods sends some results or some feedback to the main form, respectively adding a line to the memo and changing the status bar `SimpleText` property. The key method of the thread is the `Execute` method, of course. Before we look at it, however, let me show you how the thread is activated by the main form:

```
procedure TForm1.BtnFindClick(Sender: TObject);
var
  FindThread: TFindWebThread;
begin
  // create suspended, set initial values, and start
  FindThread := TFindWebThread.Create (True);
  FindThread.FreeOnTerminate := True;
  FindThread.strUrl :=
    'http://search.yahoo.com/bin/search?p=' +
    EditSearch.Text + '&n=100&h=s&b=1';
  FindThread.Resume;
end;
```

The URL string is made of the main address of the search engine, followed by some parameters. The first, *p,* indicates the words you are looking for. The second, *n=100,* indicates the number of sites to retrieve; you cannot use numbers at will but are limited to few alternatives, with 100 being the largest possible value. The *h=s* parameter indicates that the program is supposed to look for Web sites (and not for categories), and the final parameter, *b=1,* indicates the number of the starting element. To get the sites from 101 to 200, you should replace this last parameter with *b=101*.

note The WebFind program works with the server on the Yahoo Web site at the time this book was written and tested. The custom software on the site can change any day, however, which might prevent WebFind from operating correctly.

The `Execute` method of the thread, activated by the `Resume` call, is made of two parts. In the first, the program connects to the HTTP server by calling the `InternetOpen` function and then using the resulting handle in the `InternetOpenURL` call. This second call returns a handle to the URL that you can pass to the `InternetReadFile` function in order to read blocks of data. The data is stored in a local string, and while it's retrieving the data, the program also updates the status bar of the main form. When all the data has been read, the program closes the con-

nection to the URL and the Internet session by calling the `InternetCloseHandle` function twice. Here is the first part of the `Execute` method:

```
procedure TFindWebThread.Execute;
var
  hHttpSession, hReqUrl: HInternet;
  Buffer: array [0..1023] of Char;
  nRead: Cardinal;
  strRead: string;
  nBegin, nEnd: Integer;
begin
  strRead := '';
  hHttpSession := InternetOpen ('FindWeb',
    INTERNET_OPEN_TYPE_PRECONFIG, nil, nil, 0);
  try
    hReqUrl := InternetOpenURL (hHttpSession, PChar(StrUrl),
      nil, 0,0,0);
    strStatus := 'Connected to ' + StrUrl;
    Synchronize (ShowStatus);
    try
      // read all the data
      repeat
        InternetReadFile (hReqUrl, @Buffer,
          sizeof (Buffer), nRead);
        strRead := strRead + string (Buffer);
        strStatus := 'Retrieved ' + IntToStr (Length (strRead)) +
          ' of ' + StrUrl;
        Synchronize (ShowStatus);
      until nRead = 0;
    finally
      InternetCloseHandle (hReqUrl);
    end;
  finally
    InternetCloseHandle (hHttpSession);
  end;
```

The second part extracts the URLs referring to other Web sites from the result, the `strRead` string. The program looks for subsequent occurrences of the `href="http` substring, copying the text up to the closing `>` character. If the found string contains the word `yahoo`, it is considered a local link and omitted from the result. You can find this parsing code in the program source and see the output of this program in Figure 20.12. Notice that I've already gotten the result of a request, but the program is currently retrieving another page, as indicated in the status bar. You can start multiple searches at the same time, but be aware that the results will all be added to the same memo component.

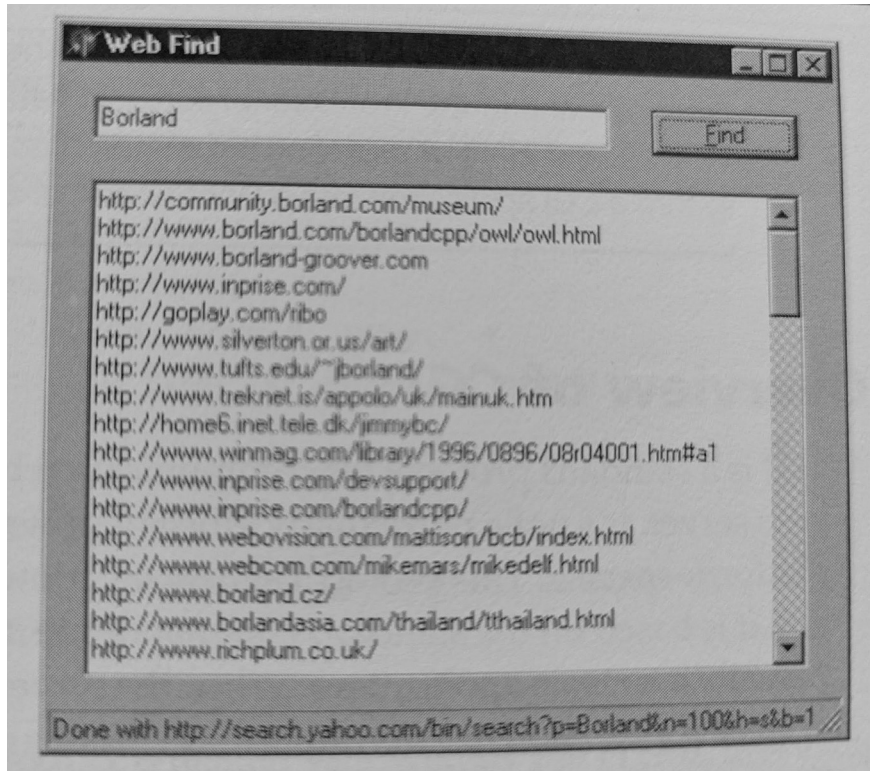


Figure 20.12: The WebFind application can be used to search for a list of sites on the Yahoo search engine. Image based on a picture of the original printed book.

Dynamic Web Pages

When you browse a Web site, you generally download static pages—HTML-format text files—from the Web server to your client computer. As a Web developer, you can create these pages manually, but for most businesses, it makes more sense to build the static pages from information in a database. Using this approach, you're basically generating a snapshot of the data, which is quite reasonable if the data isn't subject to frequent changes. This approach was discussed in the previous chapter.

As an alternative to static HTML pages, you can build dynamic ones. To do this, you extract information directly from a database in response to the browser's request, so that the HTML sent by your application displays current data, not an old snapshot of the data. This approach makes sense if the data changes frequently.

As mentioned earlier, there are a couple of ways you can program custom behavior at the Web server, and these are ideal ways for you to generate HTML pages dynamically. The two most common protocols for programming Web servers are CGI (the Common Gateway Interface) and the Web server APIs[420]. A third technique, Active Server Pages (ASP), is becoming very popular. I'll discuss ASP at the end of the chapter because Delphi 5 includes specific support for it.

note Keep in mind that Delphi's WebBroker technology (available in Delphi 5 both in the Enterprise and in the Professional editions) flattens the differences between CGI, WinCGI, and ISAPI by providing a common class framework. This way, you can easily turn a CGI application into a WinCGI one or upgrade it to use the ISAPI model.

An Overview of CGI

CGI is a standard protocol for communication between the client browser and the Web server. It's not a particularly efficient protocol, but it is widely used and is not platform-specific. This protocol allows the browser both to ask for and to send data, and it is based on the standard command-line input and output of an application (usually a console application). When the server detects a page request for the CGI application, it launches the application, passes command-line data from the page request to the application, and then sends the standard output of the application back to the client computer.

There are many tools and languages you can use to write CGI applications, and Delphi is only one of them. Given the obvious limitation that your Web server must be an Intel-based Windows NT or Windows 95 system, you can build some fairly sophisticated CGI programs in Delphi. Despite the fact that it's called a standard, there are actually different flavors of CGI. Traditional CGI uses the standard command-line input and output, along with environment variables. WinCGI uses an INI file passed as a command-line parameter to the application (instead of environment variables) and specific input and output files (instead of using command-line input/output). Server vendors developed WinCGI primarily for Visual Basic programmers, who cannot access environment variables. Another new variation, called FastCGI, is supposed to make the entire process of calling a CGI application much faster, but it's not widely supported yet.

420 The world has moved a lot in this area, but using CGI and server APIs (like in IIS and Apache modules) is still actual.

To build a CGI program without using any support class, you can simply create a Delphi console application, remove the typical project source code, and replace it with the following statements:

```
program CgiDate;
{$APPTYPE CONSOLE}

uses SysUtils;

begin
  writeln ('HTPP/1.0 200 OK');
  writeln ('CONTENT-TYPE: TEXT/HTML');
  writeln;
  writeln ('<HTML><HEAD>');
  writeln ('<TITLE>Time at this site</TITLE>');
  writeln ('</HEAD><BODY>');
  writeln ('<H1>Time at this site</H1>');
  writeln ('<HR>');
  writeln ('<H3>');
  writeln (FormatDateTime(
    '"Today is " dddd, mmmm d, yyyy,' +
      '"<br> and the time is" hh:mm:ss AM/PM',
    Now));
  writeln ('</H3>');
  writeln ('<HR>');
  writeln ('<I>Page generated by CgiDate.exe</I>');
  writeln ('</BODY></HTML>');
end.
```

CGI programs produce a header followed by the HTML text using the standard output. If you execute this program directly, you'll see the text in a terminal window. If you run it instead from a Web server and send the output to a browser, the formatted HTML text will appear, as shown in Figure 20.13.

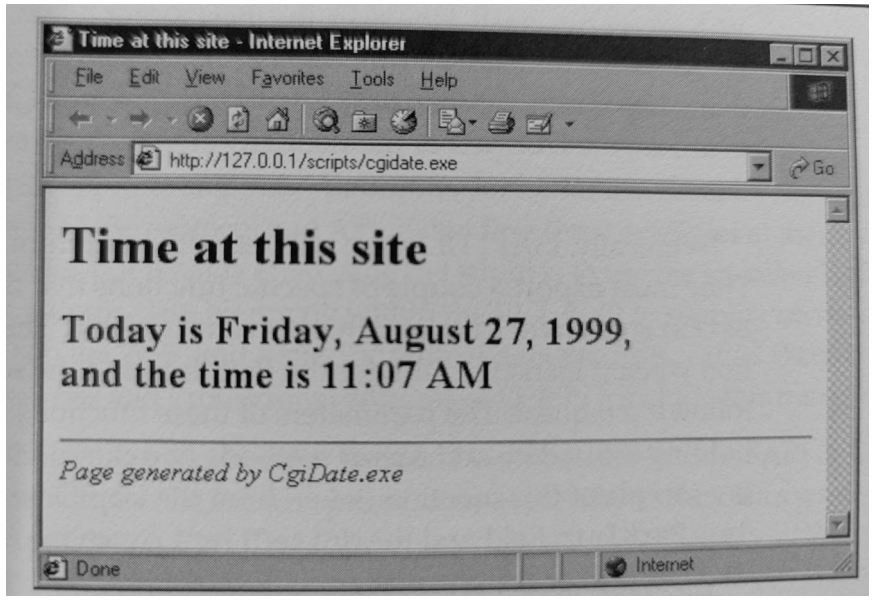


Figure 20.13: The output of the CgiDate application, as seen in Microsoft's Internet Explorer. Image based on a picture of the original printed book.

Building advanced and complex applications with plain CGI requires a lot of work. For example, to extract status information on the HTTP request, you need to access to the relevant environment variables, as in:

```
// get the path name
GetEnvironmentVariable ('PATH_INFO',
  PathName, sizeof (PathName));
```

An Overview of ISAPI/NSAPI

A completely different approach is the use of the Web server APIs, the popular ISAPI (Internet Server API, introduced by Microsoft) and the less common NSAPI (Netscape Server API)[421]. These APIs allow you to write a DLL that the server loads into its own address space and usually keeps in memory for some time. Once it loads the DLL, the server can execute individual requests via threads within the main process, instead of launching a new EXE for every request as it must in CGI applications.

421 The Netscape server is long gone. ISAPI remains a viable option. Apache has been the most popualr web server for many years (and it has a module architecture Delphi supports both on Windows and on Linux), now Nginx is going strong.

When the server receives a page request, it loads the DLL (if it hasn't done so already) and executes the appropriate code, which may launch a new thread or use an existing one to process the page request (the IIS Web server offers thread pooling support to avoid creating a new thread for each request). The DLL code then sends the appropriate data back to the client that requested the page. Because this communication generally occurs in memory, this type of application is much faster than CGI, and a given system will be able to support more simultaneous page requests this way.

The main drawback to server API DLLs is that their tight integration with the server is an Achilles' heel; if the DLL crashes or produces memory leaks, the entire Web server can crash. However, the most recent releases of Microsoft's IIS Web server fix the problem by running the DLL in a *protected* space. Another problem is that when the DLL is in memory, you cannot compile an updated version; you need to unload the DLL first or momentarily stop the Web server (an operation you can do only on a test-bed computer).

Technically, ISAPI DLLs are not very different from plain Windows DLLs. They must export a couple of specific functions that the Web server will call: `GetExtensionVersion` and `HttpExtensionProc`. The server calls the first function when it loads the DLL for the first time and the second function for every following request. The parameters of these functions are complex data structures holding input data and server methods you can call to produce the result. Here is a sample of this function (taken from the IsapiDem example), which uses the `lpszPathInfo` field and the `WriteClient` function:

```
function HttpExtensionProc (
  var ECB: TEXTENSION_CONTROL_BLOCK): DWORD; stdcall;
var
  OutStr: string;
  StrLength: Cardinal;
begin
  with ECB do
  begin
    OutStr :=
      '<HTML><HEAD><TITLE>First Isapi Demo</TITLE></HEAD><BODY>' +
      '<H2><CENTER>First Isapi Demo</CENTER></H2>' +
      'Hello Mastering Delphi Readers...<p><hr>' +
      '<b>Activated by ' + PChar (@lpszPathInfo[1]) + '</b><p>' +
      '<i>From IsapiDLL on ' + DateToStr (Now) +
      ' at ' + TimeToStr (Now) + '</i>' +
      '</body></html>';
    StrLength := Length (OutStr);
    WriteClient(ConnID, PChar (OutStr), StrLength, 0);
  end;
  Result := HSE_STATUS_SUCCESS;
```

```
end;
```

note The program doesn't simply use the `lpzPathInfo` parameter but uses the substring starting with the second characters, to get rid of the initial slash. To be more precise, the expression `PChar (@lpszPathInfo[1])` takes the string starting at the memory address of the second character of the path (a zero-based characters array).

Delphi's WebBroker Technology[422]

The CGI and ISAPI code snippets I've shown you so far demonstrate the plain, direct approach to the protocol and API. Extending these examples at that level is certainly possible, but what is interesting in Delphi is to use the so-called WebBroker technology, a specific class hierarchy within the VCL built to simplify server-side development on the Web, and a specific type of data modules, called WebModules. Both the Enterprise and Professional editions of Delphi 5 include this framework.

Using the WebBroker technology, you can begin developing an ISAPI or CGI application very easily. On the first page (*New*) of the Object Repository, select the Web Server Application icon. The subsequent dialog box will offer you three alternatives, ISAPI, CGI, and WinCGI, as you can see in Figure 20.14. If you select the first option, Delphi will generate the basic structure of an ISAPI application for you.

note As a starting point for your server-side application, you can also use the DB Web Application Wizard, available in the Business page of the File ➢ New dialog box. This wizard generates a program with a table or query connected to a DataSetTableProducer. It can be helpful, but the generated code is really very limited.

422 WebBroker remains relevant today. It has been extended with support for Apache modules and the Linux OS, it is the foundation of DataSnap REST and or RAD Server, and it can now also be used with WebStencils. WebBroker design was good and it remains a solid foundation.

Figure 20.14: The alternative options for building a Web server application in Delphi Enterprise. Image from the original book.

The application that Delphi generates (no matter which type you choose) is based on the `TWebModule` class, a container very similar to a data module. The WebModule code is similar to that of a data module, as we'll see in a moment, but the code of the library is worth looking at:

```
library Project1;

uses
  WebBroker, ISAPIApp,
  Unit1 in 'Unit1.pas' {WebModule1: TWebModule};

{$R *.RES}

exports
  GetExtensionVersion,
  HttpExtensionProc,
  TerminateExtension;

begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

note In Delphi 5, your code must refer to the new WebBroker unit. Existing WebBroker applications referring to the HTTPApp unit must be updated or they won't compile. This change has been introduced to reduce the restrictions on the use of run-time packages for components of Web server applications.

Although this is a library that exports the ISAPI functions, the code looks similar to that of an application. However, it uses a trick—the `Application` object used by this program is not the typical global object of class `TApplication` but an object of a new class. This new `Application` object is of class `TISAPIApplication` (or `TCGIApplication` if you've built that type of application), which derives from `TWebApplication`.

Although these application classes provide the foundations, you won't generally use them very often (just as you don't use the `Application` object very often in a form-based Delphi application). The most important operations take place in the WebModule. This component derives from `TCustomWebDispatcher`, which provides support for all the input and output of our programs.

In fact, the `TCustomWebDispatcher` class defines `Request` and `Response` properties, which store the client request and the response we're going to send back to the client. Each of these properties is defined using a base abstract class (`TWebRequest` and `TWebResponse`), but an application initializes them using a specific object (such as the `TISAPIRequest` and `TISAPIResponse` subclasses). These classes make available all the information passed to the server, and so you have a single, simple approach to accessing all the information. The same is true of a response, which is very easy to manipulate. One advantage to this approach is that an ISAPI DLL written with this framework is very similar to a CGI application; in fact, they are frequently identical in the source code you write.

If this is the structure of Delphi's framework, how do you write the application code? Well, in the WebModule you can use the Actions editor (shown in Figure 20.15), to define a series of actions (stored in the `Actions` array property) depending on the *path name* of the request. This path name is a portion of the CGI or ISAPI application's URL, which comes after the program name and before the parameters, such as `path1` in the following URL:

```
http://www.website.com/scripts/cgitest.exe/path1?param1=date
```

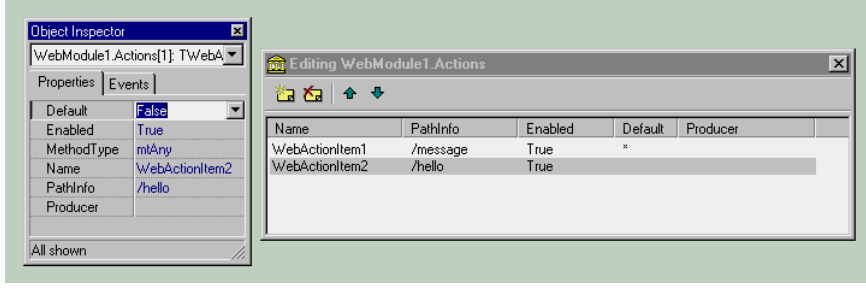


Figure 20.15: The Actions property editor of the WebModule, along with the properties of one of the actions in the Object Inspector. Image from the original book.

By providing different actions, your application can easily respond to requests with different path names, and you can assign different producer components or call a different `OnAction` event handler for every possible path name. Of course, you can omit the path name to handle a generic request. Consider also that instead of basing your application on a WebModule, you can use a plain data module and add a Web-

Dispatcher component to it. This is a good approach if you want to turn an existing Delphi application into a Web server extension. The WebModule incorporates the WebDispatcher and doesn't require it as a separate component.

note The Actions of the WebDispatcher have absolutely nothing to do with the Actions stored in a `TActionList` component.

When you define the accompanying HTML pages that launch the application, the links will make page requests to the URLs for each of those paths. Having one single ISAPI DLL that can perform different operations depending on a parameter (in this case the path name) allows the server to keep a copy of this DLL in memory and respond much faster to user requests. The same is partially true for a CGI application: The server has to run several instances but can cache the file and make it available faster.

The `OnAction` event is where you put the code to specify the *response* to a given *request,* the two main parameters passed to the event handler. Here is a simple example:

```
procedure TWebModule1.WebModule1WebActionItem1Action(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content :=
    '<HTML><HEAD><TITLE>Hello Page</TITLE></HEAD><BODY>' +
    '<H1>Hello</H1>' +
    '<hr><I>Page generated by Marco</I>' +
    '</BODY></HTML>';
end;
```

The `Content` property of the `Response` parameter is where you enter the HTML code that you want users to see. The only drawback of this code is that the output in a browser will be correctly displayed on multiple lines, but looking at the HTML source code, you'll see a single line corresponding with the entire string. To make the HTML source code more readable, by splitting it up onto multiple lines, you can insert the `#13` newline character.

To let other actions handle this request, you'll set the last parameter, `Handled`, to `False`. Otherwise, the default value is `True`, and once you've handled the request with your action, the WebModule assumes you're finished. Most of an ISAPI application's code will be in the `OnAction` event handlers for the actions defined in the WebModule container. These actions receive a request from the client and return a response using the `Request` and `Response` parameters.

When using the producer components, your `OnAction` event often returns, as `Response.Content`, the `Content` of the producer component, with a simple assignment. In Delphi 5, you can shortcut this code by assigning a producer component to the `Producer` property of the action itself, with no need to write these simple event handlers any more.

Building a Multipurpose WebModule

To demonstrate how easily you can build a feature-rich server-side application using Delphi's support, I've created the BrokDemo example. This example can be compiled as a CGI or an ISAPI application, simply by choosing the proper project file. The WebModule is shared by the two projects, without any difference in the source code, a practical proof that using the WebBroker framework you can move from ISAPI to CGI and vice versa. In practice, I tend to test the programs with CGI (to avoid having to stop the server to free the library and recompile it) and then deploy them with ISAPI.

note If your aim is to build an ISAPI application you can also use a specific ISAPI DLL debugging tool. One of such tools, called IntraBob, has been built by Bob Swart and is available on his Web site (`www.drbob42.com`) and is freeware[423].

A key element is the list of actions we're going to support with this application, which you can see in the Actions editor in Figure 20.16. Actions are visible also in the Designer of the Web data module, so you can graphically see their relationship with database objects, as shown by the figure[424]. If you examine the figure or the source code, you'll notice that I've given a specific name to every action. I've also given meaningful names to the `OnAction` event handlers. For instance, `TimeAction` as a method name should be much more understandable than the `WebModule1WebActionItem1Action` name automatically generated by Delphi.

Every action has a different path name, with one of them marked as default and executed even if no path name is specified. The first interesting idea in this program is the use of two PageProducer components, used for the initial and final portion of every page, `PageHead` and `PageTail`. Centralizing this code makes it easier to modify it, particularly if it is based on external HTML files. The HTML produced by these components is added at the beginning and the end of the resulting HTML in the `OnAfterDispatch` event handler of the Web module:

423 While Bob Swart is still active, I don't know if this tool still exists.

424 This designer doesn't exist any more, as covered in the database chapters.

```
procedure TWebModule1.WebModule1AfterDispatch(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageHead.Content +
    Response.Content + PageTail.Content;
end;
```

I'm adding the initial and final HTML at the end of the page generation simply because this allows the components to produce the HTML as if they were making all of it. Starting with some HTML in the `OnBeforeDispatch` event means that you cannot directly assign the producer components to the actions, or the producer component will override the `Content` you've already provided in the response.

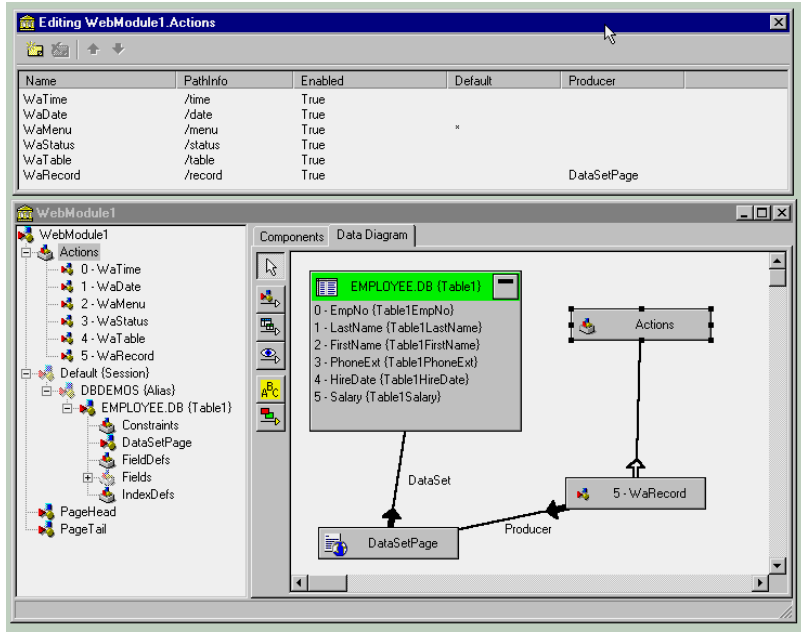


Figure 20.16: The actions of the BrokDemo example, as shown by the Actions editor and the Data Module Designer. Image from the original book. Notice that this Data Diagram doesn't exist any more.

The `OnBeforeDispatch` event fetches the script name, to make it available to the PageProducer component events (which don't receive as parameter the `Request`). Here are two code snippets showing this combined effect:

```
procedure TWebModule1.WebModule1BeforeDispatch(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  // code shared by all actions
```

```
  ScriptName := Request.ScriptName;
  Table1.Open;
end;

procedure TWebModule1.PageTailHTMLTag(Sender: TObject;
  Tag: TTag; const TagString: String; TagParams: TStrings;
  var ReplaceText: String);
begin
  if TagString = 'script' then
    ReplaceText := ScriptName;
end;
```

This code is activated to expand the `<#script>` tag of the `PageTail` component's `HTMLDoc` property. The code of the time and date actions is straightforward. The really interesting part begins with the Menu path, which is the default action. In its `OnAction` event handler, the application simply builds a list of the available actions, providing a link to each of them (with an anchor, an <a> tag) in a `for` loop:

```
procedure TWebModule1.MenuAction(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  Response.Content := '<H3>Menu</H3><ul>'#13;
  for I := 0 to Actions.Count - 1 do
    Response.Content := Response.Content +
      '<li> <a href="' + ScriptName +
      Action[I].PathInfo + '"> ' + Action[I].Name + '</a>'#13;
  Response.Content := Response.Content + '</ul>';
end;
```

Another action of the BrokDemo example provides users with a list of the system settings related to the request, something that is quite useful for debugging purposes. It is also instructive to learn how much information, and exactly what information, the HTTP protocols transfer from a browser to a Web server and vice versa. To produce this list, the program looks for the value of each property of the `TWebRequest` class, as this initial snippet demonstrates:

```
procedure TWebModule1.StatusAction(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  Response.Content := '<H3>Status</H3>'#13 +
    'Method: ' + Request.Method + '<br>'#13 +
    'ProtocolVersion: ' + Request.ProtocolVersion + '<br>'#13 +
    'URL: ' + Request.URL + '<br>'#13 +
```

```
    'Query: ' + Request.Query + '<br>'#13 + ...
```

Dynamic Database Reporting

The BrokDemo example defines two more actions, indicated by the `/table` and `/record` path names. For these two last actions, our program produces a main list of names and then displays the details of one record, using a DataSetTableProducer component to format the entire table and a DataSetPageProducer component to build the record view. Here are the properties of these two components:

```
object DataSetTableProducer1: TDataSetTableProducer
  DataSet = Table1
  OnFormatCell = DataSetTableProducer1FormatCell
end
object DataSetPage: TDataSetPageProducer
  HTMLDoc.Strings = (
    '<H3>Employee: <#LastName></H3>'
    '<ul><li> Employee ID: <#EmpNo>'
    '<li> Name: <#FirstName> <#LastName>'
    '<li> Phone: <#PhoneExt>'
    '<li> Hired On: <#HireDate>'
    '<li> Salary: <#Salary></ul>')
  OnHTMLTag = PageTailHTMLTag
  DataSet = Table1
end
```

To produce the entire table, we simply connect the DataSetTableProducer to the `Producer` property of the corresponding actions, without providing any specific event handler. The table is made more powerful by adding internal links to the specific records. The following code is executed for each cell of the table but activated only for the first column or the first row (the one with the title):

```
procedure TWebModule1.DataSetTableProducer1FormatCell(
  Sender: TObject; CellRow, CellColumn: Integer;
  var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellColumn = 0) and (CellRow <> 0) then
    CellData := '<a href="'' + ScriptName + '/record?LastName=' +
      Table1['LastName'] + '&FirstName=' + Table1 ['FirstName'] +
      '"> ' + CellData + ' </a>';
end;
```

You can see the result of this action in Figure 20.17. When the user selects one of the links, the program is called again, and it can check the `QueryFields` string list and

extract the parameters from the URL. It then uses the values corresponding to the table fields used for the record search (which is based on the `FindNearest` call).

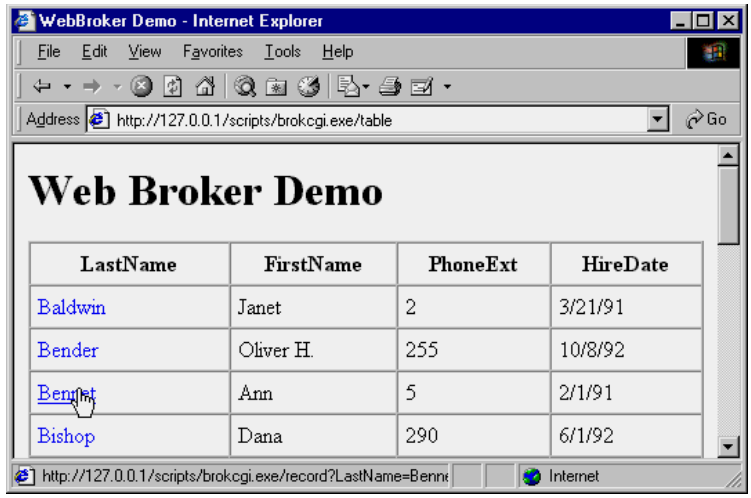


Figure 20.17: The output corresponding to the table path of the BrokDemo example, which produces an HTML table with internal links by using Delphi components. Image from the original book.

```
procedure TWebModule1.RecordAction(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Table1.Open;
  // go to the requested record
  Table1.FindNearest ([Request.QueryFields.Values['LastName'],
    Request.QueryFields.Values['FirstName']]);
  // get the output
  Response.Content := Response.Content +
    DataSetPage.Content;
end;
```

note The example we've just built accesses a Paradox table via the BDE. The CGI version executes once for every request and will actually load and unload the BDE each time it runs. As alternatives, you can consider using ISAPI, accessing the data from a plain file or running another BDE application on the server, so that the BDE will remain loaded in memory.

Of Queries and Forms

The previous example used some of the HTML producer components introduced earlier in this chapter. There is another component of this group we haven't used

yet, the QueryTableProducer. As we'll see in a moment, this component makes building even complex database programs a breeze. Suppose you want to search for some customers in a database. You might construct the following HTML form (embedded in an HTML table for better formatting):

```
<h4>Customer QueryProducer Search Form</h4>
<form action="/scripts/CustQueP.dll/search" method="POST">
<table>
<tr><td>State:</td>
  <td><input type="text" name="State"></td></tr>
<tr><td>Country:</td>
  <td><input type="text" name="Country"></td></tr>
<tr><td></td>
  <td><center><input type="Submit"></center></td></tr>
</form>
```

note As in Delphi, an HTML form hosts a series of controls (typically, things like input fields). There are visual tools to help you design these forms, or you can manually enter the proper HTML code. The available controls include buttons, input text (or edit boxes), selections (or combo boxes), and radio buttons (or input buttons). You can define buttons as specific types, such as Submit or Reset, which imply standard behaviors. An important element of forms is the *request method,* which can be either POST (data is passed behind the scenes, and you receive it in the `ContentFields` property) or GET (data is passed as part of the URL, and you extract it from the `QueryFields` property).

You can see the output of this form in Figure 20.18. There is another important element to notice: the names of the input components (*State* and *Country*), which should match the parameters of a Query component[425]:

```
select
  Company, State, Country
from
  CUSTOMER.DB
where
  State = :State or Country = :Country
```

This code is used in the CustQueP (Customer Query Producer) example. To build it, I've placed a Query component inside the WebModule and generated the field objects for it. In the same WebModule I've added a QueryTableProducer component connected to the `Producer` property of the `/search` action. The ISAPI DLL will generate the proper response. How does this work? When we activate the QueryTableProducer component by calling its `Content` function, it initializes the

425 Interesting that this SQL query based on parameters avoids the risk of "SQL injection" a common pitfall in the history of web development – but something not frequently discussed at the time this book had been written.

Query component by obtaining the parameters from the HTTP request. The component can automatically examine the request method and then use either the `QueryFields` property (if the request is a GET) or the `ContentFields` property (if the request is a POST).

Figure 20.18: The HTML form used by the CustQueP example has been formatted by placing the controls in an HTML table. Image from the original book.

One problem with using a static HTML form as we did before is that it doesn't tell us which states and countries we can search for. To address this, we can use a selection control instead of an edit control in the HTML form. However, if the user adds new records to the database table, we'll need to update the element list automatically. As a final solution, we can design the ISAPI DLL to produce a form on the fly, and we can fill the selection controls with the available elements.

We'll generate the HTML for this page in the `/form` action, which we've connected to a PageProducer component. The PageProducer contains the following HTML text, which embeds two special tags:

```
<h4>Customer QueryProducer Search Form</h4>
<form action="CustQueP.dll/search" method="POST">
<table>
<tr><td>State:</td><td><select name="State">
<#State>
</select>
</td></tr>
<tr><td>Country:</td><td><select name="Country">
<option> </option>
<#Country>
</select>
</td></tr>
<tr><td></td><td><center><input type="Submit"></center></td></tr>
</form>
```

You'll notice that the tags have the same name as some of the table's fields. When the PageProducer encounters one of these tags, it adds an `<option>` HTML tag for every distinct value of the corresponding field. Here's the `OnTag` event-handler's code, which is quite generic and reusable:

```
procedure TWebModule1.PageProducer1HTMLTag(
  Sender: TObject; Tag: TTag; const TagString: String;
  TagParams: TStrings; var ReplaceText: String);
begin
  ReplaceText := '';
  Query2.SQL.Clear;
  Query2.SQL.Add ('select distinct ' +
    TagString + ' from customer');
  try
    Query2.Open;
    try
      Query2.First;
      while not Query2.EOF do
      begin
        ReplaceText := ReplaceText +
          '<option>' + Query2.Fields[0].AsString +
          '</option>'#13;
        Query2.Next;
      end;
    finally
      Query2.Close;
    end;
  except
    ReplaceText := '{wrong field: ' + TagString + '}';
  end;
end;
```

This method used a second Query component, which I manually placed on the form and connected to the DBDemos database, and it produces the output shown in Figure 20.19.

Finally, this Web server extension, like many others we've built, allows the user to view the details of a specific record. As in the last example, we can accomplish this by customizing the output of the first column (column zero), which is generated by the QueryTableProducer component:

```
procedure TWebModule1.QueryTableProducer1FormatCell(
  Sender: TObject; CellRow, CellColumn: Integer;
  var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellColumn = 0) and (CellRow <> 0) then
    CellData := '<a HREF="' + Request.ScriptName +
      '/record?' + CellData + '">' + CellData + '</a>'#13;
end;
```

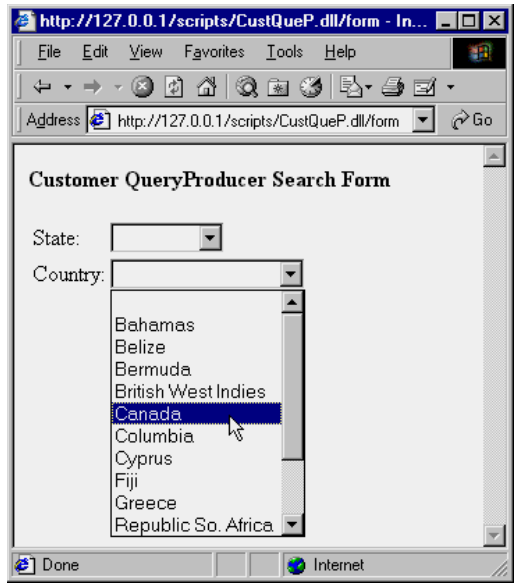


Figure 20.19: The form action of the `CustQueP` example produces an HTML form with a selection component dynamically updated to reflect the current status of the database. Image from the original book.

The action for this link is `/record`, and we'll pass a specific element after the `?` parameter (without the parameter name, which is slightly nonstandard). The code we use to produce the HTML tables for the records doesn't use the producer components as we've been doing; instead, it is very similar to the code of an early ISAPI example:

```
procedure TWebModule1.RecordAction(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  if Request.QueryFields.Count = 0 then
    Response.Content := 'Record not found'
  else
  begin
    Query2.SQL.Clear;
    Query2.SQL.Add ('select * from customer ' +
      'where Company="' + Request.QueryFields[0] + '"');
    Query2.Open;
    Response.Content :=
      '<HTML><HEAD><TITLE>Customer Record</TITLE></HEAD><BODY>'#13 +
      '<H1>Customer Record: ' + Request.QueryFields[0] +
      '</H1>'#13 +
      '<table border>'#13;
    for I := 1 to Query2.FieldCount - 1 do
```

```
      Response.Content := Response.Content +
        '<tr><td>' + Query2.Fields [I].FieldName +
        '</td>'#13'<td>' + Query2.Fields [I].AsString +
        '</td></tr>'#13;
    Response.Content := Response.Content +
      '</table><hr>'#13 +
      // pointer to the query form
      '<a HREF="' + Request.ScriptName + '/form">' +
      ' Next Query </a>'#13 +
      '</BODY></HTML>'#13;
  end;
end;
```

A Web Hit Counter

The server-side applications we've built up to now were based only on text. Of course, you can easily add references to existing graphics files. What's more interesting, however, is to build server-side programs capable of generating graphics that change over time.

A typical example is a page hit counter. To write a Web counter, we save the current number of hits to a file and then read and increase the value every time the counter program is called. How do we return this information? If all we need is some HTML text with the number of hits, the code is straightforward:

```
procedure TWebModule1.WebModule1WebActionItem1Action(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  nHit: Integer;
  LogFile: Text;
  LogFileName: string;
begin
  LogFileName := 'WebCont.log';
  System.Assign (LogFile, LogFileName);
  try
    // read if the file exists
    if FileExists (LogFileName) then
    begin
      Reset (LogFile);
      Readln (LogFile, nHit);
      Inc (nHit);
    end
    else
      nHit := 0;
    // saves the new data
    Rewrite (LogFile);
    Writeln (LogFile, nHit);
```

```
  finally
    Close (LogFile);
  end;
  Response.Content := IntToStr (nHit);
end;
```

What's a little more interesting is to create a graphical counter that can be easily embedded into any HTML page. There are basically two approaches for building a graphical counter: you can prepare a bitmap for each digit up front and then combine them in the program, or you can simply let the program draw over a memory bitmap to produce the graphic you want to return. In the WebCount program, I've chosen this second approach.

Basically, we can create an Image component that holds a memory bitmap, which we can paint on with the usual methods of the `TCanvas` class. Then we can attach this bitmap to a `TJpegImage` object. Accessing the bitmap through the JpegImage component converts the image to the JPEG format. At this point, we can save the JPEG data to a stream and return it. As you can see, there are many steps, but the code is not really complex:

```
  // create a bitmap in memory
  Bitmap := TBitmap.Create;
  try
    Bitmap.Width := 120;
    Bitmap.Height := 25;
    // draw the digits
    Bitmap.Canvas.Font.Name := 'Arial';
    Bitmap.Canvas.Font.Size := 14;
    Bitmap.Canvas.Font.Color := RGB (255, 127, 0);
    Bitmap.Canvas.Font.Style := [fsBold];
    Bitmap.Canvas.TextOut (1, 1, 'Hits: ' +
      FormatFloat ('###,###,###', Int (nHit)));
    // convert to JPEG and output
    Jpeg1 := TJpegImage.Create;
    try
      Jpeg1.CompressionQuality := 50;
      Jpeg1.Assign(Bitmap);
      Stream := TMemoryStream.Create;
      Jpeg1.SaveToStream (Stream);
      Stream.Position := 0;
      Response.ContentStream := Stream;
      Response.ContentType := 'image/jpeg';
      Response.SendResponse;
      // the response object will free the stream
    finally
      Jpeg1.Free;
    end;
  finally
    Bitmap.Free;
```

```
end;
```

The three statements responsible for returning the JPEG image are the two that set the `ContentStream` and `ContentType` properties of the `Response` and the final call to `SendResponse`. The content type must match one of the possible MIME types accepted by the browser, and the order of these three statements is relevant. There is also a `SendStream` method in the `Response` object, but it should be called only after sending the type of the data with a separate call.

You can see the effect of this program in Figure 20.20. To obtain it I've added the following code to an HTML page:

```
<img src="http://127.0.0.1/scripts/webcount.exe"
  border=0 alt="hit counter">
```

Figure 20.20: The graphical Web hit counter in action. Image from the original book.

Handling Mail Feedback

As a final example, I'm going to show you how to use a server-side application to generate mail messages with special formatting; these messages will be handled by a custom program. Why generate e-mail messages instead of saving data locally on the server computer? E-mail protocols can be used for powerful and complex transactions between two users who are not permanently connected to the Internet. In this case, in fact, using a direct socket-based program doesn't work. The mail server offers a way to desynchronize the server and the client applications.

In other words, a remote user knows when there has been activity on the site, or when someone has used a program that generated some e-mail, simply by checking

a mail account. If you have an extra e-mail account (and it is quite easy to get one for free nowadays), you can also automate the verification process by writing a program that extracts and processes the mail messages automatically.

A CGI Mail Server

To illustrate both sides of the e-mail connection, I've written two simple programs. The first is a server-side CGI application that uses the SMTP FastNet component (described earlier in this chapter). Here is the DFM file for the data modules:

```
object WebModule1: TWebModule1
  Actions = <
    item
      Default = True
      Name = 'WebActionItem1'
      OnAction = WebModule1WebActionItem1Action
    end>
  object Mail: TNMSMTP
    Host = 'XXX'
    Port = 25
    ReportLevel = 0
    UserID = 'marco'
    PostMessage.ToAddress.Strings = (
      'marco@AST')
    PostMessage.Body.Strings = (
      'Subscription')
    PostMessage.Subject = 'Subscribe'
  end
end
```

Of course, you'll need to update the program with a proper address for the SMTP host, a proper `UserID`, and the e-mail address where you want to send the message, `PostMessage.ToAddress`. The program has only one method, the handler for its only action. This method extracts the information the user must have entered in a proper HTML form, sends the mail message, and displays a result message to the user:

```
procedure TWebModule1.WebModule1WebActionItem1Action(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  OutString: string;
begin
  OutString := Request.ContentFields.Values ['firstname'];
  OutString := OutString + ' ' +
    Request.ContentFields.Values ['lastname'];
  OutString := OutString + ' [' +
    Request.ContentFields.Values ['email'] + ']';
```

```
  // send email
  Mail.PostMessage.FromAddress := OutString;
  Mail.Connect;
  Mail.SendMail;
  Mail.Disconnect;

  Response.Content := Response.Content +
    '<HTML><HEAD><TITLE>Newsletter</TITLE></HEAD>' +
    '<BODY><H1>Newsletter</H1><H2>Subscription received</H2><hr>' +
    '<H4>You''re registered in our database as <br>' +
    OutString + '</h4>' +
    '</BODY></HTML>';
end;
```

The HTML form used by the program is visible in Figure 20.21, and its HTML source code is listed below. The HTML code is interesting for two reasons. First, the edit boxes of the HTML form used for input have a `name`, which is used by the CGI application to retrieve the input data. Second, it uses a simple script written with the JavaScript language (see the `onSubmit` section of the form) to check whether the edit boxes are empty before sending the request. Here is the HTML code:

```
<HTML><HEAD>
  <TITLE>Subscription</TITLE>
</HEAD>

<BODY  bgcolor="#FFFFFF">
<H1>Subscription Module</H1>

Fill the following form to subscribe to my newsletter.

<form
name="subscribe"
action="/cgi-bin/WebMail2.exe/new"
method="post"
onSubmit="
  if(!subscribe.lastname.value ||
    !subscribe.firstname.value ||
    !subscribe.email.value)
  {
    alert('All fields must be filled');
      return false;
  };">
<TABLE>
<TR>
  <TD><B>First Name:</B></TD>
  <TD><input name="firstname">
</TD>
</TR>
<TR>
  <TD><B>Last Name:</B></TD>
```

```
  <TD><input name="lastname"></TD>
</TR>
<TR>
  <TD><B>Email:</B></TD>
  <TD><input name="email"></TD>
</TR>
<TR>
  <TD></TD>
  <TD><input type=submit value="Send"></TD>
</TR>
</TABLE>
</form>

</BODY></HTML>
```

Figure 20.21: The HTML input form of the WebMail program. Image from the original book.

If you know C or C++, you'll probably find JavaScript familiar, as it uses a similar syntax. I don't want to discuss JavaScript in detail here, but I wanted to show you that the client side of a Web-based program can be made even more powerful by incorporating scripts and using many other features of HTML[426]. A front-end built in an HTML form is not as flexible as a native Windows application can be, but HTML with some scripting offers all the basic features required to obtain a decent input form[427].

426 JavaScript was really that popular at the time this book was written, but I was correct in assuming it was going to become important – and it is today!

Retrieving Mail-Based Requests

The other side of the application is the program used to retrieve the mail messages generated by the CGI server extension. This program is called GetMail and is available in the same directory as the WebMail program. The GetMail program is based on a form and has an NMPOP3 component. Again you'll have to update the program with the proper `Host` name, `UserID`, and `Password`.

The form also has a list box and two buttons, which are used to move all the new subscribers to the list box and save them from the list box to a file. A memo component is used to show errors and log messages. The program connects to the POP3 mail server, reads the number of messages, and then scans each of the messages in reverse order. Calling the `GetMailMessage` method fills the `MailMessage` property of the component. At this point, the program checks to see if this is a subscription message (by looking at the `Subject` field), extracts the sender, adds its name and e-mail address to the list box, and removes the message from the server. Any message that has a different caption is not removed; instead, its text is added to the Memo component.

Active Server Pages

One of the biggest problems with ISAPI and CGI applications is the fact that they follow the rules of the HTTP protocol, which is stateless. Every request arriving from any user is considered a brand-new request. There are many techniques you can use to solve this problem, including the use of cookies (quite simple with the WebBroker architecture) and the use of hidden form fields that pass a user ID from page to page.

Another solution is the use of a new Microsoft technology, Active Server Pages (ASP)[428]. The idea behind ASP is to add scripts to the HTML code, so that part of the text on a Web page is directly available while other information can be added at run time on the server. The client receives a plain HTML file. The difference between this approach and ISAPI is that you don't need to recompile a program on the server

---

427 Well, a bit more than that, but it was difficult to foresee it before Gmail and Goole maps and all of the other apps that followed, along with AJAX and later web services. Still, a good desktop app can be have a better UI than most web apps and can be build for a fraction of the cost.

428 ASP was later, even if slowly, abandoned in favor of ASP.NET. Delphi offers this as well, for a little time. None of the related support remains in Delphi today.

to see a change; you simply update the script. ASP offers a complex model, where you can attach persistent data to a session (for example, a user moving from page to page of a section of your Web site) and to the entire application (the section of the Web site, regardless of the user).

ASP is quite a complex technology, and here I can only discuss it in relation to Delphi programming. One of the features of ASP is that it allows you to create COM objects within a script, and you can write those COM objects in Delphi. Delphi 5 even provides specific support classes and a wizard to help you build ASP objects. Compared to ISAPI or CGI, one of the advantages is that your ASP object built in Delphi can get access to session and application information, exactly as an ASP script does. This means we automatically get extra features as persistent user data built into our server-side object. By building a compiled ASP object, we can also increase the speed of complex server-side code. (ASP scripts are not always the best solution in term of performance.) But, again, I don't want to discuss ASP in detail, only focus on Delphi support.

To try this out, simply create a new ActiveX library, and then start the Active Server Object Wizard (from the ActiveX page of the File ➢ New dialog box). As you can see in Figure 20.22, the wizard has a couple of options. You can build an object integrated with the ASP script by selecting the Page-Level Event Methods radio button, or an internal object (which can be installed as an MTS object) by using the Object Context option. Only in the first case does the object automatically handle the `OnStartPage` method, which receives as parameter the *scripting context*. In both cases, however, the VCL classes you inherit from (`TASPObject` and `TASPMTSObject`, respectively) have properties to access the `Request`, `Response`, `Session`, `Server`, and `Application` ASP objects.

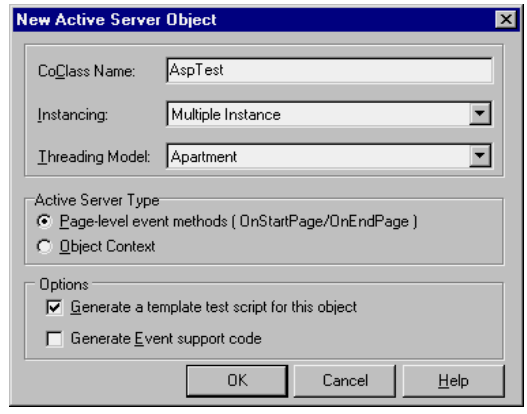


Figure 20.22: The new Active Server Object Wizard. Image from the original book.

Once you've created the ASP object with the wizard (I've used the Page-Level Event Methods option for the AspTest example), Delphi will bring up the type library editor, where you can prepare a list of properties and methods for your ASP object. Simply add the features you need, and then write their code. For example, you can write the following simple test method

```
procedure Tasptest.ShowData;
begin
  Response.Write ('<h3>Delphi wrote this text</h3>');
end;
```

and activate it from the following ASP script (only slightly modified from the demo script the Delphi wizard will generate for you):

```
<h4>Message</h4>
<% Set DelphiASPObj = Server.CreateObject("asptest1.asptest")
   DelphiASPObj.showData
%>
```

The interesting element is that the same script (or another ASP script of the same application) can also set global values our Delphi object can access. Similarly, multiple objects can communicate, setting global variables for the application and session variables for the specific user. For example, we can add the following text to the ASP page:

```
<h4>hello</h4>
<%
   Session.Value("UserName") = "Marco"
   DelphiASPObj.Hello
 %>
```

I've written the code used to set the property and the method invocation one after the other, but they can even be in different pages. This new *dynamic* property (Microsoft's term for these values added to an object) is saved in the session, so it depends on the current user. The `Hello` method can use the username to welcome him:

```
procedure Tasptest.Hello;
var
  strName: string;
begin
  strName := Session ['UserName'];
  Response.Write ('<h3>Hello, ' + strName + '</h3>');
  Response.Write ('<p>Page started at ' + TimeToStr (StartTime);
end;
```

You can see the result of this and the previous method combined in Figure 20.23. The last line of the method uses a variable I've set when the page is first loaded, in

the `OnStartPage` method (despite the name this is not an event handler, but a method the ASP engine will call as the page containing the object is activated):

```
procedure Tasptest.OnStartPage(const AScriptingContext: IUnknown);
begin
  inherited OnStartPage(AScriptingContext);
  StartTime := Now;
end;
```

Technically, this method retrieves the scripting context. The `TASPObject` base class uses the method to initialize all the ASP objects (including the two, `Response` and `Session`, I use in the code), surfacing them as properties.

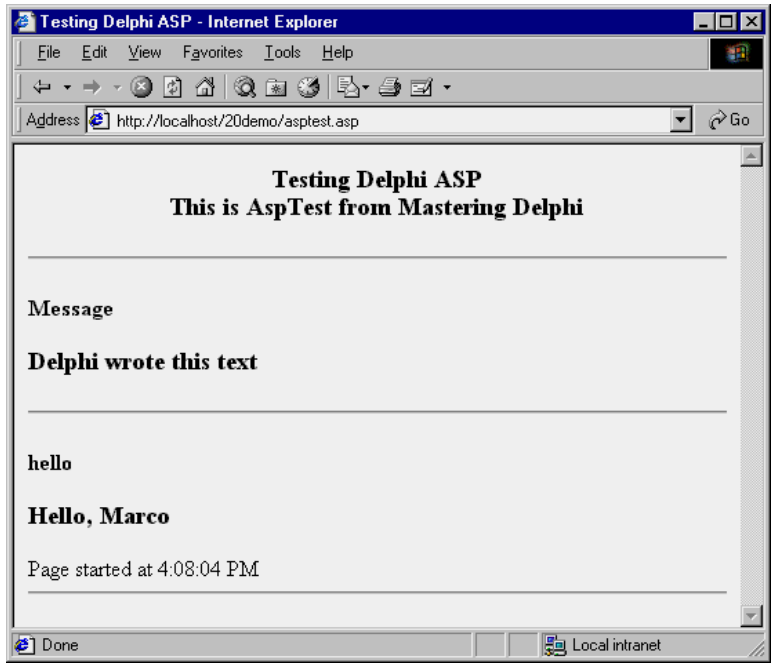


Figure 20.23: The Web page generated by the AspTest object I've built with Delphi. Image from the original book.

To generate more complex HTML from the Delphi ASP object, you can use Producer components, optionally connecting them to a dataset. In the AspTest example, I've added a Table component and a DataSetTableProducer, connected them as usual, and written the following code to activate it:

```
procedure Tasptest.ShowTable;
begin
  DataModule1 := TDataModule1.Create (nil);
  try
```

```
    Response.Write (DataModule1.DataSetTableProducer1.Content)
  finally
    DataModule1.Free;
  end;
end;
```

It will actually make more sense to create the data module when the COM object is created and destroyed (overriding `Initialize` and `Destroy`) or when the page is loaded and unloaded (with `OnStartPage` and `OnEndPage`).

What's Next?

In this long chapter, I've introduced you to some Internet-related programming techniques: the generation of HTML code, the use of ActiveX controls and ActiveForm on Web pages, the low-level socket connections, some high-level Internet protocols, CGI and ISAPI server-side technologies, the WebBroker framework, and ASP.

Internet programming technologies are getting a great deal of attention, but they are also very unstable and immature, and many alternatives often lead to similar results. For this reason, I've tried to give you a very broad overview of the available technologies, applying them to a broad range of examples. You will learn more about this topic in the next chapter's discussion of the new Internet Express architecture, which Delphi 5 adds to the MIDAS distributed database application technology.

Chapter 21: Multitier Database Applications

Large companies often have broader needs than applications using local database and SQL servers can meet. In the past few years, Borland has started addressing the needs of large corporations, and it has even changed its own name to Inprise to underline this new enterprise focus[429]. There are many different technologies Delphi is targeting: three-tier architectures based on Windows NT and DCOM, CORBA architectures based on NT and Unix servers, TCP/IP and socket applications, and—most of all—Web-based database front ends[430].

---

429 The name change was short lived. Soon afterwards the company got back to use the Borland name, only to later move the development tools software in the CodeGear business unit, which was later sold to Embarcadero, later bought by Idera, Inc.

430 There has big a big shift in these multi-tier technologies, with HTTP-based connectivity taking the lion's share. DCOM and CORBA still exist, but their use is swindling, and same for sockets.

Chapter 20 showed how to write simple distributed database applications using sockets. This chapter will introduce the key ideas of Delphi's support for three-tier architecture, using MIDAS, DCOM, TCP/IP, MTS, CORBA, and the new Internet Express technology[431]. I'll concentrate more on the programming aspects of these architectures than on installation and configuration (these aspects are subject to change across different operating systems and are too complex to cover thoroughly). The chapter is intended to serve only as an introduction to some complex technologies that are related to Delphi programming but would require an entire separate book to cover, such as CORBA.

Before proceeding, I should emphasize two important elements. First, the tools to support this kind of development are available only in the Enterprise version of Delphi[432], and second, in some cases you'll have to pay a license fee to Borland in order to deploy the necessary server-side software, MIDAS[433]. This second requirement makes this architecture cost-effective mainly for large systems (that is, servers connected to dozens or even hundreds of clients). The license fee is only required for deployment of the server application. It can be a flat fee for the server (regardless of the number of clients that will connect) or a per-client fee if you plan to have only a few clients. The license fee is not required for development or evaluation.

note You spend money on the MIDAS license, but you can save on the SQL server client licenses. Companies have saved tens of thousands of dollars in annual SQL server connection licenses, by connecting the hundreds or thousands of clients to the MIDAS server instead of the SQL server. The MIDAS server only needs one SQL server client connection license, and the end-user clients need none at all.[434]

One, Two, Three Levels

Initially, database PC applications were client-only solutions: the program and the database files were on the same computer. From there, adventuresome program-

431 A lot of these technologies are obsolete or no longer applicable. Some key elements have survived, with changes.

432 This remains true, in terms of DataSnap (the direct MIDAS successor) and RAD Server.

433 A runtime deployment fee is involved in RAD Server, but you can zero it by buying an Architect edition license.

434 This idea of saving on database licenses theoretically still applies today, however database vendors have often changed their licensing schemes to counter it, to the point of asking for payment by company employee, regardless of the actual database users.

mers moved the database files onto a network file server. The client computers still hosted the application software and the entire database engine, but the database files were now accessible to several users at the same time. You can still use this type of configuration with a Delphi application and Paradox files (or, of course, Paradox itself), but the approach was much more widespread just few years ago.

The next big transition was to client/server development, embraced by Delphi since its first version. In the client/server world, the client computer requests the data from a server computer, which hosts both the database files and a database engine to access them. This architecture downplays the role of the client, but it also reduces its requirements for processing power on the client machine. Depending on how the programmers implement client/server, the server can do most (if not all) of the data processing. In this way, a powerful server can provide data services to several less powerful clients.

Naturally, there are many other reasons for using centralized database servers, such as the concern for data security and integrity, simpler backup strategies, central management of data constraints, and so on. The database server is often called a SQL server, because this is the language most commonly used for making queries into the data, but it may also be called a DBMS (DataBase Management System), reflecting the fact that the server provides tools for managing the data, such as support for backup and replication.

Of course, some applications you build may not need the benefits of a full DBMS, so a simple client-only solution might be sufficient. On the other hand, you might need some of the robustness of a DBMS system, but on a single, isolated computer. In this case, you can use a local version of a SQL server, such as Local InterBase (included with both the Professional and Enterprise editions of Delphi). Traditional client/server development is done with a two-tier architecture. However, if the DBMS is primarily performing data storage instead of data and number crunching, the client might contain both user interface code (formatting the output and input with customized reports, data-entry forms, query screens, and so on) and code related to managing the data (also known as *business rules*). In this case, it's generally a good idea to try to separate these two sections of the program and build a logical three-tier architecture. The term *logical* here means that there are still just two computers (that is, two physical tiers), but we've now partitioned the application into three distinct elements.

Delphi 2 introduced support for a logical three-tier architecture with data modules. As you'll recall, a data module is a non-visual container for the data access components of an application, but it often includes several handlers for database-related events. You can share a single data module among several different forms and provide different user interfaces for the same data; there might be one or more data-

input forms, reports, master/detail forms, and various charting or dynamic output forms.

The logical three-tier approach solves many problems, but it also has a few drawbacks. First, you must replicate the data-management portion of the program on different client computers, which might hamper performance, but more of an issue is the complexity it adds to code maintenance. Second, when multiple clients modify the same data, there's no simple way to handle the resulting update conflicts. Finally, for logical three-tier Delphi applications, you must install and configure the BDE on every client computer.

The next logical step up from client/server is to move the data-module portion of the application to a separate server computer and design all the client programs to interact with it. This is exactly the purpose of remote data modules, which were introduced in Delphi 3. Remote data modules run on a server computer—generally called the application server. The application server in turn communicates with the DBMS (which can run on the application server or on another computer). Therefore, the client machines don't connect to the SQL server directly, but indirectly via the application server[435].

At this point there is a fundamental question: Do we still need to install the BDE? The traditional Delphi client/server architecture (even with the logical three tiers) requires you to install the BDE on each client, something quite troublesome when you must configure and maintain hundreds of machines. In the new physical three-tier architecture, you need to install and configure the BDE only on the application server, not on the client computers. This generally means installing the BDE and configuring the drivers and the aliases only on one computer! Since the client programs have only user interface code and are extremely simple to install, they now fall into the category of so-called *thin clients*. To use marketing-speak, we might even call this a *zero-configuration thin-client architecture*. But let us focus on technical issues instead of marketing terminology[436].

---

435 With the advent of extensive web interfaces and mobile applications, the use of multi-tier architectures has further grown in relevance, as the same back-end can be used by multiple clients and because the client side (web or mobile) might offer limited processing, also because of data and algorithm IP protection issues. The shift to HTTP, on top of this, favors stateless and salable architectures and helps with security overall.

436 You clearly don't want to install the BDE, but also database client libraries on the clients (most database don't have mobile clients, for example). The goal today remains zero configuration, stand-alone clients on web, mobile or desktop, which merely use HTTP to connect to the data.

The Technical Foundation: MIDAS

The foundation of Delphi's physical multitier architecture is MIDAS (Middle-tier Distributed Application Services), a collection of distinct technologies that work together to make it easier to build distributed applications with Delphi. Delphi 5 includes the third version of this technology, MIDAS 3, which is also available in C++Builder and JBuilder to allow for distributed multi-platform and multi-language projects[437].

MIDAS is a server-side technology, so you'll have to install it on the middle-tier computer, the one that should provide your client computers with the data extracted from the SQL server database or other data sources. Whether the middle-tier application server and the SQL server reside on two different machines or on the same computer isn't really important and has no effect on the MIDAS architecture. Similarly, MIDAS does not require a SQL server for data storage. MIDAS can serve up data from a wide variety of sources, including SQL, CORBA, other MIDAS servers, or just data computed on the fly.

As you would expect, the client side of MIDAS is extremely thin, and it's easy to deploy. The only file you need is now called Midas.dll (it was called DbClient.dll in past versions), a small (260KB) DLL that implements the `ClientDataSet` and `RemoteServer` components and provides the connection to the application server[438]. This DLL is basically a small, stand-alone database engine. It caches data from a remote data module and enforces the rules requested by the Constraint Broker. In MIDAS 3 you no longer need to register this DLL as a COM server.

The application server uses the same DLL to handle the datasets (called *deltas*) returned from the clients when they post updated or new records. However, the server also requires several other libraries, all of which are installed by MIDAS.

The IAppServer Interface

In previous versions of MIDAS, the two sides of the application communicated using the `IDataBroker` and `IProvider` interfaces. In Delphi 5 these two interfaces are gone, and you instead use the new `IAppServer` interface. (This limits the com-

---

437 While MIDAS as it is presented in this book doesn't exist any more, the DataSnap architecture, still available in Delphi today, is based on a lot of the original MIDAS code and the ClientDataSet component is still implemented in an external library called *midas.dll* today. As of JBuilder, this Java IDE has long been dismissed.

438 As mentioned above, this DLL is still distributed today, for the same exact purpose.

patibility of existing programs with MIDAS 3. The Delphi help file lists the changes required by an application to be ported to this updated architecture.)

The `IAppServer` interface supersedes the features of the `IProvider` interface, introducing a key feature: it is meant to be used with stateless objects. With `IProvider`, the server stored status information about the client program—for example, which records had already been passed to the client. This made it difficult to adapt the server-side objects to stateless connection layers, like CORBA message queues and MTS, and also to move toward HTTP and Web-based support.

Other reasons for moving to this new architecture were to make the system more dynamic (providers are now exported by setting a property, not by changing the type library) and to reduce the number of calls, or *round-trips*, which can affect performance. Now MIDAS makes fewer calls but delivers more data each time.

The `IAppServer` interface has the following methods[439]:

```
AS_ApplyUpdates
AS_GetRecords
AS_DataRequest
AS_GetProviderNames
AS_GetParams
AS_RowRequest
AS_Execute
```

You'll seldom need to call them directly, anyway, as there are Delphi components to be used on the client and server sides of the application that embed these calls, making them easier (and at times even hiding them completely). In practice, in the type library of a remote data module, you'll inherit your own interface from this one, possibly adding new methods. In past versions it was necessary to export specific provider interfaces from the server type library; now, this is not required, so you have less need of customizing the derived interface.

The Connection Protocol

MIDAS defines only the higher-level architecture and can use different technologies for moving the data from the middle tier to the client side. MIDAS supports most of the leading standards, including the following:

DCOM (or Distributed COM): This technology is directly available in Windows NT and 98, and it requires no additional run-time applications on the server. You

439 This interface is still the foundation of DataSnap, although the way to access to it remotely changed over time.

still have to install it on Windows 95 machines. DCOM is basically an extension of COM technology (discussed in Chapters 15 and 16) that allows a client application to use server objects that exist and execute on a separate computer[440].

MTS: DCOM also allows you to use MTS (Microsoft Transaction Server), which provides features such as security, component management, and database transactions. MTS is available in versions for Windows NT and Windows 98.

TCP/IP sockets: These are available on most systems. Using TCP/IP you might distribute clients over the Web, where DCOM cannot be taken for granted. To use sockets, the server must run the ScktSrvr.exe application provided by Borland, a single program that can run either as an application or as a service. This program receives the client requests and forwards them to the remote data module (executing on the same server) using COM. Sockets provide no protection against failure on the client side, as the server is not informed and might not release resources when a client unexpectedly shuts down[441].

HTTP: The use of HTTP as a transport protocol over the Internet simplifies connections through firewalls or proxy servers (which generally don't like custom TCP/IP sockets). You need a specific Web server application, httpsrvr.dll, which accepts client requests and creates the proper remote data modules using COM. These Web connections can use SSL security but must register themselves by adding a call to `EnableWebTransport` in the `UpdateRegistry` method. Finally, Web connections based on HTTP transport can use a new custom object-pooling support.[442]

note The MIDAS HTTP transport can use XML as the data packet format, enabling any platform or tool that can read XML to participate in MIDAS data transport. This is an extension of the native MIDAS data packet format, which is also platform independent.

CORBA: Common Object Request Broker Architecture is an official standard for object management available on most operating systems. Compared to DCOM, the advantage is that your client and server applications can be also written with Java and other products. The Inprise implementation of CORBA, Visigenic's Visibroker ORB, is available with Delphi Enterprise. CORBA provides many benefits, including

440 While technically DCOM still exists, I'd strongly recommend not using it. Same for MTS, which might not even be available any more.

441 Socket based connections are still available in DataSnap.

442 The HTTP connection is still available in DataSnap, but there is also an alternative REST layer, which targets the interface directly rather than via a redirection to the socket layer.

location transparency, load balancing, and fail-over from the ORB run-time software[443].

OLEnterprise: It is still possible to use Borland's own OLEnterprise technology. You must install the run-time version of OLEnterprise on both the client and server computers. This connection technology can be used to hook with Inprise AppServer and Entera technologies. This is not a common solution, and the corresponding connection component is no longer available on the Delphi Component Palette[444].

As an extension to this architecture in Delphi 5, you can transform the data packets into XML and deliver them to a Web browser. In this case you basically have one extra tier: the Web server gets the data from the middle tier and delivers it to the client. I'll discuss this new architecture, called Internet Express, at the end of the chapter.

Providing Data Packets

The entire Delphi multitier data-access architecture centers around the idea of *data packets*. In this context, a data packet is a block of data that moves from the application server to the client or from the client back to the server. Technically, a data packet is a sort of subset of a dataset. It describes the data it contains (usually a few records of data), and it lists the names and types of the data fields. Even more important, a data packet includes the constraints—that is, the rules to be applied to the dataset. You'll typically set these constraints in the application server, and the server sends them to the client applications along with the data.

All communication between the client and the server occurs by exchanging data packets. The provider on the server manages the transmission of several data packets within a big dataset, with the goal of responding faster to the user. As the client receives a data packet, the user can edit the records it contains. As mentioned earlier, during this process the client also receives and checks the constraints. When the client has updated the records and sends a data packet back, that packet is known as a *delta*. The delta packet tracks the difference between the original records and the updated ones, recording all the changes the client requested from the server. When the client asks to apply the updates to the server, it sends the delta to the server, and the server tries to apply each of the changes. I say *tries* because if a server is connected to several clients, the data might have changed already.

---

443 CORBA still exists, but not the Delphi support. It's another old technology I'd recommend against using, at least from Delphi. DataSnap doesn't support CORBA.

444 OLEnterprise was also deprecated, and never made it to DataSnap.

Since the delta packet includes the original data, the server can quickly determine if another client has already changed it. If so, the server fires an `OnReconcileError` event, which is one of the vital elements for the thin-client applications. In other words, the three-tier architecture uses an update mechanism similar to the one Delphi uses for cached updates. The `ClientDataSet` manages data in memory, in a sort of data cache, and it typically reads only a subset of the records available on the server side, loading more elements only as they're needed. When the client updates the records or inserts new ones, it stores these pending changes in another local cache on the client, the *delta cache*.

The client can also save the data packets to disk, which means users can work offline. Even error information and other data moves using the data packet protocol, so it is truly one of the foundation elements of this architecture.

note It's important to remember that data packets are protocol-independent. A data packet is merely a sequence of bytes, so anywhere you can move a series of bytes, you can move a data packet. This was done to make the architecture suitable for multiple transport protocols, such as DCOM, CORBA, HTTP, and TCP/IP sockets.

Delphi Support Components (Client-Side)

Now that we've examined the general foundations of the new three-tier architecture, we can focus on the Delphi components that support it. For developing client applications, Delphi provides the ClientDataSet component, which provides all the standard data-set capabilities (it derives from `TDataSet`) but doesn't require the BDE, just like the ADO and InterBase Express components. In this case, the data is delivered through the remote connection.

This connection to the server application is made via another component you'll also need in the client application. You should use one of the four specific connection components (available in the Midas page):

- The DCOMConnection component can be used on the client side to connect to a DCOM and MTS server[445], located either on the current computer or in another one indicated by the ComputerName property. The connection is with a registered object having a given `ServerGUID` or `ServerName`.

445 This component still exists today as part of DataSnap.

- The CorbaConnection component can be used to hook with a CORBA server[446]. You indicate the `HostName` (the name or IP address) to indicate the server computer, the `RepositoryID` to request a specific data module on the server, and optionally the `ObjectName` property if the data module exports multiple objects.
- The SocketConnection component can be used to connect to the server via a TCP/IP socket[447]. You should indicate the IP address or the host name, and the GUID of the server object (in the `InterceptGUID` property). In Delphi 5, this connection component has an extra property, `SupportCallbacks`, which you can disable if you are not using callbacks and want to deploy your program on Windows 95 computers that don't have Winsock 2 installed.
- The WebConnection component is used to handle an HTTP connection that can easily get through a firewall[448]. You should indicate the URL where your copy of httpsrvr.dll is located and the name or GUID of the remote object on the server.

Delphi Support Components (Server-Side)

On the server side (or actually the middle tier), you'll need to use a remote data module, a special version of the `TDataModule` class, or one of the specialized remote data modules for MTS and CORBA. There are specific wizards in Delphi to create data modules of each of these types.

The only specific component you need on the server side is the DataSetProvider[449]. You need one of these components for every table or query of the server you want to make available on the client side. The DataSetProvider supersedes the stand-alone Provider component and the internal Provider object, which was embedded in the `TBdeDataSet` sub-classes in past versions. The client applications will then use a separate `ClientDataSet` component for every exported dataset (or provider) they want to use.

---

446 The CORBA support has been removed, as already mentioned.

447 This component still exists today as part of DataSnap.

448 This component still exists today as part of DataSnap.

449 The DataSetProvider is part of the DataSnap architecture and it can work with various data access components.

Building a Sample Application

Now we're ready to build a sample program. This will allow us to observe some of the components I've just described in action, and it will also allow us to focus on some other problems, shedding some light on other pieces of the Delphi multitier puzzle. I'll build the client and application server portions of a three-tier application in two steps. The first step will simply test the technology using a bare minimum of elements. These programs will be very simple.

From that point, we'll add more power to the client and the application server. In each of the examples, we'll display data from local Paradox tables, and we'll set up everything to allow you to test the programs on a stand-alone computer. I won't cover the steps you have to follow to install the examples on multiple computers, with the various technologies—again, that would be the subject of at least one other book. I'll only show you a short introduction to MTS and CORBA later in the chapter.

The First Application Server

The server side of our basic example is very easy to build. Simply create a new application and add a remote data module to it using the corresponding icon in the Multitier page of the Object Repository. The simple Remote Dataset Wizard[450] will ask you for a class name and the instancing style. As you enter a class name, such as AppServerOne, and click the OK button, Delphi will add a data module to the program. This data module will have the usual properties and events, but its class will have the following Pascal declaration:

```
type
  TAppServerOne = class(TRemoteDataModule, IAppServerOne)
  private
    { Private declarations }
  protected
    class procedure UpdateRegistry(Register: Boolean;
      const ClassID, ProgID: string); override;
  public
    { Public declarations }
```

450 This approach is still available in Delphi today, but it's not the standard way to crate a DataSnap multitier application. The Remove Dataset approach is heavily based on COM and used TCP/IP or HTTP connectivity remapped internally to COM. The "modern" DataSnap by contract offers direct connection to Delphi classes and code, without the COM layer.

```
  end;
```

In addition to inheriting from the `TRemoteDataModule` base class (a change from past versions), this class implements a new interface, `IAppServerOne`, which derives from a default Borland interface (`IAppServer`). The class also overrides the default `UpdateRegistry` method to add the support for enabling the socket and Web transports, as you can see in the code generated by the wizard. At the end of the unit, you'll find the class factory declaration:

```
initialization
  TComponentFactory.Create(ComServer, TAppServerOne,
    Class_AppServerOne, ciMultiInstance, tmApartment);
end.
```

Now you can add a Table component to the data module, connect it to a database and a table, activate it, and finally add a DataSetProvider and hook it. You'll obtain a simple DFM file like this:

```
object AppServerOne: TAppServerOne
  object Table1: TTable
    DatabaseName = 'DBDEMOS'
    TableName = 'employee.db'
  end
  object DataSetProvider1: TDataSetProvider
    DataSet = Table1
    Constraints = True
  end
end
```

What about the main form of this program? Well, it's almost useless, so we can simply add a label to it indicating that it's the form of the server application. When you've built the server, you should compile it and run it once. This operation will automatically register it as an Automation server on your system, making it available to client applications. Of course, you should register the server on the computer where you want it to run, either the client or the middle tier.

The First Thin Client

Now that we have a working server, we can build a client that will connect to it. We'll again start with a standard Delphi application and add a DCOMConnection

component[451] to it (or the proper component for the specific type of connection you want to test). This component defines a `ComputerName` property that you'll use to specify the computer that hosts the application server. If you want to test the client and application server from the same computer, you can leave this blank.

Once you've selected an application server computer, you can simply display the `ServerName` property's combo-box list to view the available servers, the servers' registered names (by default the name of the executable file of the server followed by the name of the remote data module class). Alternatively, you can type the GUID of the server object in the `ServerGUID` property. Delphi will automatically enter this property as you set the `ServerName` property, if it can determine the GUID by looking it up in the Registry.

At this point, if you set the DCOMConnection component's `Connected` property to `True`, the server form will appear, indicating that the client has activated the server. You don't usually need to perform this operation, because the ClientDataSet component typically activates the RemoteServer component for you. I've done this simply to emphasize what's happening behind the scenes.

As you might expect, the next step is to add a ClientDataSet component to the form. You must connect the ClientDataSet to the `DCOMConnection11` component via the `RemoteServer` property, and thereby to one of the providers it exports. You can see the list of available providers in the `ProviderName` property, via the usual combo box. In this example, you'll be able to select only `DataSetProvider1`, as this is the only provider available in the selected server. This operation connects the dataset in the client's memory with the file-based dataset on the server. If you activate the client dataset and add a few data-aware controls (or a DbGrid), you'll immediately see the server data appear in them, as illustrated in Figure 21.1.

451 Again, this works today but it's not the recommended approach. Today's DataSnap offers better options, and RAD Server is significantly better and more REST-oriented, modern, and scalable.

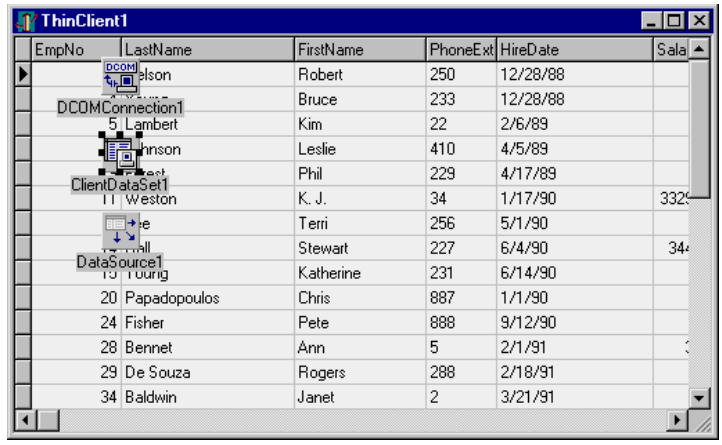


Figure 21.1: When you activate a ClientDataSet component connected to a remote data module at design time, the data from the server becomes visible as usual. Image from the original book.

Here is the DFM file for our minimal client application, ThinCli1:

```
object Form1: TForm1
  Caption = 'ThinClient1'
  object DBGrid1: TDBGrid
    Align = alClient
    DataSource = DataSource1
  end
  object DCOMConnection1: TDCOMConnection
    ServerGUID = '{09E11D63-4A55-11D3-B9F1-00000100A27B}'
    ServerName = 'AppServ1.AppServerOne'
  end
  object ClientDataSet1: TClientDataSet
    Aggregates = <>
    Params = <>
    ProviderName = 'DataSetProvider1'
    RemoteServer = DCOMConnection1
  end
  object DataSource1: TDataSource
    DataSet = ClientDataSet1
  end
end
```

Obviously, our first client and server applications are very simple, but they demonstrate how easy it is to create a data-set viewer that splits the work between two executable files. At this point, our client is only a viewer. If you edit the data on the client, it won't update the server files with those changes. To accomplish this you'll need to add some more code to the client. However, before we do that, let's add some features to the server.

Adding Constraints to the Server

When you write a traditional data module in Delphi, you can easily add some of the application logic, or business rules, by handling the data-set events, and by setting field object properties and handling their events. You should avoid doing this work on the client application; instead, write your business rules on the middle tier.

In the initial release of the MIDAS architecture, you could not use table and field events in the middle tier, because they were not activated. As an alternative, the middle tier could and still can send some constraints to the client and let the client program impose those constraints during the user input. Starting with Delphi 4, the DataSetProvider component allows you to send field properties (such as min and max values) to the client, and it can also process updates through the dataset used to access the data (or a companion UpdateSql object).

Field and Table Constraints

When the provider interface creates data packets to send to the client, it includes the field definitions, the table and field constraints, and one or more records (as requested by the ClientDataSet component). This implies you can customize the middle tier and build distributed application logic by using SQL-based constraints.

The constraints you create using SQL expressions can be assigned to an entire table or to specific fields. The provider sends the constraints to the client along with the data, and the client applies them before sending updates back to the server. This reduces network traffic, compared to having the client send updates back to the application server and eventually up to the SQL server, only to find that the data is invalid. Another advantage of coding the constraints on the server side is that if the business rules change, you need to update only the server application and not the clients. The Constraint Broker provides the basic infrastructure for this logic.

But how do you write constraints[452]? There are several properties you can use:

- The Table component has a `Constraints` property, which has a custom property editor. Similarly, the `Constraints` property of the BDEDataSet components is a collection of `TCheckConstraint` objects. Every object has a few properties, including the expression and the error message.

---

452 This approach is no longer consider the best for customizing the business logic at the UI level, although it was a very interesting model, for sure.

- The Query component defines the same `Constraints` property, plus a `Constrained` Boolean property.
- The Field objects define the `CustomConstraint`, `ImportedConstraint`, and `ConstraintErroMessage` properties, which are functionally equivalent to the `TCheckConstraint` object's properties.

note An important thing to consider is that if you are using a data dictionary, you can extract constraints directly from it[453].

Our next example adds a few constraints to a remote data module connected to the DBDEMOS database's Country.DB table. After connecting the table to the database and creating the field objects for it, you can set the following special properties (a table-wide constraint and a field-specific one):

```
object Table1: TTable
  Constraints = <
    item
      CustomConstraint = 'Name <> '''
      ErrorMessage = 'Must provide a name'
      FromDictionary = False
    end>
  TableName = 'COUNTRY.DB'
  object Table1Population: TFloatField
    CustomConstraint = 'Value > 10000'
    ConstraintErrorMessage = 'Population out of range'
    FieldName = 'Population'
  end
end
```

Including Field Properties

You can control whether the properties of the field objects on the middle tier are sent to the ClientDataSet (and copied into the corresponding field objects of the client side), by using the `poIncFieldProps` value of the `Options` property of the DataSetProvider. This flag controls the download of the field properties `Alignment`, `DisplayLabel`, `DisplayWidth`, `Visible`, `DisplayFormat`, `EditFormat`, `MaxValue`, `MinValue`, `Currency`, `EditMask`, and `DisplayValues`, if they are available in the field.

With this setting you can simply write your middle tier as usual, avoiding the use of constraints. This approach also makes it faster to move existing applications from a

453 As mentioned, the data dictionary (tied to the BDE library) no longer exists in Delphi.

client/server to a multitier architecture. The main drawback of sending fields to the client is that transmitting all the extra information takes time. Turning off `poIncFieldProps` can dramatically improve network performance of datasets with many columns.

Besides using a query, a server can filter the fields returned to the client; it does this by declaring persistent field objects with the Fields editor and omitting some of the fields. Because a field you're filtering out might be required to identify the record for future updates (if the field is part of the primary key), you can also use the field's `ProviderFlags` property on the server to send the field value to the client but make it unavailable to the ClientDataSet component (this provides some extra security, compared to sending the field to the client and hiding it there).

Field and Table Events

You can write middle-tier dataset and field event handlers as usual and let the dataset process the updates received by the client in the traditional way. This means that updates are considered to be operations on the dataset, exactly as when a user is directly editing, inserting, or deleting fields locally.

This is accomplished by setting the `ResolveToDataSet` property of the `TDatasetProvider` component, again connecting either the dataset used for input or a second one used for the updates. Note, by the way, that this provider component is not limited to the BDE but can be used with any dataset. This also allows you to remove the BDE from the middle tier, a particularly useful technique if the middle tier is part of a Web server, where you might not be allowed to install the BDE.

With either technique, the updates are performed by a dataset, which implies a lot of control (this is Pascal code!) but generally slower performance. Flexibility is much greater, as you can use standard coding practices. Also, porting existing local or client/server database applications, which use dataset and field events, is much more straightforward with this model. However, keep in mind that the user of the client program will receive your error messages only when the local cache (the delta) is sent back to the middle tier. Saying to the user that some data prepared half an hour ago is not valid might be a little awkward. If you follow this approach, you'll probably need to apply the updates in the cache at every `AfterPost` event on the client side.

Finally, if you choose this architecture, Delphi helps you a lot in handling possible exceptions. Any exceptions raised by the middle-tier update events (for example, `OnBeforePost`) are automatically transformed by Delphi into update errors, which activate the `OnReconcileError` event on the client side (more on this event later in

this chapter). No exception is shown on the middle tier, but the error travels back to the client.

Adding Features to the Client

After adding constraints to the server, it's now time to return our attention to the client application. The first version was very simple, but now there are a number of features we must add to make it work well.

We'll start by demonstrating how the client works. To do that we'll check the record status and access the delta information (the updates to be sent back to the server). Then we'll add features to the program that handle updates, reconcile errors, and support the briefcase model.

Keep in mind that while you're using this client to edit the data locally, you'll be reminded of any failure to match the business rules of the application, set up on the server side using constraints. The server will also provide us with a default value for the `Continent` field of a new record. In Figure 21.2 you can see one of the error messages this client application can display, which it receives from the server. This message is displayed while editing the data locally, not when you send it back to the server.

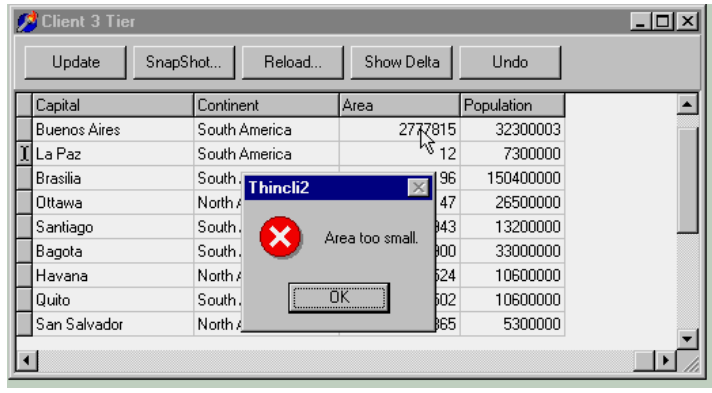


Figure 21.2: The error message displayed by the ThinCli2 example when the value of the Area field is too small. Image from the original book.

The Status of the Records

The ClientDataSet component has a feature that lets us monitor what's going on within the client/server data packets; this is the `UpdateStatus` method[454], which returns one of the following indicators for the current record:

```
type
  TUpdateStatus = (usUnmodified, usModified,
    usInserted, usDeleted);
```

To check the status of every record in the client dataset easily, we can add a string-type calculated field to the table and compute its value with the following method:

```
procedure TForm1.ClientDataSet1CalcFields(
  DataSet: TDataSet);
begin
  ClientDataSet1Status.AsString :=
    GetEnumName (TypeInfo(TUpdateStatus),
      Integer (ClientDataSet1.UpdateStatus));
end;
```

This method converts the current value of the `TUpdateStatus` enumeration to a string.

The records are moved from the server to the client depending on the value of the `PacketRecords` property of the ClientDataSet component, which determines the number of records per packet. The default value of this property is 5, which means that the provider will put five records in each packet transmitted. Alternatively, you can set this value to zero to ask the server for only the field descriptors and no actual data. At the other end of the spectrum, using `-1` transfers all the records at once (which is reasonable only for a small dataset).

Accessing the Delta

Beyond examining the status of each record, the best way to understand which changes have occurred in a given ClientDataSet (but haven't been uploaded to the server) is to look at the delta, the list of changes waiting to be applied to the server. This property is defined as follows:

```
property Delta: OleVariant;
```

---

454 This is still relevant today, also for standalone applications based on an in-memory dataset. FireDAC offers a similar feature.

The format used by the `Delta` property is the same as that used to transmit the data from the client to the server. What we can do, then, is add another ClientDataSet component to the application and connect it to the data in the `Delta` property of the first client dataset:

```
procedure TForm1.ButtonDeltaClick(Sender: TObject);
begin
  if ClientDataSet1.ChangeCount > 0 then
  begin
    ClientDataSet2.Data := ClientDataSet1.Delta;
    ClientDataSet2.Open;
    FormDelta.DataSource1.DataSet := ClientDataSet2;
    FormDelta.Show;
  end
  else
    FormDelta.Hide;
end;
```

FormDelta is a very simple form that contains a DataSource and a DBGrid component, as you can see in Figure 21.3. You'll notice that the delta dataset has two entries for each modified record: the original values and the modified fields.

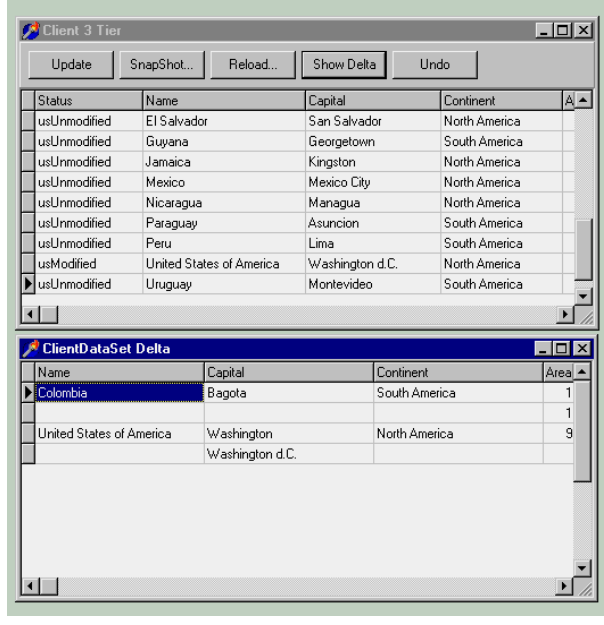


Figure 21.3: The ThinCli2 example allows you to see the temporary update requests stored in the Delta property of the ClientDataSet. Images from the original book.

This code displays the current records in the delta, but it won't automatically refresh them when the user makes other changes unless the program copies the new delta to the second ClientDataSet component every time the data changes.

You can accomplish this in the ClientDataSet component's `AfterPost` event handler, which is executed when the data changes in memory but not when it's sent to the server:

```
procedure TForm1.ClientDataSet1AfterPost(DataSet: TDataSet);
begin
  if FormDelta.Visible and
    (ClientDataSet1.ChangeCount > 0) then
  begin
    ClientDataSet2.Data := ClientDataSet1.Delta;
  end;
end;
```

Updating the Data

Now that we have a better understanding of what goes on during local updates, we can try to make this program work by sending the local update (stored in the delta) back to the application server. To apply all the updates from a dataset at once, pass `-1` to the `ApplyUpdates` method:

```
procedure TForm1.ButtonUpdateClick(Sender: TObject);
begin
  ClientDataSet1.ApplyUpdates (-1);
  FormDelta.Hide;
end;
```

The update operation might trigger the `OnReconcileError` event[455], which allows us to modify the `Action` parameter (passed by reference); this value in turn determines how the server behaves in case of an update collision:

```
procedure TForm1.ClientDataSet1ReconcileError(
  DataSet: TClientDataSet; E: EReconcileError;
  UpdateKind: TUpdateKind; var Action: TReconcileAction);
```

This method has three parameters: the client dataset component (in case more than one client application is interacting with the application server), the exception that caused the error (with the error message), and the kind of operation that failed (`ukModify`, `ukInsert`, or `ukDelete`). The return value, which you'll store in the `Action` parameter, can be any one of the following:

---

455 This event and the concepts behind it are still actual in DataSnap.

```
type
  TReconcileAction = (raSkip, raAbort, raMerge,
    raCorrect, raCancel, raRefresh);
```

- The `raSkip` value specifies that the server should skip the conflicting record, leaving it in the delta (this is the default value).
- The `raAbort` value tells the server to abort the entire update operation and not even try to apply the remaining changes listed in the delta.
- The `raMerge` value tells the server to merge the data of the client with the data on the server, applying only the modified fields of this client (and keeping the other fields modified by other clients).
- The `raCorrect` value tells the server to replace its data with the current client data, overriding all field changes already done by other clients.
- You use the `raCancel` value to cancel the update request, removing the entry from the delta and restoring the values originally fetched from the database (thus ignoring changes done by other clients).
- The `raRefresh` value tells the server to dump the updates in the client delta and to replace them with the values currently on the server (thus keeping the changes done by other clients).

If you want to test collisions on a stand-alone computer, you can simply launch two copies of the client application, change the same record in both clients, and then post the updates from both. We'll do this later to generate an error, but let's first see how to handle the `OnReconcileError` event.

This is actually a simple thing to accomplish, but only because we'll receive a little help. Since building a specific form to handle an `OnReconcileError` event is very common, Delphi already provides such a form in the Object Repository. Simply go to the Dialogs page and select the Reconcile Error Dialog item. As the source code of this unit indicates, it exports a function you can directly use to initialize and display the dialog box:

```
procedure TForm1.ClientDataSet1ReconcileError(DataSet: TClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind;
  var Action: TReconcileAction);
begin
  Action := HandleReconcileError (DataSet, UpdateKind, E);
end;
```

note As the source code of the *Reconcile Error Dialog* unit suggests, you should use the Project Options dialog to remove this form from the list of automatically created forms (if you don't, an error will occur when you compile the project). Of course, you need to do this only if you haven't set up Delphi to skip the automatic form creation.

The `HandleReconcileError` function simply creates the form of the dialog box and shows it:

```
function HandleReconcileError(DataSet: TDataSet;
  UpdateKind: TUpdateKind; ReconcileError: EReconcileError):
  TReconcileAction;
var
  UpdateForm: TReconcileErrorForm;
begin
  UpdateForm := TReconcileErrorForm.CreateForm(DataSet,
    UpdateKind, ReconcileError);
  with UpdateForm do
  try
    if ShowModal = mrOK then
    begin
      Result := TReconcileAction(ActionGroup.Items.Objects[
        ActionGroup.ItemIndex]);
      if Result = raCorrect then
        SetFieldValues(DataSet);
    end
    else
      Result := raAbort;
  finally
    Free;
  end;
end;
```

The Reconc unit, which hosts the Reconcile Error dialog, contains over 350 lines of code, so we can't describe it in detail. However, you should be able to understand the source code by studying it carefully. Alternatively, you can simply use it without caring about how everything works.

The dialog box will appear in case of an error, reporting the requested change that caused the conflict and allowing the user to choose one of the possible `TReconcileAction` values. You can see an example in Figure 21.4.

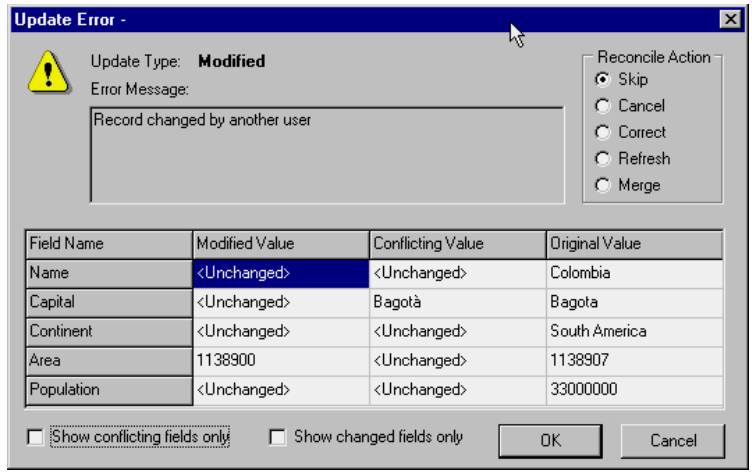


Figure 21.4: The Reconcile Error dialog provided by Delphi in the Object Repository and used by the ThinCli2 example. Image from the original book.

The Update Sequence

To summarize, here is the sequence of operations related to an update and the possible error events:

1. The client program calls the `ApplyUpdates` method of a ClientDataSet.
2. The delta is sent to the provider on the middle tier. The provider fires the `OnUpdateData` event, where you have a chance to look at the requested changes before they reach the database server. At this point you can modify the delta, which is passed in a format compatible with the data of a ClientDataSet.
3. The provider (technically, a part of the provider called the "resolver") applies each row of the delta to the database server. Before applying each update, the provider receives a `BeforeUpdateRecord` event. If you've set the `ResolveToDataSet` flag, this update will eventually fire local events of the dataset in the middle tier.
4. In case of a server error, the provider fires the `OnUpdateError` event (on the middle tier) and the program has a chance of fixing the error at that level.
5. If the middle-tier program doesn't fix the error, the corresponding update request remains in the delta. The error is returned to the client side at this point or after a given number of errors has been collected, depending on the value of the `MaxErrors` parameter of the `ApplyUpdates` call.
6. Finally, the delta packet with the remaining updates is sent back to the client, firing the `OnReconcileError` event of the ClientDataSet for each of them. In

this event handler, the client program can try to fix the problem (possibly prompting the user for help), modifying the update in the delta, and later reissuing it.

Refreshing Data

You can obtain an updated version of the data, which other users might have modified, by calling the `Refresh` method of the ClientDataSet. However, this operation can be done only if the user has no pending update operations in the cache, as calling `Refresh` clears the `Delta`. If there are pending updates, you can call `RefreshRecords` instead, which tries to reapply the changes you have on log in the delta to the new data available on the middle tier, raising an `OnReconcileError` in case of a conflict (that is, if one of the records modified on the client has already been modified on the database server).

Adding an Undo Feature

Since the update data is stored in the local memory (in the delta), besides applying the updates and sending them to the application server, we can reject them, removing entries from the delta. The ClientDataSet component has a specific `UndoLastChange` method to accomplish this. The parameter of this method allows you to *follow* the undo operation (the name of this parameter is `FollowChange`). This means the client dataset will move to the record that has been restored by the undo operation.

Here is the code connected to the Undo button of the example:

```
procedure TForm1.ButtonUndoClick(Sender: TObject);
begin
  ClientDataSet1.UndoLastChange (True);
end;
```

I suggest you try using this feature yourself, to fully understand how it works[456].

456 Again, this is still relevant and supported by in-memory datasets in general.

Supporting the Briefcase Model

The last capability of the ThinCli2 example is support for the "briefcase" model[457]. The idea here is that you might need to use the client program even when you're not physically connected to the application server. In this case, you can save all the data you expect to need in a local file for travel with a laptop (perhaps visiting client sites). You'll use the client program to access the local version of the data, edit the data normally, and when you reconnect, apply all the updates you've performed while disconnected.

The ThinCli2 example's main form has two buttons: one to save a snapshot of the data to a local file and one to restore it. The `OnClick` event handlers for these buttons use the standard OpenDialog and SaveDialog components to connect to the database file:

```
procedure TForm1.ButtonSnapClick(Sender: TObject);
begin
  if SaveDialog1.Execute then
    ClientDataSet1.SaveToFile (SaveDialog1.FileName);
end;

procedure TForm1.ButtonReloadClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    ClientDataSet1.LoadFromFile (OpenDialog1.FileName);
end;
```

After working offline and modifying the local file, you can reload it and apply the updates to the server.

When you use the briefcase model, it's best to download the whole dataset before saving it locally; as you'll recall, you can do that by setting the `PacketRecords` property of the ClientDataSet to `-1`. If you don't do this, you'll simply save the records that happened to be in the memory table, and the client application won't know about the other records still on the server.

---

457 Both ClientDataSet and FireDAC's FDMemTable offer the ability to store a snap shot of data originating from a server to a local, temporary file. This can be used for caching information or to allow editing while disconnected. It's important to save bandwidth for mobile apps, for example, and let them work smoothly even under limited connectivity. This concept overall remains fundamental and it common today, it was quite unique when Borland introduced it.

Advanced MIDAS Features

There are many more features in MIDAS than I've covered up to now. Here is a quick tour of some of the more advanced features of the architecture, partially demonstrated by the AppSPlus and ThinPlus examples. Unfortunately, demonstrating every single idea would turn this chapter into an entire book (and not every Delphi programmer can afford MIDAS), so I'll limit myself to an overview.

Besides the features discussed in the following sections, the AppSPlus and ThinPlus examples demonstrate the use of a socket connection, limited logging of events and updates on the server side, and direct fetching of a record on the client side. The last feature is accomplished with this call:

```
procedure TClientForm.ButtonFetchClick(Sender: TObject);
begin
  ButtonFetch.Caption := IntToStr (cds.GetNextPacket);
end;
```

This allows you to get more records than are actually required by the client user interface (the DbGrid)[458]. In other words, you can fetch methods directly, without waiting for the user to scroll down in the grid. I suggest you study the details of these complex examples after reading the rest of this section.

Parametric Queries

If you want to use parameters in a query or stored procedure, instead of building a custom solution (with a custom method call to the server), you can let Delphi help you. First define the query on the middle tier with a parameter, such as:

```
select * from customer
  where Country = :Country
```

Use the `Params` property to set the type and default value of the parameter. On the client side, you can use the Fetch Params command of the ClientDataSet's shortcut menu, after connecting it to the proper provider. At run time you can call the equivalent `FetchParams` method of the ClientDataSet component.

Now you can provide a local default value to the parameter by acting on the `Params` property. This will be sent to the middle tier when you fetch the data. The ThinPlus example refreshes the parameter with the following code:

---

458 This is still an option today, in the various multitier technologies Delphi support.

```
procedure TFormQuery.btnParamClick(Sender: TObject);
begin
  cdsQuery.Close;
  cdsQuery.Params[0].AsString := EditParam.Text;
  cdsQuery.Open;
end;
```

You can see the secondary form of this example, which shows the result of the parametric query in a grid, in Figure 21.5. In the figure you can also see some custom data sent by the server, as explained in the section "Customizing the Data Packets."

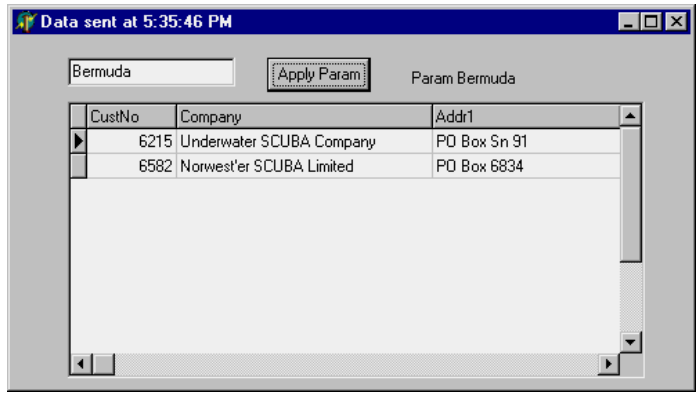


Figure 21.5: The secondary form of the ThinPlus example, showing the data of a parametric query

Custom Method Calls

Since the server has a normal COM interface, we can add more methods or properties to it and call them from the client[459]. Simply open the type library editor of the server and use it as with any other COM server. In the AppSPlus example, I've added a custom `Login` method with the following implementation:

```
procedure TAppServerPlus.Login(
  const Name, Password: WideString);
begin
  // TODO: add actual login code...
  if Password <> Name then
    raise Exception.Create (
```

459 Adding custom method calls is what modern REST servers are all about, although technologies like RAD Server allow you to easily export a database, this is done by automatically offering a number of method calls mapped to HTTP URLs.

```
      'Wrong name/password combination received')
  else
    Query.Active := True;
  ServerForm.Add ('Login:' + Name + '/' + Password);
end;
```

The program makes a simple test, instead of checking the name/password combination against a list of authorizations as a real application should do. Also, disabling the `Query` doesn't really work, as it can be activated by the provider. Disabling the DataSetProvider is actually a more robust approach. The client has a simple way to access the server, the `AppServer` property of the remote connection component. Here is a sample call from the ThinPlus example, which takes place in the `AfterConnect` event of the connection component:

```
procedure TClientForm.ConnectionAfterConnect(
  Sender: TObject);
begin
  Connection.AppServer.Login (Edit2.Text, Edit3.Text);
end;
```

Note that you can call extra methods of the COM interface through DCOM and CORBA, and also over a socket-based or HTTP connection. Because the program uses the `safecall` calling convention, the exception raised on the server is automatically forwarded and displayed on the client side. This way, when a user selects the Connect check box, the event handler used to enabled the client datasets is interrupted, and a user with the wrong password won't be able to see the data.

Besides direct method calls from the client to the server, you can also implement callbacks from the server to the client. This can be used, for example, to notify every client of specific events. Using COM events is a way to do this. As an alternative, you can add a new interface, implemented by the client, which passes the implementation object to the server. This way, the server can call the method on the client computer. Callbacks are not possible with HTTP connections, though. With socket-based connections, callbacks are possible only on computers with WinSock2 installed, which includes Windows 98 computers and Windows 95 machines with a recent version of Internet Explorer.

Master/Detail Relationships

If your middle-tier application exports multiple datasets, you can retrieve them using multiple ClientDataSet components on the client side and connect them locally to form a master/detail structure. This will create quite a few problems for the detail dataset unless you retrieve all of the records locally.

This solution also makes it quite complex to apply the updates; you cannot usually cancel a master record until all related detail records have been removed, and you cannot add detail records until the new master record is properly in place. (Actually, different servers handle this differently, but in most cases where a foreign key is used, this is the standard behavior.) What you can do to solve this problem is to write complex code on the client side to update the records of the two tables according to the specific rules.

A completely different approach is to retrieve a single dataset that already includes the detail as a dataset field, a field of type `TDatasetField`. To accomplish this, you need to set up the master/detail relationship on the server application:

```
object TableCustomer: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'customer.db'
end
object TableOrders: TTable
  DatabaseName = 'DBDEMOS'
  MasterFields = 'CustNo'
  MasterSource = DataSourceCust
  TableName = 'ORDERS.DB'
end
object DataSourceCust: TDataSource
  DataSet = TableCustomer
end
object ProviderCustomer: TDataSetProvider
  DataSet = TableCustomer
end
```

On the client side, the detail table will show up as an extra field of the ClientDataSet, and the DbGrid control will display it as an extra column with an ellipsis button. Clicking the button will display a secondary form with a grid displaying the detail table (see Figure 21.6). If you need to build a flexible user interface on the client, you can then add a secondary ClientDataSet connected to the dataset field of the master dataset, using the `DataSetField` property. Simply create persistent fields for the main ClientDataSet and then hook up the property:

```
object cdsDet: TClientDataSet
  DataSetField = cdsTableOrders
end
```

With this setting you can show the detail dataset in a separate DbGrid placed as usual in the form (the bottom grid of Figure 21.6) or in any other way you like. Note that with this structure, the updates relate only to the master table, and the server should handle the proper update sequence even in complex situations.

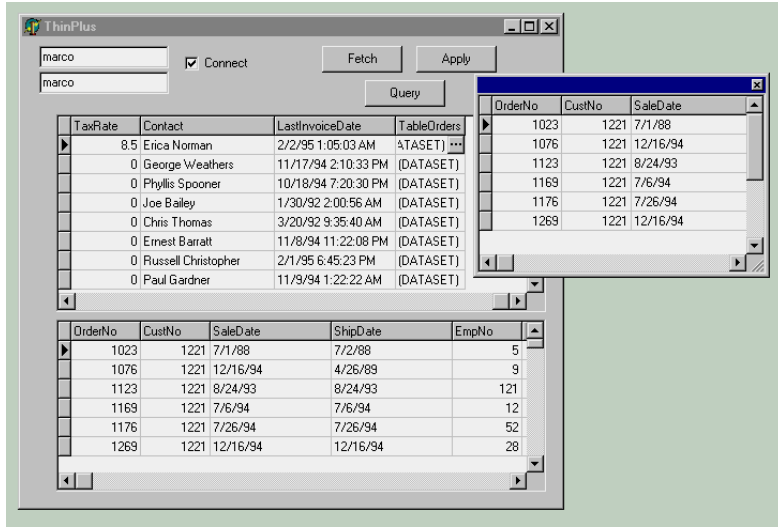


Figure 21.6: The ThinPlus example shows how a dataset field can either be displayed in a grid in a floating window or extracted by a ClientDataSet and displayed in a second form. You'll generally do one of the two things, not both! Image from the original book.

More Provider Options

I've already mentioned the `Options` property of the `DataSetProvider` component, noting that it can be used to add the field properties to the data packet. There are several other options you can use to customize the data packet and the behavior of the client program. Here is a short list:

- You can minimize downloading BLOB data with `poFetchBlobsOnDemand` option. In this case, the client application can download BLOBs by specifying the `FetchOnDemand` property of the ClientDataSet to `True` or by calling the `FetchBlobs` method for specific records. Similarly, you can disable the automatic downloading of detail records by setting the `poFetchDetailsOnDemand` option. Again, the client can use the `FetchOnDemand` property or call the `FetchDetails` method.
- When you are using a master/detail relationship, you can control cascades with either of two options. The `poCascadeDeletes` flag controls whether the provider should delete detail records before deleting a master record. You can set this option if the database server performs cascaded deletes for you as part of its referential integrity support. Similarly, you can set the `poCascadeUpdates` option

when the update of key values of a master detail relationship can be performed automatically by the server.

- You can limit the operations on the client side. The most restrictive option is `poReadOnly`, which disables any update. If you want to give the user a limited editing capability, you can use `poDisableInserts`, `poDisableEdits`, or `poDisableDeletes`.
- You can resend to the client a copy of the records the client has modified with `poAutoRefresh`, which is useful in case other users have simultaneously made other, nonconflicting changes. You can also send back to the client changes done in the `BeforeUpdateRecord` or `AfterUpdateRecord` event handlers by specifying the `poPropogateChanges` option. This option is also handy when you are using autoincrement fields, triggers, and other techniques that modify data on the server or middle tier beyond the changes requested from the client tier.

Finally, if you want the client to drive the operations, you can enable the `poAllowCommandText` option. This lets you set the SQL query or table name of the middle tier from the client, using the `GetRecords` or `Execute` methods.

The Simple Object Broker

The SimpleObjectBroker component[460] provides an easy way to locate a server application among several server computers. You simply provide a list of available computers, and the client will try each of them in order until it finds one that is available.

Moreover, if you enable the `LoadBalanced` property, the component will randomly choose one of the servers; when many clients use the same configuration, the connections will be automatically distributed among the multiple servers. If this seems like a "poor man's" object broker, consider that some highly expensive load-balancing systems don't actually offer much more than this.

Object Pooling

When multiple clients connect to your server at the same time, you have two options. The first is to create a remote data module object for each of them and let

460 It might sound surprising, but the component is still there in the current version of Delphi.

each request be processed in sequence (the default behavior for a COM server with the `ciMultiInstance` style). Alternatively, you can let the system create a different instance of the application for every client (`ciSingleInstance`). This requires more resources and more SQL server connections (and licenses), potentially overloading the BDE (we saw in Chapter 17 that the BDE cannot handle more than a set number of threads or processes).

An alternative approach is offered by the support in MIDAS 3 for object pooling. All you need to do to request this feature is add a call to `RegisterPooled` in the overridden `UpdateRegistry` method. Combined with the stateless support now built into MIDAS, the pooling capability allows you to share some middle-tier objects among a much larger number of clients.

The users on the client computers will spend most of their time reading data and typing in updates, and they generally don't continue to keep asking for data and sending updates. When the client is not calling a method of the middle-tier object, this can be used for another client. Being stateless, in fact, every request reaches the middle tier as a brand-new operation, even when a server is dedicated to a specific client.

This pooling mechanism is built into MTS and CORBA, but MIDAS 3 makes it available also for HTTP and socket-based connections, and for the Internet Express Web client.

Customizing the Data Packets

There are many ways to include custom information within the data packet handled by the `IAppServer` interface. The simplest is probably to handle the `OnGetDataSetProperties` event of the provider itself. This event has a `Sender` parameter, a dataset parameter indicating where the data is coming from, and an `OleVariant` array `Properties` parameter, in which you can place the extra information. You need to define one variant array for each extra property and include the name of the extra property, its value, and whether you want the data to return to the server along with the update delta (the `IncludeInDelta` parameter).

Of course, you can pass properties of the related dataset component, but you can also pass any other value (extra fake properties)[461]. In the AppSPlus example I pass to the client the time the query was executed and its parameters:

---

461 I'd recommend you to avoid going down this route and use a modern approach, like adding content to the JSON payload of a REST request, but that's different world. I assume this technically works today.

```
procedure TAppServerPlus.ProviderQueryGetDataSetProperties(
  Sender: TObject; DataSet: TDataSet; out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['Time', Now, True]);
  Properties[1] := VarArrayOf(['Param',
    Query.Params[0].AsString, False]);
end;
```

On the client side, the ClientDataSet component has a `GetOptionalParameter` method to retrieve the value of the extra property with the given name. The ClientDataSet also has the `SetOptionalParameter` method to add more properties to the dataset. These values will be saved to disk (in the briefcase model) and eventually sent back to the middle tier (by setting the `IncludeInDelta` member of the variant array to `True`). Here is a simple example of the retrieval of the dataset in the code above:

```
Caption := 'Data sent at ' + TimeToStr (
  TDateTime (cdsQuery.GetOptionalParam('Time')));
Label1.Caption := 'Param ' +
  cdsQuery.GetOptionalParam('Param');
```

The effect of this code was visible in Figure 21.5. An alternative and more powerful approach for customizing the data packet sent to the client is to handle the `OnGetData` event of the provider, which receives the outgoing data packet in the form of a client dataset. Using the methods of this client dataset, you can edit data before it is sent to the client. For example, you might encode some of the data or filter out sensitive records.

The Hidden Power of the ClientDataSet Component[462]

The ClientDataSet component supports many features, some of which are related to the three-tier architecture, but it can also be used in other circumstances. This component represents a database completely mapped in memory, and this makes it possible to do on-the-fly operations, like creating an index, that other datasets usu-

462 This was a very important point I made in this book. This component is so powerful that Cary Jensen wrote an entire book on it! Today, the FDMemTable component offers even more featrures, with the advantage of being 100% implemented in Delphi code, while ClientDataSet requires the distribution of the midas library.

ally don't support. To sort a query, for example, you basically reexecute it. To index a local table, you need the index to be defined. Only ADO datasets have some dynamic indexing capability like that of ClientDataSet.

Indexing is not all the ClientDataSet has to offer. When you have an index, you can define groups based on it, possibly with multiple levels of grouping. There is even specific support for determining the position of a record within a group (first, last, or middle position). Over groups or entire tables you can define aggregates; that is, you can compute the sum or average value of a column for the entire table or the current group on the fly. The data doesn't need to be posted to a physical server, because these aggregate operations take place in memory. You can even define new aggregate fields, to which you can directly connect data-aware controls.

The important thing to keep in mind is that all of these features are available not only to MIDAS applications, but also to client/server and even local thin applications. The ClientDataSet component, in fact, can get its data from a remote MIDAS connection, from a local dataset (creating a snapshot of its data), or from a local file (as in the briefcase model, but with the entire table defined only within the client dataset).

This is another huge area to explore, so I'll simply show you a couple of examples highlighting key features. These examples won't be based on MIDAS but on local tables.

Defining Abstract Data Types

An interesting feature of the VCL database support you can activate when using a ClientDataSet based on a local file is the definition of abstract data types. Simply place a ClientDataSet component onto a form, activate the editor for the `FieldDefs` property, add a couple of fields, and select the `ftADT` value for the `DataType` property of one of them. Now move to the `ChildDefs` property and define the child fields. This is the field definition of the AdtDemo example:

```
FieldDefs = <
  item
    Name = 'ID'
    DataType = ftInteger
  end
  item
    Name = 'Name'
    ChildDefs = <
      item
        Name = 'LastName'
        DataType = ftString
```

```
        Size = 20
      end
      item
        Name = 'FirstName'
        DataType = ftString
        Size = 20
      end>
    DataType = ftADT
    Size = 2
  end>
```

At this point, simply type in a name for the `FileName` property of the ClientDataSet, right-click the component, and select the Create Table command; you are ready to compile and run the application (after connecting data-aware components to it). The data will be automatically extracted from the file you've supplied and the changes will be saved onto it when you close the program.

If you use a DBGrid to view the resulting dataset, it will allow you to expand or collapse the sub-fields of the ADT field, as you can see in Figure 21.7. You can provide the condensed value of the field by defining its `OnGetText` event (there was a default value in Delphi 4, but it's not available in Delphi 5):

```
procedure TForm1.ClientDataSet1NameGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  Text := ClientDataSet1NameFirstName.AsString + ' ' +
    ClientDataSet1NameLastName.AsString;
end;
```

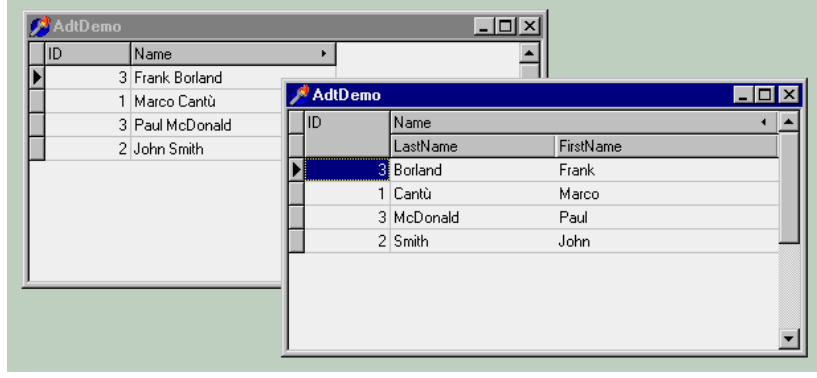


Figure 21.7: The AdtDemo example shows the support for expanding or collapsing the definition of an ADT field. Image from the original book.

On-the-Fly Indexing

Once you have data on a ClientDataSet, the data is entirely in memory. When we base the component on a local file, as in the AdtDemo example, the entire file is loaded in memory when the program starts. This is different from, say, a Paradox table, for which the BDE loads only the fields you are accessing.

An advantage of having the entire table in memory is that you can easily sort it quite fast. With the ClientDataSet component, you can do this simply by assigning a proper field name to the `IndexFieldNames` property. In the AdtDemo (as in many programs), this index change is performed as you click on a title in the DBGrid control (firing the `OnTitleClick` event):

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
  if Column.Field.FullName = 'Name' then
    ClientDataSet1.IndexFieldNames := 'Name.LastName'
  else
    ClientDataSet1.IndexFieldNames := Column.Field.FullName;
end;
```

The program uses the `FullName` property of the field (not the `FieldName` property) because of the ADT definition. For the child fields, in fact, the index should be based on `Name.LastName`, not simply on `LastName`. Also, the ADT field cannot itself be indexed, so if it is selected, the program uses as index the `LastName` subfield. These indexes are not persistent; they are not saved in a file but are simply applied to the data in memory.

---

note A ClientDataSet can have an index based on a calculated field, specifically an internally calculated field, a type of field available only for this dataset.

Grouping

Once you've defined an index for a ClientDataSet, you can group the data by that index. In practice, a group is defined as a list of consecutive records (according to the index) for which the value of the indexed field doesn't change. For example, if you have an index by state, all the addresses within that state will fall in the group.

The CdsCalcs example has a ClientDataSet component that extracts its data from the Country table of the familiar DBDEMOS database. This operation can be performed at design time, using the Assign Local Data command of the ClientDataSet component's shortcut menu. To extract the data at run time, getting an updated

snapshot, you can add a DataSetProvider component to the form, connecting the three components as follows:

```
object Table1: TTable
  Active = True
  DatabaseName = 'DBDEMOS'
  TableName = 'COUNTRY.DB'
end
object DataSetProvider1: TDataSetProvider
  DataSet = Table1
end
object ClientDataSet1: TClientDataSet
  ProviderName = 'DataSetProvider1'
end
```

Now we can focus on the definition of the group. This is obtained, along with the definition of an index, by specifying a grouping level for the index itself:

```
object ClientDataSet1: TClientDataSet
  IndexDefs = <
    item
      Name = 'ClientDataSet1Index1'
      Fields = 'Continent'
      GroupingLevel = 1
    end>
  IndexName = 'ClientDataSet1Index1'
```

When you have a group active, you can make this obvious to the user by displaying the grouping structure in the DBGrid, as shown in Figure 21.8. Simply handle the `OnGetText` event for the grouped field (the `Continent` field in the example), and show the text only if the record is the first of the group:

```
procedure TForm1.ClientDataSet1ContinentGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  if gbFirst in ClientDataSet1.GetGroupState (1) then
    Text := Sender.AsString
  else
    Text := '';
end;
```

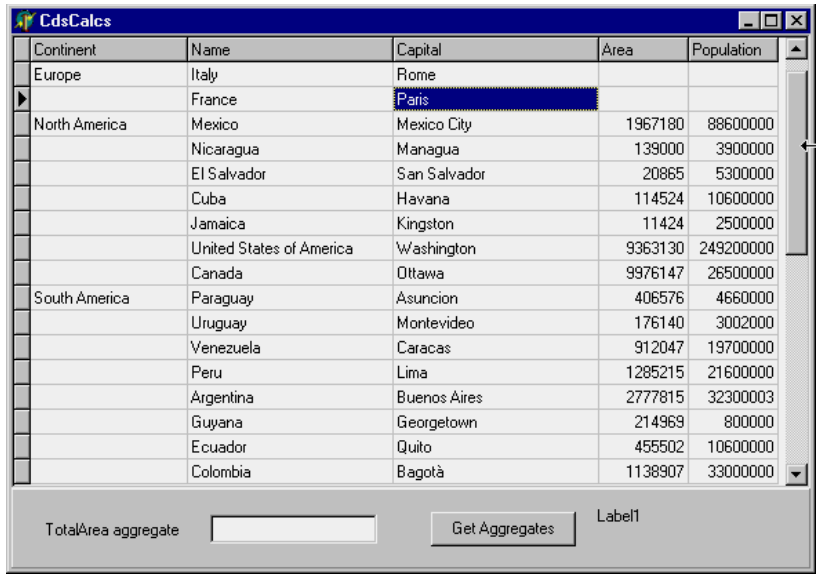


Figure 21.8: The CdsCalcs example demonstrates that by writing a little code, you can have the DBGrid control visually show the grouping defined in the ClientDataSet. Image from the original book.

Defining Aggregates

Another extremely powerful feature of the ClientDataSet component is the support for *aggregates*. An aggregate is a calculated value based on multiple records, such as the sum or the average value of a field for the entire table or a group of records (defined with the grouping logic I've just discussed). Aggregates are maintained; that is, they are recalculated immediately if one of the records changes. For example, the total of an invoice can be maintained automatically while the user types in the invoice items.

note Aggregates are maintained incrementally, not by recalculating all the values every time one value changes. Aggregate updates take advantage of the deltas tracked by the ClientDataSet. For example, to update a Sum when a field is changed, the ClientDataSet subtracts from the aggregate the old value and adds the new value. Only two calculations are needed, even if there are thousands of rows in that aggregate group. For this reason, aggregate updates are instantaneous.

There are two ways to define aggregates. You can use the `Aggregates` property of the ClientDataSet, which is a collection, or you can define aggregate fields using the Fields editor. In both cases you define the aggregate expression, give it a name, and

connect it to an index and a grouping level (unless you want to apply it to the entire table). Here is the `Aggregates` collection of the CdsCalcs example:

```
object ClientDataSet1: TClientDataSet
  Aggregates = <
    item
      Active = True
      AggregateName = 'Count'
      Expression = 'COUNT (NAME)'
      GroupingLevel = 1
      IndexName = 'ClientDataSet1Index1'
      Visible = False
    end
    item
      Active = True
      AggregateName = 'TotalPopulation'
      Expression = 'SUM (POPULATION)'
      Visible = False
    end>
  AggregatesActive = True
```

Notice in the last line above that you have to activate the support for aggregates, in addition to activating each specific aggregate you want to use. Disabling aggregates is important, because having too many of them can slow down a program. The alternative approach, as I mentioned, is to use the Fields editor, select the New Field command of its shortcut menu, and choose the Aggregate option (available, along with the InternalCalc option, only in a ClientDataSet). This is the definition of an aggregate field:

```
object ClientDataSet1: TClientDataSet
  object ClientDataSet1TotalArea: TAggregateField
    FieldName = 'TotalArea'
    ReadOnly = True
    Visible = True
    Active = True
    DisplayFormat = '###,###,###'
    Expression = 'SUM(AREA)'
    GroupingLevel = 1
    IndexName = 'ClientDataSet1Index1'
  end
```

The aggregate fields are displayed in the Fields editor in a separate group, as you can see in Figure 21.9. The advantage of using an aggregate field, compared to a plain aggregate, is that you can define the display format and hook the field directly to a data-aware control, such as a DBEdit in the CdsCalcs example. Because the aggregate is connected to a group, as soon as you select a record of a different group, the output will be automatically updated. Also, if you change the data, the total will immediately show the new value.

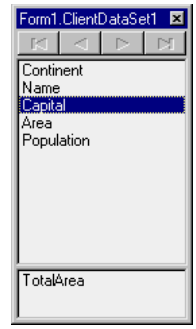


Figure 21.9: The bottom portion of the Fields editor of a ClientDataSet displays aggregate fields. Image from the original book.

To use plain aggregates, instead, you have to write a little code, as in the following example (notice that the `Value` of the aggregate is a variant):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption :=
     'Area: ' + ClientDataSet1TotalArea.DisplayText +
    #13'Population : ' + FormatFloat ('###,###,###',
    ClientDataSet1.Aggregates [1].Value) + #13'Number : ' +
    IntToStr (ClientDataSet1.Aggregates [0].Value);
end;
```

High-End Distributed Services (MTS and CORBA)[463]

Besides using plain socket, DCOM, or HTTP-based connections, Delphi and MIDAS support two higher-level and more powerful transport and object-broker architectures. MTS is the Microsoft solution to distributed computing, while CORBA is a more open standard embraced by most Unix vendors and often used in Java environments. Both topics would require separate books for complete coverage. My aim here is only to highlight a few of their features and discuss Delphi's support for them.

463 Considering these features are no longer actual, this section is more or less useless today. I won't add a footnte fo reach obsolete feature, because everything is obsolete!

Microsoft Transaction Server

In addition to plain DCOM servers, Delphi also allows you to create Microsoft Transaction Server components. You can actually build plain MTS components or an MTS remote data module. In both cases you'll start the development by using one of the available Delphi wizards. MTS is an operating-system service you can install on Windows NT and 98. MTS is also one of the cornerstones of the Windows 2000 COM+ technology; MTS is actually going to be folded into COM+, and the MTS name will eventually disappear.

MTS is a run-time environment that provides database transaction services, security, resource pooling, and an overall improvement in robustness for DCOM applications. The MTS run-time environment manages objects called *MTS components*. These are COM objects stored in an in-process server (that is, a DLL). While other COM objects run directly in the client application, MTS objects are handled by the MTS run-time environment. The MTS libraries, in fact, are installed into the MTS environment. MTS objects must support specific COM interfaces, starting with `IObjectControl`, which is the base interface (like `IUnknown` for a COM object).

Before getting into too many technical and low-level details, let's consider MTS from a different perspective. What are the benefits of this approach? MTS provides a few interesting features, including:

- **Role-based security:** The role assigned to a client determines if it has the right to access the interface of a data module.
- **Reduced database resources:** You can reduce the number of database connections, as the middle tier logs on to the server and uses the same connections for multiple clients (although you cannot have more clients connected at once than you have licenses for the server). Moreover, you can set up the component so that the MTS server will instantiate the data modules only for the minimum necessary time.
- **Database transactions:** MTS transaction support includes multiple database support, although few SQL server databases support MTS.

Creating an MTS Data Module

If you select the MTS Data Module icon in the Multitier page of the File ➢ New dialog box (in Delphi Enterprise), you can easily set up an MTS-compliant remote server. The MTS Data Module Wizard (see Figure 21.10) allows you to enter a name

for the class of the MTS component, the threading model (because MTS serializes all the requests, Single or Apartment will generally do), and a transaction model:

- *Requires a transaction* indicates that each call from the client to the server is considered to be an MTS transaction (unless the caller supplies an existing transaction context).
- *Requires a new transaction* indicates that each call is considered a new MTS transaction.
- *Supports transactions* indicates that the client must explicitly provide a transaction context.
- *Does not support transaction* (the default choice) indicates that the remote data module won't be involved in any MTS transaction.

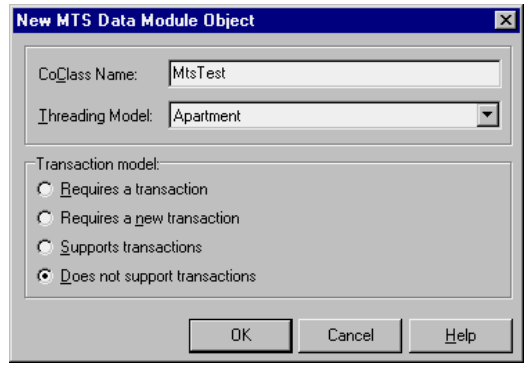


Figure 21.10: Delphi's MTS Data Module Wizard. Image from the original book. This tool no longer exists.

Once you've created an MTS data module, you can easily build it as we've done in earlier examples for remote data modules, adding a DataSet component and exporting its provider property. You can also add custom methods to the data module type library. Within the MTS data module, you can use the `GetObjectContext` method, which returns the `IObjectContext` interface of the MTS object.

The `IObjectContext` interface provides support for transactions. You can use `SetComplete` to tell the MTS environment that the object has finished working and can be deactivated, so that the transaction can be committed. Call `EnableCommit` to indicate that the object hasn't finished but the transaction should be committed; `DisableCommit` to stop the commit operation even if the method is done, disabling the object deactivation between method calls; `SetAbort` to say that the object has finished and can be activated but the transaction cannot be committed; or `IsInTransaction` to check whether the object is part of a transaction. Other meth-

ods of the `IContextObject` interface include `CreateInstance`, which creates another MTS object in the same context and within the current transaction, `IsCallerInRole`, which checks if the object's caller is in a particular "security" role, and `IsSecurityEnabled` (whose name is self-explanatory).

Once you've built an MTS server library, you can easily install it by using the Run ➢ Install MTS Object option. You can add the new library to an existing MTS package (not to be confused with a Delphi component package) or create a new one right from the Delphi environment. After the MTS object has been installed, it will be directly available to other applications and visible in the Transaction Server Explorer application. This configuration program by Microsoft should be installed on your computer along with the MTS support.

After you've built this server, you can connect to it in a client application using the DCOMConnection component, as we've done in earlier examples. This brief introduction should have given you an idea of how to use MTS for the development of multitier applications. The advantages in terms of installation, if compared with the direct use of DCOM, are really worth the extra effort of using MTS.

CORBA

After a fast overview of MTS, we are ready for a similar excursion into the world of the Common Object Request Broker Architecture. The CORBA standard is defined by the Object Management Group (OMG) and addresses the complexity of building and deploying distributed applications based on objects.

One of the key elements of CORBA is that is it completely independent of platforms and operating systems. CORBA opens up the non-Microsoft world to Delphi programmers: Although modules built with Delphi can run only on Windows, they can connect with other CORBA objects running on different operating systems. For example, CORBA provides a good integration with Java and you can leverage the Borland/Inprise multitier architecture from both Delphi and JBuilder, Borland's Java development environment. You can actually use a MIDAS CORBA architecture to build Delphi clients with the Java MIDAS server on Unix (or Linux) boxes or you can build a Java front end for your Windows NT MIDAS servers built with Delphi.

The CORBA specification defines how client-side programs communicate with server-side objects through an Object Request Broker (ORB). Inprise's VisiBroker ORB is the request broker you can find in each copy of Delphi Enterprise. Of course, you'll need a deployment license to ship applications built with this ORB, as with any other.

A Simple CORBA Server

You can build a simple CORBA server, based on a data module. When you select the CORBA Data Module option in the Multitier page of the Object Repository, Delphi displays the simple CORBA Data Module Wizard illustrated in Figure 21.11.

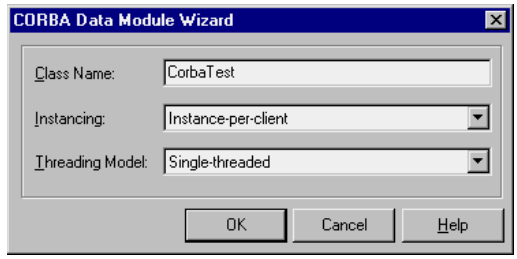


Figure 21.11: The CORBA Data Module Wizard lets you build a remote data module with CORBA support. Image from the original book. This feature no longer exists.

In this wizard you can specify the instancing and threading models. These models are different from the one used by COM:

- *Per-client* instancing indicates that a new instance of the data module is created for each connection.
- *Shared* instancing indicates that a single instance of the data module handles all the requests of multiple clients. This second approach is possible only for a stateless protocol, which is a no-brainer in MIDAS 3.

In a single-threaded server each data module receives only one client request at a time, so that the data of the instance is safe from possible conflicts. In a multi-threaded server, by contrast, the client can send multiple simultaneous requests, and this imposes some extra care on the developer.

Once the data module has been created, you can continue the development as for any other remote data module. The class `TCorbaDataModule` inherits directly from `TRemoteDataModule`. The data module also has a corresponding entry in the type library, as you can see by opening the corresponding editor or looking at the translated Pascal code. In the translated Corba1_TLB file, you can see the `IFirstCorba` interface and the `IFirstCorbaDisp` dispinterface, as for COM applications.

In fact, in Delphi 5, the CORBA support is still limited. The main limiting factor to Delphi's CORBA support is that it allows only method calls through CORBA's late-bound dispatch, called DII. This is analogous to calling COM dispinterface methods via variants (variant method calls). In fact, Delphi's CORBA method calls can be made through variants, but are dispatched through CORBA services, not COM. Like

dispinterface calls in COM, DII in CORBA is not the fastest way to invoke CORBA methods.

note That's why there is interest in an IDL-to-Pascal converter—to generate stubs and skeletons that communicate with the CORBA ORB directly without the overhead of late binding. Borland has recently announced that it's working on native CORBA support, including a CORBA IDL-to-Pascal converter. This support is expected some time after Delphi 5 ships, and it might actually be available by the time you read this.

The role of the CORBA stub is to mimic the remote object in the local address space, by taking the calls, writing the arguments into a buffer, transmitting them to the remote server, and returning the result. This way, the program can work as if the remote object was local. The role of the skeleton is the opposite: it runs in the server address space and receives the requests from the client, unpacking the buffer received from the stub, and making the server believe that the client is local.

The type library editor also allows you to convert the COM-based IDL code into the CORBA version, using the Export button at the far end of the toolbar. The CORBA IDL can be used by other programming languages to generate the proper interfaces or to register the IDL in the Interface Repository Server, managed with the `IREP` and `IDL2IR` command line utilities. These steps are not needed to build and run a simple CORBA example.

Getting back to the example, once you've compiled it, you can run it. Contrary to COM, you don't need to statically register the server, but you have to run it to make it available to the ORB. Because the program tries to register itself as it starts, you need to have the ORB installed on the system and a Smart Agent running on your network before you can run it. The CORBA Smart Agent is a dynamic, distributed directory service that locates an available server, which in turn provides the implementation of the CORBA object.

To test the program on a stand-alone system, simply run the Visibroker Smart Agent from the Visibroker menu (available under the Borland Delphi 5 entry of the Windows Start ➢ Program menu). At this point you can simply run the server (but not from the Delphi debugger, because you'll need to create a separate client application).

A Simple CORBA Client

While the server program is running, a client program can connect to it. If you want to test the client with live data at design time, you need to keep the server running while you are building the client program.

The development of this client program can proceed simply by adding a CorbaConnection component to a form. To indicate the proper server, in this case, there is no combo box to choose from. You should simply enter in the `RepositoryID` property the program name and data module name separated by a slash (not a period, as for a COM server). For example, you can enter the string *'Corba1/FirstCorba'*. To test if this value is correct and everything is working properly, simply toggle the `Connected` property of the component.

In this client program you can add a ClientDataSet component, select the `CorbaConnection1` component for the `RemoteServer` property, and select one of the available provider interfaces in the `ProviderName` property (this time using the combo box). Finally, add a DataSource component and some data-aware controls.

As you compile and run the program, it will simply hook with the CORBA data module to fetch the data, something you can already do at design time. Again, the server must be manually started (which was not the case in the COM-based examples).

ActiveForm Thin Clients[464]

In this chapter we've built thin-client programs, which did not directly access a database but got the data from an application server running on the so-called middle tier. Some networks might want to move the front end of these applications to an intranet or the Internet, deploying them through a browser. That's what ActiveForms are for, and it's what we'll do to demonstrate a client program that requires no installation or configuration whatsoever.

If we build an ActiveForm that connects to a MIDAS application server, the server will deliver live SQL data to us as necessary, we'll be able to send back updates, and we won't need to install the BDE on the client computer.

The ActiveForm of this example, called AfRemote, connects to one the application servers we built earlier in this chapter (specifically, AppServ2). If you haven't already done so, you'll need to build and run this server application to register it and make it available to this client ActiveX control. The registration should take place on the server only.

---

464 Taking in consideration what I wrote about the ActiveForm technology, a terribly insecure web technology working only in Internet Explorer, this is another section with zero value today, even if the support was not removed from Delphi.

The connection, this time, will be based on a SocketConnection component, so that we can use TCP/IP; this means that every Windows 95 computer connected to the Internet will be capable of running it. Here are the key properties of the components of the client program, which is actually an ActiveForm:

```
object ActiveRemote: TActiveRemote
  Caption = 'ActiveRemote'
  object DBGrid1: TDBGrid
    Align = alClient
    DataSource = DataSource1
  end
  object Panel1: TPanel
    Align = alTop
    object CheckActive: TCheckBox
      Caption = 'Active'
      OnClick = CheckActiveClick
    end
    object BtnApply: TButton
      Caption = 'Apply Updates'
      OnClick = BtnApplyClick
    end
  end
  object ClientDataSet1: TClientDataSet
    ProviderName = 'DataSetProvider1'
    RemoteServer = SocketConnection1
    OnReconcileError = ClientDataSet1ReconcileError
  end
  object DataSource1: TDataSource
    DataSet = ClientDataSet1
  end
  object SocketConnection1: TSocketConnection
    ServerGUID = '{C5DDE903-2214-11D1-98D0-444553540000}'
    ServerName = 'AppServTwo.RdmCount'
    Address = '127.0.0.1'
  end
end
```

In this test case, the remote computer is the current one, so I've used the address `127.0.0.1`. You should update it with whatever IP address your server has. To make the example work, the server will need to run the Borland Socket Server. The code of the AfRemote example's three methods is very straightforward:

```
procedure TActiveRemote.CheckActiveClick(Sender: TObject);
begin
  if CheckActive.Checked and not SocketConnection1.Connected then
    SocketConnection1.Connected := True;
  ClientDataSet1.Active := CheckActive.Checked;
end;

procedure TActiveRemote.BtnApplyClick(Sender: TObject);
begin
```

```
  if ClientDataSet1.Active then
    ClientDataSet1.ApplyUpdates (-1);
end;

procedure TActiveRemote.ClientDataSet1ReconcileError(
  DataSet: TClientDataSet; E: EReconcileError;
  UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  Action := HandleReconcileError (DataSet, UpdateKind, E);
end;
```

As we did before, on an update error we display the standard Reconcile dialog box. The output of the AfRemote program in Microsoft's Internet Explorer appears in Figure 21.12.

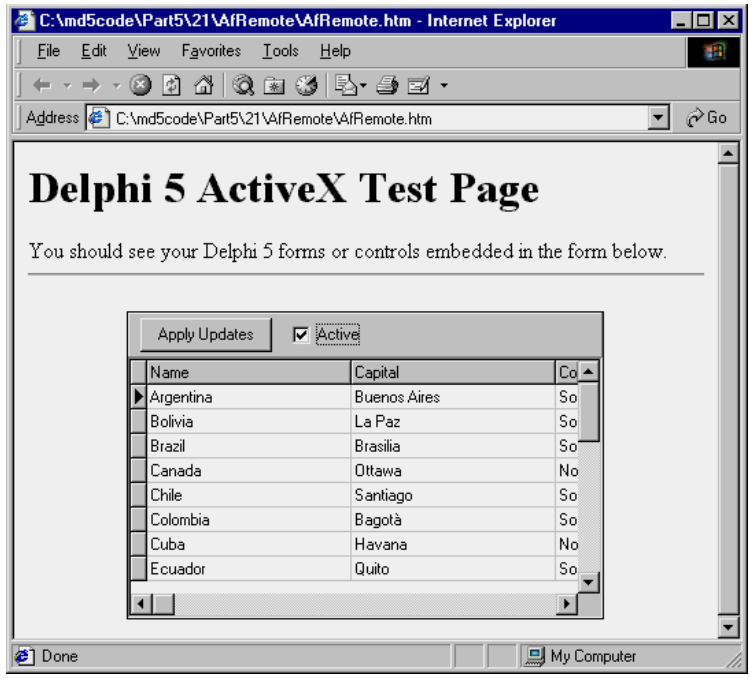


Figure 21.12: The ActiveForm of the AfRemote example uses the live data provided from an application server, following Delphi's three-tier architecture. Image from the original book.

Although you can certainly use an ActiveForm as a MIDAS client, this approach has a few problems. First, users must have a Win32 computer and Internet Explorer (not another browser) and probably a recent version of it. Second, downloading ActiveX components exposes the user to considerable risks, exactly like running a program downloaded from the Web on a computer. I'm not particularly referring to the viruses the ActiveX control could implant on your computer, but to the fact that

the ActiveX control can grab any file from your machine and send it to a server—without your knowledge. In practice, many Web surfers disable the ActiveX control support, even in browsers that allow it. Finally, downloading an ActiveX control can require a lot of time over a slow Internet connection.

As I mentioned earlier, this approach is reasonable on an intranet but not for the open world of the Internet, particularly if you want to get as many visitors as possible on your Web site. The alternative approach Delphi 5 introduces is the Web MIDAS client, made possible by a technology called Internet Express.

Internet Express[465]

Now that we know how to build MIDAS servers and client programs, we might want to open up this architecture and generate HTML pages on the Web to let any user interact with our middle-tier server through a Web server. The idea behind Internet Express is that you write a Web server extension (CGI or ISAPI, as discussed in the previous chapter), which in turn produces Web pages hooked to your MIDAS server. Your custom application acts as a MIDAS client and produces pages for a browser client. Internet Express offers the services required to build this custom application easily.

I know this sounds confusing, but Internet Express is a four-tier architecture: SQL server, application server (the MIDAS server), the Web server with a custom application, and finally the Web browser. Of course, you can place the first three levels on a single computer, but there is still a logical division into four levels. Also, you can shortcut the MIDAS level, hooking the Web server to a local file.

Internet Express uses multiple technologies to accomplish this:

- The MIDAS data packets (based on OleVariants in the Delphi implementation) are converted to XML format, to let the program embed the data in the HTML page. Actually, the Delta data packet is also represented in XML. These operations are performed by the new XMLBroker component, a dataset similar to the ClientDataSet, which can handle XML and provide data to the new JavaScript components.

---

465 Like many other areas of this chapter, this technology is noi longer actual and this section has very little value, outside of the historical perpective.

- There is a new MidasPageProducer component, which allows you to generate HTML forms from datasets, in a visual way similar to the development of a Delphi form. Instead of using VCL components, you use JavaScript ones.
- To make the editing operations on the client side powerful, the MidasPageProducer uses special JavaScript components and code. Delphi 5 embeds a rather large JavaScript library, which the browser will have to download. This might seem a nuisance, but it is the only way the browser interface (which is based on dynamic HTML) can be rich enough to support field constraints and other business rules with the browser. This is really impossible with plain HTML.

Of course, to deploy this architecture you don't need anything special on the client side, as any browser up to the HTML 4 standard (which cuts a lot of them out of the picture) can be used on any operating system! The Web server, instead, must be a Win32 server and you must deploy MIDAS on it (after paying the proper license fee, even if you hook up the Web server application to a local file).

Building a First Example

My first Internet Express example, called IeFirst, is a very simple one, including only the minimal elements required to move a simple MIDAS client (in this case the ThinCli1 example) to a browser-based interface. I've created a new CGI application and added a DCOMConnection component to it, hooked to the AppServ1 server. The next step is to add an XMLBroker component and connect it with the remote server and a provider:

```
object XMLBroker1: TXMLBroker
  ProviderName = 'DataSetProvider1'
  RemoteServer = DCOMConnection1
  WebDispatch.MethodType = mtAny
  WebDispatch.PathInfo = 'XMLBroker1'
  ReconcileProducer = PageProducer1
  OnGetResponse = XMLBroker1GetResponse
end
```

The `ReconcileProducer` is required to show a proper error message in case of an update conflict. As we'll see later, one of the Delphi demos includes some custom code, but in this simple example I've simply connected a traditional PageProducer with a generic HTML error message.

After setting up the XML broker, you can add a MidasPageProducer[466] to the Web data module. This component has a standard HTML skeleton you can customize without touching the special entries:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<#INCLUDES><#STYLES><#WARNINGS><#FORMS><#SCRIPT>
</BODY>
</HTML>
```

The special tags are automatically expanded using the JavaScript files of the directory specified by the `IncludePathURL` property. You must set this property to refer to the Web server directory where these files reside. You can find them in the Source/Webmidas sub-directory of the Delphi5 folder.

note In the MidasPageProducer component you can refer to an external style sheet file or an embedded set of styles. The support for style sheets in this component and in the entire Internet Express architecture is quite complete.

To customize the resulting HTML of the MidasPageProducer you can use its editor, which is a rather complex tool. Just double-click on the component, and Delphi opens up a window like the one you can see in Figure 21.13. In this editor you can create complex structures, starting with a query form, a data form, or a generic layout group. In the data form of my simple example, I've added a DataGrid and a DataNavigator component, without customizing them any further (an operation you do by adding child buttons, columns, and other objects, which fully replace the default ones).

466 A DataSetPageProducer component and a DataSnapTableProducer component were later added to DataSnap, with a similar role. So it's still possible to use a similar feature today, but not recommended at all.

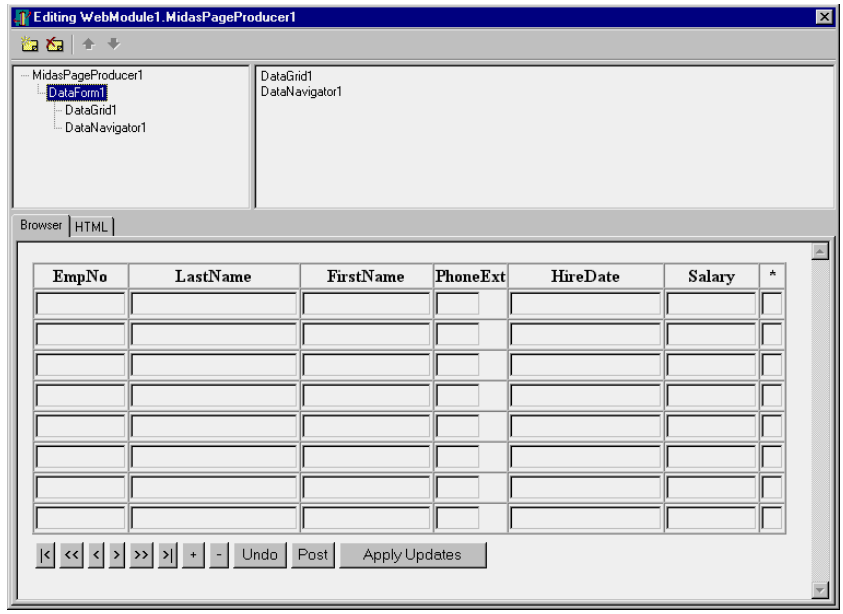


Figure 21.13: The MidasPageProducer editor allows you to build complex HTML forms visually. Image from the original book.

The DFM code for these components in my example is the following:

```
object DataForm1: TDataForm
  object DataGrid1: TDataGrid
    XMLBroker = XMLBroker1
    DisplayRows = 5
    TableAttributes.CellSpacing = 0
  end
  object DataNavigator1: TDataNavigator
    XMLComponent = DataGrid1
  end
end
```

But the value of these components is in the HTML they generate, which you can preview by selecting the HTML tab of the MidasPageProducer editor. The initial part of the table definition in this HTML script looks like this (only one data cell is visible here):

```
<FORM NAME=DataForm1>
<TABLE><TR><TD COLSPAN=2>
<TABLE CELLSPACING=0 BORDER=1><TR>
<TH>EmpNo</TH>
<TH>LastName</TH>
...
</TR><TR><TD><DIV><INPUT TYPE=TEXT
  NAME="DataGrid1_EmpNo"
```

```
  SIZE=10
  onFocus='if(xml_ready)DataGrid1_Disp.xfocus(this);'
  onkeydown='if(xml_ready)DataGrid1_Disp.keys(this);'>
</DIV></TD>
...
```

When the HTML generator is set up, you can go back to the Web data module, add an action to it, and connect the actions with the MidasPageProducer via the `Producer` property. This should be enough to make the program work through a browser, as you can see in Figure 21.14.

If you look at the HTML file received by the browser, you'll find the table mentioned definition above, some JavaScript code here and there, and the database data in XML format (preceded by the metadata):

```
<XML ID=XMLBroker1_Doc>
  <DATAPACKET Version="2.0">
    <METADATA>
      <FIELDS>
        <FIELD attrname="EmpNo" fieldtype="i4"/>
        <FIELD attrname="LastName" fieldtype="string"
          WIDTH="20"/>
        ...
      </FIELDS>
      <PARAMS DEFAULT_ORDER="1"
        PRIMARY_KEY="1" LCID="1033"/>
    </METADATA>
    <ROWDATA>
      <ROW EmpNo="2" LastName="Nelson" FirstName="Robert"
        PhoneExt="250" HireDate="19881228" Salary="40000"/>
      <ROW EmpNo="4" LastName="Young" FirstName="Bruce"
        PhoneExt="233" HireDate="19881228" Salary="55500"/>
```

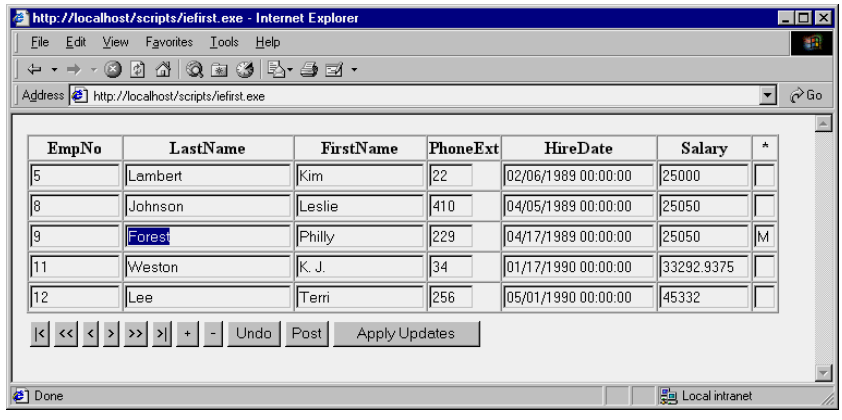


Figure 21.14: The IeFirst example in a Web browser. Notice the M code in the last column, indicating that the record has been modified. Image from the original book.

This data is assembled by the XML broker and passed to the producer component to be embedded in the HTML file. Notice that the number of records sent to the client depends on the XMLBroker, not on the number of lines in the grid. Once the XML data is sent to the browser, in fact, you can use the buttons of the navigator component to move around in the data without requiring further access to the server to fetch more.

At the same time, the JavaScript classes in the system allow the user to type in new data, following the rules imposed by the JavaScript code hooked to dynamic HTML events. Notice that the grid, by default, has an extra asterisk column, indicating which records have been modified. The update data is collected in an XML data packet in the browser, and sent back to the server when the user clicks the Apply Updates button. At this point the browser activates the action specified by the `WebDispath.PathInfo` property of the XMLBroker. There is no need to export this action from the Web data module, as this operation is automatic (although you can disable it by setting `WebDispath.Enable` to `False`).

The XMLBroker applies the changes to the server, returning the content of the provider connected to the `ReconcileProvider` property (or raising an exception if this is not defined). When everything works fine, the XMLBroker redirects the control to the main page that contains the data. However, I've experienced some problems with this technique when using Personal Web Server for Windows 98, and for this reason the IeFirst example handles the `OnGetReponse` events with this code:

```
procedure TWebModule1.XMLBroker1GetResponse(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
begin
  Response.Content := '<h1>Updated</h1><p>' +
    MidasPageProducer1.Content;
  Handled := True;
end;
```

Master/Detail on the Web

My second and last Internet Express example goes a little beyond the basics by providing a master/detail data packet for Web browsing. The program uses the AppPlus server, which defines the master/detail relationship. The dataset field embedded in the table will be transformed into a nested XML structure, delivering the same information.

The program uses a combination of XMLBroker, MidasPageProducer, and DCOMConnection, as in the last example. This time, however, I've customized the Web

components, creating fields and selecting the information to display. You can see part of this structure in the Tree view of the Web data module in Figure 21.15, along with the intricate connections between some of the components in the Data Diagram view. Here is a rather long listing of this structure: I've removed some extra information but think it is worth looking at it:

```
object XMLBroker1: TXMLBroker
  ProviderName = 'ProviderCustomer'
  RemoteServer = DCOMConnection1
  WebDispatch.PathInfo = 'XMLBroker1'
end
object MidasPageProducer1: TMidasPageProducer
  IncludePathURL =
    'C:/Program Files/Borland/Delphi5/Source/Webmidas/'
  object DataForm1: TDataForm
    object DataNavigator1: TDataNavigator
      XMLComponent = FieldGroup1
      object FirstButton1: TFirstButton
        XMLComponent = FieldGroup1
        Caption = '|<'
      end
      object PriorButton1: TPriorButton
        XMLComponent = FieldGroup1
        Caption = '<'
      end
      object NextButton1: TNextButton
        XMLComponent = FieldGroup1
        Caption = '>'
      end
      ...
      object ApplyUpdatesButton1: TApplyUpdatesButton
        Caption = 'Apply Updates'
        XMLBroker = XMLBroker1
        XMLUseParent = True
      end
    end
    object FieldGroup1: TFieldGroup
      XMLBroker = XMLBroker1
      object CustNo: TFieldText
        DisplayWidth = 10
        Caption = 'CustNo'
        FieldName = 'CustNo'
      end
      object Company: TFieldText
        DisplayWidth = 30
        Caption = 'Company'
        FieldName = 'Company'
      end
      ...
    end
    object DataNavigator2: TDataNavigator
      XMLComponent = DataGrid1
```

```
      object FirstButton2: TFirstButton
        XMLComponent = DataGrid1
        Caption = '|<'
      end
      object PriorPageButton1: TPriorPageButton
        XMLComponent = DataGrid1
        Caption = '<<'
      end
      ...
    end
    object DataGrid1: TDataGrid
      XMLBroker = XMLBroker1
      XMLDataSetField = 'TableOrders'
      DisplayRows = 8
      object OrderNo: TTextColumn
        DisplayWidth = 10
        Caption = 'OrderNo'
        FieldName = 'OrderNo'
      end
      object SaleDate: TTextColumn
        DisplayWidth = 18
        Caption = 'SaleDate'
        FieldName = 'SaleDate'
      end
      ...
    end
  end
end
object DCOMConnection1: TDCOMConnection
  Connected = True
  ServerName = 'AppSPlus.AppServerPlus'
end
```

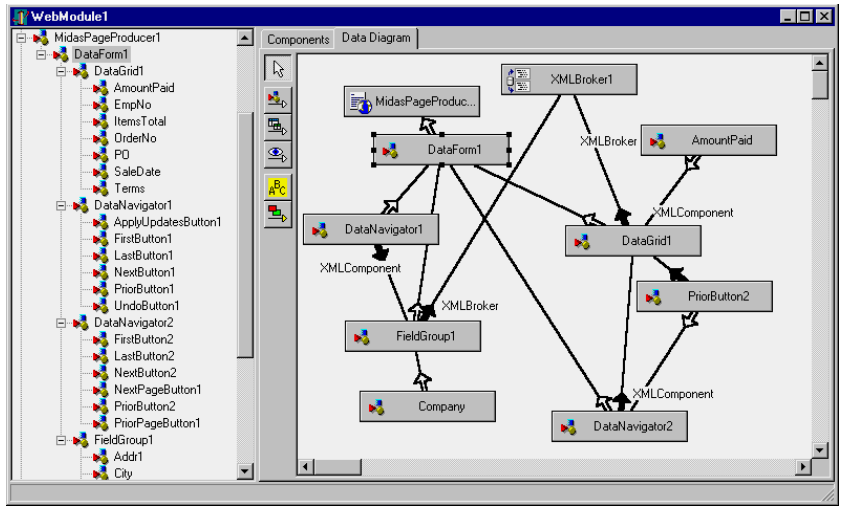


Figure 21.15: The structure of the MidasPageProducer of the IeMd example. Image from the original book. As mentioned the Data Diagram doesn't exist any more.

Once the structure is set up, you can deploy the CGI executable on the Web server and see the effect illustrated in Figure 21.16 directly in a browser. Notice that the HTML you receive is rather large, as it includes the entire master/detail structure. Once you've received it, however, you can browse the master table and the detail grid without having to ask the server for more data.

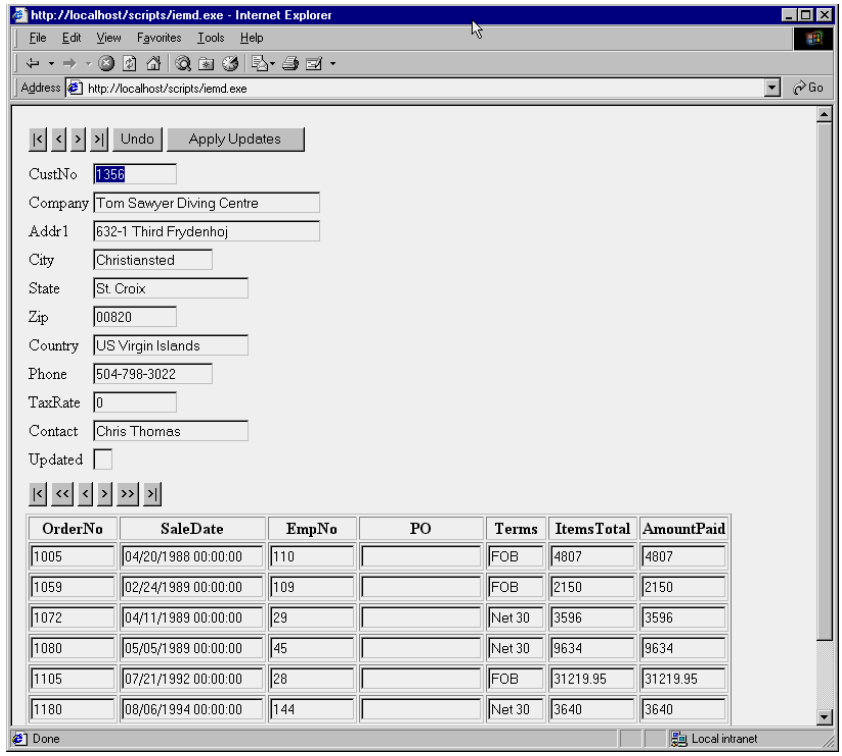


Figure 21.16: A master/detail relationship displayed in a browser by the IeMd example. Image from the original book.

Obviously, much more could be said about the capabilities of the Internet Express MIDAS components of Delphi 5 and the technologies behind it, as I haven't really introduced you to XML and JavaScript. My point was simply to give you an idea of what can be done and how fast it can be achieved using this brand-new Delphi 5 architecture, which is definitely promising in this fast-evolving area of Web development.

What's Next?

Borland/Inprise introduced support for a true three-tier architecture for the first time in Delphi 3 and has extended it in Delphi 4 and Delphi 5 to support TCP/IP sockets, MTS, CORBA, HTTP support, and even Web browser MIDAS clients. The company is continually extending this architecture to play a fundamental role in the future of client/server computing. It is also focused on the CORBA support provided

by the Visigenic ORB, and we'll hopefully have native IDL-to-Pascal mapping support very soon. At the same time, the agreement Inprise signed with Microsoft in the spring of 1999 provides a solid foundation for one of the best development tools for COM, DCOM, and MTS (and COM+ in the near future).

Of course, I don't want to delve too much into these nontechnical issues, but I thought it was worth mentioning them at the end of this book, as I try to give a few hints providing a sort of "what's next" for Delphi programmers. Delphi is indeed a strong player both in the Windows market and in the client/server and enterprise application markets, and it's probably the best tool if you need power and control in both directions. Now, with Delphi 5, it also becomes a complete platform for Web development.

Just as Borland wants to provide the best tools to developers, I hope this book has helped you master Delphi, the most successful tool Borland has brought to the market in the last few years. If you want to delve further into the secrets of the VCL library Delphi is based upon, try *Delphi Developer's Handbook* (also from Sybex), which I've co-authored with Tim Gooch and John Lam. Also check the reference, foundations, and advanced material I've collected on my Web site (`www.marcocantu.com`). This material could not be included in the book, simply because of space constraints.

There is also a bonus chapter, discussing graphics in Delphi, which is Chapter 22[467].

467 This was originally a bonus chapter available as an additional download, but I've now merged back in the ebook of this "annotated edition".

Chapter 22: Graphics In Delphi

In Chapter 6 of *Mastering Delphi 5,* I introduced the `Canvas` object, Windows painting process, and the `OnPaint` event. In this bonus chapter[468], I'm going to start from this point and continue covering graphics, following a number of different directions. (For all the code discussed here and in *Mastering Delphi 5,* check the Sybex Web site.[469])

I'll start with the development of a complex program to demonstrate how the Windows painting model works. Then I'll focus on some graphical components, such as graphical buttons and grids. During this part of the chapter we'll also add some animation to the controls.

---

468 This chapter wasn't in the printed book, but available as a separate download. I merged it in the text, but it's out of the logical order (it should have been Chapter 7 or 8). Moving it and renumbering everything would have been a lot of change, so I kept it at the end.

469 The code is in the GitHub repository, at https://github.com/MarcoDelphiBooks/MasteringDelphi5/tree/master/WebBonus/22

Finally, this chapter will discuss the use of bitmaps, covering some advanced features for fast graphics rendering, metafiles, the TeeChart component (including its use on the Web), and few more topics related to the overall issue of graphics.

Drawing on a Form

In Chapter 6, we saw that it is possible to paint directly on the surface of a form in response to a mouse event. To see this behavior, simply create a new form with the following `OnMouseDown` event handler:

```
procedure TForm1.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Canvas.Ellipse (X-10, Y-10, X+10, Y+10);
end;
```

The program *seems* to work fairly well, but it doesn't. Every click produces a new circle, but if you minimize the form, they'll all go away. Even if you cover a portion of your form with another window, the shapes behind that other form will disappear, and you might end up with partially painted circles.

As I detailed in Chapter 6, this direct drawing is not automatically supported by Windows. The standard approach is to store the painting request in the `OnMouseDown` event and then reproduce the output in the `OnPaint` event. This event, in fact, is called by the system every time the form requires *repainting*. However, you'll need to force its activation by calling the `Invalidate` or `Repaint` methods in the mouse-event handler. In other words, Windows knows when the form has to be repainted because of a system operation (such as placing another window in front of your form), but your program must notify the system when painting is required because of user input or other program operations.

The Drawing Tools

All the output operations in Windows take place using objects of the `TCanvas` class. The output operations usually don't specify colors and similar elements but use the current drawing tools of the canvas. Here is a list of these drawing tools (or *GDI*

objects, from the Graphics Device Interface, which is one of the Windows system libraries[470]):

- The `Brush` property determines the color of the enclosed surfaces. The brush is used to fill closed shapes, such as circles or rectangles. The properties of a brush are `Color`, `Style`, and optionally, `Bitmap`.
- The `Pen` property determines the color and size of the lines and of the borders of the shapes. The properties of a pen are `Color`, `Width`, and `Style`, which includes several dotted and dashed lines (available only if the `Width` is 1 pixel). Another relevant subproperty of the `Pen` is the `Mode` property, which indicates how the color of the pen modifies the color of the drawing surface. The default is simply to use the pen color (with the `pmCopy` style), but it is also possible to merge the two colors in many different ways and to reverse the current color of the drawing surface.
- The `Font` property determines the font used to write text in the form, using the `TextOut` method of the canvas. A font has a `Name`, `Size`, `Style`, `Color`, and so on.

note Experienced Windows programmers should note that a Delphi canvas technically represents a Windows device context. The methods of the `TCanvas` class are similar to the GDI functions of the Windows API. You can call extra GDI methods by using the `Handle` property of the canvas, which is a handle of an `HDC` type.

Colors

Brushes, pens, and fonts (as well as forms and most other components) have a `Color` property. However, to change the color of an element properly, using non-standard colors (such as the color constants in Delphi), you should know how Windows treats the color. In theory, Windows uses 24-bit RGB colors. This means you can use 256 different values for each of the three basic colors (red, green, and blue), obtaining 16 million different shades.

However, you or your users might have a video adapter that cannot display such a variety of colors, although this is increasingly less frequent. In this case, Windows either uses a technique called *dithering,* which basically consists of using a number of pixels of the available colors to simulate the requested one; or it approximates the

470 Notice that while the foundations of painting (via GDI) are still at the core of Delphi, the VCL also offers some Direct2D painting support (with a specific, alternative canvas) and recently also Skia-based painting.

color, using the nearest available match. For the color of a brush (and the background color of a form, which is actually based on a brush), Windows uses the dithering technique; for the color of a pen or font, it uses the nearest available color[471].

In terms of pens, you can read (but not change) the current pen position with the `PenPos` property of the canvas. The pen position determines the starting point of the next line the program will draw, using the `LineTo` method. To change it, you can use the canvas's `MoveTo` method. Other properties of the canvas affect lines and colors, too. Interesting examples are `CopyMode` and `ScaleMode`. Another property you can manipulate directly to change the output is the `Pixels` array, which you can use to access (read) or change (write) the color of any individual point on the surface of the form. As we'll see in the BmpDraw example, per pixel operations are very slow in GDI, compared to line access available through the `ScanLines` property.

Finally, keep in mind that Delphi's `TColor` values do not always match plain RGB values of the native Windows representation (`COLORREF`), because of Delphi color constants. You can always convert a Delphi color to the RGB value using the `ColorToRGB` function. You can find the details of Delphi's representation in the *TColor type* Help entry.

Drawing Shapes

Now I want to extend the Mouse1 example built at the end of Chapter 6 and turn it into the Shapes application. In this new program I want to use the *store-and-draw* approach with multiple shapes, handle color and pen attributes, and provide a foundation for further extensions.

Because you have to remember the position and the attributes of each shape, you can create an object for each shape you have to store, and you can keep the objects in a list. (To be more precise, the list will store references to the objects, which are allocated in separate memory areas.) I've defined a base class for the shapes and two inherited classes that contain the painting code for the two types of shapes I want to handle, rectangles and ellipses.

The base class has a few properties, which simply read the fields and write the corresponding values with simple methods. Notice that the coordinates can be read using the `Rect` property but must be modified using the four positional properties.

---

471 Modern computers offer graphic cards and monitors capable of displaying the actual colors.

The reason is that if you add a `write` portion to the `Rect` property, you can access the rectangle as a whole but not its specific sub-properties. Here are the declarations of the three classes:

```
type
  TBaseShape = class
  private
    FBrushColor: TColor;
    FPenColor: TColor;
    FPenSize: Integer;
    procedure SetBrushColor(const Value: TColor);
    procedure SetPenColor(const Value: TColor);
    procedure SetPenSize(const Value: Integer);
    procedure SetBottom(const Value: Integer);
    procedure SetLeft(const Value: Integer);
    procedure SetRight(const Value: Integer);
    procedure SetTop(const Value: Integer);
  protected
    FRect: TRect;
  public
    procedure Paint (Canvas: TCanvas); virtual;
  published
    property PenSize: Integer read FPenSize write SetPenSize;
    property PenColor: TColor read FPenColor write SetPenColor;
    property BrushColor: TColor read FBrushColor write SetBrushColor;
    property Left: Integer write SetLeft;
    property Right: Integer write SetRight;
    property Top: Integer write SetTop;
    property Bottom: Integer write SetBottom;
    property Rect: TRect read FRect;
  end;

type
  TEllShape = class (TBaseShape)
    procedure Paint (Canvas: TCanvas); override;
  end;

  TRectShape = class (TBaseShape)
    procedure Paint (Canvas: TCanvas); override;
  end;
```

Most of the code in the methods is very simple. The only relevant code is in the three `Paint` procedures:

```
procedure TBaseShape.Paint (Canvas: TCanvas);
begin
  // set the attributes
  Canvas.Pen.Color := fPenColor;
  Canvas.Pen.Width := fPenSize;
  Canvas.Brush.Color := fBrushColor;
end;
```

```
procedure TEllShape.Paint(Canvas: TCanvas);
begin
  inherited Paint (Canvas);
  Canvas.Ellipse (fRect.Left, fRect.Top,
    fRect.Right, fRect.Bottom)
end;

procedure TRectShape.Paint(Canvas: TCanvas);
begin
  inherited Paint (Canvas);
  Canvas.Rectangle (fRect.Left, fRect.Top,
    fRect.Right, fRect.Bottom)
end;
```

All of this code is stored in the secondary ShapesH (Shapes Hierarchy) unit. To store a list of shapes, the form has a `TList` object data member, named `ShapesList`, which is initialized in the `OnCreate` event handler and destroyed at the end; the destructor also frees all the objects in the list (in reverse order, to avoid refreshing the internal list data too often):

```
procedure TShapesForm.FormCreate(Sender: TObject);
begin
  ShapesList := TList.Create;
end;

procedure TShapesForm.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  // delete each object
  for I := ShapesList.Count - 1 downto 0 do
    TBaseShape (ShapesList [I]).Free;
  ShapesList.Free;
end;
```

The program adds a new object to the list each time the user starts the dragging operation. Since the object is not completely defined, the form keeps a reference to it in the `CurrShape` field. Notice that the type of object created depends on the status of the mouse keys:

```
procedure TShapesForm.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    // activate dragging
    fDragging := True;
    SetCapture (Handle);

    // create the proper object
```

```
    if ssShift in Shift then
      CurrShape := TEllShape.Create
    else
      CurrShape := TRectShape.Create;

    // set the style and colors
    CurrShape.PenSize := Canvas.Pen.Width;
    CurrShape.PenColor := Canvas.Pen.Color;
    CurrShape.BrushColor := Canvas.Brush.Color;

    // set the initial position
    CurrShape.Left := X;
    CurrShape.Top := Y;
    CurrShape.Right := X;
    CurrShape.Bottom := Y;
    Canvas.DrawFocusRect (CurrShape.Rect);

    // add to the list
    ShapesList.Add (CurrShape);
  end;
end;
```

During the dragging operation we draw the line corresponding to the shape, as I did in the Mouse1 example:

```
procedure TShapesForm.FormMouseMove(Sender: TObject; Shift:
TShiftState;
  X, Y: Integer);
var
  ARect: TRect;
begin
  // copy the mouse coordinates to the title
  Caption := Format ('Shapes (x=%d, y=%d)', [X, Y]);

  // dragging code
  if fDragging then
  begin
    // remove and redraw the dragging rectangle
    ARect := NormalizeRect (CurrShape.Rect);
    Canvas.DrawFocusRect (ARect);
    CurrShape.Right := X;
    CurrShape.Bottom := Y;
    ARect := NormalizeRect (CurrShape.Rect);
    Canvas.DrawFocusRect (ARect);
  end;
end;
```

This time, however, I've also added a fix to the program. In the Mouse1 example, if you move the mouse toward the upper-left corner of the form while dragging, the `DrawFocusRect` call produces no effect. The reason is that the rectangle passed as a parameter to `DrawFocusRect` must have a `Top` value that is less than the `Bottom`

value, and the same is true for the `Left` and `Right` values. In other words, a rectangle that extends itself on the negative side doesn't work properly. However, at the end it paints correctly, because the `Rectangle` drawing function doesn't have this problem.

To fix this problem I've written a simple function that inverts the coordinates of a rectangle to make it reflect the requests of the `DrawFocusRect` call:

```
function NormalizeRect (ARect: TRect): TRect;
var
  tmp: Integer;
begin
  if ARect.Bottom < ARect.Top then
  begin
    tmp := ARect.Bottom;
    ARect.Bottom := ARect.Top;
    ARect.Top := tmp;
  end;
  if ARect.Right < ARect.Left then
  begin
    tmp := ARect.Right;
    ARect.Right := ARect.Left;
    ARect.Left := tmp;
  end;
  Result := ARect;
end;
```

Finally, the `OnMouseUp` event handler sets the definitive image size and refreshes the painting of the form. Instead of calling the `Invalidate` method, which would cause all of the images to be repainted with a lot of flickering, the program uses the `InvalidateRect` API function:

```
procedure InvalidateRect(Wnd: HWnd; Rect: PRect; Erase: Bool);
```

The three parameters represent the handle of the window (that is, the `Handle` property of the form), the rectangle you want to repaint, and a flag indicating whether or not you want to erase the area before repainting it. This function requires, once more, a *normalized* rectangle. (You can try replacing this call with one to `Invalidate` to see the difference, which is more obvious when you create many forms.) Here is the complete code of the `OnMouseUp` handler:

```
procedure TShapesForm.FormMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ARect: TRect;
begin
  if fDragging then
  begin
    // end dragging
```

```
    ReleaseCapture;
    fDragging := False;

    // set the final size
    ARect := NormalizeRect (CurrShape.Rect);
    Canvas.DrawFocusRect (ARect);
    CurrShape.Right := X;
    CurrShape.Bottom := Y;

    // optimized invalidate code
    ARect := NormalizeRect (CurrShape.Rect);
    InvalidateRect (Handle, @ARect, False);
  end;
end;
```

note When you select a large drawing pen (we'll look at the code for that shortly), the border of the frame is painted partially inside and partially outside the frame, to accommodate the large pen. To allow for this, we should invalidate a frame rectangle that is inflated by half the size of the current pen. You can do this by calling the `InflateRect` function. As an alternative, in the `FormCreate` method I've set the `Style` of the `Pen` of the form `Canvas` to `psInsideFrame`. This causes the drawing function to paint the pen completely inside the frame of the shape.

In the method corresponding to the `OnPaint` event, all the shapes currently stored in the list are painted, as you can see in Figure 22.1. Since the painting code affects the properties of the `Canvas`, we need to store the current values and reset them at the end. The reason is that, as I'll show you later in this chapter, the properties of the form's canvas are used to keep track of the attributes selected by the user, who might have changed them since the last shape was created. Here is the code:

```
procedure TShapesForm.FormPaint(Sender: TObject);
var
  I, OldPenW: Integer;
  AShape: TBaseShape;
  OldPenCol, OldBrushCol: TColor;
begin
  // store the current Canvas attributes
  OldPenCol := Canvas.Pen.Color;
  OldPenW := Canvas.Pen.Width;
  OldBrushCol := Canvas.Brush.Color;

  // repaint each shape of the list
  for I := 0 to ShapesList.Count - 1 do
  begin
    AShape := ShapesList.Items [I];
    AShape.Paint (Canvas);
  end;

  // reset the current Canvas attributes
  Canvas.Pen.Color := OldPenCol;
```

```
  Canvas.Pen.Width := OldPenW;
  Canvas.Brush.Color := OldBrushCol;
end;
```

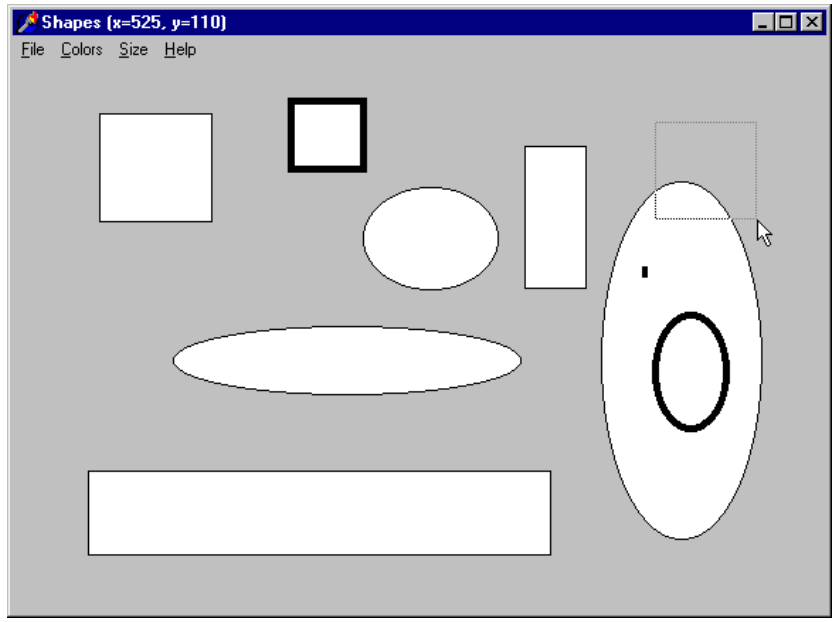


Figure 22.1: The Shapes example can be used to draw multiple shapes, which it stores in a list. Image from the original book.

The other methods of the form are simple. Three of the menu commands allow us to change the colors of the background, the shape borders (the pen), and the internal area (the brush). These methods use the ColorDialog component and store the result in the properties of the form's canvas. This is an example:

```
procedure TShapesForm.PenColor1Click(Sender: TObject);
begin
  // select a new color for the pen
  ColorDialog1.Color := Canvas.Pen.Color;
  if ColorDialog1.Execute then
    Canvas.Pen.Color := ColorDialog1.Color;
end;
```

The new colors will affect shapes created in the future but not the existing ones. The same approach is used for the width of the lines (the pen), although this time the program also checks to see whether the value has become too small, disabling the menu item if it has:

```
procedure TShapesForm.DecreasePenSize1Click(Sender: TObject);
```

```
begin
  Canvas.Pen.Width := Canvas.Pen.Width - 2;
  if Canvas.Pen.Width < 3 then
    DecreasePenSize1.Enabled := False;
end;
```

To change the colors of the border (the pen) or the surface (the brush) of the shape, I've used the standard Color dialog box. Here is one of the two methods:

```
procedure TShapesForm.PenColor1Click(Sender: TObject);
begin
  ColorDialog1.Color := Canvas.Pen.Color;
  if ColorDialog1.Execute then
   Canvas.Pen.Color := ColorDialog1.Color;
end;
```

In Figure 22.2 you can see another example of the output of the Shapes program, this time using multiple colors for the shapes and their background. The program asks the user to confirm some operations, such as exiting from the program or removing all the shapes from the list (with the File ➢ New command):

```
procedure TShapesForm.New1Click(Sender: TObject);
begin
  if (ShapesList.Count > 0) and (MessageDlg (
    'Are you sure you want to delete all the shapes?',
    mtConfirmation, [mbYes, mbNo], 0) = idYes) then
  begin
    // delete each object
    for I := ShapesList.Count - 1 downto 0 do
      TBaseShape (ShapesList [I]).Free;
    ShapesList.Clear;
    Refresh;
  end;
end;
```

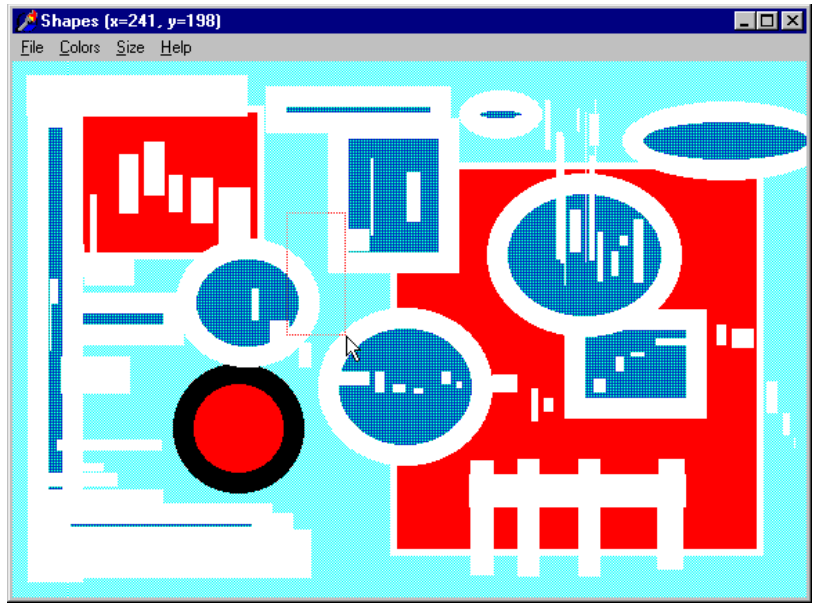


Figure 22.2: Changing the colors and the line size of shapes allows you to use the Shapes example to produce any kind of result. Image from the original book.

Printing Shapes

Besides painting the shapes on a form canvas, we can paint them on a printer canvas, effectively printing them! Because it is possible to execute the same methods on a printer canvas as on any other canvas, you might be tempted to add to the program a new method for printing the shapes. This is certainly easy, but an even better option is writing a single output method to use for both the screen and the printer.

As an example of this approach, I've built a new version of the program, called ShapesPr. The interesting point is that I've moved the code of the `FormPaint` example into another method I've defined, called `CommonPaint`. This new method has two parameters, the canvas and a scale factor (which defaults to 1):

```
procedure CommonPaint(Canvas: TCanvas; Scale: Integer = 1);
```

The `CommonPaint` method outputs the list of shapes to the canvas passed as parameters, using the proper scale factor:

```
procedure TShapesForm.CommonPaint (
  Canvas: TCanvas; Scale: Integer);
```

```
var
  I, OldPenW: Integer;
  AShape: TBaseShape;
  OldPenCol, OldBrushCol: TColor;
begin
  // store the current Canvas attributes
  OldPenCol := Canvas.Pen.Color;
  OldPenW := Canvas.Pen.Width;
  OldBrushCol := Canvas.Brush.Color;

  // repaint each shape of the list
  for I := 0 to ShapesList.Count - 1 do
  begin
    AShape := ShapesList.Items [I];
    AShape.Paint (Canvas, Scale);
  end;

  // reset the current Canvas attributes
  Canvas.Pen.Color := OldPenCol;
  Canvas.Pen.Width := OldPenW;
  Canvas.Brush.Color := OldBrushCol;
end;
```

Once you've written this code, the `FormPaint` and `Print1Click` methods are simple to implement. To paint the image on the screen, you can call `CommonPaint` without a scaling factor (so that the default value 1 is used):

```
procedure TShapesForm.FormPaint(Sender: TObject);
begin
  CommonPaint (Canvas);
end;
```

To paint the contents of the form to the printer instead of the form, you can reproduce the output on the printer canvas, using a proper scaling factor. Instead of choosing a scale, I decided to compute it automatically. The idea is to print the shapes on the form as large as possible, by sizing the form's client area so that it takes up the whole page. The code is probably simpler than the description:

```
procedure TShapesForm.Print1Click(Sender: TObject);
var
  Scale, Scale1: Integer;
begin
  Scale := Printer.PageWidth div ClientWidth;
  Scale1 := Printer.PageHeight div ClientHeight;
  if Scale1 < Scale then
    Scale := Scale1;
  Printer.BeginDoc;
  try
    CommonPaint (Printer.Canvas, Scale);
    Printer.EndDoc;
  except
```

```
    Printer.Abort;
    raise;
  end;
end;
```

Of course, you need to remember to call the specific commands to start printing (`BeginDoc`) and commit the output (`EndDoc`) before and after you call the `CommonPaint` method. If an exception is raised, the program calls `Abort` to terminate the printing process anyway.

Delphi Graphical Components

The Shapes example uses almost no components, aside from a standard color-selection dialog box. As an alternative, we could have used some Delphi components that specifically support graphics:

- You use the PaintBox[472] component when you need to paint on a certain area of a form and that area might move on the form. For example, PaintBox is useful for painting on a dialog box without the risk of mixing the area for the output with the area for the controls. The PaintBox might fit within other controls of a form, such as a toolbar or a status bar, and avoid any confusion or overlapping of the output. In the Shapes example, using this component made no sense, because we always worked on the whole surface of the form.
- You use the Shape component to paint shapes on the screen, exactly as we have done up to now. You could indeed use the Shape component instead of the manual output, but I really wanted to show you how to accomplish some direct output operations. This approach was not much more complex than the one Delphi suggests. Using the Shape component would have been useful to extend the example, allowing a user to drag shapes on the screen, remove them, and work on them in a number of other ways.
- You can use the Image component to display an existing bitmap, possibly loading it from a file, or even to paint on a bitmap, as I'll demonstrate in the next two examples and discuss in the next section.

---

472 There is now also a Skia-based PaintBox component.

- If it is included in your version of Delphi, you can use the TeeChart control to create business graphics output, as we'll see toward the end of this chapter.
- You can use the graphical support provided by the bitmap buttons and speed button controls, among others. We'll see later in this chapter how to extend the graphical capabilities of these controls.
- You can use the Animate component to make the graphics more—well, animated. Besides using this component, you can manually create animations by displaying bitmaps in sequence or scrolling them, as we'll see other examples.

As you can see, we have a long way to go to cover Delphi's graphics support from all of its angles.

Drawing in a Bitmap

I've already mentioned that by using an Image component, you can draw images directly in a bitmap. Instead of drawing on the surface of a window, you draw on a bitmap in memory and then copy the bitmap to the surface of the window. The advantage is that instead of having to repaint the image each time an `OnPaint` event occurs, the component copies the bitmap back to video.

Technically, a `TBitmap` object has its own canvas. By drawing on this canvas, you can change the contents of the bitmap. As an alternative, you can work on the canvas of an Image component connected to the bitmap you want to change. You might consider choosing this approach instead of the typical painting approach if any of the following conditions are true:

- The program has to support freehand drawing or very complex graphics (such as fractal images).
- The program should be very fast in drawing a number of images.
- RAM consumption is not an issue.
- You are a lazy programmer.

The last point is interesting because painting generally requires more code than drawing, although it allows more flexibility. In a graphics program, for example, if you use painting, you have to store the location and colors of each shape. On the other hand, you can easily change the color of an existing shape or move it. These operations are very difficult with the painting approach and may cause the area behind an image to be lost. If you are working on a complex graphical application,

you should probably choose a mix of the two approaches. For casual graphics programmers, the choice between the two approaches involves a typical speed-versus-memory decision: painting requires less memory; storing the bitmap is faster.

Drawing Shapes

Now let's look at an Image component example that will paint on a bitmap. The idea is simple. I've basically written a simplified version of the Shape example, by placing an Image component on its form and redirecting all the output operations to the canvas of this Image component.

In this example, ShapeBmp, I've also added some new menu items to save the image to a file and to load an existing bitmap. To accomplish this, I've added to the form a couple of default dialog components, OpenDialog and SaveDialog. One of the properties I had to change was the background color of the form. In fact, when you perform the first graphical operation on the image, it creates a bitmap, which has a white background by default. If the form has a gray background, each time the window is repainted, some flickering occurs. For this reason, I've chosen a white background for the form, too.

The code of this example is still quite simple, considering the number of operations and menu commands. The drawing portion is linear and very close to Mouse1, except that the mouse events now relate to the image instead of the form; I've used the `NormalizeRect` function during the dragging; and the program uses the canvas of the image. Here is the `OnMouseMove` event handler, which reintroduces the drawing of points when moving the mouse with the Shift key pressed:

```
procedure TShapesForm.Image1MouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
var
  ARect: TRect;
begin
  // display the position of the mouse in the caption
  Caption := Format ('ShapeBmp (x=%d, y=%d)', [X, Y]);
  if fDragging then
  begin
    // remove and redraw the dragging rectangle
    ARect := NormalizeRect (fRect);
    Canvas.DrawFocusRect (ARect);
    fRect.Right := X;
    fRect.Bottom := Y;
    ARect := NormalizeRect (fRect);
    Canvas.DrawFocusRect (ARect);
  end
  else
```

```
    if ssShift in Shift then
      // mark point in red
      Image1.Canvas.Pixels [X, Y] := clRed;
end;
```

Notice that the temporary focus rectangle is painted directly on the form, over the image (and thus not stored in the bitmap). What is different is that at the end of the dragging operation, the program paints the rectangle on the image, storing it in the bitmap. This time the program doesn't call `Invalidate` and has no `OnPaint` event handler:

```
procedure TShapesForm.Image1MouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if fDragging then
  begin
    ReleaseCapture;
    fDragging := False;
    Image1.Canvas.Rectangle (fRect.Left, fRect.Top,
      fRect.Right, fRect.Bottom);
  end;
end;
```

To avoid overly complex file support, I decided to implement the File ➢ Load and File ➢ Save As commands and not handle the Save command, which is generally more complex. I've simply added an `fChanged` field to the form to know when an image has changed, and I've included code that checks this value a number of times (before asking the user to confirm).

The `OnClick` event handler of the File ➢ New menu item calls the `FillArea` method to paint a big white rectangle over the whole bitmap. In this code you can also see how the `Changed` field is used:

```
procedure TShapesForm.New1Click(Sender: TObject);
var
  Area: TRect;
  OldColor: TColor;
begin
  if not fChanged or (MessageDlg (
    'Are you sure you want to delete the current image?',
    mtConfirmation, [mbYes, mbNo], 0) = idYes) then
  begin
    {repaint the surface, covering the whole area,
    and resetting the old brush}
    Area := Rect (0, 0, Image1.Picture.Width,
      Image1.Picture.Height);
    OldColor := Image1.Canvas.Brush.Color;
    Image1.Canvas.Brush.Color := clWhite;
    Image1.Canvas.FillRect (Area);
    Image1.Canvas.Brush.Color := OldColor;
```

```
    fChanged := False;
  end;
end;
```

Of course, the code has to save the original color and restore it later on. A realignment of the colors is also required by the File ➢ Load command-response method. When you load a new bitmap, in fact, the Image component creates a new canvas with the default attributes. For this reason, the program saves the pen's colors and size and copies them later to the new canvas:

```
procedure TShapesForm.Load1Click(Sender: TObject);
var
  PenCol, BrushCol: TColor;
  PenSize: Integer;
begin
  if not fChanged or (MessageDlg (
      'Are you sure you want to delete the current image?',
      mtConfirmation, [mbYes, mbNo], 0) = idYes) then
    if OpenDialog1.Execute then
    begin
      PenCol := Image1.Canvas.Pen.Color;
      BrushCol := Image1.Canvas.Brush.Color;
      PenSize := Image1.Canvas.Pen.Width;
      Image1.Picture.LoadFromFile (OpenDialog1.Filename);
      Image1.Canvas.Pen.Color := PenCol;
      Image1.Canvas.Brush.Color := BrushCol;
      Image1.Canvas.Pen.Width := PenSize;
      fChanged := False;
    end;
end;
```

Saving the current image is much simpler:

```
procedure TShapesForm.Saveas1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
  begin
    Image1.Picture.SaveToFile (
      SaveDialog1.Filename);
    fChanged := False;
  end;
end;
```

Finally, here is the code of the `OnCloseQuery` event of the form, which uses the `Changed` field:

```
procedure TShapesForm.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if not fChanged or (MessageDlg (
      'Are you sure you want to delete the current image?',
```

```
      mtConfirmation, [mbYes, mbNo], 0) = idYes) then
    CanClose := True
  else
    CanClose := False;
end;
```

ShapeBmp is an interesting program (see Figure 22.3), with limited but working file support. The real problem is that the Image component creates a bitmap of its own size. When you increase the size of the window, the Image component is resized but not the bitmap in memory. Therefore, you cannot draw on the right and bottom areas of the window. There are a number of possible solutions: use the `Constraints` property to set the maximum size of the form, use a fixed border, visually mark the *drawing area* on the screen, and so on. However, I've decided to leave the program as is because it does its job of demonstrating how to draw in a bitmap well enough.

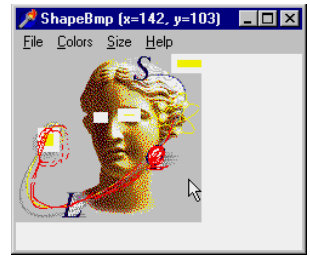


Figure 22.3: The ShapeBmp example has limited but working file support: you can load an existing bitmap, draw shapes over it, and save it to disk. Image from the original edition of the book.

An Image Viewer

The ShapeBmp program can be used as an image viewer, because you can load any bitmap in it. In general, in the Image control you can load any graphic file type that has been registered with the VCL `TPicture` class. The default file formats are bitmap files (BMP), icon files (ICO), or Windows metafiles (WMF). Bitmap and icon files are well-known formats. Windows metafiles, however, are not so common. They are a collection of graphical commands, similar to a list of GDI function calls that need to be executed to rebuild an image[473]. Metafiles are usually referred to as *vector graphics* and are similar to the graphics file formats used for clip-art libraries. Delphi also ships with JPG support for `TImage`, and third parties have GIF and other file formats covered.

473 The component now supports many more formats, thanks to the integration of the Windows Imaging Component (WIC)

note To produce a Windows metafile, a program should call GDI functions, redirecting their output to the file. In Delphi, you can use a `TMetafileCanvas` and the high-level `TCanvas` methods. Later on, this metafile can be *played* or executed to call the corresponding functions, thus producing a graphic. Metafiles have two main advantages: the limited amount of storage they require compared to other graphical formats, and the device-independence of their output. I'll cover Delphi metafile support later in this chapter.

To build a full-blown image viewer program, ImageV, around the Image component, we only need to create a form with an image that fills the whole client area, a simple menu, and an OpenDialog component:

```
object ViewerForm: TViewerForm
  Caption = 'Image Viewer'
  Menu = MainMenu1
  object Image1: TImage
    Align = alClient
  end
  object MainMenu1: TMainMenu
    object File1: TMenuItem...
      object Open1: TMenuItem...
      object Exit1: TMenuItem...
    object Options1: TMenuItem
      object Stretch1: TMenuItem
      object Center1: TMenuItem
    object Help1: TMenuItem
      object AboutImageViewer1: TMenuItem
  end
  object OpenDialog1: TOpenDialog
    FileEditStyle = fsEdit
    Filter = 'Bitmap (*.bmp)|*.bmp|
      Icon (*.ico)|*.ico|Metafile (*.wmf)|*.wmf'
    Options = [ofHideReadOnly, ofPathMustExist,
      ofFileMustExist]
  end
end
```

Surprisingly, this application requires very little coding, at least in its first basic version. The File ➢ Exit and Help ➢ About commands are trivial, and the File ➢ Open command has the following code:

```
procedure TViewerForm.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    Image1.Picture.LoadFromFile (OpenDialog1.FileName);
    Caption := 'Image Viewer - ' + OpenDialog1.FileName;
  end;
end;
```

The fourth and fifth menu commands, Options ➤ Stretch and Options ➤ Center, simply toggle the component's `Stretch` property (see Figure 22.4 for the result) or `Center` property and add a check mark to themselves. Here is the `OnClick` event handler of the `Stretch1` menu item:

```
procedure TViewerForm.Stretch1Click(Sender: TObject);
begin
  Image1.Stretch := not Image1.Stretch;
  Stretch1.Checked := Image1.Stretch;
end;
```

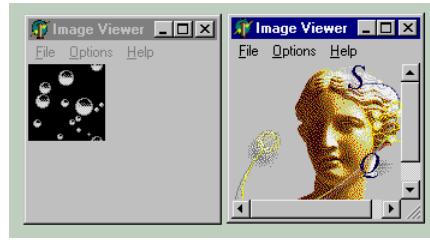


Figure 22.4: Two copies of the ImageV program, which display the regular and stretched versions of the same bitmap. Image from the original book.

Keep in mind that when stretching an image, you can change its width-to-height ratio, possibly distorting the shape, and that not all images can be properly stretched. Stretching black-and-white or 256-color bitmaps doesn't always work correctly.

Besides this problem, the application has some other drawbacks. If you select a file without one of the standard extensions, the Image component will raise an exception. The exception handler provided by the system behaves as we would expect; the wrong image file is not loaded, and the program can safely continue. Another problem is that if you load a large image, the viewer has no scroll bars. You can maximize the viewer window, but this might not be enough. The Image components do not handle scroll bars automatically, but the form can do it. I'll further extend this example to include scroll bars in the following paragraph.

Scrolling an Image

An advantage of the way automatic scrolling works in Delphi is that if the size of a single big component contained in a form changes, scroll bars are added or removed automatically. A good example is the use of the Image component. If the `AutoSize` property of this component is set to `True` and you load a new picture into it, the

component automatically sizes itself, and the form adds or removes the scroll bars as needed.

If you load a large bitmap in the ImageV example, you will notice that part of the bitmap remains hidden. To fix this, you can set the `AutoSize` property of the Image component to `True` and disable its alignment with the client area. You should also set a small initial size for the image. You don't need to make any adjustments when you load a new bitmap, because the size of the Image component is automatically set for you by the system. You can see in Figure 22.5 that scroll bars are actually added to the form. The figure shows two different copies of the program. The difference between the copy of the program on the left and the one on the right is that the first has an image smaller than its client area, so no scroll bars were added. When you load a larger image in the program, two scroll bars will automatically appear, as in the example on the right.

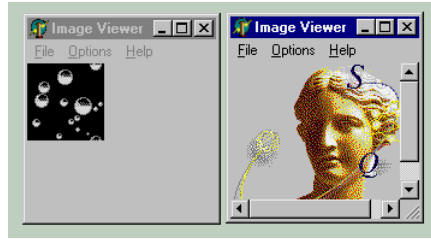


Figure 22.5: In the ImageV2 example, the scroll bars are added automatically to the form when the whole bitmap cannot fit into the client area of the form displayed. Image from the original book.

Some more coding is required to disable the scroll bars and change the alignment of the image when the Stretch menu command is selected and to restore them when this feature is disabled. Again, we don't act directly on the scroll bars themselves but simply change the alignment of the panel, using its `Stretch` property, and manually calculate the new size, using the size of the picture currently loaded. This code mimics the effect of the `AutoSize` property, which works only when loading new files.

```
procedure TViewerForm.Stretch1Click(Sender: TObject);
begin
  Image1.Stretch := not Image1.Stretch;
  Stretch1.Checked := Image1.Stretch;
  if Image1.Stretch then
    Image1.Align := alClient
  else
  begin
   Image1.Align := alNone;
   Image1.Height := Image1.Picture.Height;
   Image1.Width := Image1.Picture.Width;
  end;
end;
```

Bitmaps to the Max

When the Image control is connected to a bitmap, there are some additional operations you can do, but before we examine them, I have to introduce bitmap formats. There are different types of bitmaps in Windows. Bitmaps can be *device-independent* or not, a term used to indicate whether the bitmap has extra palette management information. BMP files are usually device-independent bitmaps.

Another difference relates to the color depth—that is, the number of different colors the bitmap can use or, in other words, the number of bits required for storing each pixel. In a 1-bit bitmap, each point can be either black or white (to be more precise, 1-bit bitmaps can have a color palette, allowing the bitmap to represent any two colors and not just black and white). An 8-bit bitmap usually has a companion palette to indicate how the 256 different colors map to the actual system colors, a 24-bit bitmap indicates the system color directly. To make things more complex, when the system draws a bitmap on a computer with a different color capability, it has to perform some conversion.

Internally the bitmap format is very simple, whatever the color depth. All the values that make up a line are stored sequentially in a memory block. This is efficient for moving the data from memory to the screen, but it is not an effective way to store information; BMP files are generally very large, and they perform no compression.

note The BMP format actually has a very limited form of compression, known as Run-Length Encoding (RLE), in which subsequent pixels with the same color are replaced by the number of such pixels followed by the color. This can reduce the size of the image, but in some cases it will make it grow. For compressed images in Delphi, you can use the `TJpegImage` class and the support for the JPEG format offered by the `TPicture` class. Actually, all `TPicture` does is to manage a registered list of graphic classes.

The BmpDraw example uses this information about the internal structure of a bitmap and some other technical features to take direct handling of bitmaps to a new level. First, it extends the ImageV example by adding a menu item you can use to display the color depth of the current bitmap, by using the corresponding `PixelFormat` property:

```
procedure TBitmapForm.ColorDepth1Click(Sender: TObject);
var
  strDepth: String;
begin
  case Image1.Picture.Bitmap.PixelFormat of
    pfDevice: strDepth := 'Device';
    pf1bit: strDepth := '1-bit';
    pf4bit: strDepth := '4-bit';
```

```
    pf8bit: strDepth := '8-bit';
    pf15bit: strDepth := '15-bit';
    pf16bit: strDepth := '16-bit';
    pf24bit: strDepth := '24-bit';
    pf32bit: strDepth := '32-bit';
    pfCustom: strDepth := 'Custom';
  end;
  MessageDlg ('Bitmap color depth: ' + strDepth,
    mtInformation, [mbOK], 0);
end;
```

You can try loading different bitmaps and see the effect of this method, as shown in Figure 22.6.

What is more interesting is to study how to access the memory image held by the bitmap object. A simple solution is to use the `Pixels` property, as I've done in the ShapeBmp example, to draw the red pixels during the dragging operation. In this program I've added a menu item to create an entire new bitmap pixel by pixel, using a simple mathematical calculation to determine the color. (The same approach can be used, for example, to build fractal images.)

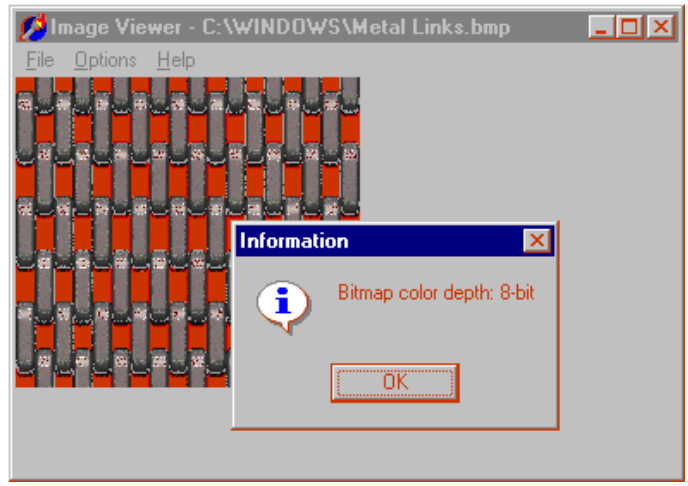


Figure 22.6: The color depth of a standard Windows bitmap, as displayed by the BmpDraw example. Image from the original book (the red borders are incorrect, it looks like the file got corrupted)

Here is the code of the method, which simply scans the bitmap in both directions and defines the color of each pixel. Because we are doing many operations on the bitmap, I can store a reference to it in the local `Bmp` variable for simplicity:

```
procedure TBitmapForm.GenerateSlow1Click(Sender: TObject);
var
  Bmp: TBitmap;
```

```
  I, J, T: Integer;
begin
  // get the image and modify it
  Bmp := Image1.Picture.Bitmap;
  Bmp.PixelFormat := pf24bit;
  Bmp.Width := 256;
  Bmp.Height := 256;

  T := GetTickCount;
  // change every pixel
  for I := 0 to Bmp.Height - 1 do
    for J := 0 to Bmp.Width - 1 do
      Bmp.Canvas.Pixels [I, J] := RGB (I*J mod 255, I, J);
  Caption := 'Image Viewer - Memory Image (MSecs: ' +
    IntToStr (GetTickCount - T) + ')';
end;
```

Notice that the program keeps track of the time required by this operation, which on my computer takes about six seconds. As you see from the name of the function, this is the slow version of the code.

We can speed it up considerably by accessing the bitmap one entire row at a time. This little-known feature is available through the `ScanLine` property of the bitmap, which returns a pointer to the memory area of the bitmap line. By taking this pointer and accessing the memory directly, we make the program much faster. The only problem is that we need to know the internal representation of the bitmap. In the case of a 24-bit bitmap, every point is represented by three bytes defining the amount of blue, green, and red (the reverse of the RGB sequence). Here is the alternative code, with a slightly different output (as I've deliberately modified the calculation of the color):

```
procedure TBitmapForm.GenerateFast1Click(Sender: TObject);
var
  Bmp: TBitmap;
  I, J, T: Integer;
  Line: PByteArray;
begin
  // get the image and modify it
  Bmp := Image1.Picture.Bitmap;
  Bmp.PixelFormat := pf24bit;
  Bmp.Width := 256;
  Bmp.Height := 256;

  T := GetTickCount;
  // change every pixel, line by line
  for I := 0 to Bmp.Height - 1 do
  begin
    Line := PByteArray (Bmp.ScanLine [I]);
    for J := 0 to Bmp.Width - 1 do
    begin
```

```
      Line [J*3] := J;
      Line [J*3+1] := I*J mod 255;
      Line [J*3+2] := I;
    end;
  end;
  // refresh the video
  Image1.Invalidate;
  Caption := 'Image Viewer - Memory Image (MSecs: ' +
    IntToStr (GetTickCount - T) + ')';
end;
```

Simply moving a line in memory doesn't cause a screen update, so the program calls `Invalidate` at the end. The output produced by this second method (see Figure 22.7) is very similar, but the time it takes on my computer is about 60 milliseconds. That's about one hundredth the time of the other approach! This technique is so fast that we can use it for scrolling the lines of the bitmap and still produce a fast and smooth effect. The scrolling operation has a few options, so as you select the corresponding menu items, the program simply shows a panel inside the form. This panel has a trackbar you can use to adjust the speed of the scrolling operation (reducing its smoothness as the speed increases). The position of the trackbar is saved in a local field of the form:

```
procedure TBitmapForm.TrackBar1Change(Sender: TObject);
begin
  nLines := TrackBar1.Position;
  TrackBar1.Hint := IntToStr (TrackBar1.Position);
end;
```

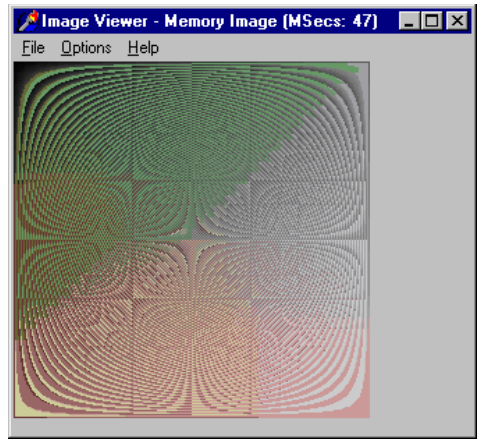


Figure 22.7: The drawing you see on the screen is generated by the BmpDraw example in a fraction of a second (as reported in the caption). Image from the original book.

In the panel there are also two buttons used to start and stop the scrolling operation. The code of the Go button has two `for` loops. The external loop is used to repeat the scrolling operation, as many times as there are lines in the bitmap. The internal loop does the scrolling operation by copying each line of the bitmap to the previous one. The first line is temporarily stored in a memory block and then copied to the last line at the end. This temporary memory block is kept in a dynamically allocated memory area (`AllocMem`) large enough to hold one line. This information is obtained by computing the difference in the memory addresses of two consecutive lines.

The core of the moving operation is accomplished using Delphi's `Move` function. Its parameters are the variable to be moved, not the memory addresses. For this reason, you have to de-reference the pointers. (Well, this method is really a good exercise on pointers!) Finally, notice that this time we cannot invalidate the entire image after each scrolling operation, as this produces too much flickering in the output. The opposite solution is to invalidate each line after it has been moved, but this makes the program far too slow. As an in-between solution, I decided to invalidate a block of lines at a time, as determined by the `J mod nLines = 0` expression. When a given number of lines has been moved, the program refreshes those lines:

```
Rect (0, PanelScroll.Height + H - nLines,
  W, PanelScroll.Height + H);
```

As you can see, the number of lines is determined by the position of the TrackBar control.

A user can even change the speed by moving their thumb during the scrolling operation. We also allow the user to press the Cancel button during the operation. This is made possible by the call to `Application.ProcessMessages` in the external `for` loop. The Cancel button changes the `fCancel` flag, which is checked at each iteration of the external `for` loop:

```
procedure TBitmapForm.BtnCancelClick(Sender: TObject);
begin
  fCancel := True;
end;
```

So, after all this description, here is the complete code of the Go button's `OnClick` event handler:

```
procedure TBitmapForm.BtnGoClick(Sender: TObject);
var
  W, H, I, J, LineBytes: Integer;
  Line: PByteArray;
  Bmp: TBitmap;
  R: TRect;
```

```
begin
  // set the user interface
  fCancel := False;
  BtnGo.Enabled := False;
  BtnCancel.Enabled := True;

  // get the bitmap of the image and resize it
  Bmp := Image1.Picture.Bitmap;
  W := Bmp.Width;
  H := Bmp.Height;

  // allocate enough memory for one line
  LineBytes := Abs (Integer (Bmp.ScanLine [1]) -
    Integer (Bmp.ScanLine [0]));
  Line := AllocMem (LineBytes);

  // scroll as many items as there are lines
  for I := 0 to H - 1 do
  begin
    // exit the for loop if Cancel was pressed
    if fCancel then
      Break;

    // copy the first line
    Move ((Bmp.ScanLine [0])^, Line^, LineBytes);

    // for every line
    for J := 1 to H - 1 do
    begin
      // move line to the previous one
      Move ((Bmp.ScanLine [J])^, (Bmp.ScanLine [J-1])^, LineBytes);
      // every nLines update the output
      if (J mod nLines = 0) then
      begin
        R := Rect (0, PanelScroll.Height + J-nLines,
          W, PanelScroll.Height + J);
        InvalidateRect (Handle, @R, False);
        UpdateWindow (Handle);
      end;
    end;

    // move the first line back to the end
    Move (Line^, (Bmp.ScanLine [Bmp.Height - 1])^, LineBytes);
    // update the final portion of the bitmap
    R := Rect (0, PanelScroll.Height + H - nLines,
      W, PanelScroll.Height + H);
    InvalidateRect (Handle, @R, False);
    UpdateWindow (Handle);

    // let the program handle other messages
    Application.ProcessMessages;
  end;
```

```
  // reset the UI
  BtnGo.Enabled := True;
  BtnCancel.Enabled := False;
end;
```

You can see a bitmap during the scrolling operation in Figure 22.8. Notice that the scrolling can take place on any type of bitmap, not just the 24-bit bitmaps generated by this program. You can, in fact, load another bitmap into the program and then scroll it, as I did to create the illustration.

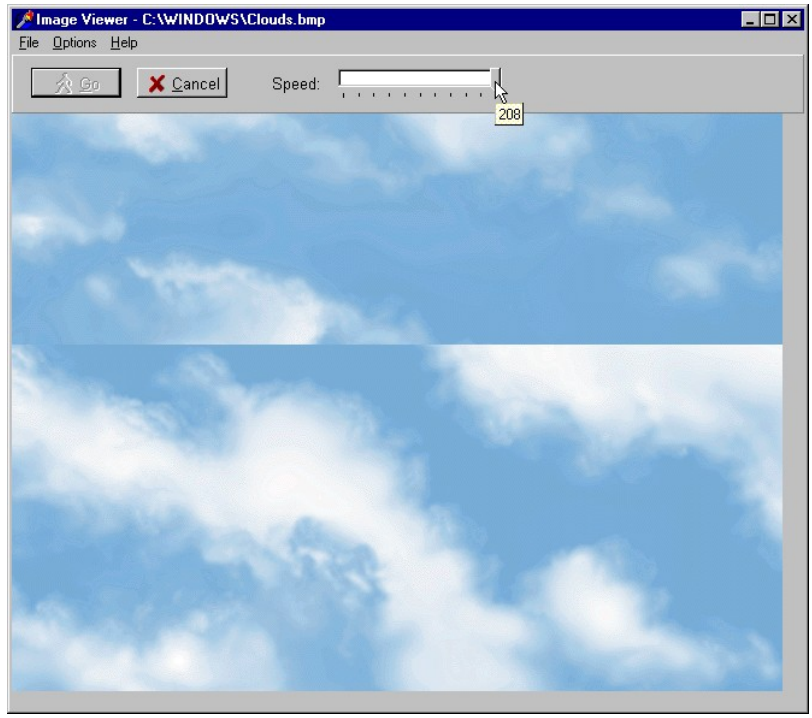


Figure 22.8: The BmpDraw example allows fast scrolling of a bitmap. Image from the original book.

An Animated Bitmap in a Button

Bitmap buttons are easy to use and can produce better-looking applications than the standard push buttons (the Button component). To further improve the visual effect of a button, we can also think of *animating* the button. There are basically two kinds

of animated buttons—buttons that change their glyph slightly when they are pressed and buttons having a moving image, regardless of the current operation. I'll show you a simple example of each kind, Fire and World. For each of these examples, we'll explore a couple of slightly different versions.

A Two-State Button

The first example, the Fire program, has a very simple form, containing only a bitmap button. This button is connected to a `Glyph` representing a cannon. Imagine such a button as part of a game program. As the button is pressed, the glyph changes to show a firing cannon. As soon as the button is released, the default glyph is loaded again. In between, the program displays a message if the user has actually clicked the button.

To write this program, we need to handle three of the button's events: `OnMouseDown`, `OnMouseUp`, and `OnClick`. The code of the three methods is extremely simple:

```
procedure TForm1.BitBtnFireMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  // load firing cannon bitmap
  if Button = mbLeft then
    BitBtnFire.Glyph.LoadFromFile ('fire2.bmp');
end;

procedure TForm1.BitBtnFireMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  // load default cannon bitmap
  if Button = mbLeft then
    BitBtnFire.Glyph.LoadFromFile ('fire.bmp');
end;

procedure TForm1.BitBtnFireClick(Sender: TObject);
begin
  PlaySound ('Boom.wav', 0, snd_Async);
  MessageDlg ('Boom!', mtWarning, [mbOK], 0);
end;
```

I've added some sound capabilities, playing a WAV file when the button is pressed with a call to the `PlaySound` function of the MmSystem unit. When you hold down the left mouse button over the bitmap button, the bitmap button is pressed. If you then move the mouse cursor away from the button while holding down the mouse button, the bitmap button is released, but it doesn't get an `OnMouseUp` event, so the firing cannon remains there. If you later release the left mouse button outside the

surface of the bitmap button, it receives the `OnMouseUp` event anyway. The reason is that all buttons in Windows capture the mouse input when they are pressed.

Many Images in a Bitmap

The Fire example used a manual approach. I loaded two bitmaps and changed the value of the `Glyph` property when I wanted to change the image. The BitBtn component, however, can also handle a number of bitmaps automatically. You can prepare a single bitmap that contains a number of images (or glyphs) and set this number as the value of the `NumGlyphs` property. All such "sub-bitmaps" must have the same size because the overall bitmap is divided into equal parts.

If you provide more than one glyph in the bitmap, they are used according to the following rules:

- The first bitmap is used for the released button, the default position.
- The second bitmap is used for the disabled button.
- The third bitmap is used when the button is clicked.
- The fourth bitmap is used when the button remains down, as in buttons behaving as check boxes.

Usually you provide a single glyph and the others are automatically computed from it, with simple graphical changes. However, it is easy to provide a second, a third, and a fourth customized picture. If you do not provide all four bitmaps, the missing ones will be computed automatically from the first one.

In our example, the new version of Fire (named Fire2), we only need the first and third glyphs of the bitmap but are obliged to add the second bitmap. To see how this glyph (the second of the bitmap) can be used, I've added a check box to disable the bitmap button. To build the new version of the program, I've prepared a bitmap of 32 × 96 pixels (see Figure 22.9) and used it for the `Glyph` property of the bitmap. Delphi automatically set the `NumGlyphs` property to 3, because the bitmap is three times wider than it is high.

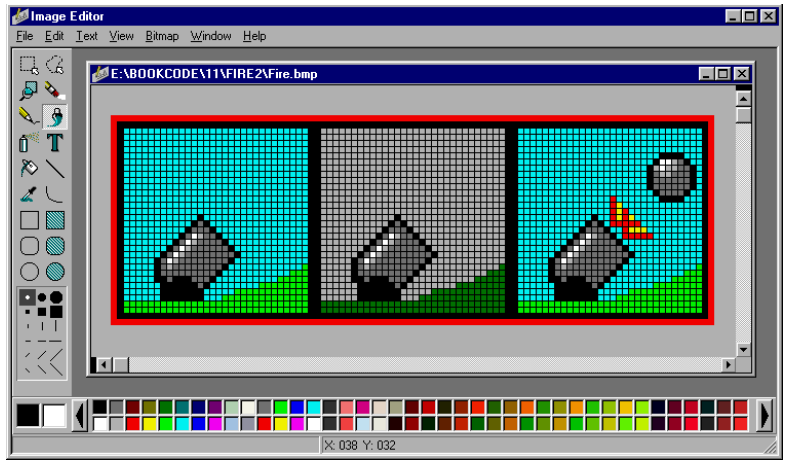


Figure 22.9: The bitmap with three images of the Fire2 example, as seen in the Delphi Image Editor. Image from the original book (as mentioned the Image Editor doesn't exist any more in Delphi).

The check box, used to enable and disable the button (so we can see the glyph corresponding to the disabled status), has the following `OnClick` event:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  BitBtnFire.Enabled := CheckBox1.Checked;
end;
```

When you run the program, there are two ways to change the bitmap in the button. You can disable the bitmap button by using the check box (see Figure 22.10), or you can press the button to see the cannon fire. In the first version (the Fire example), the image with the firing cannon remained on the button until the message box was closed. Now (in the Fire2 example) the image is shown only while the button is pressed. As soon as you move outside the surface of the button, or release the button after having pressed it (activating the message box), the first glyph is displayed.

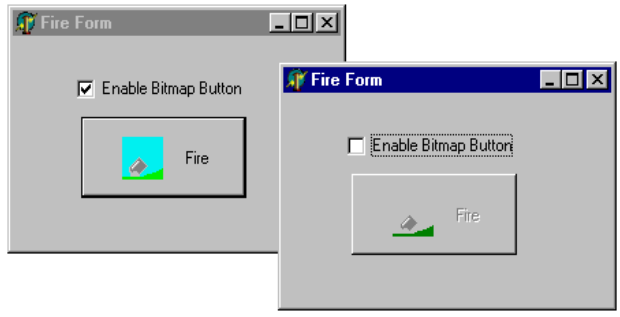


Figure 22.10: The enabled and disabled bitmap buttons of the Fire2 example, in two different copies of the application. Image from the original book.

The Rotating World

The second example of animation, World, has a button featuring the earth, which slowly rotates, showing the various continents. You can see some samples in Figure 22.11, but, of course, you should run the program to see its output. In the previous example, the image changed when the button was pressed. Now the image changes by itself, automatically. This occurs thanks to the presence of a Timer component, which receives a message at fixed time intervals.

Here is a summary of the component properties:

```
object WorldForm: TWorldForm
  Caption = 'World'
  OnCreate = FormCreate
  object Label1: TLabel...
  object WorldButton: TBitBtn
    Caption = '&Start'
    OnClick = WorldButtonClick
    Glyph.Data = {w1.bmp}
    Spacing = 15
  end
  object Timer1: TTimer
    Enabled = False
    Interval = 500
    OnTimer = Timer1Timer
  end
end
```

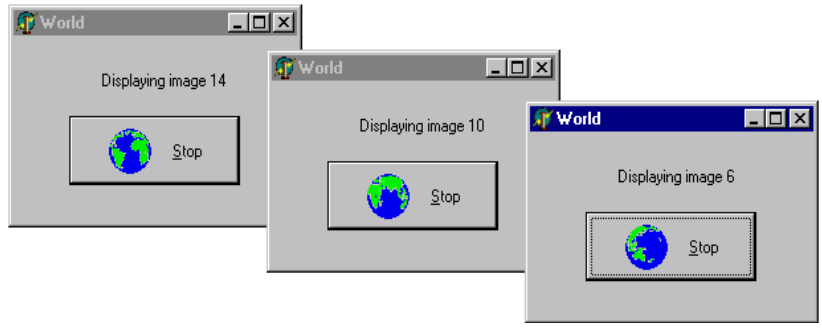


Figure 22.11: Some examples of the running World program. Image from the original book.

The timer component is started and stopped (enabled and disabled) when the user presses the bitmap button with the world image:

```
procedure TWorldForm.WorldButtonClick(Sender: TObject);
begin
  if Timer1.Enabled then
```

```
  begin
    Timer1.Enabled := False;
    WorldButton.Caption := '&Start';
  end
  else
  begin
    Timer1.Enabled := True;
    WorldButton.Caption := '&Stop';
  end;
end;
```

As you can see in Figure 22.11, a label above the button indicates which of the images is being displayed. Each time the timer message is received, the image and label change:

```
procedure TWorldForm.Timer1Timer(Sender: TObject);
begin
  Count := (Count mod 16) + 1;
  Label1.Caption := 'Displaying image ' +
    IntToStr (Count);
  WorldButton.Glyph.LoadFromFile (
     'w' + IntToStr (Count) + '.bmp');
end;
```

In this code, `Count` is a field of the form that is initialized to `1` in the `FormCreate` method. At each timer interval, `Count` is increased modulus `16` and then converted into a string (preceded by the letter *w*). The reason for this limit is simple—I had 16 bitmaps of the earth to display. Naming the bitmap files `W1.BMP`, `W2.BMP`, and so on makes it easy for the program to access them, building the strings with the name at run time.

note The modulus operation returns the remainder of the division between integers. This means that `Count mod 16` invariably returns a value in the range 0–15. Adding one to this return value, we obtain the number of the bitmap, which is in the range 1–16.

A List of Bitmaps, the Use of Resources, and a ControlCanvas

The World program works, but it is very slow, for a couple of reasons. First of all, at each timer interval, it needs to read a file from the disk, and although a disk cache can make this faster, it is certainly not the most efficient solution. Besides reading the file from disk, the program has to create and destroy Windows bitmap objects, and this takes some time. The second problem depends on how the image is updated: When you change the button's bitmap, the component is completely

erased and repainted. This causes some flickering, as you can see by running the program.

To solve the first problem (and to show you a different approach to handling bitmaps), I've created a second version of the example, World2. Here I've added a `TObjectList` Delphi 5 container, storing a list of bitmaps, to the program's form. The form has also some more fields:

```
type
  TWorldForm = class(TForm)
    ...
  private
    Count, YPos, XPos: Integer;
    BitmapsList: TObjectList;
    ControlCanvas: TControlCanvas;
  end;
```

All the bitmaps are loaded when the program starts and destroyed when it terminates. At each timer interval, the program shows one of the list's bitmaps in the bitmap button. By using a list, we avoid loading a file each time we need to display a bitmap, but we still need to have all the files with the images in the directory with the executable file. A solution to this problem is to move the bitmaps from independent files to the application's resource file. This is easier to do than to explain.

To use the resources instead of the bitmap files, we need to first create this file. The best approach is to write a resource script (an RC file), listing the names of the bitmap files and of the corresponding resources. Open a new text file (in any editor) and write the following code:

```
W1  BITMAP  "W1.BMP"
W2  BITMAP  "W2.BMP"
W3  BITMAP  "W3.BMP"
// ... and so on
```

Once you have prepared this RC file (I've named it `WorldBmp.RC`), you can compile it into a RES file using the resource compiler included and the BRCC32 command-line application you can find in the BIN directory of Delphi, and then include it in the project by adding the `{$R WORLDBMP.RES}` directive in the project source code file or in one of the units.

In Delphi 5, however, you can use a simpler approach. You can take the RC file and simply add it to the project using the Project Manager Add command or simply dragging the file to the project. Delphi 5 will automatically activate the resource compiler, and it will then bind the resource file into the executable file. These operations are controlled by an extended resource inclusion directive added to the project source code:

```
{$R 'WORLDBMP.res' WORLDBMP.RC'}
```

Once we have properly defined the resources of the application, we need to load the bitmaps from the resources. For a `TBitmap` object we can use the `LoadFromResourceName` method, if the resource has a string identifier, or the `LoadFromResourceID` method, if it has a numeric identifier. The first parameter of both methods is a handle to the application, known as `HInstance`, available in Delphi as a global variable.

note Delphi defines a second global variable, `MainInstance`, which refers to the `HInstance` of the main executable file. Unless you are inside a DLL, you can use one or the other interchangeably.

This is the code of the `FormCreate` method:

```
procedure TWorldForm.FormCreate(Sender: TObject);
var
  I: Integer;
  Bmp: TBitmap;
begin
  Count := 1;
  // load the bitmaps and add them to the list
  BitmapsList := TList.Create;
  for I := 1 to 16 do
  begin
    Bmp := TBitmap.Create;
    Bmp.LoadFromResourceName (HInstance,
      'W' + IntToStr (I));
    BitmapsList.Add (Bmp);
  end;
end;
```

note As an alternative, we could have used the ImageList component, but for this example I decided to use a low-level approach to show you all the details involved.

One problem remains to be solved: obtaining a smooth transition from one image of the world to the following one. The program should paint the bitmaps in a canvas using the Draw method. Unfortunately, the bitmap button's canvas is not directly available (and not event protected), so I decided to use a `TControlCanvas` (usually the internal canvas of a control, but one you can also associate to externally) To use it to paint over a button, we can assign the button to the control canvas in the `FormCreate` method:

```
  ControlCanvas := TControlCanvas.Create;
  ControlCanvas.Control := WorldButton;
  YPos := (WorldButton.Height - Bmp.Height) div 2;
```

```
  XPos := WorldButton.Margin;
```

The horizontal position of the button where the image is located (and where we should paint) depends on the `Margin` of the icon of the bitmap button and on the height of the bitmap. Once the control canvas is properly set, the `Timer1Timer` method simply paints over it—and over the button:

```
procedure TWorldForm.Timer1Timer(Sender: TObject);
begin
  Count := (Count mod 16) + 1;
  Label1.Caption := Format ('Displaying image %d', [Count]);
  // draw the current bitmap in the control canvas
  ControlCanvas.Draw (XPos, YPos,
    BitmapsList.Items[Count-1] as TBitmap);
end;
```

The last problem is to move the position of the image when the left mouse button is pressed or released over it (that is, in the `OnMouseDown` and `OnMouseUp` events of the button). Besides moving the image by few pixels, we should update the glyph of the bitmap, because Delphi will automatically display it while redrawing the button. Otherwise, a user would see the initial image until the timer interval elapsed and the component fired the `OnTimer` event. (That might take a while if you've stopped it!) Here is the code of the first of the two methods:

```
procedure TWorldForm.WorldButtonMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    // paint the current image over the button
    WorldButton.Glyph.Assign (
      BitmapsList.Items[Count-1] as TBitmap);
    Inc (YPos, 2);
    Inc (XPos, 2);
  end;
end;
```

The Animate Control

There is a better way to obtain animation than displaying a series of bitmaps in sequence. Use the Win32 Animate common control. The Animate control is based on the use of AVI (Audio Video Interleaved) files, a series of bitmaps similar to a movie.

note Actually, the Animate control can display only those AVI files that have a single video stream, are uncompressed or compressed with RLE8 compression, and have no palette changes; and if they have sound, it is ignored. In practice, the files corresponding to this requirement are those made of a series of computer bitmaps, not those based on an actual film.

The Animate control can have two possible sources for its animation:

- It can be based on any AVI file that meets the requirements indicated in the note above; to use this type of source, set a proper value for the `FileName` property.
- It can use a special internal Windows animation, part of the common control library; to use this type of source, choose one of the possible values of the `CommonAVI` property (which is based on an enumeration).

If you simply place an Animate control on a form, choose an animation using one of the methods just described, and finally, set its `Active` property to `True`, you'll start seeing the animation performed even at design time. By default, the animation runs continuously, restarting it as soon as it is done. However, you can regulate this effect by using the `Repetitions` property. The default value `-1` causes infinite repetition; use any other value to specify a number of repetitions.

You can also specify the initial and final frame of the sequence, with the `StartFrame` and `StopFrame` properties. These three properties (initial position, final position, and number of repetitions) correspond to the three parameters of the `Play` method, which you'll often use with an Animate control. As an alternative, you can set the properties and then call the `Start` method. At run time, you can also access the total number of frames using the `FrameCount` property: you can use this to execute the animation from the beginning to the end. Finally, for finer control, you can use the `Seek` method, which displays a specific frame.

I've used all of these methods in a simple demo program, which can use both files and the Windows standard animations. The program allows you to choose a file or one of the animations by using a ListBox. I've added an item to this ListBox for each element of the `TCommonAVI` enumeration and used the same order:

```
object ListBox1: TListBox
  Items.Strings = (
    '[Use an AVI file]'
    'Find Folder'
    'Find File'
    'Find Computer'
    'Copy Files'
    'Copy File'
    'Recycle File'
    'Empty Recycle'
    'Delete File')
```

```
  OnClick = ListBox1Click
end
```

Thanks to this structure, when the user clicks on the ListBox, simply casting the number of the selected items to the enumerated data type will get the proper value for the `CommonAVI` property.

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  Animate1.CommonAVI := TCommonAVI (ListBox1.ItemIndex);
  if (ListBox1.ItemIndex = 0) and
      OpenDialog1.Execute then
    Animate1.FileName := OpenDialog1.FileName
end;
```

As you can see, when the first item is selected (the value is `caNone`), the program automatically loads an AVI file, using an OpenDialog component. The most important component of the form is the Animate control. Here is its textual description:

```
object Animate1: TAnimate
  AutoSize = False
  Align = alClient
  CommonAVI = aviFindFolder
  OnOpen = Animate1Open
end
```

It's aligned to the client area, so that a user can easily resize it depending on the actual size of the frames of the animation. As you can see, I've also defined a handler for an event of this component, `OnOpen`:

```
procedure TForm1.Animate1Open(Sender: TObject);
begin
  LblFrames.Caption := 'Frames ' +
    IntToStr (Animate1.FrameCount);
end;
```

When a new file (or common animation) is opened, the program simply outputs the number of its frames in a label. This label is hosted together with several buttons and a few SpinEdit controls into a big panel, acting as a toolbar. You can see them in the design-time form of Figure 22.12.

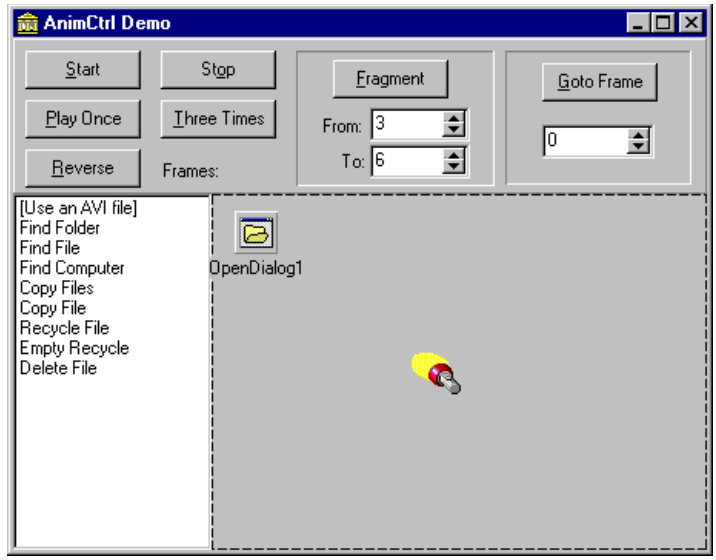


Figure 22.12: The form of the AnimCtrl example at design time. The Animate control is actually showing an animation, even before running the program. Image from the original book.

The Start and Stop buttons are completely trivial, but the Play Once button has some simple code:

```
procedure TForm1.BtnOnceClick(Sender: TObject);
begin
  Animate1.Play (0, Animate1.FrameCount, 1);
end;
```

Things start getting more interesting with the code used to play the animation three times or to play only a fragment of it. Both of these methods are based on the `Play` method:

```
procedure TForm1.BtnTriceClick(Sender: TObject);
begin
  Animate1.Play (0, Animate1.FrameCount, 3);
end;

procedure TForm1.BtnFragmentClick(Sender: TObject);
begin
  Animate1.Play (SpinEdit1.Value, SpinEdit2.Value, -1);
end;
```

The last two button event handlers are based on the `Seek` method. The Goto button simply moves to the frame indicated by the corresponding SpinEdit component, while the Reverse buttons move to each frame in turn, starting with the last one and pausing between each of them:

```
procedure TForm1.BtnGotoClick(Sender: TObject);
begin
  Animate1.Seek (SpinEdit3.Value);
end;

procedure TForm1.BtnReverseClick(Sender: TObject);
var
  Init: TDateTime;
  I: Integer;
begin
  for I := Animate1.FrameCount downto 1 do
  begin
    Animate1.Seek (I);
    // wait 50 milliseconds
    Init := Now;
    while Now < Init + EncodeTime (0, 0, 0, 50) do
      Application.ProcessMessages;
  end;
end;
```

The Animate Control in a Button

Now that you know how the Animate control works, we can use it to build another animated button. Simply place an Animate control and a large button (possibly with a large font as well) in a form. Then write the following code to make the button the parent window of the Animate control at run time and position it properly:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  hDiff: Integer;
begin
  Animate1.Parent := Button1;
  hDiff := Button1.Height - Animate1.Height;
  Animate1.SetBounds (hDiff div 2, hDiff div 2,
    Animate1.Width, Animate1.Height);
  Animate1.Active := True;
end;
```

You can see an example of this effect in Figure 22.13. (The project has the name AnimBtn.) This is indeed the simplest approach to producing an animated button, but it also permits the least control.

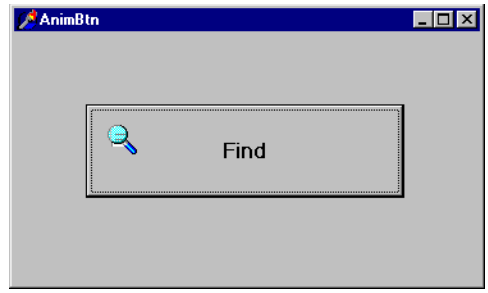


Figure 22.13: The effect of the Animate control inside a button, as shown by the AnimBtn program. Image from the original book.

Graphical Grids

Grids represent another interesting group of Delphi graphical components. The system offers different grid components: a grid of strings, one of images, database-related grids, and a sample grid of colors. The first two kinds of grids are particularly useful because they allow you to represent a lot of information and let the user navigate it. Of course, grids are extremely important in database programming, and they can be customized with graphics as we've seen in Chapter 10 of *Mastering Delphi 5*.

The DrawGrid and StringGrid components are closely related. In fact, the `TStringGrid` class is a subclass of `TDrawGrid`. What use are these grids? Basically, you can store some values, either in the strings related to the StringGrid or in other data structures, and then display selected values, using specific criteria. While grids of strings can be used almost as they are (because they already provide editing capabilities), the grids of generic objects usually require more coding.

Grids, in fact, define the way information is organized for display, not how it is stored. The only grid that stores the data it displays is the StringGrid. All other grids (including the DrawGrid and the DBGrid components) are just data viewers, not data containers. The DBGrid doesn't own the data it displays; it fetches the data from the connected data source. This is sometimes a source of confusion.

The basic structure of a grid includes a number of fixed columns and rows, which indicate the nonscrollable region of the grid (as you can see in Figure 22.14). Grids are among the most complex components available in Delphi, as indicated by the high number of properties and methods they contain. There are a great many

options and properties for grids, controlling both their appearance and their behavior.

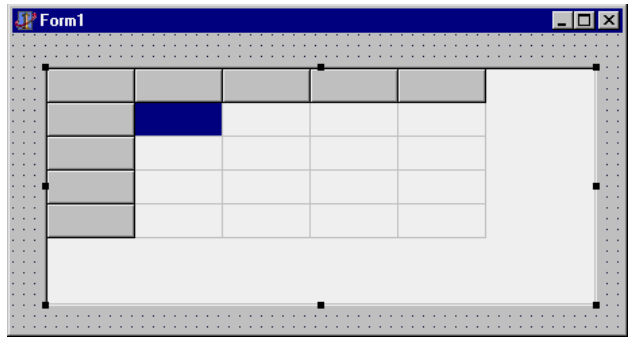


Figure 22.14: When you place a new grid component on a form, it contains one fixed row and one fixed column by default. Image from the original book.

In its appearance, the grid can have lines of different sizes, or it can have no lines. You can set the size of each column or row independently of the others because the `RowSize`, `ColWidth`, and `RowHeight` properties are arrays. For the grid's behavior, you can let the user resize the columns and the rows (`goColSizing` and `goRowSizing`), drag entire columns and rows to a new position (`goRowMoving` and `goColumnMoving`), select automatic editing, and allow range selections. Because various options allow users to perform a number of operations on grids, there are also a number of events related to grids, such as `OnColumnMoved`, `OnDrawCell`, or `OnSetEditText`.

The most important event is probably `OnDrawCell`. In response to this event, a program has to paint a certain cell of the grid. Only string grids can automatically display their contents. The DrawGrid, in fact, doesn't have support for storing data. It is simply a tool for arranging a portion of the screen to display information in a regular format. It is a simple tool but also a powerful one. Methods such `CellRect`, which returns the rectangle corresponding to the area of a cell, or `MouseToCell`, which returns the cell in a specific location, are a joy to use. By handling resizable rows and columns and scrollable grids, they simplify complex tasks and spare the programmer from tedious calculations.

What can you use a grid for? Building a spreadsheet is probably the first idea that comes to mind, but that's probably a little too complex for an example. I've decided to use the StringGrid control in a program that shows the fonts installed in the system and the DrawGrid control in a program that emulates the MineSweeper game.

A Grid of Fonts

If you place a StringGrid component on a form and set its options properly, you have a full working editor of strings arranged in a grid, without doing any programming at all. To make the example more interesting, I've decided to draw each cell of the grid with a different font, varying both its size and its typeface. You can see the result of the FontGrid program in Figure 22.15.

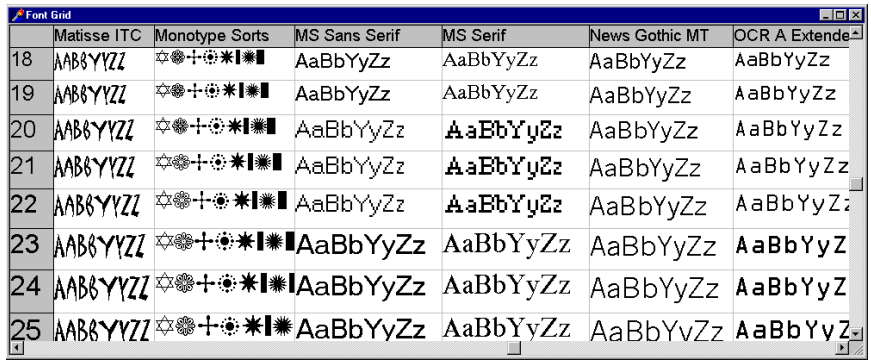


Figure 22.15: An example of the output of the FontGrid application. Image from the original book.

The form of this program is very simple. You need only place a grid component on a form, align it with the client area, set a few properties and options, and let the program do the rest. The number of columns and rows and their size, in fact, are computed at run time. The important properties you need to set are `DefaultDrawing`, which should be `False` to let us paint the grid as we like, and `Options`:

```
object Form1: TForm1
  Caption = 'Font Grid'
  OnCreate = FormCreate
  object StringGrid1: TStringGrid
    Align = alClient
    DefaultColWidth = 200
    DefaultDrawing = False
    Options = [goFixedVertLine, goFixedHorzLine,
      goVertLine, goHorzLine, goDrawFocusSelected,
      goColSizing, goColMoving, goEditing]
    OnDrawCell = StringGrid1DrawCell
  end
end
```

As usually happens in Delphi, the simpler the form is, the more complex the code. This example follows that rule, although it has only two methods, one to initialize

the grid at start-up and the other to draw the items. The editing, in fact, has not been customized and takes place using the system font. The first of the two methods is `FormCreate`. At the beginning, this method uses the global `Screen` object to access the fonts installed in the system.

The grid has a column for each font as well as a fixed column with numbers representing font sizes. The name of each column is copied from the `Screen` object to the first row of each column (which has a zero index):

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I, J: Integer;
begin
  {the number of columns equals the number of fonts plus
  1 for the first fixed column, which has a size of 20}
  StringGrid1.ColCount := Screen.Fonts.Count + 1;
  StringGrid1.ColWidths [0] := 50;

  for I := 1 to Screen.Fonts.Count do
  begin
    // write the name of the font in the first row
    StringGrid1.Cells [I, 0] :=
      Screen.Fonts.Strings [I-1];

    {compute maximum required size of column, getting the width
    of the text with the biggest size of the font in that column}
    StringGrid1.Canvas.Font.Name :=
      StringGrid1.Cells [I, 0];
    StringGrid1.Canvas.Font.Size := 32;
    StringGrid1.ColWidths [I] :=
      StringGrid1.Canvas.TextWidth ('AaBbYyZz');
  end;
  ...
```

In the last part of the code above, the program computes the width of each column. This is accomplished by evaluating the space occupied by the custom string of text *AaBbYyZz,* using the font of the column (written in the first row, `Cells [I, 0]`) and the biggest font size used by the program (`32`). To compute the space required by the text, you can apply the `TextWidth` and `TextHeight` methods to a canvas with the proper font selected.

The rows, instead, are always `26` and have an increasing height, computed with the approximate formula: `15 + I x 2`. In fact, computing the highest text means checking the height of the text in each column, certainly too complex an operation for this example. The approximate formula works well enough, as you can see in Figure 22.15 and by running the program. In the first cell of each row, the program writes the size of the font, which corresponds to the number of the line plus seven.

The last operation is to store the string "AaBbYyZz" in each nonfixed cell of the grid. To accomplish this, the program uses a nested `for` loop. Expect to use nested `for` loops often when working with grids. Here is the second part of the `FormCreate` method:

```
  // defines the number of columns
  StringGrid1.RowCount := 26;
  for I := 1 to 25 do
  begin
    // write the number in the first column
    StringGrid1.Cells [0, I] := IntToStr (I+7);
    // set an increasing height for the rows
    StringGrid1.RowHeights [I] := 15 + I*2;
    // insert default text in each column
    for J := 1 to StringGrid1.ColCount do
      StringGrid1.Cells [J, I] := 'AaBbYyZz'
  end;
  StringGrid1.RowHeights [0] := 25;
end;
```

Now we can study the second method, `StringGrid1DrawCell`, which corresponds to the grid's `OnDrawCell` event. This method has a number of parameters:

- `Col` and `Row` refer to the cell we are currently painting.
- `Rect` is the area of the cell we are going to paint.
- `State` is the state of the cell, a set of three flags, which can be active at the same time: `gdSelected` (the cell is selected), `gdFocused` (the cell has the input focus), and `gdFixed` (the cell is in the fixed area, which usually has a different background color). It is important to know the state of the cell because this usually affects its output.

The `DrawCell` method paints the text of the corresponding element of the grid, with the font used by the column and the size used for the row. Here is the listing of this method:

```
procedure TForm1.StringGrid1DrawCell (Sender: TObject;
  Col, Row: Integer; Rect: TRect; State: TGridDrawState);
begin
  // select a font, depending on the column
  if (Col = 0) or (Row = 0) then
    StringGrid1.Canvas.Font.Name := I
  else
    StringGrid1.Canvas.Font.Name :=
      StringGrid1.Cells [Col, 0];

  // select the size of the font, depending on the row
  if Row = 0 then
    StringGrid1.Canvas.Font.Size := 14
```

```
  else
    StringGrid1.Canvas.Font.Size := Row + 7;

  // select the background color
  if gdSelected in State then
    StringGrid1.Canvas.Brush.Color := clHighlight
  else if gdFixed in State then
    StringGrid1.Canvas.Brush.Color := clBtnFace
  else
    StringGrid1.Canvas.Brush.Color := clWindow;

  // output the text
  StringGrid1.Canvas.TextRect (
    Rect, Rect.Left, Rect.Top,
    StringGrid1.Cells [Col, Row]);

  // draw the focus
  if gdFocused in State then
    StringGrid1.Canvas.DrawFocusRect (Rect);
end;
```

The font's name is retrieved by the row 0 of the same column. The font's size is computed by adding 7 to the number of the row. The fixed columns use some default values. Having set the font and its size, the program selects a color for the background of the cell, depending on its possible states: selected, fixed, or normal (that is, no special style). The value of the style's `gdFocused` flag is used a few lines later to draw the typical focus rectangle. When everything is set up, the program can perform some real output, drawing the text and if necessary the focus rectangle, with the last two statements of the `StringGrid1DrawCell` method above.

note To draw the text in the grid's cell, I've used the `TextRect` method of the canvas instead of the more common `TextOut` method. The reason is that `TextRect` clips the output to the given rectangle, preventing drawing outside this area. This is particularly important in the case of grids because the output of a cell should not cross its borders. Since we are painting on the canvas of the whole grid, when we are drawing a cell, we can end up corrupting the contents of neighboring cells, too.

As a final observation, remember that when you decide to draw the contents of a grid's cell, you should not only draw the default image but also provide a different output for the selected item, properly draw the focus, and so on.

Mines in a Grid

The StringGrid component uses the `Cells` array to store the values of the elements and also has an `Objects` property to store custom data for each cell. The DrawGrid

component, instead, doesn't have a predefined storage. For this reason, the next example defines a two-dimensional array to store the value of the grid's cells—that is, of the playing field.

The Mines example is a clone of the MineSweeper game included with Windows. If you have never played this game, I suggest you try it and read its rules in the Help file since I'll give only a basic description. When the program starts, it displays an empty field (a grid) in which there are some hidden mines. By clicking the left mouse button on a cell, you test whether or not there is a mine in that position. If you find a mine, it explodes, and the game is over. You have lost.

If there is no mine in the cell, the program indicates the number of mines in the eight cells surrounding it. Knowing the number of mines near the cell, you have a good hint for the following turn. To help you further on, when a cell has zero mines in the surrounding area, the number of mines for these cells is automatically displayed, and if one of them has zero surrounding mines, the process is repeated. So if you are lucky, with a single click you might uncover a good number of clear cells (see Figure 22.16).

When you think you have found a mine, simply right-click on the cell; this places a flag there. The program does not say whether your inference is correct; the flag is only a hint for your future attempts. If you later change your mind, you can again right-click on the cell to remove the flag. When you have found all of the mines, you have won, and the game terminates.

Those are the rules of the game. Now we have to implement them, using a DrawGrid as starting point. In this example, the grid is fixed and cannot be resized or modified in any way at run time. In fact, it has square cells of 30 × 30 pixels, which will be used to display bitmaps of the same size.

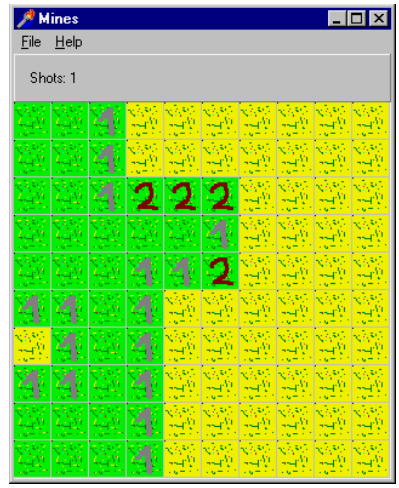


Figure 22.16: The Mines program after a single lucky click. A group of cells with no mines is displayed at once. Image from the original book.

The code of this program is complex, and it is not easy to find a starting point to describe it. For this reason, I've added more comments than usual to the source code (in the download files) so you can browse through it to understand what it does. Nonetheless, I'll describe its most important elements. First of all, the program's data is stored in two arrays (declared as `private` fields of the form):

```
Display: array [0 .. NItems - 1, 0 .. NItems -1] of Boolean;
Map: array [0 .. NItems - 1, 0 .. NItems -1] of Char;
```

The first is an array of `Boolean` values that indicate whether an item should be displayed or remain hidden. Notice that the number of rows and columns of this array is `NItems`. You can freely change this constant, but you should resize the grid accordingly. The second array, `Map`, holds the positions of the mines and flags and the numbers of the surrounding mines. It uses character codes instead of a proper enumeration data type, in order to use the digits `0–8` to indicate the number of mines around the cell. Here is a list of the codes:

- *M: Mine* indicates the position of a mine that the user still has not found.
- *K: Known mine* indicates the position of a mine already found by the user and having a flag.
- *W: Wrong mine* indicates a position where the user has set a flag but where there is no mine.
- *0 to 8: Number of mines* indicates the number of mines in the surrounding cells.

The first method to explore is `FormCreate`, executed at start-up. This method initializes a number of fields of the form class, fills the two arrays with default values (using two nested `for` loops), and then sets the mines in the grid. For the number of times defined in a constant (that is, the number of mines), the program adds a new mine in a random position. However, if there was already a mine, the loop should be executed once more because the final number of mines in the `Map` array should equal the requested number. Otherwise the program will never terminate, because it tests when the number of mines found equals the number of mines added to the grid. Here is the code of the loop; it can be executed more than `NMines` times, thanks to the use of the `MinesToPlace` integer variable, which is increased when we try to place a mine over an existing one:

```
Randomize;
// place 'NMines' non-overlapping mines
MinesToPlace := NMines;
while MinesToPlace > 0 do
begin
  X := Random (NItems);
  Y := Random (NItems);
  // if there isn't a mine
  if Map [X, Y] <> 'M' then
  begin
    // add a mine
    Map [X, Y] := 'M';
    Dec (MinesToPlace)
  end;
end;
```

The last portion of the initialization code computes the number of surrounding mines for each cell that doesn't have a mine. This is accomplished by calling the `ComputeMines` procedure for each cell. The code of this function is fairly complex because it has to consider the special cases of the mines near a border of the grid. The effect of this call is to store, in the `Map` array, the character representing the number of mines surrounding each cell.

The next logical procedure is `DrawGrid1MouseDown`. This method first computes the cell on which the mouse has been clicked, with a call to the grid's `MouseToCell` method. Then there are three alternative portions of code: a small one when the game has ended, and the other two for the two mouse buttons. When the left mouse button is pressed, the program checks whether there is a mine (hidden or not), and if there is, it displays a message and terminates the program with an explosion (see Figure 22.17).

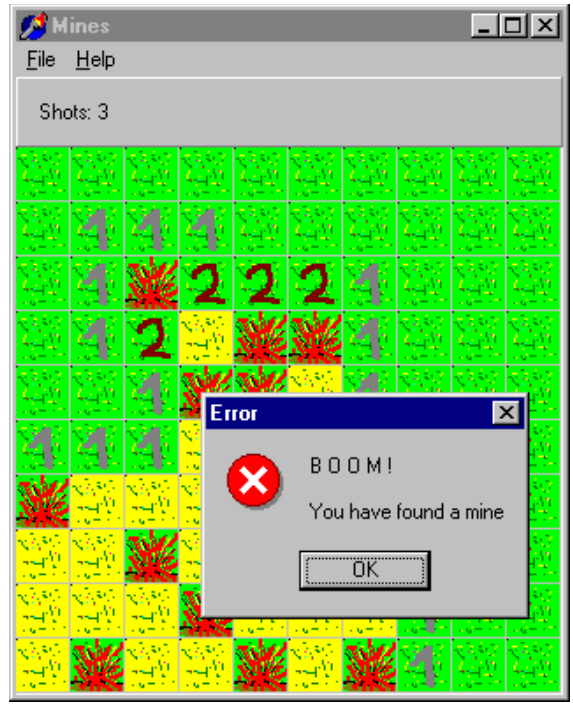


Figure 22.17: Ouch! You have stepped on a mine. Image from the original book.

If there is no mine, the program sets the `Display` value for the cell to `True`, and if there is a `0`, it starts the `FloodZeros` procedure. This method displays the eight items near a visible cell having a value of 0, repeating the operation over and over if one of the surrounding cells also has a value of 0. This recursive call is complex because you have to provide a way to terminate it. If there are two cells near each other, both having a value of 0, each one is in the surrounding area of the other, so they might continue forever to ask the other cell to display itself and its surrounding cells. Again, the code is complex, and the best way to study it may be to step through it in the debugger.

When the user presses the right mouse button, the program changes the status of the cell. The right mouse button action is to toggle the flag on the screen, so a user can always remove an existing flag, if he or she thinks the earlier decision was wrong. For this reason the status of a cell that contains a mine can change from *M* (hidden Mine) to *K* (Known mine) and vice versa; and the status of a cell with no mine can change from a number to *W* (Wrong mine) and vice versa. When all the mines have been found, the program terminates with a congratulation message.

A very important piece of code is at the end of the `OnMouseDown` event response method. Each time the user clicks on a cell and its contents change, that cell should be repainted. If you repaint the whole grid, the program will be slower. For this reason, I've used the Windows API function `InvalidateRect`:

```
MyRect := DrawGrid1.CellRect (Col, Row);
InvalidateRect (DrawGrid1.Handle, @MyRect, False);
```

The last important method is `DrawGrid1DrawCell`. We already used this painting procedure in the last example, so you should remember that it is called for each cell that needs repainting. Fundamentally, this method extracts the code corresponding to the cell, which shows a corresponding bitmap, loaded from a file. Once again, I've prepared a bitmap for each of the images in a new resource file, which is included in the project thanks to Delphi 5's improved Project Manager.

Recall that when using resources, the code tends to be faster than when using separate files, and again, we end up with a single executable file to distribute. The bitmaps have names corresponding to the code in the grid, with a character ('M') in front since the name '0' would have been invalid. The bitmaps can be loaded and drawn in the cell with this code:

```
Bmp.LoadFromResourceName (HInstance, 'M' + Code);
DrawGrid1.Canvas.Draw (Rect.Left, Rect.Top, Bmp);
```

Of course, this takes place only if the cell is visible—that is, if `Display` is `True`. Otherwise, a default undefined bitmap is displayed. (The bitmap name is *'UNDEF'*.) Loading the bitmaps from the resources each time seems slow, so the program could have stored all the bitmaps in a list in memory, as the World2 example earlier in this chapter did. However, this time, I decided to use a different, although slightly less efficient, approach: a cache. This makes sense because we already use resources instead of files to speed up things.

The bitmap cache of Mines is small since it has just one element, but its presence speeds up the program considerably. The program stores the last bitmap it has used and its code; then, each time it has to draw a new item, if the code is the same, it uses the cached bitmap. Here is the new version of the code above:

```
if not (Code = LastBmp) then
begin
  Bmp.LoadFromResourceName (HInstance, 'M' + Code);
  LastBmp := Code;
end;
DrawGrid1.Canvas.Draw (Rect.Left, Rect.Top, Bmp);
```

Increasing the size of this cache will certainly improve its speed. You can consider a list of bitmaps as a big cache, but this is probably useless because some bitmaps

(those with high numbers) are seldom used. As you can see, some improvements can be made to speed up the program, and much can also be done to improve its user interface. If you have understood this version of the program, I think you'll be able to improve it considerably.

Using TeeChart

TeeChart is a VCL-based charting component built by David Berneda and licensed to Borland for inclusion in the Developer and Client/Server versions of Delphi[474]. The TeeChart component is very complex: Delphi includes a Help file and other reference material for this component, so I won't spend time listing all of its features. I'll just build a couple of examples. TeeChart comes in three versions: the standalone component (in the Additional page of the Component Palette), the data-aware version (in the Data Controls page), and the Report version (in the QuickReport page). Delphi Client/Server also includes a DecisionChart control in the Decision Cube page of the palette. The data-aware version of TeeChart is presented in Chapter 9 of *Mastering Delphi 5,* and I'll use it again later in a Web-oriented example.

note Of course, it would be simpler to build an example using the TeeChart Wizard, but seeing all the steps will give you a better understanding of this component's structure.

The TeeChart component provides the basic structure for charting, through a complex framework of charting and series classes and the visual container for charts (the actual control). The actual charts are objects of class `TChartSeries` or derived classes. Once you've placed the TeeChart component on a form, you should create one or more series. To accomplish this, you can open the Chart Component Editor: select the component, right-click to show the local menu of the form designer, and choose the Edit Chart command. Now press the Add button, and choose the graph (or series) you want to add from the many available (as you can see in Figure 22.18).

474 The TeeChart component is still available and installed in Delphi today, as an optional feature.

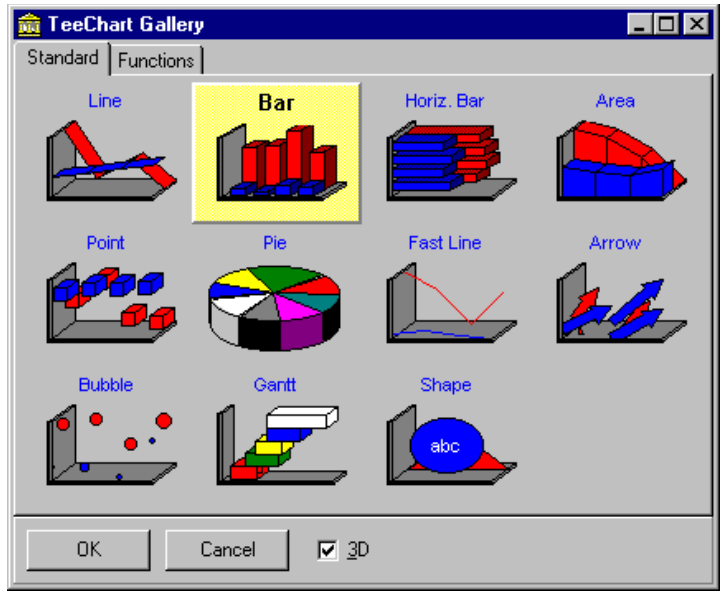


Figure 22.18: The TeeChart Gallery allows you to choose the type of graph, or series. Image from the original book.

As soon as you create a new series, a new object of a `TChartSeries` subclass is added to your form. This is the same behavior as the MainMenu component, which adds objects of the `TMenuItem` class to the form. You can then edit the properties of the `TSeries` object in the Chart Component Editor, or you can select the `TChartSeries` object in the Object Inspector (with the Object Selector combo box) and edit its many properties.

The different `TChartSeries` subclasses—that is, the different kinds of graph—have different properties and methods (although some of them are common to more than one subclass). Keep in mind that a graph can have multiple series: if they are all of the same type they will probably integrate better, as in the case of multiple bars. Anyway, you can also have a complex layout with graphs of different types visible at the same time. At times, this is an extremely powerful option.

Building a First Example

To build this example I placed a TeeChart component in a form and then simply added four 3D Bar series—that is, four objects of the `TBarSeries` class. Then I set up some global properties, such as the title of the chart, and so on. Here is a summary of this information, taken from the textual description of the form:

```
object Chart1: TChart
  AnimatedZoom = True
  Title.Text.Strings = (
    'Simple TeeChart Demo for Mastering Delphi')
  BevelOuter = bvLowered
  object Series1: TBarSeries
    SeriesColor = clRed
    Marks.Visible = False
  end
  object Series2: TBarSeries
    SeriesColor = clGreen
    Marks.Visible = False
  end
  object Series3: TBarSeries
    SeriesColor = clYellow
    Marks.Visible = False
  end
  object Series4: TBarSeries
    SeriesColor = clBlue
    Marks.Visible = False
  end
end
```

Next I added to the form a string grid and a push button labeled *Update*. This button is used to copy the numeric values of the string grid to the chart. The grid is based on a 5×4 matrix as well as a line and a column for the titles. Here is its textual description:

```
object StringGrid1: TStringGrid
  ColCount = 6
  DefaultColWidth = 50
  Options = [goFixedVertLine, goFixedHorzLine,
    goVertLine, goHorzLine, goEditing]
  ScrollBars = ssNone
  OnGetEditMask = StringGrid1GetEditMask
end
```

The value `5` for the `RowCount` property is a default, and it doesn't show up in the textual description. (The same holds for the value of `1` for the `FixedCols` and `FixedRows` properties.) An important element of this string grid is the edit mask used by all of its cells. This is set using the `OnGetEditMask` event:

```
procedure TForm1.StringGrid1GetEditMask(Sender: TObject;
  ACol, ARow: Longint; var Value: string);
begin
  // edit mask for the grid cells
  Value := '09;0';
end;
```

There is actually one more component, a check box used to toggle the visibility of the marks of the series. (The *marks* are small yellow tags describing each value; you'll need to run the program to see them.) You can see the form at design time in Figure 22.19. In this case the series are populated with random values; this is a nice feature of the component, as it allows you to preview the output without entering real data.

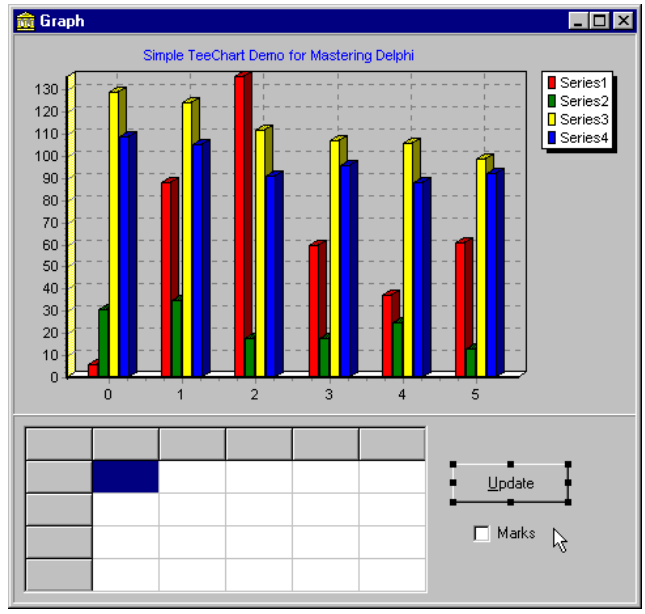


Figure 22.19: The Graph1 example, based on the TeeChart component, at design time. Image from the original book.

Adding Data to the Chart

Now we simply initialize the data of the string grid and copy it to the series of the chart. This takes place in the handler of the `OnCreate` event of the form. This method fills the fixed items of the grid and the series names, then fills the data portion of the string grid, and finally calls the handler of the `OnClick` event of the *Update* button, to update the chart:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I, J: Integer;
begin
  with StringGrid1 do
```

```
  begin
    {fills the fixed column and row,
    and the chart series names}
    for I := 1 to 5 do
      Cells [I, 0] := Format ('Group%d', [I]);
    for J := 1 to 4 do
    begin
      Cells [0, J] := Format ('Series%d', [J]);
      Chart1.Series [J-1].Title := Format ('Series%d', [J]);
    end;

    // fills the grid with random values
    Randomize;
    for I := 1 to 5 do
      for J := 1 to 4 do
        Cells [I, J] := IntToStr (Random (100));
  end; // with

  // update the chart
  UpdateButtonClick (Self);
end;
```

We can access the series using the component name (as `Series1`) or using the `Series` array property of the chart, as in `Chart1.Series[J-1]`. In this expression, notice that the actual data in the string grid starts at row and column one—the first line and column, indicated by the zero index, are used for the fixed elements—while the chart `Series` array is zero-based.

Another example of updating each series is present in the `OnClick` event handler for the check box; this method toggles the visibility of the marks:

```
procedure TForm1.ChBoxMarksClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 4 do
    Chart1.Series [I-1].Marks.Visible :=
      ChBoxMarks.Checked;
end;
```

But the really interesting code is in the `UpdateButtonClick` method, which updates the chart. To accomplish this, the program first removes the existing data of each chart, and then it adds new data (or *data points,* to use a jargon term):

```
procedure TForm1.UpdateButtonClick(Sender: TObject);
var
  I, J: Integer;
begin
  for I := 1 to 4 do
  begin
    Chart1.Series [I-1].Clear;
```

```
    for J := 1 to 5 do
      Chart1.Series [I-1].Add (
        StrToInt (StringGrid1.Cells [J, I]),
        '', Chart1.Series [I-1].SeriesColor);
  end;
end;
```

The parameters of the `Add` method (used when you don't want to specify an `X` value, but only an `Y` value) are the actual value, the label, and the color. In this example the label is not used, so I've simply omitted it. I could have used the default value, `clTeeColor`, to get the proper color of the series. You might use specific colors to indicate different ranges of data.

Once you've built the graph, TeeChart allows you a lot of viewing options. You can easily zoom into the view (simply indicate the area with the left mouse button), zoom out (using the mouse in the opposite way, dragging towards the top left corner), and use the right mouse button to pan the view. You can see an example of zooming in Figure 22.20.

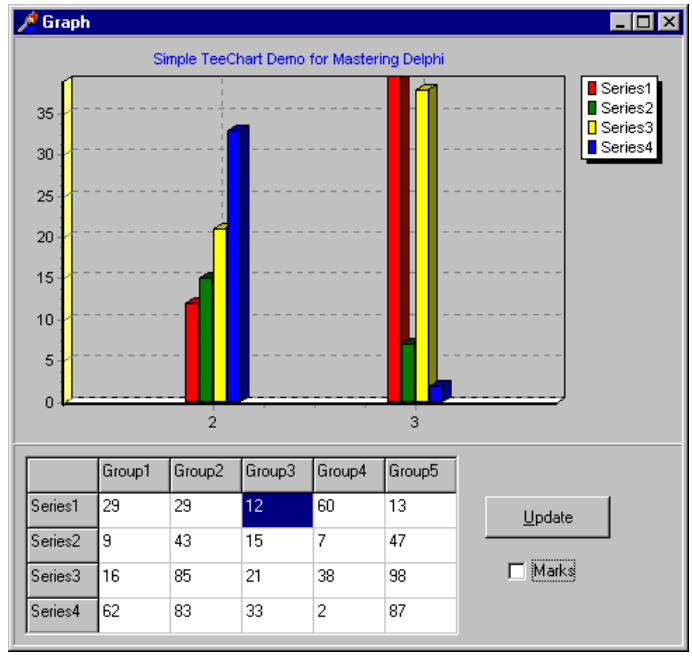


Figure 22.20: The form of the Graph1 example at run time. Notice that I've zoomed into the graph. Image from the original book.

Creating Series Dynamically

The Graph1 example shows some of the capabilities of the TeeChart component, but it is based on a single, fixed type of graph. I could have improved it by allowing some customization of the shape of the vertical bars; instead I chose a more general approach, allowing the user to choose different kinds of series (graphs).

The TeeChart component initially has the same attributes as in the previous example. But the form now has four combo boxes, one for each row of the string grid. Each combo box has four values (Line, Bar, Area, and Point), corresponding to the four types of series I want to handle. To handle these combo boxes in a more flexible way in the code, I've added an array of these controls to the private fields of the form:

```
private
  Combos: array [0..3] of TComboBox;
```

This array is filled with the actual component in the `FormCreate` method, which also selects the initial item of each of them. Here is the new code of `FormCreate`:

```
  // fill the Combos array
  Combos [0] := ComboBox1;
  Combos [1] := ComboBox2;
  Combos [2] := ComboBox3;
  Combos [3] := ComboBox4;
  // show the initial chart type
  for I := 0 to 3 do
    Combos [I].ItemIndex := 1;
```

note This example demonstrates a common way to create an array of controls in Delphi, something Visual Basic programmers often long for. Actually Delphi is so flexible that arrays of controls are not built-in; you can create them as you like. This approach relies on the fact that you can generally associate the same event handler with different events, something that VB doesn't allow you to do.

All these combo boxes share the same `OnClick` event handler, which destroys each of the current series of the chart, creates the new ones as requested, and then updates their properties and data:

```
procedure TForm1.ComboChange(Sender: TObject);
var
  I: Integer;
  SeriesClass: TChartSeriesClass;
  NewSeries: TChartSeries;
begin
  // destroy the existing series (in reverse order)
```

```
  for I := 3 downto 0 do
    Chart1.Series [I].Free;
  // create the new series
  for I := 0 to 3 do
  begin
    case Combos [I].ItemIndex of
      0: SeriesClass := TLineSeries;
      1: SeriesClass := TBarSeries;
      2: SeriesClass := TAreaSeries;
    else // 3: and default
      SeriesClass := TPointSeries;
    end;
    NewSeries := SeriesClass.Create (self);
    NewSeries.ParentChart := Chart1;
    NewSeries.Title :=
      Format ('Series %d', [I + 1]);
  end;
  // update the marks and update the data
  ChBoxMarksClick (self);
  UpdateButtonClick (self);
  Modified := True;
end;
```

The central part of this code is the `case` statement, which stores a new class in the `SeriesClass` class reference variable, used to create the new series objects and set each one's `ParentChart` and `Title`. I could have also used a call to the `AddSeries` method of the chart in each case branch and then set the `Title` with another `for` loop. In fact, a call such as

```
Chart1.AddSeries (TBarSeries.Create (self));
```

creates the series objects and sets its parent chart at the same time.

Notice that this new version of the program allows you to change the type of graph for each series independently. You can see an example of the resulting effect in Figure 22.21.

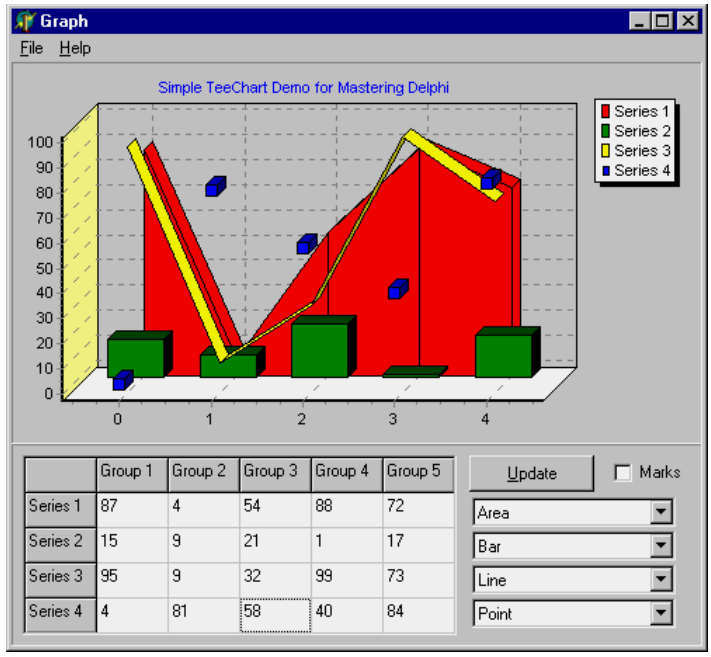


Figure 22.21: Various kinds of graphs, or chart series, displayed by the Graph2 example. Image from the original book.

Finally, the Graph2 example has support for saving the current data it is displaying on a file and loads existing files. The program has a `Modified` Boolean variable, used to track whether the user has changed any of the data, and it prompts the user to confirm closing the form when the data has changed. The file support is based on streams and is not particularly complex, because the number of elements to save is fixed (all the files have the same size). Here are the two methods connected with the Open and Save menu items:

```
procedure TForm1.Open1Click(Sender: TObject);
var
  LoadStream: TFileStream;
  I, J, Value: Integer;
begin
  if OpenDialog1.Execute thenbegin
    CurrentFile := OpenDialog1.Filename;
    Caption := 'Graph [' + CurrentFile + ']';
    // load from the current file
    LoadStream := TFileStream.Create (
      CurrentFile, fmOpenRead);
    try
      // read the value of each grid element
      for I := 1 to 5 do
        for J := 1 to 4 do
```

```
        begin
          LoadStream.Read (Value, sizeof (Integer));
          StringGrid1.Cells [I, J] := IntToStr(Value);
        end;
      // load the status of the checkbox and the combo boxes
      LoadStream.Read (Value, sizeof (Integer));
      ChBoxMarks.Checked := Boolean(Value);
      for I := 0 to 3 do
      begin
        LoadStream.Read (Value, sizeof (Integer));
        Combos [I].ItemIndex := Value;
      end;
    finally
      LoadStream.Free;
    end;
    // fire udpate events
    ChBoxMarksClick (Self);
    ComboChange (Self);
    UpdateButtonClick (Self);
    Modified := False;
  end;
end;

procedure TForm1.Save1Click(Sender: TObject);
var
  SaveStream: TFileStream;
  I, J, Value: Integer;
begin
  if Modified then
    if CurrentFile = '' then // call save as
      SaveAs1Click (Self)
    else
    begin
      // save to the current file
      SaveStream := TFileStream.Create (
        CurrentFile, fmOpenWrite or fmCreate);
      try
        // write the value of each grid element
        for I := 1 to 5 do
          for J := 1 to 4 do
          begin
            Value := StrToIntDef (Trim (
              StringGrid1.Cells [I, J]), 0);
            SaveStream.Write (Value, sizeof (Integer));
          end;
        // save check box and combo boxes
        Value := Integer (ChBoxMarks.Checked);
        SaveStream.Write (Value, sizeof (Integer));
        for I := 0 to 3 do
        begin
          Value := Combos [I].ItemIndex;
          SaveStream.Write (Value, sizeof (Integer));
        end;
```

```
        Modified := False;
      finally
        SaveStream.Free;
      end;
    end;
end;
```

A Database Chart on the Web

In Chapter 20 of *Mastering Delphi 5,* we saw how to create a simple graphic image and return it from a CGI application. We can apply the same approach in returning a complex and dynamic graph built with the `TDBChart` component. Using this component in memory is a little more complex than setting all of its properties at design time, as you'll have to set the properties in the Pascal code. (You cannot use a visual component, such as a DBChart, in a Web Module or any other data module).

In the WebChart ISAPI application I've used the Country.DB table to produce a pie chart with the area and population of the American countries, as in the ChartDb example of Chapter 9 in *Mastering Delphi 5*. The two graphs are generated by two different actions, indicated by the paths `/population` and `/area`. As most of the code is used more than once, I've collected it in the `OnCreate` and `OnAfterDispatch` events of the WebModule.

---

note As written, this program doesn't support concurrent users. You'll need to add some threading or synchronization code to this ISAPI DLL to make it work with multiple users at the same time. An alternative is to place all the code in the Action event handlers, so that no global object is shared among multiple requests. Or you can turn it into a CGI application.

The data module has a table object, which is properly initialized at design time, and three private fields:

```
private
  Chart: TDBChart;
  Series: TPieSeries;
  Image: TImage;
```

The objects corresponding to these fields are created along with the Web module (and used by subsequent calls):

```
procedure TWebModule1.WebModule1Create(Sender: TObject);
begin
  // open the database table
  Table1.Open;
  // create the chart
```

```
  Chart := TDBChart.Create (nil);
  Chart.Width := 600;
  Chart.Height := 400;
  Chart.AxisVisible := False;
  Chart.Legend.Visible := False;
  Chart.BottomAxis.Title.Caption := 'Name';
  // create the pie series
  Series := TPieSeries.Create (Chart);
  Series.ParentChart := Chart;
  Series.DataSource := Table1;
  Series.XLabelsSource := 'Name';
  Series.OtherSlice.Style := poBelowPercent;
  Series.OtherSlice.Text := 'Others';
  Series.OtherSlice.Value := 2;
  Chart.AddSeries (Series);
  // create the memory bitmap
  Image := TImage.Create (nil);
  Image.Width := Chart.Width;
  Image.Height := Chart.Height;
end;
```

The next step is to execute the handler of the specific action, which sets the pie chart series to the specific data field and updates a few captions:

```
procedure TWebModule1.WebModule1ActionPopulationAction(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  // set specific values
  Chart.Title.Text.Clear;
  Chart.Title.Text.Add ('Population of Countries');
  Chart.LeftAxis.Title.Caption := 'Population';
  Series.Title := 'Population';
  Series.PieValues.ValueSource := 'Population';
end;
```

This creates the proper DBChart in memory. The final step, again common to the two actions, is to save the chart in a bitmap image, and then format it as a JPEG on a stream, to be later returned from the server-side application. The code is actually similar to that of the previous example:

```
procedure TWebModule1.WebModule1AfterDispatch(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  Jpeg: TJpegImage;
  MemStr: TMemoryStream;
begin
  // paint the chart on the memory bitmap
  Chart.Draw (Image.Canvas, Image.BoundsRect);
  // create the jpeg and copy the image to it
```

```
  Jpeg := TJpegImage.Create;
  try
    Jpeg.Assign (Image.Picture.Bitmap);
    MemStr := TMemoryStream.Create;
    // save to a stream and return it
    Jpeg.SaveToStream (MemStr);
    MemStr.Position := 0;
    Response.ContentType := 'image/jpeg';
    Response.ContentStream := MemStr;
    Response.SendResponse;
  finally
    Jpeg.Free;
  end;
end;
```

The result, visible in Figure 22.22, is certainly interesting. Optionally, you can extend this application by hooking it to an HTML table showing the database data. Simply write a program with a main action returning the HTML table and a reference to the embedded graphics, which will be returned by a second activation of the ISAPI DLL with a different path.

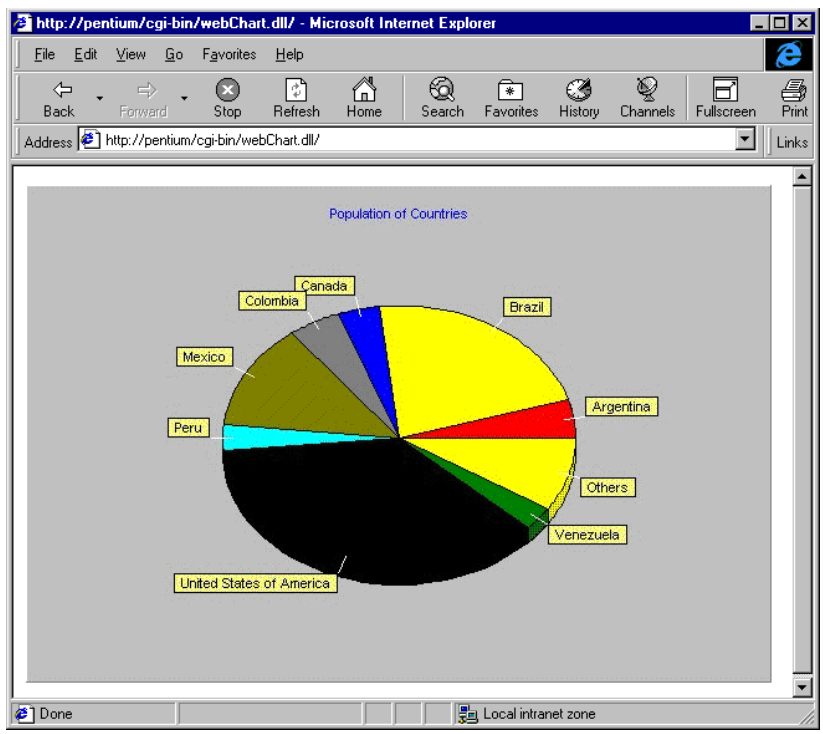


Figure 22.22: The JPEG with the population chart generated by the WebChart application. Image from the original book.

Using Metafiles

The bitmap formats covered earlier in this chapter store the status of each pixel of a bitmap, although they usually compress the information. A totally different type of graphic format is represented by vector-oriented formats. In this case the file stores the information required to re-create the picture, such as the initial and final point of each line or the mathematics that define a curve. There are many different vector-oriented file formats, but the only one supported by the Windows operating system is the Windows Metafile Format (WMF). This format has been extended in Win32 into the Extended Metafile Format (EMF), which stores extra information related to the mapping modes and the coordinate system.

A Windows metafile is basically a series of calls to the GDI primitive functions. After you've stored the sequence of calls, you can *replay* them, reproducing the graphics. Delphi supports Windows metafiles through the `TMetafile` and `TMetaFileCanvas` classes, so it's very simple to build an example.

The `TMetafile` class is used to handle the file itself, with methods for loading and saving the files, and properties determining the key features of the file. One of them is the `Enhanced` property, which determines the type of metafile format. Note that when Windows is reading a file, the `Enhanced` property is set depending on the file extension—WMF for Windows 3.1 metafiles and EMF for the Win32 enhanced metafiles.

To generate a metafile, you can use an object of the `TMetafileCanvas` class, connected to the file through its constructors, as shown by the following code fragment:

```
Wmf := TMetafile.Create;
WmfCanvas := TMetafileCanvas.CreateWithComment(
  Wmf, 0, 'Marco', 'Demo metafile');
```

Once you've created the two objects, you can paint over the canvas object with regular calls and, at the end, save the connected metafile to a physical file.

Once you have the metafile (either a brand-new one you've just created or one you've built with another program) you can show it in an Image component, or you can simply call the `Draw` or `StretchDraw` methods of any canvas.

In the WmfDemo example I've written some simple code, just to show you the basics of this approach. The `OnCreate` event handler of the form creates the enhanced metafile, a single object that is used both for reading and writing operations:

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

```
begin
  Wmf := TMetafile.Create;
  Wmf.Enhanced := True;
  Randomize;
end;
```

The form of the program has two buttons and the PaintBox components, plus a check box. The first button creates a metafile by generating a series of partially random lines. The result is both shown in the first PaintBox and saved to a fixed file:

```
procedure TForm1.BtnCreateClick(Sender: TObject);
var
  WmfCanvas: TMetafileCanvas;
  X, Y: Integer;
begin
  // create the virtual canvas
  WmfCanvas := TMetafileCanvas.CreateWithComment(
    Wmf, 0, 'Marco', 'Demo metafile');

  try
    // clear the background
    WmfCanvas.Brush.Color := clWhite;
    WmfCanvas.FillRect (WmfCanvas.ClipRect);

    // draws 400 lines
    for X := 1 to 20 do
      for Y := 1 to 20 do
      begin
        WmfCanvas.MoveTo (15 * (X + Random (3)),
          15 * (Y + Random (3)));
        WmfCanvas.LineTo (45 * Y, 45 * X);
      end;
  finally
    // end the drawing operation
    WmfCanvas.Free;
  end;

  // show the current drawing and save it
  PaintBox1.Canvas.Draw (0, 0, Wmf);
  Wmf.SaveToFile (ExtractFilePath (Application.ExeName) + 'test.emf');
end;
```

note If you draw or save the metafile before the connected metafile canvas is closed or destroyed, these operations will produce no effect at all! This is the reason I call the `Free` method before calling `Draw` and `SaveToFile`.

Reloading and repainting the metafile is even simpler:

```
procedure TForm1.BtnLoadClick(Sender: TObject);
begin
```

```
  // load the metafile
  Wmf.LoadFromFile (ExtractFilePath (
    Application.ExeName) + 'test.emf');

  // draw or stretch it
  if cbStretched.Checked then
    PaintBox2.Canvas.StretchDraw (PaintBox2.Canvas.ClipRect, Wmf)
  else
    PaintBox2.Canvas.Draw (0, 0, Wmf);
end;
```

Notice that you can reproduce exactly the same drawing but also modify it with the `StretchDraw` call. (The result of this operation is visible in Figure 22.23.) This operation is different from stretching a bitmap, which usually degrades or modifies the image, because here we are scaling by changing the coordinate mapping. This means that while printing a metafile, you can enlarge it to fill an entire page with a rather good effect, something very hard to do with a bitmap.

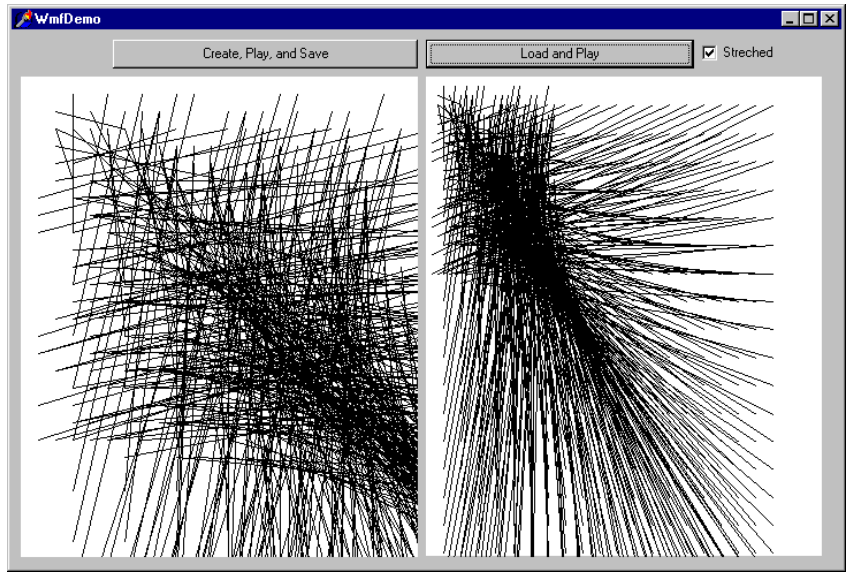


Figure 22.23: The output of the WmfDemo with a stretched metafile. Image from the original book.

Rotating Text

In this chapter, we've covered a lot different examples of the use of bitmaps, and we've created graphics of many types. However, the most important type of graphics we usually deal with in Delphi applications is text. In fact, even when showing a label or the text of an Edit box, Windows still paints it in the same was as any other graphical element. I've actually presented an example of font painting earlier in this chapter in the FontGrid example. Now I'm getting back to this topic with a slightly more unusual approach.

When you paint text in Windows, there is no way to indicate the direction of the font: Windows seems to draw the text only horizontally. However, to be precise, Windows draws the text in the direction supported by its font, which is horizontal by default. For example, we can change the text displayed by the components on a form by modifying the font of the form itself, as I've done in the SideText example. Actually you cannot modify a font, but you can create a new one similar to an existing font:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  ALogFont: TLogFont;
  hFont: THandle;
begin
  ALogFont.lfHeight := Font.Height;
  ALogFont.lfWidth := 0;
  ALogFont.lfEscapement := -450;
  ALogFont.lfOrientation := -450;
  ALogFont.lfWeight := fw_DemiBold;
  ALogFont.lfItalic := 0; // false
  ALogFont.lfUnderline := 0; // false
  ALogFont.lfStrikeOut := 0; // false
  ALogFont.lfCharSet := Ansi_CharSet;
  ALogFont.lfOutPrecision := Out_Default_Precis;
  ALogFont.lfClipPrecision := Clip_Default_Precis;
  ALogFont.lfQuality := Default_Quality;
  ALogFont.lfPitchAndFamily := Default_Pitch;
  StrCopy (ALogFont.lfFaceName, PChar (Font.Name));
  hFont := CreateFontIndirect (ALogFont);
  Font.Handle := hFont;
end;
```

This code produced the desired effect on the label of the example's form, but if you add other components to it, the text will generally be printed outside the visible portion of the component. In other words, you'll need to provide this type of support within components, if you want everything to show up properly. For labels, however, you can avoid writing a new component; instead, simply change the font

associated with the form's `Canvas` (not the entire form) and use the standard text drawing methods. The SideText example changes the font of the `Canvas` in the `OnPaint` method, which is similar to `OnCreate`:

```
procedure TForm1.FormPaint(Sender: TObject);
var
  ALogFont: TLogFont;
  hFont: THandle;
begin
  ALogFont.lfHeight := Font.Height;
  ALogFont.lfEscapement := 900;
  ALogFont.lfOrientation := 900;
  ...
  hFont := CreateFontIndirect (ALogFont);
  Canvas.Font.Handle := hFont;
  Canvas.TextOut (0, ClientHeight, 'Hello');
end;
```

The font orientation is modified also by a third event handler, associated with a timer. Its effect is to rotate the form over time, and its code is very similar to the procedure above, with the exception of the code to determine the font *escapement* (the angle of the font rotation):

```
  ALogFont.lfEscapement := - (GetTickCount div 10) mod 3600;
```

With these three different font rotating techniques (label caption, painted text, text rotating over time) the form of the SideText program at runtime looks like Figure 22.24.

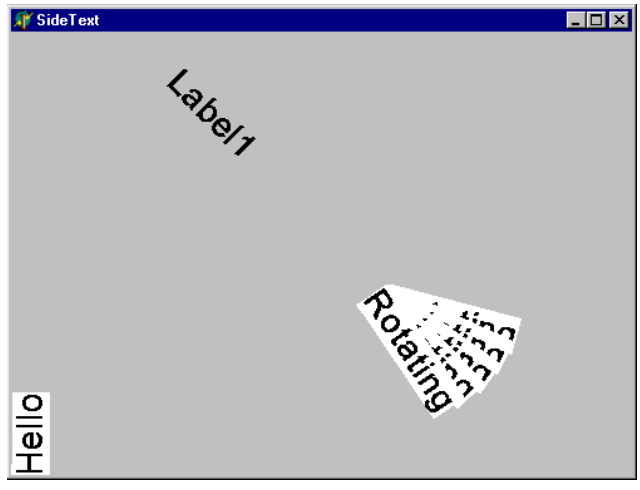


Figure 22.24: The effects of the SideText example, with some text actually rotating. Image from the original book.

Where Do You Go from Here?

In this chapter, we have explored a number of different techniques you can use in Delphi to produce graphical output. We've used the `Canvas` of the form, bitmaps, metafiles, graphical components, grids, and other techniques. There are certainly many more techniques related with graphics programming in Delphi and in Windows in general, including the large area of high-speed games programming.

Allowing you to hook directly with the Windows API, Delphi support for graphics is certainly extensive. However, most Delphi programmers never make direct calls to the GDI system but rely instead on the support offered by existing Delphi components. This topic was introduced in Chapter 13 of *Mastering Delphi 5*.

If you've already read *Mastering Delphi 5,* I hope you've also enjoyed this extra bonus chapter. If you've started by this chapter, the rest of the book has plenty to offer, even in the context of graphics but certainly not only limited to that. Refer to `www.sybex.com` and `www.marcocantu.com` for more information about the book and to download the free source code of this chapter and of the entire book.

Mastering™ Delphi™ 5

Praise for the Previous Edition

Delphi Informant

Of all the Delphi books available, only a handful have achieved special status. *Mastering Delphi 4* is clearly among that handful of books that set the standard. This latest edition of *Mastering* continues the tradition established by its previous three versions....

Throughout the book, Cantù highlights the many new features of Delphi 4, and does a remarkably good job of it....As in previous editions, Marco provides strong coverage of database topics, as well as graphics and user-interface programming issues. The book is filled with excellent code examples illustrating many new and powerful techniques. The vast majority of these [are] small and elegant pieces of code....

Marco has gone to considerable effort to keep the material in *Mastering* fresh and current....Another thing I enjoy about Marco's *Mastering* books is that they're not just technical reference books. I find them excellent just for reading. And—believe me—that doesn't apply to many computer books!

There is definitely something here for everyone....I highly recommend it.

—Robert Vivrette

Developers Review

Marco has done it again with *Mastering Delphi 4*. This book will indeed help you master Delphi 4 programming....Compared to previous editions, I'm pleased to see more in-depth database and Internet coverage....Every topic or component is demonstrated with a small working example, showing how it can be used. The examples are small and to the point....

Mastering Delphi is for "advanced beginners" and up, covering just about any topic related to Delphi programming. I can recommend it to anyone.

—Bob Swart ("Dr. Bob"), co-winner of Borland's 1999 Spirit of Delphi award

Readers on Amazon.com Praise the Previous Edition

If you only buy one Delphi 4 book, this is the one! This is the most comprehensive and useful book on Delphi 4 programming that I have come across (and I've browsed ALL the books on Delphi 4 that Amazon lists).

A reader from Sydney, Australia

Excellent work. Cantù has an excellent summary of the new language features added to Delphi 4, and notes pitfalls that may occur when compiling old code. I generally only refer to books when I'm stuck and need help beyond the online docs. I can sit down and just read this one; there's so much useful information in it that I always find something helpful.

A reader from Fort Wayne, Indiana

Excellent intermediate-level book. This book does an excellent job of walking the user through all the basics of Delphi and Object Pascal. It does assume some familiarity with Pascal, but if the reader has that, this is an extremely easy to follow book. The examples are useful, and are explained well enough that the reader can easily expand them to meet his own needs.

A reader from California

The literary equivalent of Delphi 4. Unmatched by any other. Marco's treatment of each Delphi topic is concise, well thought out, and succinctly written. I cannot praise his book more than this. It's his best yet!

A reader from San Francisco

Don't miss this one. For Bob Swart to review a book highly should be enough for any aspiring Delphi programmer, but I'll add my voice in saying this is another great book from Marco. Unlike other version books, Marco uses tons of new examples and avoids simply providing an updated version of his previous work....

Great book.

A reader from Atlanta, Georgia

Visit Marco's Delphi Developer Web Site

The book's author, Marco Cantù, has created a site specifically for Delphi developers, at `www.marcocantu.com`. It's a great resource for all of your Delphi programming needs.

The site includes:

- The source code of the book (also available on the Sybex site)
- Extra examples and tips
- A number of Delphi components, wizards, and tools built by the author.
- The online book *Essential Pascal*
- Delphi reference material not found in the Help file
- Some papers the author has written about Delphi, C++, and Java
- Extensive links to Delphi-related Web sites and documents
- Other material related to the author's books, the conferences he speaks at, and his training seminars

The site also hosts a newsgroup[475], which has a specific section devoted to the author's books, so that readers can discuss the book content with him and among themselves. There are also other sections of the newsgroup for discussing Delphi programming and general topics.

475 This isn't the case any more

Table Of Contents

Table of Contents