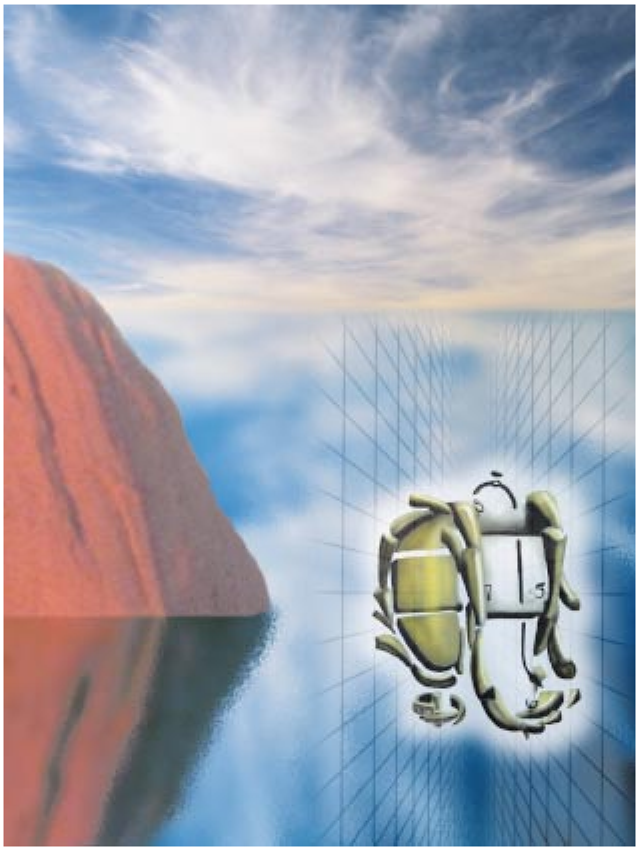
October 1997, Volume 3, Number 10

The Persistence of Objects

Presenting a Generic, Registry-Aware Component



Cover Art By: *Louis Lillegard*

TCompLHA 2.0 and TCompress Component Set 3.5 Ship from SPIS Ltd

SPIS Ltd, makers of the TCompress Component Suite for Delphi and C++Builder, has released *TCompLHA 2.0* and *TCompress Component Set 3.5*.

TCompLHA supports Delphi 1, 2, and 3, as well as C++Builder. It helps create and manage archives compatible with the freeware LHArc and LHA utilities — archives which can also be used by utilities including WinZip.

One-step methods such as *Scan*, *Compress*, *Expand*, *Delete*, and *Verify* simplify archive management. Key properties such as *ArchiveName*, *CompressionMethod*, *Confirm*, and *FilesToProcess* provide control over how files are processed, including wildcard expansion and user interface customization. TCompLHA also offers segmented (multi-disk) archive processing.

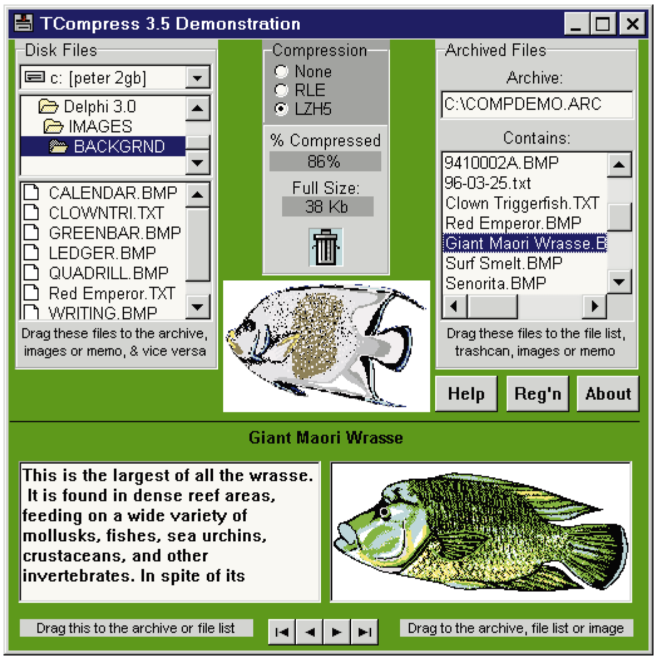
Additional features such as full archive verify and safe mode processing enable TCompLHA to be used in communications applications for the Internet.

TCompLHA ships with a demonstration, source examples, and Help files.

SPIS Ltd also announced a new version of their compression components for Delphi 1, 2, and 3, and C++Builder. In addition to new Delphi 3 and C++Builder support, TCompress 3.5 adds a compression method, a rich text BLOb compression component, and new utility functions for more flexible archive management.

TCompress 3.5 provides native components for the creation of multi-file compressed archives, as well as database, file, resource, and in-memory compression using streams. Two compression methods (RLE and LZH) are built in, with hooks for adding custom compression formats.

TCompress 3.5 also includes drop-and-play components for automatic database BLOb, image, rich text, and memo compression, based on the Delphi VCL's *TDBMemo*, *TDBRichText*, and *TDBImage* components. TCompress 3.5 includes a demonstration, source code, examples, and Help files.


TCompress 3.5 Demonstration

Disk Files
c: [peter 2gb]
Delphi 3.0
IMAGES
BACKGRND
CALENDAR.BMP
CLOWNTRI.TXT
GREENBAR.BMP
LEDGER.BMP
QUADRILL.BMP
Red Emperor.TXT
WRITING.BMP
Drag these files to the archive, images or memo, & vice versa

Compression
None
RLE
[✓ LZH5]
% Compressed
86%
Full Size:
38 Kb

Archived Files
Archive:
C:\COMPDEMO.ARC
Contains:
9410002A.BMP
96-03-25.txt
Clown Triggerfish.TXT
Red Emperor.BMP
Giant Maori Wrasse.B
Surf Smelt.BMP
Senorita.BMP
Drag these files to the file list, trashcan, images or memo

Help | Reg'n | About

Giant Maori Wrasse

This is the largest of all the wrasse. It is found in dense reef areas, feeding on a wide variety of mollusks, fishes, sea urchins, crustaceans, and other invertebrates. In spite of its

Drag this to the archive or file list

Drag to the archive, file list or image

SPIS Ltd
Price: TCompLHA 2.0 is US$65 for registration; US$40 for TCompLHA component source. TCompress Component Set 3.5 is US$65 for registration; US$40 for TCompress component source; and US$34 for source of the compressed DB controls.
Fax: +64-3-384-5138
E-Mail: software@spis.co.nz
Web Site: http://www.spis.co.nz

Design Systems' New VCL Component Collection Adds Time-Saving Features

Design Systems announced *Remember This*, a VCL component collection that allows developers to add options and settings to screens in Delphi applications.

With Remember This, developers can modify any dialog box to remember a user's favorite settings, to see when any component on a form has been modified, and to roll back those modifications.

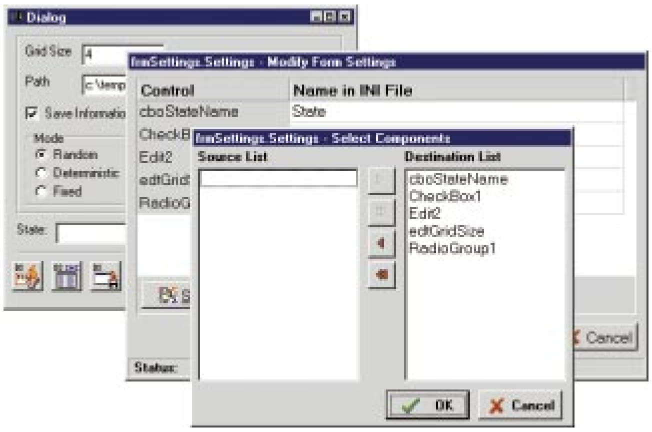
Remember This also provides simpler access to Windows .INI files and the registry.

Similar products require retrofitting existing applications into their framework, but Remember This works with new or existing applications.

Remember This uses Delphi's Run-Time Type Information (RTTI) to automatically handle any component. It relies on a unique and extensible *StateSaver* technology to customize the saving of all Delphi standard controls.

Remember This is available in a Delphi edition; a C++Builder edition is scheduled for release in the 3rd quarter of 1997. Both versions include online Help, sample applications, unlimited support, and a 60-day money-back guarantee. A trial version is available from Design Systems' Web site.

Design Systems
Price: The developers kit of the Delphi or C++Builder version is US$119; with source code it's US$179. A bundle of both versions is US$139; the bundle with source code is US$199.
Phone: (508) 888-4964
E-Mail: support@dsgnsystms.com
Web Site: http://www.-dsgnsystms.com


Dialog
Grid Size 4
Path c:\temp
[✓ Save Informatio]
Mode
[✓ Random]
Deterministic
Fixed
State:

frmSettings.Settings - Modify Form Settings

Control	Name in INI File
cboStateName	State
CheckB	
Edit2	
edtGrid	
RadioG	

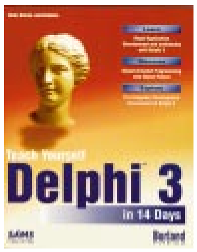
Status:
Cancel

frmSettings.Settings - Select Components
Source List
Destination List
cboStateName
CheckBox1
Edit2
edtGridSize
RadioGroup1
OK | Cancel

Delphi TOOLS

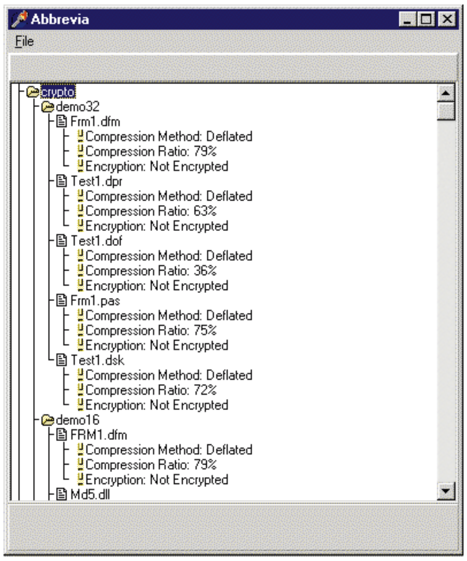
New Products and Solutions

Teach Yourself Delphi 3 in 14 Days
Dan Osier, et al.
Sams Publishing



ISBN: 0-672-31114-3
Price: US$29.99 (599 pages)
Phone: (800) 545-5914 or (317) 581-5801

TurboPower Launches Abbrevia; Releases Free Delphi 3 Upgrades



```
crypto
 demo32
  Frm1.dfm
   Compression Method: Deflated
   Compression Ratio: 79%
   Encryption: Not Encrypted
  Test1.dpr
   Compression Method: Deflated
   Compression Ratio: 63%
   Encryption: Not Encrypted
  Test1.dof
   Compression Method: Deflated
   Compression Ratio: 36%
   Encryption: Not Encrypted
  Frm1.pas
   Compression Method: Deflated
   Compression Ratio: 75%
   Encryption: Not Encrypted
  Test1.dsk
   Compression Method: Deflated
   Compression Ratio: 72%
   Encryption: Not Encrypted
 demo16
  FRM1.dfm
   Compression Method: Deflated
   Compression Ratio: 79%
   Encryption: Not Encrypted
  Md5.dll
```

TurboPower Software Co. has released *Abbrevia 1.0*, enabling Delphi programmers to access and manipulate PKZIP-compatible files.

Abbrevia can create ZIP files; it includes access to advanced features such as disk spanning, self-extracting archives, ZIP-file comments, and password protection. With disk spanning, large ZIP files can be automatically spread across a series of floppy disks. A user receiving a self-extracting ZIP file can expand its contents by executing a small Windows program bound with the ZIP file.

In addition, Abbrevia includes visual components that enable programmers to offer access to ZIP files graphically in their programs. It also has API-level routines for compressing and decompressing streams of data on-the-fly.

Unlike other libraries for handling ZIP files, Abbrevia components are written in Delphi, and will compile directly into an application. Abbrevia requires no additional files be shipped with an application that uses its components. This product ships with documentation, online Help, free technical support, and a 60-day money-back guarantee.

TurboPower also announced the availability of free Delphi 3 upgrades for owners of LockBox, OnGuard, Orpheus, and SysTools. These upgrade patches can be downloaded from http://www.turbopower.com/updates.

In addition to the patches, TurboPower has released a list of the changes necessary to use its flagship product, Async Professional, with Delphi 3. Free updates for other TurboPower products, including FlashFiler, will be available soon.

TurboPower Software Co.
Price: Abbrevia 1.0 (supports 16- and 32-bit development with Delphi 1, 2, and 3), US$199. Current TurboPower customers receive a 20 percent discount.
Phone: (800) 333-4160 or (719) 260-9639
E-Mail: info@turbopower.com
Web Site: http://www.turbopower.com

LMD Innovative Ships LMD-Tools 3.0 for Delphi

LMD Innovative of Siegen, Germany has released *LMD-Tools 3.0*, a component package of over 100 native Delphi VCL components and routines. System, multimedia, visual, data sensitive, and dialog components are available.

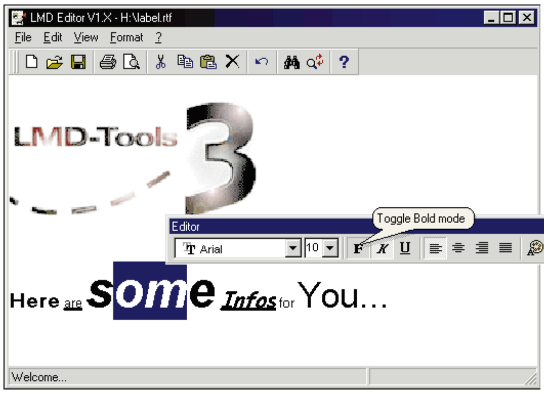
Version 3 adds new components, such as a text editor with RTF support, dockable toolbars, 10 new button classes, and unique handling of data containers. Any component in the set can integrate with Delphi 1, 2, or 3.

About 20 components in the set can be used to manipulate standard system behavior (system menu, caption bars, exploding effects for forms, appearance of hints, timerpool for sharing timers, and more).

LMD-Tools features data container supporting resource files, form data, or custom formats (including compression). Several components can be linked to these containers to share resources, and to make an application smaller and faster.

It also offers standard dialog boxes for immediate use, multimedia components (e.g. for playing .WAV files, creating screensavers, or supporting joysticks), and a design-time helper for hiding non-visual components on a form.

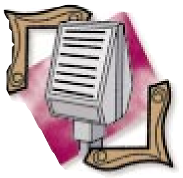
Trial and demonstration versions are available from the LMD Innovative Web site.



LMD Innovative
Price: Standard Edition, US$99; Professional Edition, US$169.
Phone: +49-271-355489
E-Mail: sales@lmd.de
Web Site: http://www.lmd.de

News LINE

October 1997



Borland's Yocam Announces New Business Strategy

Nashville, TN — Borland's Chairman and CEO Delbert W. Yocam outlined a new strategy to establish Borland as the leading provider of tools and technologies to help users build corporate InfoNets. An extension of their Golden Gate strategy, this InfoNet strategy combines Borland's object-oriented development tools, middleware technologies, and open architecture to enable developers to create applications that provide cross-platform access and analysis of data over intranets, extranets, and the Internet.

According to Yocam, Borland is qualified to deliver the technologies needed to help build next-generation information networks. He believes Borland can offer the tools and middleware for the information application life-cycle, including development, analysis, reporting, deployment, and management of information resources and applications.

Yocam noted the key foundations of the InfoNet architecture include enterprise-class application development tools, Java/Web integration, and rich information analysis.

In addition, Yocam said the forces driving the need for InfoNets include: the adoption of thin-client, browser-based applications; increasing growth of Windows NT as an InfoNet application server platform; the maturity of Java as a cross-platform language; the emergence of low-cost network computers and network PCs; and expanded deployment of Decision Support Solutions (DSS) and Online Analytic Processing Server (OLAP) applications, as a result of the growth of the Internet.

Borland Returns to Profitability

Scotts Valley, CA — Borland's first quarter revenues for Fiscal 1998 were US$41,970,000, compared to revenues of US$38,146,000 for the first quarter of the previous fiscal year. (First quarter revenues were up 10 percent over the first quarter of fiscal year 1997.)

Net income for the first quarter of fiscal year 1998 was US$79,000, break-even in terms of earnings per share.

This is compared with a net loss of US$21,809,000 (or US$.60 per share) for the first quarter of fiscal 1997.

The company's cash balance on June 30, 1997 was US$76,600,000, compared to an ending cash balance of US$54,400,000 on March 31, 1997.

The increase in cash was primarily the result of the closing of the privately placed equity financing of approximately US$25 million, and exercises of employee stock options of about US$1.6 million.

When Delbert W. Yocam, Chairman and CEO, joined Borland in December 1996, he pledged a return to profitability in fiscal year 1998, which began on April 1, 1997. To achieve this goal, he cut 30 percent of the workforce, and initiated new cost-reduction programs.

Borland Launches AS/400 Client/Server Developer Program with IBM

Borland announced a developer-relations and marketing program in conjunction with IBM's AS/400 Division. A three-part partnership program, AS/400 ADI assists Borland and IBM AS/400 developers in supporting, registering, and marketing their client/server applications on the IBM AS/400 platform.

As part of the partnership, Borland and IBM are hosting the ADI Web site that includes technical information on Borland's client/server development tools, as well as how to deliver applications to the AS/400. Developers interested in registering their applications on the ADI Web site may send an e-mail to ADI@corp.borland.com, or visit http://www.borland.com/-borland400/.

Borland and the IBM AS/400 Division also plan to launch a world-wide series of technical seminars in more than 15 cities on client/server development for the AS/400.

Borland's AS/400 development tools include Delphi/400 Client/Server Suite and C++Builder/400 Client/Server Suite.

Borland Announces Entera Update

Nashville, TN — Borland has updated its Entera intelligent middleware, which supports Delphi, ODBC data connectivity, and Gradient/DCE on Windows NT.

The company also announced it has extended Entera 3.2's platform support with a new port to IBM MVS/Open Edition.

Borland Entera is an independent framework for building, managing, and deploying middle-tier client/server applications.

The new update to Entera 3.2 is available to all registered Borland Entera support subscribers.

Borland plans to ship a new version of Entera later this year.

For more information, visit Borland's Web site, located at http://www.-borland.com.

Borland Announces JBuilder Products

Nashville, TN — Borland announced its JBuilder family of visual development tools for Java. JBuilder Standard and JBuilder Professional are shipping now, and JBuilder Client/Server Suite is scheduled for release later this year.

All three versions of JBuilder feature JavaBean component creation, a scalable database architecture, visual development tools, and the ability to produce 100% Pure Java applications, applets, and JavaBeans.

As part of Borland's Golden Gate strategy, JBuilder also supports standards such as JDK 1.1, JFC, AFC, RMI, CORBA, JDBC, ODBC, and most database servers. JBuilder also provides an open architecture for adding third-party tools and JavaBean components.

JBuilder's developer-specific features include the Borland RAD WorkBench, pure Java two-way tools, productivity wizards, a new application browser, a graphical debugger and SmartChecker compiler, as well as Local InterBase and Borland SQL Tools.

JBuilder Standard is US$99.95. JBuilder Professional is US$299.95. Owners of other Borland tools can purchase JBuilder Professional for US$249.95. Pricing for JBuilder Client/Server Suite was unavailable at press time. For more information, call Borland at (800) 233-2444.

table of CONTENTS

ON THE COVER

Delphi 3

By Robert Vivrette

The Persistence of Objects

Make Your Applications Registry-Aware with a Generic Object-Oriented Component

Your parents taught you to be persistent. It's the only way to get noticed these days! Well, persistence isn't just a good quality for humans; computer users have come to expect a certain level of persistence in the programs they use as well.

Think about the applications you use every day. No doubt each has its own problems with persistence that annoy you. For example, I use CompuServe for Windows to manage my CompuServe mail, and I am always irritated at the number of places within the program where I have to perform boring tasks over and over again; when I launch the program, it never leaves me where I left off. I have to click on the Mail Center button to look at my list of incoming mail; and whenever I add attachments to messages, the Open dialog is never looking at the right directory. Virtually every program I use has these kinds of minor shortcomings. I'm sure when they were developed, saving the existing state of certain controls was not considered a priority. In fact, many programmers consider such programming to be drudgery.

Yet programming an application to be persistent is a vital piece of your job as an application developer. Your application might be the greatest thing since they put soap in Brillo pads, but if it requires users to unnecessarily provide repetitive input to the program, it will quickly become an annoyance. Here are just a few of the kinds of information that a program should save:

- User information (name, address, passwords, etc.)
- Sizes and locations of forms (including the main application form)
- States of program options
- History lists of file Open and Save dialogs
- Last directory visited by Open and Save dialogs
- History list of recently-accessed document files
- Custom wildcard specifications for file Open and Save dialogs
- Typefaces used in the application
- Last visited tab in tab-sheet controls

The list is virtually endless, and I could go on with additional items. Probably the best indicator of when you have a problem with persistence is to watch a user run your program repeatedly. How many times did they have to perform the same steps? How many times did they have to resize a form, or change to another tab? Every place where there was a repetitive action is a place where you could save that state from the previous execution of the application.

The Windows Registry

32-bit Windows provides us with an excellent place to store persistent data. It's an internal database of information managed by Windows called the *registry*. Before the registry existed, application developers stored persistent data in .INI files. While that technique can still be used, it's no longer recommended. The registry adds capabilities that don't automatically exist in .INI files (such as storing data for different users of the same machine). However, some habits die hard and many application developers still prefer .INI files. For the purpose of this article, we are going to be dealing with only the registry, but if you're stuck on .INI files (or are developing 16-bit applications) the same principles apply.

The format of the registry is a tree structure. Each branch of the tree (or *key* in registry terminology) holds information about a particular user, application, or piece of hardware. At the end of these keys, there are Name/Value pairs. This is where persistent data is saved. For example, for the HKEY_CURRENTUSER\ControlPanel\Colors key, there is a list of key names (e.g. ActiveBorder, ActiveTitle, AppWorkspace, etc.) that have color assignments as values. When you open the Control Panel and access the colors applet, Windows retrieves these color assignments as your color options.

This article assumes you have a basic understanding of how the registry works, so before we go any farther, I must provide this warning: The registry holds data essential to the operation of Windows; if you don't know what you're doing and you accidentally corrupt data in the registry, you quite possible might render Windows inoperative. Therefore, please make certain you understand how the registry functions and make backups of the registry before doing any experimentation. Windows 95 and NT have a program called RegEdit to manipulate the registry database. The online Help for RegEdit will help you if you have questions on its use. With that out of the way, let's go on.

So we want to save persistent application data, and we want to use the registry to hold that data. How can you get a component to save pieces of itself (properties, values, etc.) automatically? Well, there have been many efforts in the Delphi community to create persistent objects, and to understand the issues a bit more, lets look at some of these along with their advantages and disadvantages.

Subclassed Registry-Aware Components

In this scheme, a developer creates a component that is aware of the registry, and has the ability to save and load data to and from the registry. Say for example that you wanted an Edit component that automatically saved the value of its *Text* property to a certain registry key. Using this technique, you would create a new component (descended from a *TEdit*), and add support for the registry. This support would likely be in the form of two methods (one each to save and restore the *Text* property to the registry), as well as some new property that lets the user define what key to use in the registry.

The result would be something named, say, *TRegistryEdit*. It would look and behave exactly as an Edit component, but would have one additional property named, say, *RegistryKey*. To use the control, a developer simply takes a *TRegistryEdit* control, drops it on the form, provides the name of a key in *RegistryKey* and then adds a line of code in the *FormCreate* and *FormDestroy* methods to access the registry load and save methods of the new subclassed control.

What are the disadvantages of such an approach? There are plenty. The key one is that you have to create a new subclassed control for every component to which you want to add registry support. If you want to add registry support to ListBoxes, ComboBoxes, StringGrids, RadioButtons, and CheckBoxes, you will have to create a new subclassed control for each of these and add the same registry support code that you did for *TRegistryEdit*. The code you add would be essentially identical, but would of course save the checked state of a CheckBox and RadioButton, the list of items in a ListBox and ComboBox, etc. However, when you add all these new controls to your form, you're adding quite a lot of redundant registry support code. Remember, each subclassed control you added had its own registry support! Also, your Component palette will have to house all these new registry-aware components.

In addition, you still will need to manage (in code) each of these controls. Your *FormCreate* method might start to look like this:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Edit1.LoadFromRegistry;
  Edit2.LoadFromRegistry;
  Edit2.LoadFromRegistry;
  Edit2.LoadFromRegistry;
  ListBox1.LoadFromRegistry;
  StringGrid1.LoadFromRegistry;
  CheckBox1.LoadFromRegistry;
  CheckBox2.LoadFromRegistry;
  CheckBox3.LoadFromRegistry;
  CheckBox4.LoadFromRegistry;
  CheckBox5.LoadFromRegistry;
end;
```

Obviously, this adds a lot of maintenance responsibilities when developing the application, and it will be prone to error (particularly if you're saving and restoring dozens or hundreds of values).

Another disadvantage with this approach is that the new registry-aware control is only going to save data that you as the component developer have programmed it to save. For many controls this is obvious: An Edit component saves its *Text* property, a CheckBox saves its *Checked* state, and so on. But what if someone using your registry-aware control wants to save the Edit component's *ShowHint* property, or the CheckBox's *Enabled* property? They probably would have to toss your control and make their own subclassed version saving the values they want to save. Obviously, this is not a great example for promoting code re-use.

Now, granted this kind of idea could have worked only if the Delphi development team decided to add automatic registry support to the component hierarchy. They could have added some kind of *RegistryKey* property in the *TPersistent* class (or somewhere else nearby) and then all components descending from it would have that same basic behavior. However, it would have clearly made the VCL a bit more cluttered. Who knows, they may have not even thought of it. Regardless, it isn't there now so we have to find another solution.

A Better Solution

Let's step back a bit and look at the problem with a broader perspective. We want to save data related to controls, but as we already learned, we don't want to have to create new registry-aware versions of each control. This means that our improved solution must work with unmodified components and should require the absolute minimum amount of maintenance.

Enter the Persistent Object Manager (ObjMgr). For those of you who have used my IniOut Component Property Manager, the Persistent Object Manager will be familiar. Although IniOut is a shareware product, I wanted Delphi programmers to get a feel for some of the concepts it uses. As a result, I created a completely new component, based loosely on the ideas used in IniOut. That way, I can provide all the source code necessary to show you how it works in this article without sabotaging the success of IniOut.

First, let's go through a brief example of how ObjMgr works. Then we can dig into how it's implemented in code.

Using the Persistent Object Manager

Let's say we are writing a simple piece of code that will save the name of the program's user and some other related details. This form might look something like Figure 1.

The first time your application runs, you want to gather this information and save it to the registry. You will probably also want the ability for the user to get back to this form to modify or update the information. Granted, you could just create an instance of a *TRegistry* object and write out the values when the form closes and read them back in when it opens again. However, the objective here is to not have to write or manage this kind of code, particularly for each component on the form.

Instead, let's add an ObjMgr component to the form. It's a non-visual component, and once you place it on a form and double-click it, you will see the property editor shown in Figure 2. This is the project we're going to create in this article. The ListBox on the left shows what component properties you're managing. For each item managed, you will provide data in the controls on the right. We add a new item by clicking on the **Add** button. The **Component** combo box enables you to pick a control on the current form. Note that

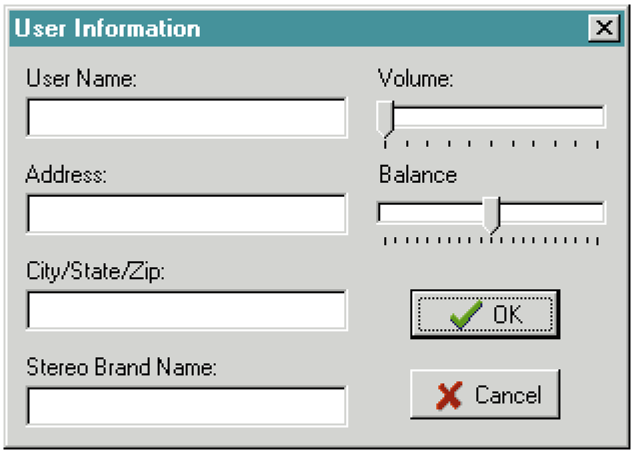


Figure 1: A sample form.

by "dropping down" this combo box, we automatically see all the controls that are present on the form (see Figure 3).

We will start with the **User Name** edit box (*edtName*). When you select that component, the **Property** combo box is populated with all the supported properties of that control (see Figure 4). In this case, we want to manage the *Text* property. After choosing this property, we provide the registry subkey that this value will be saved as. Let's enter `Software\Informant\ObjMgrDemo\UserData` for this value. Then, we provide the name of this particular key (in this

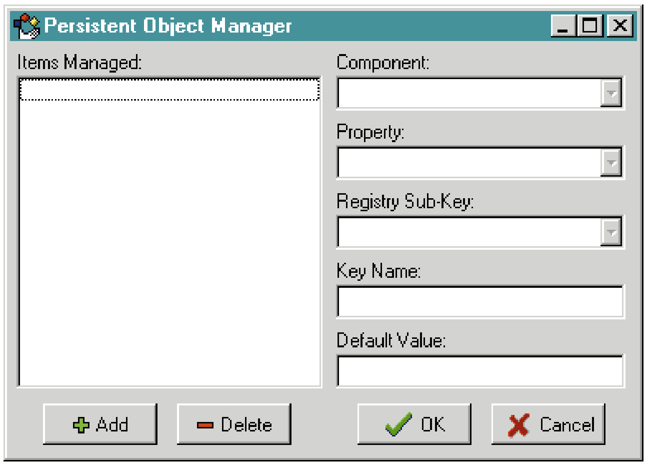


Figure 2: The property editor.

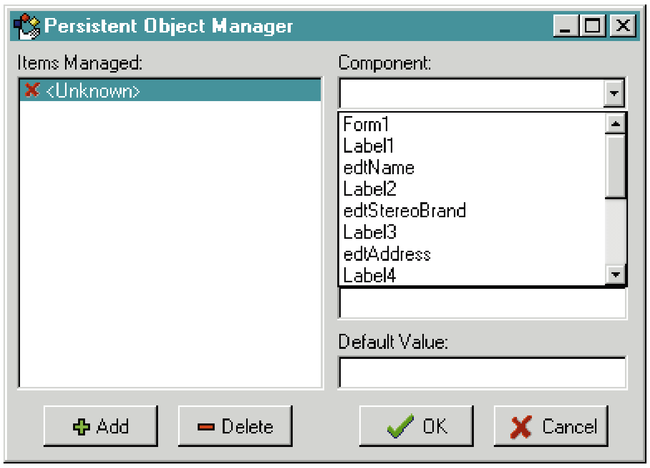


Figure 3: Accessing the controls.

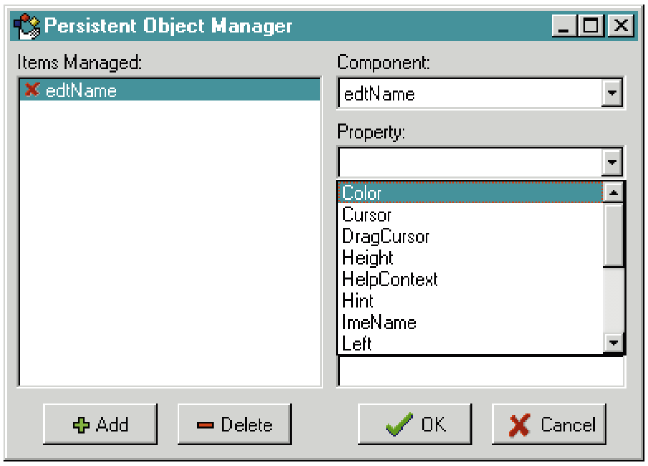
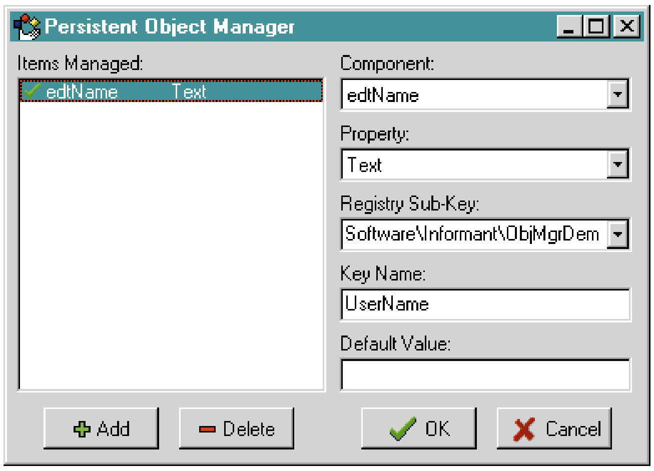


Figure 4: Viewing the supported properties.

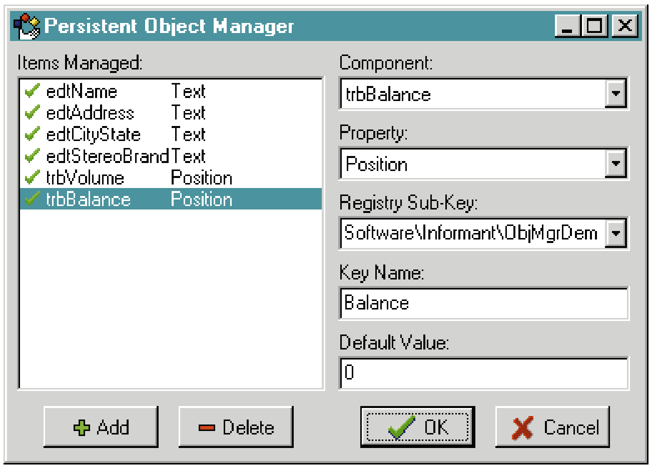


Items Managed:

edtName	Text

Component: edtName
Property: Text
Registry Sub-Key: Software\Informant\ObjMgrDem
Key Name: UserName
Default Value:

Figure 5: A completed property editor.

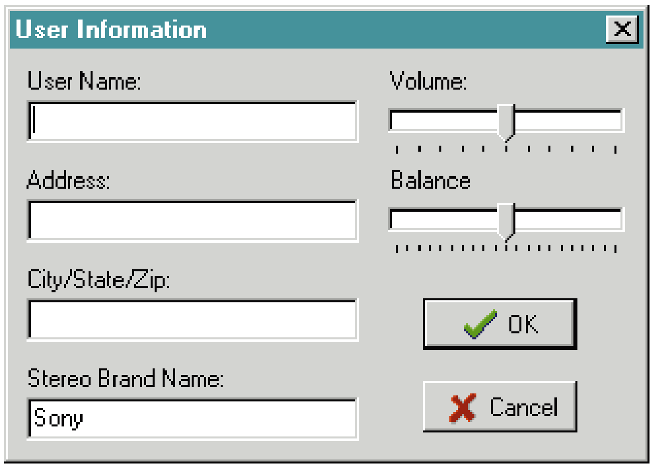


Items Managed:

edtName	Text
edtAddress	Text
edtCityState	Text
edtStereoBrand	Text
trbVolume	Position
trbBalance	Position

Component: trbBalance
Property: Position
Registry Sub-Key: Software\Informant\ObjMgrDem
Key Name: Balance
Default Value: 0

Figure 6: All the items we want to manage.

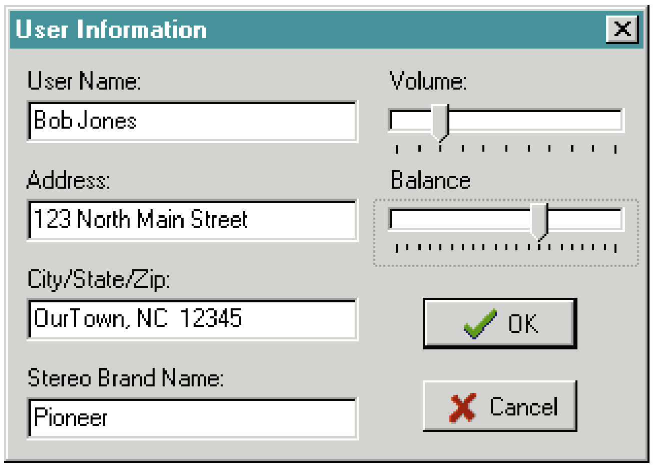


User Name:
Address:
City/State/Zip:
Stereo Brand Name: Sony

Figure 7: The running application.

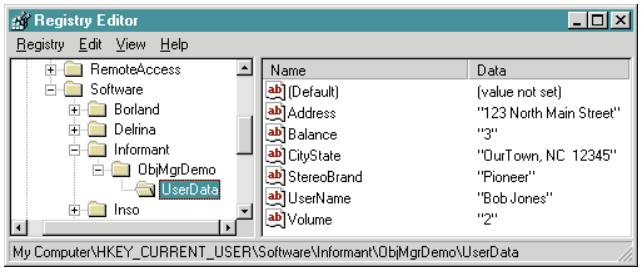
case, we'll call it UserName). Lastly, we have a field that allows us to provide a default value for the property. It wouldn't make sense to provide a default for a user name, so we will leave this blank. After all these steps have been done, the property editor would look like Figure 5.

Now, let's add all the other items we want to manage. Specifically, we would want to save the *Text* properties of the **Address**, **City/State/Zip**, and **Stereo Brand Name** edit boxes. Also, let's save the *Position* property of each of the two track



User Name: Bob Jones
Address: 123 North Main Street
City/State/Zip: OurTown, NC 12345
Stereo Brand Name: Pioneer

Figure 8: Enter data then close the application.



Name	Data
(Default)	(value not set)
Address	"123 North Main Street"
Balance	"3"
CityState	"OurTown, NC 12345"
StereoBrand	"Pioneer"
UserName	"Bob Jones"
Volume	"2"

My Computer\HKEY_CURRENT_USER\Software\Informant\ObjMgrDemo\UserData

Figure 9: The data in RegEdit.

bars (**Volume** and **Balance**). Note that the registry subkey value will be the same for each item (more on this later). After specifying each of these, the property editor would look like Figure 6.

We dismiss the property editor by clicking **OK**. Now there is only one thing left to do: We need to tell ObjMgr when to read values from the registry and when to save them. The *FormCreate* and *FormDestroy* methods are good spots; all we do is add a single line of code to each to access the *Load* and *Save* methods of ObjMgr as follows:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ObjMgr1.Load;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  ObjMgr.Save;
end;
```

Now run the application (see Figure 7). See what happened? ObjMgr set up the form as we requested through its property editor. Because this is the first time the program has been executed, there is no entry in the registry for any of this data. Therefore each control was given its default values.

Now let's enter data in each field (see Figure 8) and close the application. When the application closed, the *FormDestroy* method was triggered, which told ObjMgr to save its managed properties to the registry. Now if you run RegEdit, you'll note that the registry data in Figure 9 has been entered.

The location of these keys comes, of course, from the subkey field data we provided to ObjMgr. Now comes the cool part: Run the program again. See what happened? Because the registry key existed along with all the values we provided to ObjMgr, each of the controls has been restored to how it was when the form last closed. This is because we told ObjMgr to load its values in the form's *FormCreate* method.

Keep in mind that all we needed to do to accomplish this was to add an ObjMgr component to the form, use its property editor to provide some basic information of properties to manage, then add two lines of code (one each in the *FormCreate* and *FormDestroy* methods). We used stock controls from Delphi's Component palette and turned them into persistent objects. Not bad! Furthermore, if you were to expand on the form to hold a few more controls, and you wanted to save property values from those controls, you only need to access ObjMgr's property editor and add those items!

How It Works

Now that we've seen ObjMgr at work, let's check under the hood to see *how* it works. I won't be covering every aspect of ObjMgr, but rather hitting the highlights. But fear not! If you want to use ObjMgr yourself, the complete source code is available on (see the end of this article for details).

The key piece of ObjMgr is of course the property editor that you see when you double-click on the component (again, see Figure 2). The whole purpose of this property editor is to manage the contents of a single *TStrings* object. This string list holds the essential bits of information necessary for ObjMgr to save and restore properties from the registry. Although this list is internal to the component and you never see it directly, a brief discussion of its structure would be beneficial.

Each item in this string list holds information for a single property being managed. The string is a collection of smaller strings separated by semicolons. Each string list contains the following items:

- Component Name
- Property Name
- Registry SubKey
- Key Name
- Default Value
- Valid Flag

In our previous discussion, therefore, the first item we entered into ObjMgr would be stored internally like this:

```
edtName;Text;Software\Informant\ObjMgrDemo\UserData;UserName;;1
```

First, there is the component name (*edtName*) then a semicolon, followed by the property name (*Text*), another semicolon, and so on. Towards the end, you see two semicolons together. This is where the default is stored, and we didn't provide one for *edtName*. Last, there is a 1, which is a flag to tell ObjMgr whether the item is valid. If the item is valid, it's included when saving or restoring from the registry. If it isn't valid, it's skipped.

```
procedure TObjMgrDialog.FormShow(Sender: TObject);
var
  a : Integer;
begin
  // Add the form name to the list so we
  // can access its properties.
  cmbComponent.Items.Add(TheObjEd.Designer.Form.Name);
  // Populate combo with all components on form.
  with TheObjEd.Designer.Form do
    for a := 0 to ComponentCount-1 do
      cmbComponent.Items.Add(Components[a].Name);
  ValidateAll;
  // Are there items in the items managed list?
  if lbItemsManaged.Items.Count > 0 then
    begin
      // Select the first one.
      lbItemsManaged.ItemIndex := 0;
      // Simulate a click on it to populate other controls.
      lbItemsManagedClick(lbItemsManaged);
    end;
end;
```

Figure 10: The *FormShow* method.

Run-Time Type Information

With that piece of architectural design behind us, let's move on. When the ObjMgr is activated, how does it get the list of components from the form? Well, as it turns out, the creators of Delphi knew that component developers would want this feature. These capabilities are all collected under one main concept, namely what Borland calls RTTI (run-time type information). This is a mechanism in the internal structure of Delphi that allows you to interrogate any component at run time to determine its name, properties, methods, etc. RTTI is essential for the design-time pieces of Delphi to function, and ObjMgr relies heavily on RTTI to work its magic. If you want to learn more about RTTI, look through the Delphi units Typinfo.pas and Dsgnintf.pas. Both of these files are reasonably well documented, and many of the procedures mentioned therein are also referenced in the Delphi Help system.

So to get the list of components on the form, we need to talk to RTTI. We first do this in the *FormShow* method of the property editors form (see Figure 10). The first step is to add the name of the form itself to the **Component** combo box. This will allow us to access the form's properties. Next, we need a list of all the components on the form. This is done by means of a reference to the Form Designer, which is defined in Delphi's Dsgnintf.pas unit. Basically, this is a way a property editor can obtain a connection to the form that is in use. In our case, we use this reference to get access to the form's *Components* array. We simply loop through all the elements of this array and add each component name to the **Component** combo box.

Once a component is selected from this list, we need to get a list of its properties to populate the **Property** combo box. After a user has chosen a component (actually on its *OnChange* event), we call the *UpdatePropertiesCombo* method. The first line of code in this method extracts the selected item from the **Items Managed** list box (as the compound strings we were talking about earlier). I then use a short function called *Parse* to

```
procedure TObjMgrDialog.lbItemsManagedClick(
  Sender: TObject);
begin
  UpdateComponentsCombo;
  UpdatePropertiesCombo;
  UpdateSubKeyCombo;
  UpdateKeyNameEditBox;
  UpdateDefaultsEditBox;
end;
```

Figure 11: Calling the *Update* methods when a new item is clicked.

extract the individual piece we're looking for — the name of the component, in this case. Referring to the Form Designer again, we obtain a reference to the component (which can be either a component on the form or the form itself).

Next, we use the *GetPropList* function in Delphi's Typinfo.pas unit. This function returns all properties matching a certain criteria in its class or its ancestors. That criteria is a set of the supported property types ObjMgr can handle (specifically, *tkInteger*, *tkChar*, *tkString*, and *tkLString*). If you call this function and provide a **nil** as the last argument, the function simply returns the number of properties for the component. After we have this count, we allocate a chunk of memory of the appropriate size and then call the function again — this time including a pointer to that block of memory. Then it's just a matter of scanning through this block of data (records of type *TPropInfo*) and extracting the name of each property we encounter. After that, we make sure that we free the memory block; the whole section is wrapped in a **try..finally** construct to make sure this happens.

Briefly then, whenever a user clicks on an item in the Items Managed list box to the left, ObjMgr needs to fill each control on the right with the appropriate data (if available). Obviously, if you're adding a new item to the list, some of the controls will have no data to display. But after you've entered a few items, you will be able to scroll up and down the list and all the controls on the right would be automatically updated. The effect is much like that of data-aware controls. In the preceding paragraphs, I discussed how *UpdatePropertiesCombo* works, but there is actually a similar *Update* method for each of the controls on the form. This is shown in Figure 11.

One additional routine I wanted to comment on is *NewRegSubKeyCheck*. This method is called whenever the user hits Enter in, or exits from, the **Registry Sub-Key** field in the property editor. We want the drop-down list to show all the subkeys entered so far. This routine looks at any entry into the drop-down combo box and checks to see if it already exists in the list. If it does, that item is selected. If it doesn't, the new string is added to the list. This is a good general purpose routine for this kind of drop-down combo box management.

The Units

The code for ObjMgr is split up into two units: Objmgr.pas and Objmgr2.pas. The first unit houses all the behavior of the property editor as a whole, i.e. the way the controls work, the way it allows you to add and delete items managed, and so on. The second unit, Objmgr2.pas, contains the internal string list of managed properties as well as the registry-support code.

When you install ObjMgr into the Delphi Component Library, you're adding Objmgr.pas, which references Objmgr2.pas.

At run time, however, the only file that needs to be included in a project is Objmgr2.pas. Because that is where the internal string list and registry code are, Delphi knows that it doesn't need to link in all of the design-time material into your run-time executable. If you looked at the compiled DCU files for each of these units, you would see that Objmgr.dcu is about 17KB and Objmgr2.dcu is only 6KB. Because the design-time and run-time material has been segregated in this way, we save the 17KB of Objmgr.dcu that doesn't link into the final executable.

The Run-Time Implementation

This brings us to the run-time piece of the code, Objmgr2.pas (shown in its entirety in Listing One beginning on page 12). The unit is straightforward. First, there is the definition of *TObjMgr*, which descends from *TComponent*. There is a single private variable, *FItems*, which is the internal string of items that ObjMgr is managing. Then there are the *Load* and *Save* methods that do the actual reading and writing of registry data.

Let's take a look at the *Save* method (the *Load* method is essentially the same). The first step is to create an instance of a *TRegistry* object, and set its *RootKey* property to `HKEY_CURRENTUSER`. ObjMgr would of course be more flexible if this could be changed by means of a property, but for the sake of simplicity, I have hard-coded it here. After setting up the registry object instance, we now need to loop through all the items managed. We first verify that the object is valid by checking the 1 or 0 at the end of each item string. If the item isn't valid, it's ignored.

Assuming the item is valid, we next extract each piece of the item (component name, property name, registry key, etc.). We obtain a reference to either the form or a component on the form, and once we have that component reference, we pass it into the *GetPropInfo* function (in Delphi's Typinfo.pas unit). *GetPropInfo* returns a pointer to a *TPropInfo* structure. That structure holds all the basic information necessary to gain access to the property and its stored value.

Now we need to write the data out to the registry. We first close any key that might be open and then open the newly specified key. The second parameter of the *OpenKey* method is set to *True,* which tells the *TRegistry* object to create the key if it doesn't exist. Now we simply write out the key with *TRegistry*'s *WriteString* method. We give it the key name the user provided and a string value to write out. That string value is actually the return result of a function in this unit called *GetPropAsString*. Its purpose is to return the current value of the specified property as a string value. It is that value that is written to the registry.

So let's look at *GetPropAsString*. When we called it, we passed in a reference to the component we are working with, along with the *PropInfo* record of the specific property we are interested in. We first examine the *Kind* property of *PropInfo* to see if this property is an integer, char, or string.

If it is a string, we simply use *GetStrProp* (also defined in Delphi's Typinfo unit) to return the current value of that string property. If it is of Char type, we use *GetOrdProp* and convert the ordinal value that comes back into a Char value. If it is an *Integer* property, we want to do a little more than simply return a number. In the event that a user wants to save off a *Color* or *Cursor* property, it wouldn't be very handy to require the user to lookup that the color *clFuchsia* is the number 16711935. Instead, we use some handy functions provided with Delphi that will convert a numeric value to and from either a *Color* or *Cursor*. Then it is simply a matter of determining if we are looking at a *TColor* or *TCursor* and calling the appropriate conversion routine (*ColorToIdent* or *CursorToIdent* — both defined in Graphics.pas). Because we add this extra step, we now will have a registry entry that says that the *Color* key is *clFuchsia* instead of the meaningless value 16711935. Also, we can use the same technique when asking the user for a default value for such a property. In the **Default Value** edit box in the property editor, we can enter `clYellow` and the property editor will make the appropriate conversion.

Some Additional Features

There are some additional features of ObjMgr that I am going to mention briefly here. As you can see, the property editor form is resizable. I trapped the WMGetMinMaxInfo message that Windows processes to tell it the minimum size the form can be. Also, on the form's *Resize* method, I put in code that stretches the controls so that they take advantage of additional form space. This is particularly useful to view long registry key names.

I also added a component editor to ObjMgr to add the capability of double-clicking on the component to get its property editor up. It also adds a **Edit Managed Properties** menu item on the right-click popup menu for the ObjMgr component.

The **Items Managed** list box is also different in that it is an owner-draw list box. I wanted to indicate whether a particular item was valid or not, so I added a bitmap resource to the project that shows either a green check or a red "X" drawn to the left of each item to indicate its validity.

IniOut and Where to Go from Here

As I mentioned before, ObjMgr is based on principles used in my IniOut utility (yes, I hate the name too, but I am stuck with it now). ObjMgr was created to show how you can apply these principles of object persistence in your applications. However, it clearly falls short in many areas. For one thing, it only supports three property types, i.e. integer, string, and char. These are the easiest to implement and I chose them strictly to limit the focus of this article.

ObjMgr can be extended to add other property types with varying degrees of effort. Floating-point numbers would probably be the next easiest to implement. After that would come set types, which would be a little more difficult because you will need to manage the individual elements of the set and convert them to and from a string. Getting even more difficult would be enumerated types, which would allow you to manage something like a component's *Align* property. When you chose the *Align* property, the **Default Value** edit box could become a drop-down list populated with all the valid elements of that enumerated type; in this instance it would hold *alBottom*, *alClient*, *alLeft*, *alNone*, *alRight*, and *alTop*. You would also need to convert these to and from textual representations (e.g. *alClient* as a string) into its actual type (*alClient* as an item of the type *TAlign*).

IniOut takes those extra steps to be a far more comprehensive solution. It manages virtually any property type (including floating-point properties, sets, enumerated types, classes, methods, and compound properties types such as Fonts, StringLists, and so on. It also allows you to add non-published properties, so you can add variables at run time to its list of managed properties.

As a special offer to readers, I have decided to provide a free copy of the registered version of IniOut to any *Delphi Informant* subscriber. The shareware version has a few limitations, but the registered version is completely functional and includes compiled units for all versions of Delphi. The registered version doesn't include the source to IniOut, but it is available for a modest fee. For those of you who have previously paid for a registered copy of IniOut, don't fret! The money you paid for the registered version will be applied toward the cost of the source code version if you wish to get the IniOut source.

To get your free registered copy of IniOut, simply send me an e-mail at RobertV@csi.com. Include in your e-mail the number directly to the left of your subscription expiration date. The number should look something like "C 12345." I will send you the registered version of IniOut by return e-mail.

Conclusion

With the Persistent Object Manager, making a Delphi object or application persistent is as easy as dropping a component on a form. Saving persistent data helps users of your applications be more productive and eliminates annoying repetitive chores or input of data.

In the past, making objects persistent might have been a chore that you as an application developer would rather not deal with. Now you have no excuse!

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\OCT\DI9710RV.

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

Begin Listing One — Objmgr2.pas

```
unit OBJMGR2;

interface

uses
  WinProcs, WinTypes, Messages, SysUtils, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls, Buttons,
  TypInfo, DsgnIntF, Registry;

const
  // These are the supported property types.
  tkProps = [tkInteger,tkChar,tkString,tkLString];
  ppComponent = 0;
  ppProperty  = 1;
  ppSubKey    = 2;
  ppName      = 3;
  ppDefault   = 4;
  ppValid     = 5;

type
  TObjMgr = class(TComponent)
  private
    FItems     : TStrings;
    procedure SetItems(Value: TStrings);
  protected
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Load;
    procedure Save;
  published
    property Items: TStrings read FItems write SetItems;
  end;

  function Parse(TmpS: string;
                 A: Integer; C: Char): string;

implementation

function Parse(TmpS: string; A: Integer; C: Char): string;
var
  SCNum,X : Integer;
begin
  Result := '';
  SCNum := 0;
  for X := 1 to Length(TmpS) do
    if TmpS[X] = C then
       Inc(SCNum)
    else
      if SCNum = A then
         Result := Result + TmpS[X];
  Result := Trim(Result);
end;

function IsValid(a: string): Boolean;
begin
  Result := (Parse(a,ppComponent,';')<>'');
  if not Result then
    Result := (Parse(a,ppProperty,';')<>'');
  if not Result then
    Result := (Parse(a,ppSubKey,';')<>'');
  if not Result then
    Result := (Parse(a,ppName,';')<>'');
end;

function GetPropAsString(Obj: TObject;
  Info: PPropInfo): string;
var
  IntVal : LongInt;
begin
  Result := '';
  with Info^ do
    case PropType^.Kind of
      tkInteger :
        begin
          // Get the integer value.
          IntVal := LongInt(GetOrdProp(Obj,Info));
          // If TColor or TCursor, convert to text.
          if (PropType^.Name = 'TColor') and
             ColorToIdent(IntVal,Result) then
          else if (PropType^.Name = 'TCursor') and
                  CursorToIdent(IntVal,Result) then
          else
            Result := IntToStr(IntVal);
        end;
      tkChar    : Result := Chr(GetOrdProp(Obj,Info));
      tkString,
      tkLString : Result := GetStrProp(Obj,Info);
    end;
end;

procedure SetPropFromString(Obj: TObject; Info: PPropInfo;
  Str: string);
var
  IntVal  : LongInt;
  CharVal : Char;
begin
  try with Info^ do
    case PropType^.Kind of
      tkInteger :
        if Str<>'' then
          if PropType^.Name = 'TColor' then
            if IdentToColor(Str,IntVal) then
               SetOrdProp(Obj,Info,IntVal)
            else
              SetOrdProp(Obj,Info,StrToInt(Str))
          else
            if PropType^.Name = 'TCursor' then
              if IdentToCursor(Str,IntVal) then
                 SetOrdProp(Obj,Info,IntVal)
              else
                SetOrdProp(Obj,Info,StrToInt(Str))
            else
              SetOrdProp(Obj,Info,StrToInt(Str));
      tkChar :
        begin
          CharVal := Str[1];
          SetOrdProp(Obj,Info,Ord(CharVal));
        end;
      tkString,
      tkLString : SetStrProp(Obj,Info,Str);
    end;
  except
    // This catches invalid property assignments.
  end;
end;

{ TObjMgr - Persistent Object Manager }
constructor TObjMgr.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FItems := TStringList.Create;
  TStringList(FItems).OnChange := nil;
end;

destructor TObjMgr.Destroy;
begin
  FItems.Free;
  inherited Destroy;
end;

procedure TObjMgr.SetItems(Value: TStrings);
begin
  FItems.Assign(Value);
end;

procedure TObjMgr.Load;
var
  Reg      : TRegistry;
  A        : Integer;
  TmpCmp   : TComponent;
  PropInfo : PPropInfo;
  S1       : string;
  CmpName  : string;
  PrpName  : string;
  SubKey   : string;
  KeyName  : string;
  DefVal   : string;
begin
  Reg := TRegistry.Create;
  try
    Reg.RootKey := HKEY_CURRENT_USER;
    // Loop through all items managed.
    for A := 0 to Items.Count-1 do begin
      S1 := Items[A];
      // Verify item is valid.
      if not IsValid(S1) then Continue;
      // Extract the individual elements from the item.
      CmpName := Parse(S1,ppComponent,';');
      PrpName := Parse(S1,ppProperty,';');
      SubKey  := Parse(S1,ppSubKey,';');
      KeyName := Parse(S1,ppName,';');
      DefVal  := Parse(S1,ppDefault,';');
      // Check to see if this is a form.
      if CmpName = (Owner as TForm).Name then
        // Yes - use the form.
        TmpCmp := (Owner as TForm)
      else
        // Find the component on the form.
        TmpCmp := (Owner as TForm).FindComponent(CmpName);
      // Couldn't find component - go on to next.
      if TmpCmp = nil then Continue;
        // Get the info record on this component.
        PropInfo := GetPropInfo(TmpCmp.ClassInfo,PrpName);
      if PropInfo = nil then Continue;
      try
        Reg.CloseKey;
        // Open the Subkey.
        if Reg.OpenKey(SubKey,False) then
           // Does this key name exist?
           if Reg.ValueExists(KeyName) then
             // Yes - set the property.
             SetPropFromString(TmpCmp,PropInfo,
                               Reg.ReadString(KeyName))
           else
             // No it doesn't exist - use the default.
             SetPropFromString(TmpCmp,PropInfo,DefVal)
         else
           // Couldn't open the key - use the default.
           SetPropFromString(TmpCmp,PropInfo,DefVal);
      except
      end;
    end;
  finally
    Reg.Free;
  end;
end;

procedure TObjMgr.Save;
var
  Reg      : TRegistry;
  A        : Integer;
  TmpCmp   : TComponent;
  PropInfo : PPropInfo;
  S1       : string;
  CmpName  : string;
  PrpName  : string;
  SubKey   : string;
  KeyName  : string;
begin
  Reg := TRegistry.Create;
  try
    Reg.RootKey := HKEY_CURRENT_USER;
    // Loop through all items managed.
    for A := 0 to Items.Count-1 do begin
      S1 := Items[A];
      // Verify item is valid.
      if not IsValid(S1) then Continue;
      // Extract the individual elements from the item.
      CmpName := Parse(S1,ppComponent,';');
      PrpName := Parse(S1,ppProperty,';');
      SubKey  := Parse(S1,ppSubKey,';');
      KeyName := Parse(S1,ppName,';');
      // Check to see if this is a form.
      if CmpName = (Owner as TForm).Name then
        // Yes - use the form.
        TmpCmp := (Owner as TForm)
      else
        // Find the component on the form.
        TmpCmp := (Owner as TForm).FindComponent(CmpName);
      // Couldn't find component - go on to next.
      if TmpCmp = nil then Continue;
        // Get the info record on this component.
        PropInfo := GetPropInfo(TmpCmp.ClassInfo,PrpName);
      if PropInfo = nil then Continue;
      try
        Reg.CloseKey;
        if Reg.OpenKey(Subkey,True) then
           Reg.WriteString(
             KeyName,GetPropAsString(TmpCmp,PropInfo));
      except
      end;
    end;
  finally
    Reg.Free;
  end;
end;

end.
```

End Listing One

Delphi 2 / Delphi 3

By *Jay Cole*

Down to the Metal

High-Performance Computing with Delphi

Delphi generates incredible code. It can pass procedure parameters in registers instead of setting up a stack frame; it uses pointer arithmetic instead of array index calculations when it can. All in all, the code Delphi generates is at least on a par with any C/C++ compiler out there.

Are you tired of C/C++ programmers claiming you can't do low-level programming efficiently in Delphi? They maintain that only languages such as C/C++ are suited for low-level tasks, and that Delphi is great for user-interface programming and prototyping, but for time-critical, "serious" applications, only C/C++ or assembler will do. Well, you can out-perform even the craftiest C/C++ programmer — if you are willing to break a few rules.

So what can you do when you need even more performance? Obviously, first visit the algorithm used to solve the problem. A well designed algorithm can create orders-of-magnitude improvements in performance; the classic example is a bubble sort versus a quick sort. That is exactly what we at OpNek did with NitroSort: We invented a new sorting algorithm that was several times faster than the traditional quick sort. But that wasn't enough. We couldn't get the performance numbers we wanted without breaking some of the cardinal rules of computer science, and generating some very tight code. (Note: The material in this article applies only to Delphi 2/3 running on Windows 95/NT.)

Creating a Dynamic Compiler

Remember back in college when your professor said, "Don't ever, ever, ever write self-modifying code?" Well, this article puts a bit of a twist on that. It doesn't actually modify existing code, but it does generate entirely new code, in memory, on-the-fly, based on a small, embedded language specific to the problem being addressed. The technique is simple: Allow users to specify what they want to accomplish through a small language geared to the problem at hand.

In NitroSort, this is a sorting language that allows the user to specify the file type, record size, and sort keys, among other things. Then the system allocates a block of executable memory, and compiles the machine code directly into the allocated memory block. Finally, a standard function pointer in Delphi is assigned to the allocated memory block, and is called as a function pointer. Essentially, this is what Borland did with Turbo Pascal 3.0 when it compiled and ran within the DOS environment.

Sound difficult? After all, who knows machine code? Who knows how to write a compiler? How does one debug and test the code? Actually, it looks more difficult than it is. Surprisingly, all this is fairly easy for most embedded-language applications. With tools such as Visual Parse++ for the parsing and Turbo Debugger for generating the machine code and debugging compiler output, the process is quick and straightforward. (This article's references to the Turbo Debugger refer specifically to the Turbo Debugger for Win32. It allows assembling and disassembling of 32-bit instruction codes. Generally, the more complex and general a language becomes, the more difficult the compilation and debugging becomes.)

An Old, Trusty Example

This article assumes a basic knowledge of compiler theory and assembly language. For more information, read *Compilers: Principles, Techniques and Tools* by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman [Addison-Wesley, 1985]. It's the "Dragon Book" — the bible of compiler theory. For 386 assembler, almost any introductory text will do. Peter Norton puts out a pretty good book on Intel Assembler; for a full treatment, read *Assembly Language Programming for the Intel 80XXX Family* by William B. Giles [Prentice Hall, 1991].

To illustrate the technique without getting bogged down in the details, we'll create a simple version of a classic computer-science homework assignment, "The Formula Compiler."

Our Formula Compiler

Our example will be very limited; it will do calculations only on the IEEE floating-point data type (the Delphi Double), and handle only addition, subtraction, multiplication, and division. The compiler will be a simple, recursive-descent parser, with an emphasis on code generation, not parsing. The techniques presented here, however, can be generalized for powerful languages. Of course, the compiler will handle the expressions in correct mathematical precedence.

Two key elements. Users must be able to specify the formula via an edit box or some other mechanism at run time. They also need access to internally-held variables in the compiled program with which they are interacting. For example, when creating a graphing function for an X-Y coordinate system, users would need access to the X and Y variables the program will be plotting, in order to enter their formulas. They also need a place to enter the formula to be compiled and used by the graphing function. If users are allowed to enter formulas that act on internal fields and variables, they would need access to the names of those variables. Essentially, the user-defined, run-time formula must interact with the static, compile-time variables. A symbol table manager handles this.

Who Writes Machine Code, Anyway?

Take a look at the Turbo Assembler quick reference guide, or any x86 assembler reference. In it are cryptic entries such as:

```
OpCode        Instruction          Clocks     Description
$81 /0 id     ADD r/m32, imm32     ...        ...
```

The *OpCode* on the left is the machine code that represents the *instruction* (in the Instruction column). If the computer's instruction pointer were currently pointing at a memory location containing $81, and it was to execute, the CPU would know to perform an ADD instruction, and the remaining bytes immediately after the $81 would tell it what and how to add. To perform this operation, generate a $81 hex followed by the mod r/m and the immediate value as a 4-byte integer. The mod r/m is an indicator to tell whether the operation is to memory, or to a register, and what memory location or what register. It's not difficult to generate the machine code by hand, but it's tedious.

Fortunately, there's an easier way. I don't think in, or write, machine code, so I cheat. Turbo Debugger has a built-in facility that makes this step easy. Load the Turbo Debugger, select **File | Open**, then open any executable program (even if it wasn't created in Delphi). If the program wasn't compiled with Turbo Debugger information, the message "Program has no symbol table" will be displayed. Click on the **OK** button if that message box appears. If the selected program has Turbo Debugger information embedded, the debugger will show source code instead of CPU instructions. In that case, select **View | CPU** to switch from source-level to CPU-level debugging. A screen similar to Figure 1 will appear immediately after opening the file and viewing the CPU window.

Now, to see how the assembly statement would be represented in machine code, enter it into the debugger:

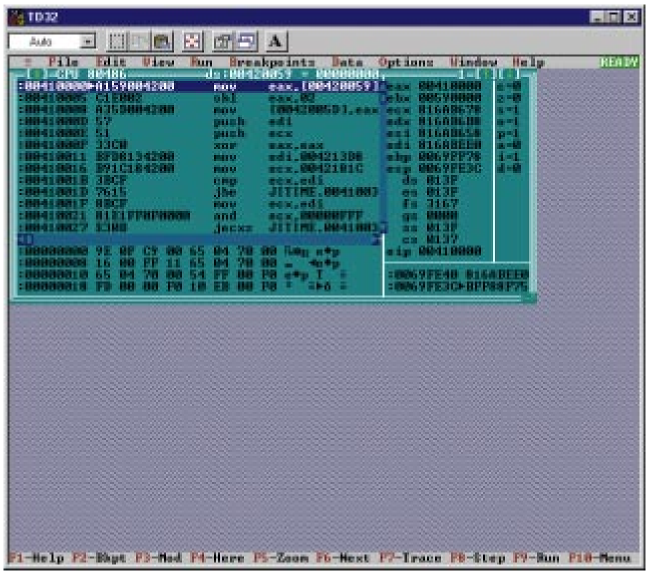
```
ADD EDX, 200
```

By pressing A, an input box is displayed with the character "A" already in it. Enter the rest of the ADD statement as shown in Figure 2. After pressing Enter, the assembly statement is transformed into machine code.

The first column in the CPU window is the instruction pointer for the machine, i.e. the location of the add instruction in memory (:00410000). The next column contains the machine code bytes equivalent to ADD EDX, 00000200. The final column is the assembler statement itself. So, in our example, we have:

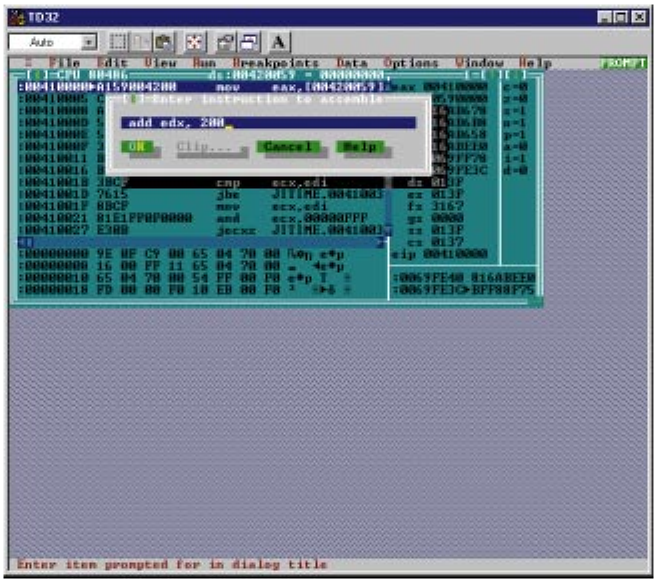
```
:00410000     81C200020000     add edx, 00000200
```

The $81 byte is the add instruction we saw earlier in the Turbo Assembler reference. The $C2 byte indicates the EDX register, and $00200000 is the 32-bit value represent-



```
[■]=CPU 80486=============ds:00420059 = 00000000=========1=[↑][↓]=
:00410000►A159004200   mov   eax,[00420059]  eax 00410000  c=0
:00410005 C1E002       shl   eax,02          ebx 00590000  z=0
:00410008 A35D004200   mov   [0042005D],eax  ecx 816A8678  s=1
:0041000D 57           push  edi             edx 816A8608  o=1
:0041000E 51           push  ecx             esi 816A8658  p=1
:0041000F 33C0         xor   eax,eax         edi 816ABEE0  a=0
:00410011 BFD8134200   mov   edi,004213D8    ebp 0069FF78  i=1
:00410016 B91C184200   mov   ecx,0042181C    esp 0069FE3C  d=0
:0041001B 3BCF         cmp   ecx,edi          ds 013F
:0041001D 7615         jbe   JITIME.0041003   es 013F
:0041001F 8BCF         mov   ecx,edi          fs 3167
:00410021 81E1FFFF0000 and   ecx,0000FFFF     gs 0000
:00410027 E308         jecxz JITIME.0041003   ss 013F
                                              cs 0137
:00000000 9E 0F C9 00 65 04 70 00            eip 00410000
:00000008 16 00 FF 11 65 04 70 00
:00000010 65 04 70 00 54 FF 00 F0           :0069FE40 816ABEE0
:00000018 F0 00 00 F0 10 E8 00 F0           :0069FE3C►BFF88F75

F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 1: Turbo Debugger.



```
TD32
  File  Edit  View  Run  Breakpoints  Data  Options  Window  Help      PROMPT
[ ]-CPU 80486               ds:00420059 = 00000000
:00410000►A159004200    mov   eax,[00420059]
   [ ]-Enter instruction to assemble
   add edx, 200
   OK   Clip...   Cancel   Help
:00410018 3BCF          cmp   ecx,edi
:0041001D 7615          jbe   JITIME.0041003
:0041001F 8BCF          mov   ecx,edi
:00410021 81E1FF0F0000  and   ecx,00000FFF
:00410027 E300          jecxz JITIME.0041003
:00000000 9E 0F C9 00 65 04 70 00
:00000008 16 00 FF 11 65 04 70 00
:00000010 65 04 70 00 54 FF 00 F0
:00000018 F0 00 00 F0 10 EB 00 F0
eax 00410000   c=0 z=0 s=1 o=1 p=1 a=0 i=1 d=0
ds 013F  es 013F  fs 3167  gs 0000  ss 013F  cs 0137  eip 00410000
:0069FE40 816ABEE0
:0069FE3C►BFF88F75
Enter item prompted for in dialog title
```

Figure 2: The instruction entry screen.

ing the hex number 200. (Remember, on the Intel platform, words are stored backwards.)

How Do We Allocate the Code Space?

Now that there is an easy method to quickly access the machine code for any assembly statement, we need an executable memory block to hold the code to be generated. In Windows 95, you can allocate a block of memory using *GetMem*, assign a function pointer to it, and call the function directly. That's nice, but not very safe. Fortunately, Windows NT is a bit more protective. If you tried the Windows 95 approach, a protection violation would be issued under NT. In NT, a memory block must be allocated with the *VirtualAlloc* function, which contains more details as to the memory's exact use. The code in the article will work in both Windows 95 and NT, and is the safest code for performing this type of technique. In NT, it's necessary to declare the memory as executable, as well as readable and writeable.

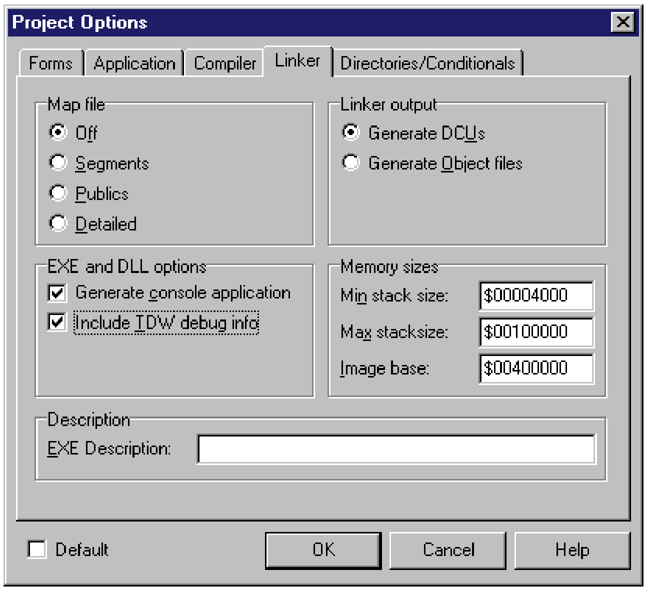
VirtualAlloc specifies the size of the memory space to allocate, as well as the attributes of the memory block. In Figure 3, the memory block is allocated as an executable, readable, and writeable block of memory. PAGE_EXECUTE_READWRITE is a flag to indicate to Windows NT not only to read and write to this memory location, but later to actually allow executing it as code. It returns a pointer to an executable memory block of size *codeSize*. This pointer to memory can be used just as any other pointer variable would be used.

Generating Good Code

So far, this article has demonstrated translating assembly instructions to machine code, and how to allocate an executable block of memory. The only thing left is to generate code for the application. Ideally, the run-time formulas should execute as fast as if they had been compiled directly into Delphi. But, instead of being compiled statically, they will be dynamic and user-defined well after compilation.

```
fCodeBuff :=
  VirtualAlloc(nil, codeSize, MEM_COMMIT + MEM_RESERVE,
               PAGE_EXECUTE_READWRITE);
```

Figure 3: Virtual allocation of a code page.



Project Options

Forms | Application | Compiler | Linker | Directories/Conditionals

Map file
- (●) Off
- ( ) Segments
- ( ) Publics
- ( ) Detailed

Linker output
- (●) Generate DCUs
- ( ) Generate Object files

EXE and DLL options
- [✓] Generate console application
- [✓] Include TDW debug info

Memory sizes

Min stack size:	$00004000
Max stacksize:	$00100000
Image base:	$00400000

Description

EXE Description:

[ ] Default   OK   Cancel   Help

Figure 4: The Project Options dialog box for our test program.

By examining how Delphi generates a formula, we can size up the machine code involved. If the code Delphi generated isn't efficient, a better algorithm or method can be substituted and typed into the debugger (as shown in Figure 2). For this example, a small console application is created, containing all the operators to be implemented: addition, multiplication, subtraction, and division. The application is compiled with Turbo Debugger information embedded, and pulled into Turbo Debugger for examination. By doing this, the source code and the associated assembly language generated by Delphi are viewed together. The project options are shown in Figure 4.

I knew Delphi would do some optimizations over and above what I would handle in this simple example. So, I put some floating-point assembly-language code inside an **asm** block I knew I would need in the code generation process. By doing this, I can translate the other needed instructions at the same time, instead of typing them in separately. Don't execute this code! It would most likely result in a nasty crash. Now, with a basic understanding of how Delphi generates the code, an efficient formula parser can be implemented.

Before proceeding, make sure the compiler tab has all three **Debugging** options checked. With the sample code in Figure 5, the machine code needed to implement the next part of our formula parser is displayed. First, compile this program; go into Turbo Debugger and select **File | Load** for the executable. Then select **View | CPU** and begin stepping through the code until the call to *Eval* in Figure 5. At this point, use F7 to step into the function. A display similar to Figure 6 will appear.

```
program ConsoleApp;

uses
  Forms;

{$R *.RES}

var
  a,b,c,d, tmpResult : Double;

function Eval : Double;
begin
  b := a * b + c * d;
  asm
    FMULP     ST(1), ST
    FDIVP     ST(1), ST
    FADDP     ST(1), ST
    FSUBP     ST(1), ST
  end;
  Result := a + b;
end;

begin
  b := 12.445;
  c := 13.345;
  d := 445014.112;
  tmpResult := Eval;
  WriteLn(tmpResult);
end.
```

Figure 5: The sample code.

Here, Delphi generates some pretty simple code. It loads the EAX register with the address of the variable *A*. It uses a combination of FLD, FSTP, FADD, FDIV, FMUL, and FSUB. These operators load values on the floating-point stack from a memory pointer, pop the values off the stack to a memory address, and perform binary arithmetic operations on the floating-point stack. Most assembly language texts will give a complete treatment of these floating-point operators. Now all the information is present to generate a user-defined function in machine code.

Each function in Delphi has function entry and exit code. All the entry and exit code information can simply be copied directly to the formula compiler. All Delphi functions must adhere to this standard calling methodology as long as they are not **stdcall** type functions. Delphi subtracts eight from the stack pointer, and loads the EAX register with the address of *A* upon entry into this function. It then performs several floating-point stack operations, and finally, moves the result to the ESP stack, as well as pushing the result on to the floating-point stack, the code after:

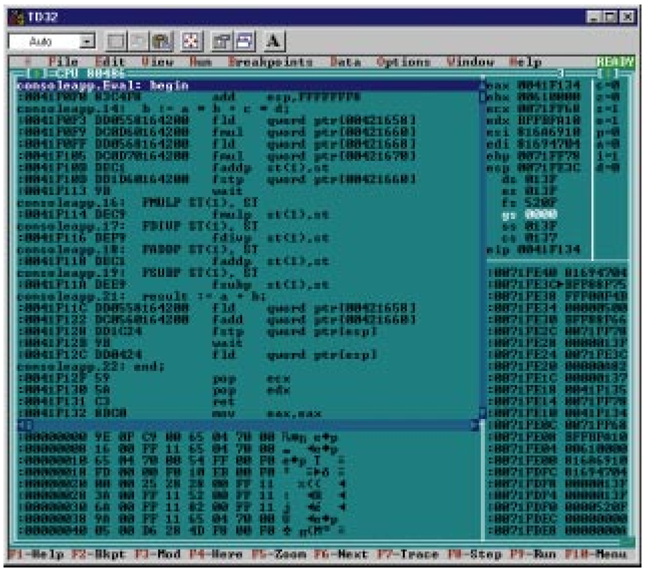
```
Result := a+b
```

With the value on the floating-point stack, the function result could easily be used in an expression instead of just an assignment. Finally, it pops off two four-byte integers, so the stack frame is returned to the original setting before calling. The user-defined procedure generated in the formula compiler will need to mimic the entry and exit code of the function, and, based on the user formula, generate machine code similar to that in Figure 6, between the entry and exit code.

Take one of the instructions as an example of how to use the machine code:

```
DD0560164200      fld      qword ptr [00421660]
```

The first two bytes are the FLD (Load Real) instruction ($DD followed by $05). The remaining bytes are the address of the Double variable push onto the floating-point stack. To


TD32
Auto
File Edit View Run Breakpoints Data Options Window Help
READY
CPU 80486
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
consoleapp.Eval: begin
:0041F0F0 83C4F8            add     esp,FFFFFFF8
consoleapp.14:  b := a * b + c * d;
:0041F0F3 DD0558164200      fld     qword ptr[00421658]
:0041F0F9 DC0D60164200      fmul    qword ptr[00421660]
:0041F0FF DD0568164200      fld     qword ptr[00421668]
:0041F105 DC0D70164200      fmul    qword ptr[00421670]
:0041F10B DEC1              faddp   st(1),st
:0041F10D DD1D60164200      fstp    qword ptr[00421660]
:0041F113 9B                wait
consoleapp.16:   FMULP ST(1), ST
:0041F114 DEC9              fmulp   st(1),st
consoleapp.17:   FDIVP ST(1), ST
:0041F116 DEF9              fdivp   st(1),st
consoleapp.18:   FADDP ST(1), ST
:0041F118 DEC1              faddp   st(1),st
consoleapp.19:   FSUBP ST(1), ST
:0041F11A DEE9              fsubp   st(1),st
consoleapp.21:   result := a + b;
:0041F11C DD0558164200      fld     qword ptr[00421658]
:0041F122 DC0560164200      fadd    qword ptr[00421660]
:0041F128 DD1C24            fstp    qword ptr[esp]
:0041F12B 9B                wait
:0041F12C DD0424            fld     qword ptr[esp]
consoleapp.22: end;
:0041F12F 59                pop     ecx
:0041F130 5A                pop     edx
:0041F131 C3                ret
:0041F132 8BC0              mov     eax,eax
```

Figure 6: Mixed view of source and generated code.

create this instruction in memory, write out the first two bytes for the instruction, then the address of the variable to push. When the machine's instruction pointer (IP) comes to that instruction ($DD,$05), it will push the value contained at that memory location to the floating-point stack, just as if Delphi had generated it.

So, all the building blocks necessary to generate user-defined code are now in place. The only thing that remains is the formula parser and code generator. The formula compiler's code model will push the left side, then the right side of each expression to the floating-point stack. It will perform the arithmetic operation on the top two stack elements and push that result back to the floating-point stack. For example, in the expression x + y, the value for x, then the value for y, would be pushed onto the stack. The + operation would be performed on the values for x and y, and the resulting value would be pushed back to the floating-point stack. This is a traditional post-fix stack execution model. It's not always the most effective code, but because the floating-point processor on a Pentium takes one or three clock cycles to perform these instructions, the code will be quick. Optimization is outside the scope of this article, so it's just brute force for now. Even this brute-force method, however, would outperform the cleverest interpreted scheme.

The Code Generator

Code generation is the simplest of the two remaining sections. Machine code must be output and space reserved for the user constants and variables. The symbol table is key to sharing compile-time variables with the user's run-time formula.

The symbol table entries are basic. Pass in the name and address of the compile-time variable. Those variables are then ready to be processed against a user-entered formula. The compiler has memory-address knowledge, through the symbol table, of the variables inside the compiled program, and therefore can access and set the values in those variables. Figure 7 shows the class

```
TCodeGenerator = class(TObject)
  fCodeBuff, fCodeBuffEnd,
  fDataBuffPtr, fCurrCodeBuffOfs : Pointer;
  fCodeBuffSize : Integer;

  public
    constructor Create(const codeBuffSize : Integer);
    destructor Destroy; override;

    function GenBytes(
      const bytes : array of Byte) : Boolean;
    function GenWords(
      const words : array of Word) : Boolean;
    function GenQuads(
      const quads : array of Cardinal) : Boolean;
    function AllocDataSpace(numBytes : Longint) : Pointer;
    function GetFunctionAddress : Pointer;
  end;
```

Figure 7: The code generator interface.

definition of the code generator. Basically, the constructor reports the combined size of the code and data space. In parsing the formula, the compiler will begin to generate bytes, words, and quad-words to the code space, and sub-allocate data space from the block of memory allocated for the machine code. By calling the *Gen...* functions, the machine code is generated as the expression is parsed. When done with this step, only the exit code and return must be generated before the code is ready to execute as a standard Delphi function.

The *AllocDataSpace* function allocates a block of *numBytes* size from the end of the code buffer. The code is generated from the beginning of the block to the end. The data is allocated from the end of the block to the beginning. As long as these two pointers don't overlap, there is enough space to generate code and data. If the program runs out of space, the *Gen...* functions return *False*. Of course, it would be nice to have some simple error recovery functions. It's not difficult, but it's left as a problem for the reader. The *Create* constructor uses the *VirtualAlloc* function, described previously, to allocate an executable memory block. Finally, calls to *GetFunctionAddress* get the pointer to the memory block containing the generated machine code, so a function pointer can be assigned to the newly generated code, and subsequently be called and executed. The implementation code is shown in Listing Two, beginning on page 20.

The Parser and Scanner

A parser and scanner could be written from scratch, but it turns out the Dragon Book provides an adequate parser and scanner. The only remaining task is to integrate the code generation and the data-space allocation. The parser takes the in-fix formula, parses it, then produces post-fix code. Post-fix code pushes both operands to a stack, then generates the binary operator to act on the stack elements. In other words, the operator is evaluated after (post) the operands. Except for a few changes to make the parser/lexer more efficient and "Delphi-izing" the code, most of the original Dragon Book code for parsing and lexing is retained. Instead of outputting a character as in the Dragon Book, the equivalent machine code is generated.

Due to space limitations, the lexer is greatly simplified. In reality, a tool like Lex, Yacc, or Visual Parse++ would be used to put together a real grammar, parser, and lexical analyzer. But, for this example, a quick and dirty, recursive-descent parser is best for illustration. Note the glaring omission of almost all error-checking and recovery code. This is also left as a problem for the reader, as there is not sufficient space to include it. Full source code for this article is available (see end of article for details).

It's best to explain this overall process through an example. Take the expression A+B. The parser would push the value in A, then the value for B to the floating-point stack. The + operator generates an FADDP ST(1), ST that takes the top two values off the floating-point stack, adds them, then pushes the resultant value back to the floating-point stack for the next operation. The entry and exit code for the function is the same as the code generated by Delphi. The actual code generated for the mathematical operators is quite small. In fact, most of the code is spent in the mundane task of lexing the input stream into recognizable tokens. This grammar is rudimentary; it doesn't handle the negate operator or the power operator, let alone mathematical functions such as Sine, Cosine, etc., but this article is not about how to write grammars.

Running the Formula

To execute the code, a function pointer is needed to assign to the memory. This particular function pointer must point to a function that returns a Double:

```
type
  TDoubleFunction = function : Double;

var
  funcPtr : TDoubleFunction;
  tmpDouble : Double;
```

Now that the types are declared, all that remains is assignment and execution. To assign the function pointer, cast the pointer returned by the *GetFunctionPointer* method to type *TDoubleFunction*:

```
FuncPtr := TFunctionType(codeGenObj.GetFunctionPointer);
```

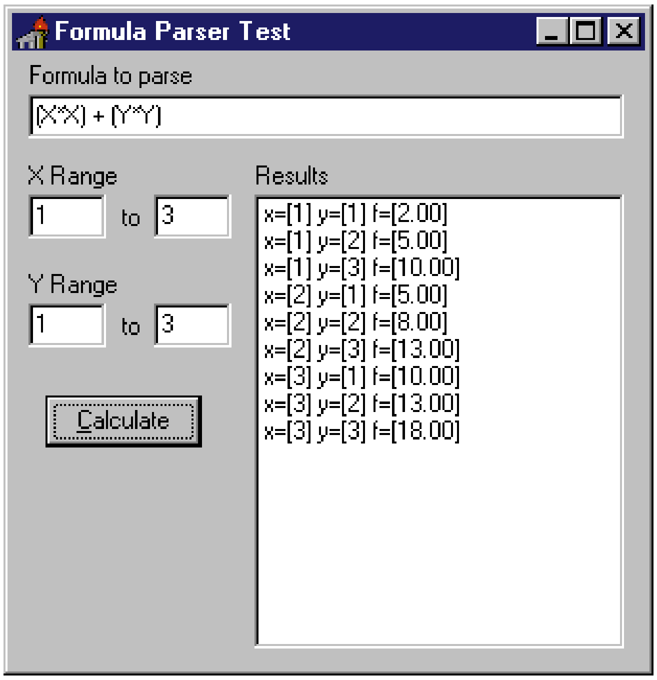
And execute the function:

```
tmpDouble := FuncPtr;
```

Now if the run-time formula will have access to internal variables, it must use the previous symbol table functions to set them (before the compilation process). A symbol table consists of each variable name and its address. Right after the parser is created, but before the user's formula is compiled and executed, add the internal variables and their respective addresses to the symbol table:

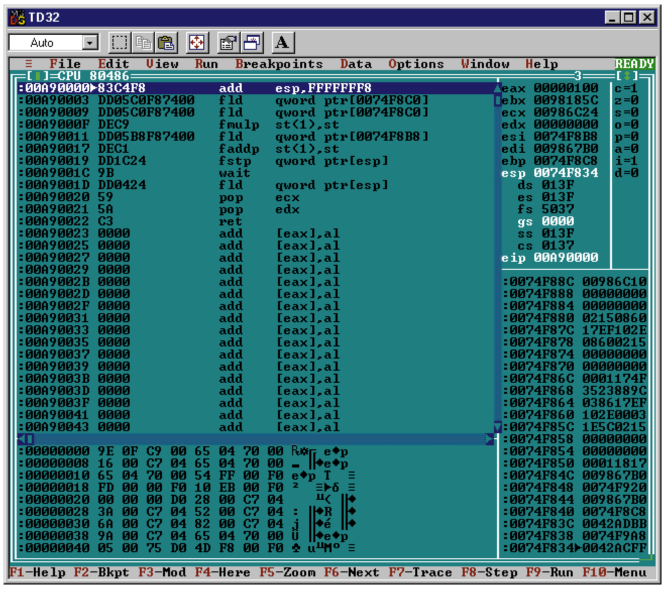
```
aParser := TParser.Create;

aParser.AddSymbol('X', @x);
aParser.AddSymbol('Y', @y);

if aParser.ParseFormula('x+y * 2');
  ...
```

Because this example uses the variables *x* and *y*, and uses the address of the variables (not the values), the values don't need



```
Formula Parser Test

Formula to parse
(X*X) + (Y*Y)

X Range            Results
1      to  3       x=[1] y=[1] f=[2.00]
                   x=[1] y=[2] f=[5.00]
                   x=[1] y=[3] f=[10.00]
Y Range            x=[2] y=[1] f=[5.00]
1      to  3       x=[2] y=[2] f=[8.00]
                   x=[2] y=[3] f=[13.00]
                   x=[3] y=[1] f=[10.00]
  Calculate        x=[3] y=[2] f=[13.00]
                   x=[3] y=[3] f=[18.00]
```

Figure 8: The user input screen.



```
TD32
 ≡  File  Edit  View  Run  Breakpoints  Data  Options  Window  Help          READY
[ ]=CPU 80486
:00A90000▸83C4F8          add    esp,FFFFFFF8            eax 00000100  c=1
:00A90003 DD05C0F87400    fld    qword ptr[0074F8C0]     ebx 0098185C  z=0
:00A90009 DD05C0F87400    fld    qword ptr[0074F8C0]     ecx 00986C24  s=0
:00A9000F DEC9            fmulp  st(1),st                edx 00000000  o=0
:00A90011 DD05B8F87400    fld    qword ptr[0074F8B8]     esi 0074F8B8  p=0
:00A90017 DEC1            faddp  st(1),st                edi 009867B0  a=0
:00A90019 DD1C24          fstp   qword ptr[esp]          ebp 0074F8C8  i=1
:00A9001C 9B              wait                           esp 0074F834  d=0
:00A9001D DD0424          fld    qword ptr[esp]            ds 013F
:00A90020 59              pop    ecx                       es 013F
:00A90021 5A              pop    edx                       fs 5037
:00A90022 C3              ret                              gs 0000
:00A90023 0000            add    [eax],al                  ss 013F
:00A90025 0000            add    [eax],al                  cs 0137
:00A90027 0000            add    [eax],al                eip 00A90000
:00A90029 0000            add    [eax],al
:00A9002B 0000            add    [eax],al                :0074F88C 00986C10
:00A9002D 0000            add    [eax],al                :0074F888 00000000
:00A9002F 0000            add    [eax],al                :0074F884 00000000
:00A90031 0000            add    [eax],al                :0074F880 02150860
:00A90033 0000            add    [eax],al                :0074F87C 17EF102E
:00A90035 0000            add    [eax],al                :0074F878 08600215
:00A90037 0000            add    [eax],al                :0074F874 00000000
:00A90039 0000            add    [eax],al                :0074F870 00000000
:00A9003B 0000            add    [eax],al                :0074F86C 0001174F
:00A9003D 0000            add    [eax],al                :0074F868 3523889C
:00A9003F 0000            add    [eax],al                :0074F864 038617EF
:00A90041 0000            add    [eax],al                :0074F860 102E0003
:00A90043 0000            add    [eax],al                :0074F85C 1E5C0215
                                                         :0074F858 00000000
:00000000 9E 0F C9 00 65 04 70 00                        :0074F854 00000000
:00000008 16 00 C7 04 65 04 70 00                        :0074F850 00011817
:00000010 65 04 70 00 54 FF 00 F0                        :0074F84C 009867B0
:00000018 FD 00 00 F0 10 EB 00 F0                        :0074F848 0074F920
:00000020 00 00 00 D0 28 00 C7 04                        :0074F844 009867B0
:00000028 3A 00 C7 04 52 00 C7 04                        :0074F840 0074F8C8
:00000030 6A 00 C7 04 82 00 C7 04                        :0074F83C 0042ADBB
:00000038 9A 00 C7 04 65 04 70 00                        :0074F838 0074F9A8
:00000040 05 00 75 D0 4D F8 00 F0                        :0074F834▸0042ACFF
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 9: The generated code.

to be initialized in any other way than just setting them in the Pascal code. For example:

```
for x := 1 to 100 do begin
  for y := 1 to 100 do begin
    tmpDouble := FuncPtr;
  end;
end;
```

Because the compiler is accessing *x* and *y* via their address in the function routine, the assignment in the Delphi **for** loop is sufficient to evaluate the formula versus the current values in *x* and *y*.

Reviewing the Code

Once the code is generated, take a look at it. When the TestFormParser program is loaded into the Turbo Debugger and run, the form in Figure 8 will be displayed. Step into the code and see the results. First, enter:

```
x*x + y
```

into the formula input box. Next, set a breakpoint at this statement:

```
tDouble := funcPtr;
```

This code is located in the *OnClick* event for the Calculate button in the inner-most loop. By setting the breakpoint here, the debugger will stop short of the newly generated routine. Now, run the program by pressing the Calculate button. The debugger will break at the line above. Select View | CPU in Turbo Debugger and switch to mixed-mode source and assembly. Press F7 to step into the generated code. A screen similar to that shown in Figure 9 will appear.

First is the generated function entry code, ADD ESP, FFFFFFF8. Next is the push of the variable *X* twice and a call to FMULP, followed by a push of the *Y* variable and a call to FADDP. Finally, the resultant value is pushed to the stack, and the function exit code is executed. The RET function returns to the function call in the *OnClick* event. It is now an official compiler! It has also produced some pretty efficient code — not quite to the level of Delphi, but close.

Where to Go from Here

Okay, now that the basic techniques have been illustrated, how useful is this? Whenever an application needs user-dynamic interaction and speed, this technique provides a powerful solution. Compiled code will typically yield three to five times the speed advantage over an interpreted version.

Dynamic compilation is a great encapsulation tool, and yields dramatic performance and flexibility advantages to users. Its simple language interface means that application programmers don't need to understand any of the implementation details. Just the simple formula compiler we have built can be embedded into any other application that needs user-defined formulas.

As you investigate this technique, you'll notice that you have to handle problems such as general register allocation and stack and temporary name handling, as well as relative jump addressing and near/long jumps, among others. These are not difficult topics (save register allocation), but they do take time to implement — and are certainly beyond the scope of a magazine article. In fact, when I do technical interviews, I use the forward and backward jump references as an interview question to see the thought processes of potential job candidates. It is not a trivial problem, but certainly solvable within a few days of coding.

My recommendation is to start small, with the formula parser. Begin to add small functions as the application's needs expand. These compiler tools and objects will evolve out of necessity. Soon, all the tools necessary to generate even a "full" language compiler such as Delphi or C/C++ will be available. It takes some time, but small steps add up quickly. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\OCT\DI9710JC.

Jay Cole is the owner of OpNek Research, a consulting and Financial Markets firm that consults for many Fortune 500 companies. OpNek also produces the high-speed sort utility NitroSort. For more information on OpNek or NitroSort, visit http://www.nitrosort.com.

Begin Listing Two — Code Generator Implementation

```
implementation

uses SysUtils, Dialogs;

const
  cTokToStr : array[TTokenType] of string[50] =
     ('+', '-', '*', '/', 'Number', 'None', 'Error',
      'Id', '(', ')', 'Done');
type
  TRefObj = class(TObject)
  public
    fAddr : Pointer;
  end;

constructor TParser.Create;
begin
  // Create the symbol table and an 8KB
  // code/data space for our formula.
  fSymbolTable := TStringList.Create;
  fCodeGen := TCodeGenerator.Create(8192);
end;

destructor TParser.Destroy;
begin
  fSymbolTable.Free;
  fCodeGen.Free;
end;

function TParser.LookUp(tokenStr : string;
  var addrVal : Pointer) : Boolean;
var
  ndx : Integer;
begin
  ndx := fSymbolTable.IndexOf(tokenStr);
  if (ndx >= 0) then begin
     addrVal := (fSymbolTable.Objects[ndx] as TRefObj).fAddr;
     Result := True;
  end else begin
    Result := False;
    addrVal := nil;
  end;
end;

procedure TParser.AddSymbol(sym : string; addr : Pointer);
  var refObj : TRefObj;
begin
  refObj := TRefObj.Create;
  refObj.fAddr := addr;
  fSymbolTable.AddObject(UpperCase(sym), refObj);
end;

procedure TParser.DeleteSymbol(sym : string);
begin
  fSymbolTable.Delete(
    fSymbolTable.IndexOf(UpperCase(sym)));
end;

function TParser.GetChar : char;
begin
  if (fCurrPos <= Length(fInputStr)) then
    begin
      Result := fInputStr[fCurrPos];
      Inc(fCurrPos);
    end
  else
    Result := Chr(0);
end;

procedure TParser.UnGetChar;
begin
  Dec(fCurrPos);
end;
function TParser.Lexan : TTokenType;
  var
    t : char;
    tokenStr : string;
begin
  Result := ttNone;
  while True do begin
    t := GetChar;
    t := UpCase(t);
    if (t in [' ', Chr(9)]) then begin
      // Skip the white space.
      while (t in [' ', Chr(9)]) do
        t := GetChar;
      UnGetChar;
    end else if (t in ['0'..'9']) then begin
      // We have a number.
      tokenStr := '';
      while (t in ['0'..'9', '-', '.']) do
        begin
          tokenStr := tokenStr + t;
          t := GetChar;
        end;
      UnGetChar;

      try
        Result := ttNumber;
        fFloatVal := StrToFloat(tokenStr);
      except
        on EConvertError do Result := ttError;
      end;

    Break;
    end else if (t in ['A'..'Z']) then begin
      // We have a character, so we have an ID.
      tokenStr := '';
      while (t in ['A'..'Z','0'..'9']) do
        begin
          tokenStr := tokenStr + t;
          t := GetChar;
          t := UpCase(t);
        end;
      UnGetChar;

      if (LookUp(tokenStr, fAddrVal)) then
        Result := ttId
      else
        Result := ttError;

      Break;
    end else
      case t of
        '+' : Result := ttAdd;
        '-' : Result := ttSubtract;
        '/' : Result := ttDivide;
        '*' : Result := ttMultiply;
        '(' : Result := ttLParen;
        ')' : Result := ttRParen;
        Chr(0) : Result := ttDone;
      else
        Result := ttNone;
    Break;
  end;
end;

procedure TParser.SyntaxError(anErrorStr : string);
begin
  fError := True;
  MessageDlg(anErrorStr, mtError, [mbOk], 0);
end;

function TParser.ParseFormula(aFormula : string) :
Boolean;
begin
  // Generate the entry code.
  fError := False;
  fInputStr := aFormula;
  fCurrPos := 1;
  fCodeGen.GenBytes([$83, $c4, $f8]); // ADD ESP,
$FFFFFFF8

  lookAhead := Lexan;
  if (lookAhead <> ttDone) then Expr;

  // Generate the exit code. We must have space to store
  // result, and that space is the ESP position that we
  // backed up. We also need to leave the result on the
  // floating-point stack.
  fCodeGen.GenBytes([$dd,$1c,$24]); // FSTP QWORD PTR
[ESP]
  fCodeGen.GenBytes([$9B]);         // WAIT
  fCodeGen.GenBytes([$dd,$04,$24]); // FLD  QWORD PTR
[ESP]

  // Pop off the 8 stack bytes and return.
  fCodeGen.GenBytes([$59]);         // POP  ECX
  fCodeGen.GenBytes([$5a]);         // POP  EDX
  fCodeGen.GenBytes([$c3]);         // RET

  Result := not fError;
end;

procedure TParser.Expr;
begin
  Term;
  while (True) do begin
    if  (lookAhead = ttAdd) then begin
      Match(ttAdd); Term;
      // Generate Add code.
      fCodeGen.GenBytes([$de,$c1]);  // FADDP ST(1), ST
    end else if (lookAhead = ttSubtract) then begin
      Match(ttSubtract); Term;
      // Generate Subtract Code.
      fCodeGen.GenBytes([$de,$e9]);  // FSUBP ST(1), ST
    end else
      Break;
  end;
end;

procedure TParser.Term;
begin
  Factor;
  while (True) do begin
    if (lookAhead = ttMultiply) then begin
      Match(ttMultiply); Factor;
      // Generate the multiply code.
      fCodeGen.GenBytes([$de,$c9]);  // FMULP ST(1), ST
    end else if (lookAhead = ttDivide) then begin
      Match(ttDivide); Factor;
      // Generate the divide code.
      fCodeGen.GenBytes([$de,$f9]);  // FDIVP ST(1), ST
    end else
      Break;
  end;
end;

procedure TParser.Factor;
var
  tmpPtr : ^double;
begin
  if (lookAhead = ttLParen) then begin
    Match(ttLParen); Expr;
    Match(ttRParen);
  end else if (lookAhead = ttNumber) then begin
    // Add the number to the data space and then generate
    // the load code for that number.
    tmpPtr := fCodeGen.AllocDataSpace(SizeOf(fFloatVal));
    tmpPtr^ := fFloatVal;
    fCodeGen.GenBytes([$dd,$05]);
    // FLD QWORD PTR [tmpPtr]
    fCodeGen.GenQuads([Cardinal(tmpPtr)]);
    Match(ttNumber);
  end else if (lookAhead = ttId) then begin
    // Get the ID's address. Generate the push for the
    // value held in the Id address space.
    fCodeGen.GenBytes([$dd,$05]);
    // FLD QWORD PTR [@id]
    fCodeGen.GenQuads([Cardinal(fAddrVal)]);
    Match(ttId);
  end else begin
    // We have a syntax error. Let the user know.
    SyntaxError('Unrecognized token at ' +
                IntToStr(fCurrPos));
  end;
end;

procedure TParser.Match(t : TTokenType);
begin
  if (lookAhead = t) then
    lookAhead := Lexan
  else begin
    // Syntax Error
    SyntaxError('Expecting '+cTokToStr[t]);
  end;
end;

procedure TParser.GetProcAddress(
  var funcPtr : TDoubleFunction);
var
  ptr : Pointer;
begin
  ptr := fCodeGen.GetFunctionPointer;
  funcPtr := TDoubleFunction(ptr);
end;
```

End Listing Two

InterBase / Delphi

By Bill Todd

What's Multi-Generational Architecture, Anyway?

Inside InterBase: Part V

This final installment of the InterBase series exposes what really sets InterBase apart from other database servers: its multi-generational, or *versioning*, architecture. Most new technologies are developed to solve a problem posed by a previous technology, and the InterBase versioning architecture is no exception. Before exploring versioning, it's important to look at transaction processing (the previous technology, in this case), how it works, and why you need it.

The Complete Transaction

A *transaction* is a logical unit of work that consists of one or more changes — to one or more tables in a database — that must all succeed or fail as a unit. Why do you need transactions? Here's the example that appears in every database text: You go to your ATM to transfer $1,000 from your savings account to your checking account. To do this, the software must make two changes to the database. It must decrease the balance in your savings account, and increase the balance in your checking account. A problem arises if the database software decreases the balance in your savings account, but crashes before it increases the balance in your checking account. At this point, you have lost $1,000.

You want both changes to succeed, or neither; that's what transaction processing ensures. The software would start a transaction, reduce the balance in your savings account, increase the balance in your checking account, then *commit* the transaction. Committing makes all the transaction's changes a permanent part of the database. Up to the moment the transaction is committed, you can roll it back and undo the changes. If the database server crashes before the transaction is committed, the transaction automatically will be rolled back when the server restarts.

Dwelling in Isolation

When you work with transactions, picking a *transaction-isolation level* is critical. It controls how — or if — it sees changes made by other transactions. The *TransIsolation* property of Delphi's Database component lets you choose one of three levels.

The first isolation level is *tiDirtyRead*. If you choose this isolation level, your transaction can see every change made to the database by other transactions — including those that haven't yet committed. This is undesirable, because it allows your transaction to read a value that may yet be rolled back, rendering that value incorrect. That's why many servers don't support "dirty reads."

The second isolation level is *tiReadCommitted*, which allows your transaction to see only those values written to the database by transactions that have committed. This ensures that the values used have become a permanent part of the database. However, it doesn't protect your transaction from reading a value, then having another transaction change the value — again, rendering it incorrect.

The third and most restrictive isolation level is *tiRepeatableRead*. With repeatable read, your transaction is guaranteed that a value will remain constant, no matter how many times it's read from the database. Another way of looking at repeatable-read isolation is that it gives your transaction a snapshot of the database at a moment in time, and that's the way your transaction will see the database throughout its life. But a database server built on the locking model can provide repeatable-read isolation only by preventing other transactions from updating the database until your transaction commits or rolls back.

Not all database servers support all these isolation levels. To determine what each level maps to for your server, see the Delphi online Help for the *TransIsolation* property. InterBase doesn't support dirty reads; selecting *tiDirtyRead* with InterBase will produce the read-committed level.

The Case of the Phantom Platinum

I asserted that the versioning architecture used in InterBase was developed to solve a problem. To understand the problem, consider the case of a large business whose many warehouses are scattered across the country. One of the features in this company's new executive-information system is the ability to generate an inventory-valuation report by warehouse. This allows the president and CFO to see the exact value of the goods in each warehouse. But the report may not be accurate if the data is stored in a database server that uses the older locking architecture. To understand why, consider what happens when this report is run.

The first and most important step in preparing the report is to run a query that totals the value of all the items by warehouse. If the database is large, this query will take some time to run. Suppose the database server begins running the query, and totals the value of the inventory in the Albany warehouse, then moves on to Buffalo and then to Chicago. While the server is totaling the value for the Chicago warehouse, someone in Albany commits a transaction that transfers 10,000 platinum bars from Albany to El Paso. Meanwhile, the query finishes Chicago, moves on to Denver, and finally arrives at El Paso. The result is that the 10,000 bars of platinum have been counted twice. Of course, there would be no problem if the query transaction used repeatable-read transaction isolation. Unfortunately, the only way to provide repeatable read with a database server that uses locking architecture is to lock the tables involved in the query, so that no changes are made while the query is running. While this solves the report problem, it imposes severe restrictions on concurrent database access by multiple users.

The Versioning Solution

The solution to this problem is a totally different database architecture: *versioning*. In a versioning database, the status of each transaction is tracked on the transaction-inventory pages (TIP) of the database. Transactions can be active, committed, rolled back, or "in limbo." When a new transaction starts, it's assigned a unique transaction number; it also gets a copy of the TIP, so that it knows the status of all other transactions at the time it started. When a transaction needs to update a row in the database, it checks the TIP for any other active transactions. If none are found, the row is updated. However, if other transactions are active, the transaction writes a new version of the record in the database, and leaves the original version intact. When a transaction updates a row in the database, it also stores its transaction number as part of the record. InterBase does not actually write a complete new copy of the row; instead, it writes a *difference record*, which contains only the fields that have changed. This conserves space in the database.

As a read transaction requests each row in a table, the database software checks to see if the transaction number for the latest version of the row is greater than the number of the transaction that's requesting it, and if the transaction was committed at the time the read transaction started. If it *is* greater, or if the transaction that created the latest version *was* active, the database software looks back through the chain of prior versions until it encounters one whose transaction number is less than that of the transaction trying to read the row, and whose transaction status was committed at the time the read transaction started. When the database manager finds the most recent version that meets these criteria, it returns that version. The result is repeatable-read transaction isolation without preventing updates during the life of the read transaction. This is what sets InterBase apart from other database servers.

Consider the following example of a row for which four versions exist:

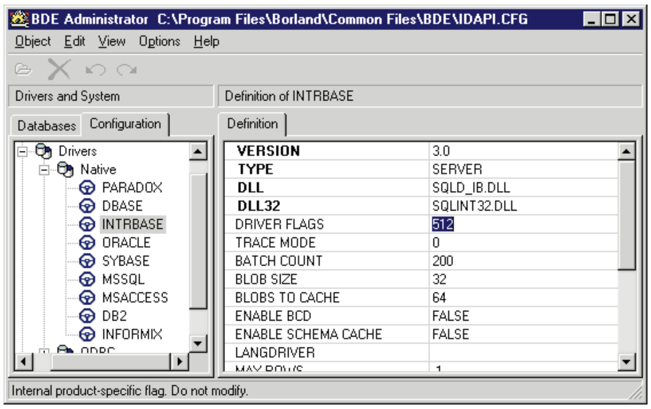
```
Tran=100 (status=committed)
  Tran=80 (status=active when read started)
    Tran=60 (status=rolled back)
      Tran=40 (status=committed when read started)
```

Assume that a read transaction with transaction number 90 attempts to read this row. The read transaction will not see the version of the row created by transaction 100, because the update that created this version took place after transaction 90 began. Transaction 90 also will not be able to read the version created by transaction 80, even though it has a lower transaction number, because transaction 80 has not yet committed. Although the version for transaction 60 still exists on disk, transaction 60 has rolled back, and rolled back versions are always ignored. Therefore, the version that transaction 90 will read is the version created by transaction 40.

In this example, transaction 80 won't be allowed to commit. When transaction 80 attempts to commit, the database manager will discover that transaction 100 has committed, and transaction 80 will be rolled back.

Crash Proof

Another significant advantage of InterBase's versioning architecture is instantaneous crash recovery. Locking-model databases support transaction processing by maintaining a log file that contains information about all changes made to the database. If the database software crashes and is restarted, it must scan the entire log file to determine which transactions were active at the time of the crash, then roll them back. This can take some time. InterBase doesn't maintain a log file, because it doesn't need one. The record versions in the database provide all the information



VERSION	3.0
TYPE	SERVER
DLL	SQLD_IB.DLL
DLL32	SQLINT32.DLL
DRIVER FLAGS	512
TRACE MODE	0
BATCH COUNT	200
BLOB SIZE	32
BLOBS TO CACHE	64
ENABLE BCD	FALSE
ENABLE SCHEMA CACHE	FALSE
LANGDRIVER	
MAX ROWS	-1

Figure 1: The BDE Administrator. Changing the InterBase SQL Link driver DRIVER FLAGS setting to 512, changes the default mode from read-committed (the default) to snapshot.

needed to roll back an active transaction. This means that if an InterBase server crashes, it will restart instantly, because the only recovery it must perform is to scan the transaction-inventory pages and change the status flags of all active transactions to "rolled back." No changes to the data are required, because any record version whose status is rolled back will be ignored when the record is read.

An Unfortunate Default

The ability to provide repeatable-read transaction isolation without blocking update transactions is obviously a big advantage for InterBase. However, the BDE doesn't take advantage of it. For some strange reason, the BDE InterBase driver defaults to read-committed transaction isolation. To overcome this, you must use the BDE Administrator to set the DRIVER FLAGS property of the InterBase SQL Link driver to 512 (see Figure 1). This changes the default mode from read-committed to snapshot.

Conclusion

InterBase's multi-generational architecture truly sets it apart from other database servers. The ability to provide repeatable-read transaction isolation without blocking update transactions makes it ideal for executive-information systems or any situation that must mix long-running read transactions with updates. Its instantaneous crash recovery is another big advantage for systems in which maximum up-time is important. Just remember that when you access InterBase with the BDE, you must set DRIVER FLAGS to get these advantages. ∆

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

table of CONTENTS

DBNAVIGATOR

Delphi 3

By Cary Jensen, Ph.D.

Trying on TeeCharts

A New Component Gets Its Due

With its new interfaces, COM support, packages, remote data-broker technology, and support for creating Web server extensions, Delphi has taken Windows software development to a higher level. Unfortunately, these important improvements can easily obscure some of the wonderful new, updated components available in Delphi 3.

Among these are TeeCharts, which I'll address this month, and Decision Cube components, which I'll cover in next month's column. This is not, however, a comprehensive treatment; that would require hundreds of pages. To get the most from TeeChart, read the teeMach *TeeChart version 3 Chart Guide*. This guide, which is in Microsoft Word format, is installed with Delphi 3. Its default location is C:\Program Files\Borland\Delphi 3\TeeChart\TChartV3.DOC. (If you don't have Word, use the Windows 95 WordPad, or download the Word Viewer from http://www.microsoft.com.)

TeeChart Basics

Delphi 3 ships with TeeChart version 3, a powerful component for displaying and printing graphs. Four components are related to TeeChart in Delphi 3; three of these are associated with the following classes: *TChart*, *TDBChart*, and *TDecisionGraph*. The fourth component, *TQRChart* on the QReport page of the Component palette, is not an actual TeeChart. Rather, it's an interface object, much like DataSource, that points to a *TChart* or *TDBChart*.

You can include the Delphi 3 version of TeeChart in the applications you create, royalty-free. This version, however, doesn't ship with source code. Also, it doesn't work with Delphi 1 or 2. Both the source code and the 16- and 32-bit versions of TeeChart are part of TeeChart Pro. For purchase information, right-click a TeeChart component and select **About TeeChart**, or visit http://ourworld.-compuserve.com/homepages/dberneda.

TeeChart is a parent to one or more series, or graphs. Consequently, it's possible to define more than one graph with a single TeeChart component. Each series defines a chart type and data. The version of TeeChart that ships with Delphi includes 11 series: Line, Area, Point (scatter), Bar (Bar, Pyramid, and Cylinder), Horizontal Bar (Bar, Pyramid, and Cylinder), Pie, Shape, Fast Line, Arrow, Gantt, and Bubble.

Each series type has methods for defining graph dimensions. While these can be manipulated at run time, it's much easier to prepare a TeeChart at design time, using the Chart Editor shown in Figure 1. To display this editor, either double-click a TeeChart component (Chart, DBChart, or DecisionGraph), or right-click the TeeChart and select **Edit Chart**.

The Chart Editor has two primary pages. The first page, labeled Chart (again, see Figure 1), permits you to create, remove, and copy series, as well as define overall properties for the chart. The second page, labeled Series, applies to the current series selected on the Chart page. For example, if a series named Series1 is defined on the Chart page and highlighted, selecting Series permits you to customize Series1. Likewise, if a series named Series2 is selected on the Chart page, the Series page permits you to control the characteristics of Series2.

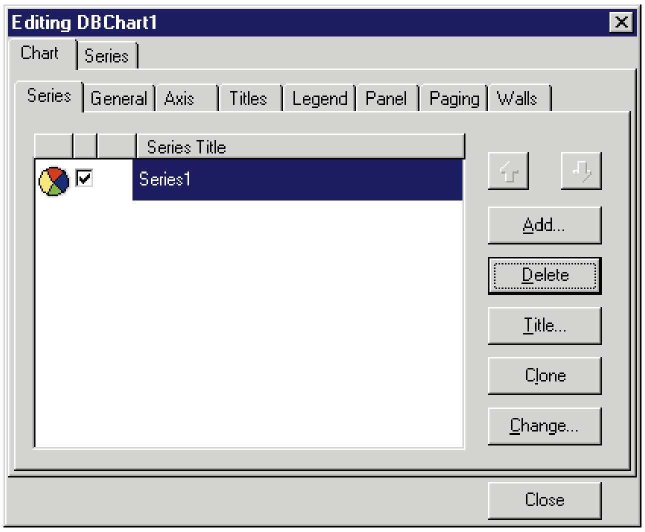
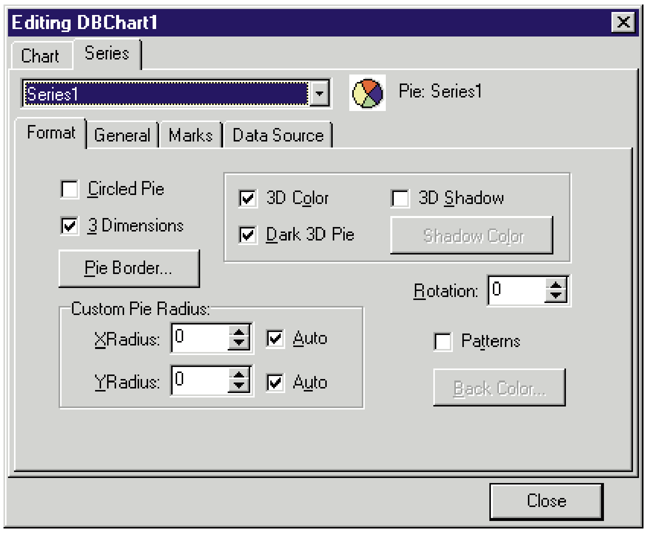


Figure 1: Use the Chart Editor to define the series for a TeeChart.



Series1

[ ] Circled Pie

[✓] 3 Dimensions

[✓] 3D Color

[ ] 3D Shadow

[✓] Dark 3D Pie

Rotation: 0

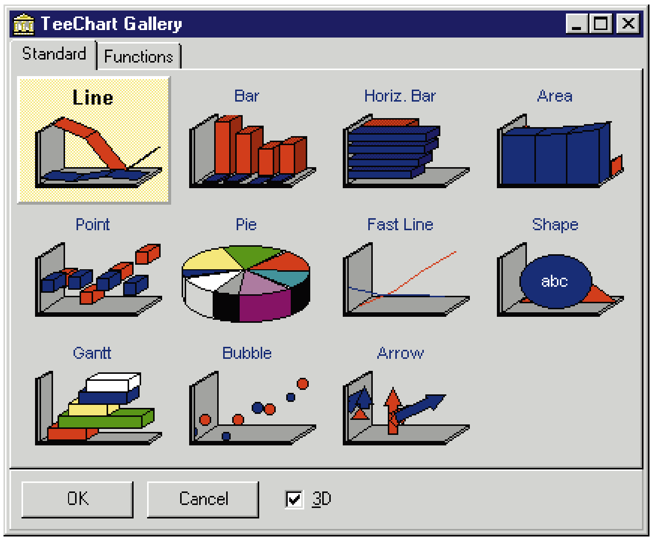
Custom Pie Radius:

XRadius: 0 [✓] Auto

YRadius: 0 [✓] Auto

[ ] Patterns

Figure 2: The Series page of the Chart Editor.



[✓] 3D

Figure 3: The TeeChart Gallery offers design-time choices.

The Series page, shown in Figure 2, displays only those properties appropriate for a given series type. For example, if the selected series is a pie chart, the Series page displays controls for customizing a pie chart. You can switch between the various defined series by using the ComboBox at the top of the Series page.

TeeChart Step-by-Step

The following steps demonstrate how to use a TeeChart to create a pie chart.

Create a new project. Place Panel, DBChart, and Query components on the main form of the project. Set the Panel's *Align* property to *alBottom*, and its *Caption* property to a blank string. Next, set the DBChart's *Align* property to *alClient*.

Define the data. Set the Query's *DatabaseName* property to DBDEMOS. Next, define a query that asks the question "What is the relative contribution of each country to overall sales?" Calculate the sum of the amounts paid in the orders table, and summarize this value for each country in which customers reside. The following query performs this calculation, so enter it in the *SQL* property of the Query:

```
  SELECT CUSTOMER."Country", SUM(ORDERS."AmountPaid")
    FROM "CUSTOMER.DB" CUSTOMER, "ORDERS.DB" ORDERS
   WHERE (CUSTOMER.CustNo = ORDERS.CustNo)
GROUP BY CUSTOMER."Country"
```

Now set the Query's *Active* property to *True*. While this last step isn't essential for creating the chart, it permits you to see it at design time.

Define the DBChart properties. Right-click the DBChart object, and select **Edit Chart**. Define a new series by clicking the **Add** button on the Series sub-page of the Chart page. TeeChart displays the TeeChart Gallery shown in Figure 3. Select **Pie** from the TeeChart Gallery, and click **OK**. This returns your view to the Chart Editor.

With Series1 now selected, go to the Series page of the Chart Editor. To link to the data provided by the Query, select the Data Source page of the Series page, and select **Dataset** from the displayed ComboBox. TeeChart now permits you to select a specific Dataset, the field to use for labels, and the field that represents the quantities to graph. Set **Labels** to **Country**, and set **Pie** to **SUM OF AmountPaid**. Your screen should resemble Figure 4.

Now click the **Close** button on the Chart Editor to accept the changes. Because the Query is active, the data it returns is immediately visible in the DBChart component (see Figure 5). Run the form by pressing F9; notice that some of the labels overlap. This can be solved by rotating the angle of the pie chart. Close the application and return to the designer.

Rotate the pie chart. Pie-chart rotation is subject to the properties of the series itself, rather than those of the DBChart component. Once defined, a series can be selected directly in

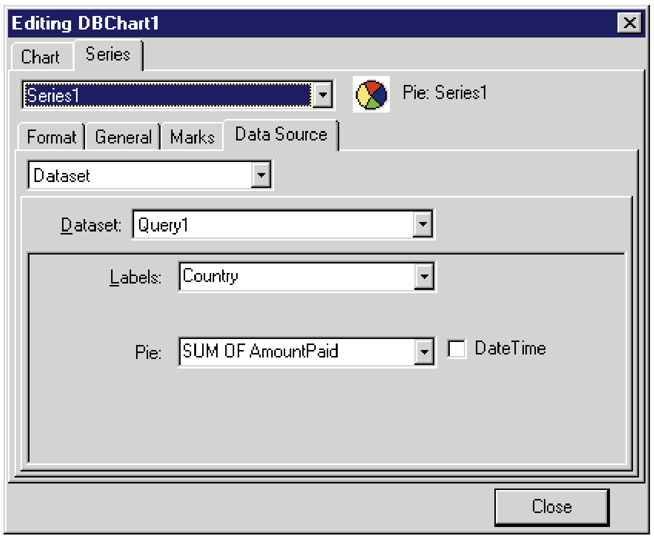

Editing DBChart1

Chart | Series

Series1 — Pie: Series1

Format | General | Marks | Data Source

Dataset

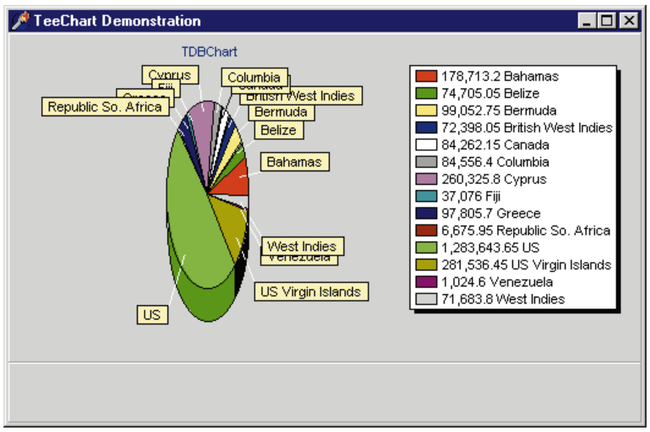
Dataset: Query1

Labels: Country

Pie: SUM OF AmountPaid ☐ DateTime

Close

Figure 4: Defining the data to display in a pie chart, using the Chart Editor.



🟥	178,713.2 Bahamas
🟩	74,705.05 Belize
🟨	99,052.75 Bermuda
🟦	72,398.05 British West Indies
⬜	84,262.15 Canada
▨	84,556.4 Columbia
🟪	260,325.8 Cyprus
🟦	37,076 Fiji
🟦	97,805.7 Greece
🟥	6,675.95 Republic So. Africa
🟩	1,283,643.65 US
🟨	281,536.45 US Virgin Islands
🟪	1,024.6 Venezuela
⬜	71,683.8 West Indies

Figure 5: A pie chart displayed in a DBChart. Note the overlapping labels.

the Object Inspector, permitting you to control its properties at design time. Therefore, you could select Series1 in the Object Inspector, and set the *RotationAngle* property to a value other than 0. (The Chart Editor also permits you to modify this property.) However, like all published properties, this one can be modified at run time, as well. Providing a run-time interface to rotate the chart offers the user extra flexibility; let's do that, rather than limit the chart to a static, design-time adjustment.

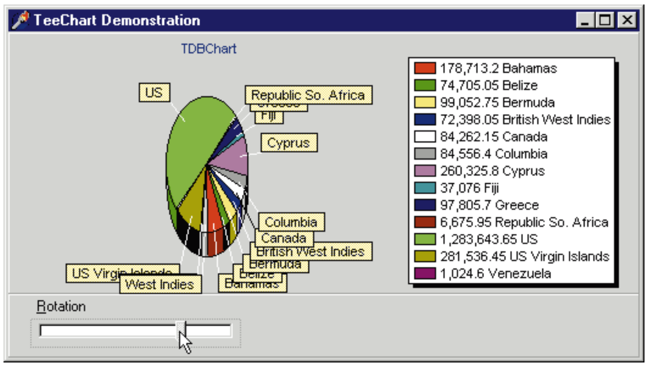
Place a TrackBar in the Panel. Set its *Min* property to `0`, and its *Max* property to `360`. Next, set the TrackBar *TickStyle* property to *tsNone*. Double-click the TrackBar to create an *OnChange* event handler, and modify it as follows:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  Series1.RotationAngle := TrackBar1.Position;
end;
```

Run the form. When you move the TrackBar, the pie chart rotates (see Figure 6).

Adding Additional Series

A single TeeChart can have many different series; you can use a single form to display multiple charts. A particular



🟥	178,713.2 Bahamas
🟩	74,705.05 Belize
🟨	99,052.75 Bermuda
🟦	72,398.05 British West Indies
⬜	84,262.15 Canada
▨	84,556.4 Columbia
🟪	260,325.8 Cyprus
🟦	37,076 Fiji
🟦	97,805.7 Greece
🟥	6,675.95 Republic So. Africa
🟩	1,283,643.65 US
🟨	281,536.45 US Virgin Islands
🟪	1,024.6 Venezuela

Figure 6: With the TrackBar, a user can adjust the pie-chart rotation at run time.

series is visible when its *Active* property is set to *True*. Normally, you have only one series active at a time. However, you can overlap charts, if you want, by setting more than one series to *Active*.

While overlapping series provide a powerful analytical tool for users, they're useful only under certain circumstances. The series of a given TeeChart share characteristics such as scale, background gradient, and so forth; you should permit the display of overlapping series only when their characteristics are compatible.

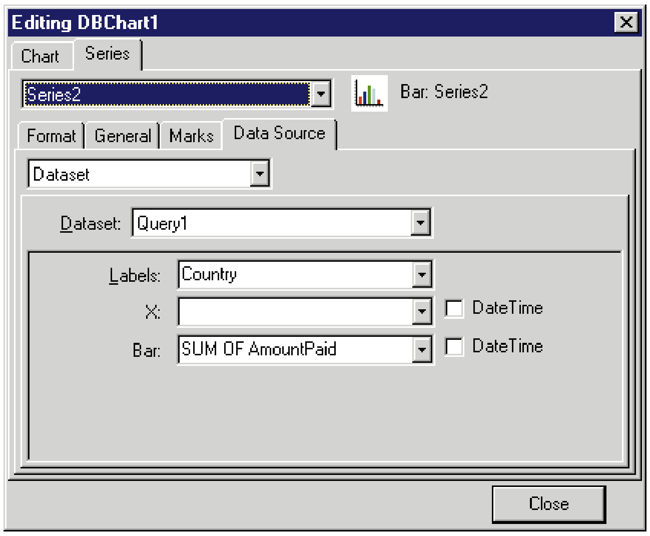
The next example demonstrates how to create a second series — a bar chart — with the DBChart created earlier. A bar chart is used to display relative quantities, much like a pie chart. TeeChart permits you to base each series on a different Dataset — or even the same Dataset, if the data is consistent with the chart style you choose.

Create a new series. Right-click the TeeChart, and select **Edit Chart**. Click the **Add** button, then select **Bar** from the TeeChart Gallery. Press **OK** to return to the Chart Editor. Now that you have two series, you can select which is active by using the check box next to the series name, in the Series sub-page of the Chart page (refer to Figure 1). Uncheck the check box next to Series1, so that only Series2 is displayed.

Set the properties. Select the Series page of the Chart Editor. If Series2 is not selected in the ComboBox, select it. Move to the DataSource page. Set **Data Source** to **Dataset**, **Dataset** to **Query1**, **Labels** to **Country**, and **Bar** to **SUM OF AmountPaid**. If the data could be further divided, such as amount paid by year, you could set **X** to **Year**. In this case, leave **X** blank. Your screen should resemble Figure 7.

Now select the General page of the Series page. Uncheck the **Show in Legend** check box to turn off the legend for Series2. Now return to the Series sub-page of the Chart page. Uncheck Series2, and check Series1. Again, the pie chart is displayed. Accept the Chart Editor by clicking **Close**.

Add a button. Add a button to the panel at the bottom of the form. Set the caption of this button to `Show Bars`. Double-click this button, and enter the event handler shown



Editing DBChart1

Chart | Series

Series2 — Bar: Series2

Format | General | Marks | Data Source

Dataset

Dataset: Query1

Labels: Country

X: ☐ DateTime

Bar: SUM OF AmountPaid ☐ DateTime

Close

Figure 7: Setting up a bar chart in the Chart Editor.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  for i := 0 to DBChart1.SeriesCount - 1 do
    DBChart1.Series[I].Active := False;
  if Button1.Caption = 'Show Bars' then
    begin
      Button1.Caption := 'Show Pie';
      Series2.Active := True;
      TrackBar1.Enabled := False;
    end
  else
    begin
      Button1.Caption := 'Show Bars';
      Series1.Active := True;
      TrackBar1.Enabled := True;
    end;
end;
```

Figure 8: The button's event handler.

in Figure 8. Now run the form. When you click **Show Bars**, the bar chart appears, as shown in Figure 9. Clicking this button a second time reverts the display to the pie chart.

Controlling Series at Run Time

You can control series at run time by calling the methods and controlling the properties of *TChart* and the individual series. For example, to add a new series, create a new *TChartSeries* descendant, then add it to the TeeChart by calling the *AddSeries* method:

```
procedure AddSeries(ASeries : TChartSeries);
```

where *TChartSeries* is one of the following classes: *TAreaSeries*, *TBarSeries*, *TCircledSeries*, *TCustomBarSeries*, *TCustomSeries*, *TFastLineSeries*, *THorizBarSeries*, *TLineSeries*, *TPieSeries*, or *TPointSeries*. For example, to add a new bar-chart series to *TDBChart1*, you would first declare a *TBarSeries* instance variable in the form's type declaration. For example, you could add the following line to the public section of your form's **type** declaration:

```
Series3: TBarSeries;
```

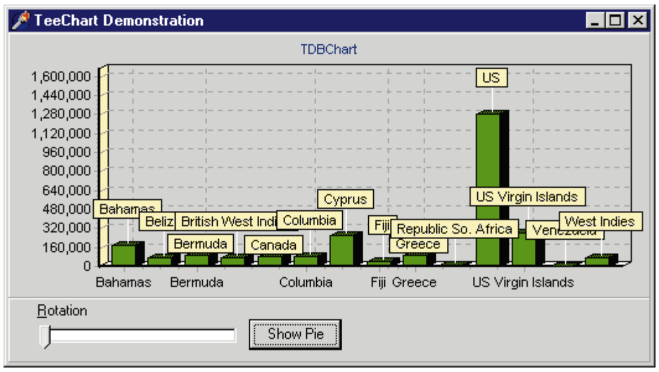


Figure 9: The bar chart in DBChart.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  i: Integer;
begin
  if Series3 = nil then
    begin
      Series3 := TBarSeries.Create(Self);
      DBChart1.AddSeries(Series3);
      Series3.AddBar(16,'West',clBlue);
      Series3.AddBar(25,'Central',clRed);
      Series3.AddBar(35,'NorthEast',clYellow);
      Series3.AddBar(25,'South',clGreen);
    end;

  for i := 0 to DBChart1.SeriesCount -1 do
    DBChart1.Series[i].Active := False;

  Series3.Active := True;

end;
```

Figure 10: The code to produce *Series3*.

Next, call the *TBarSeries* constructor, and assign it to the instance variable. Finally, use *AddSeries* to add this series to your TeeChart. These last two steps look like this:

```
Series3 := TBarSeries.Create(Self);
DBChart1.AddSeries(Series3);
```

Once you've created the series and added it to the TeeChart, you can work with the properties and methods of the series to produce the chart. The methods and properties available depend on the type of series you've created. After you finish setting properties, you can make a particular series visible by setting its *Active* property to *True*. Although more than one series at a time can be active, you will generally want to turn off other series by setting their *Active* properties to *False*.

This technique is demonstrated in the TEECHART project, which is available for download (see the download instructions at the end of this article). Attached to the button labeled **Add/Show New Bar** the code shown in Figure 10, which tests for the existence of a series named *Series3*. If it doesn't exist, *Series3* is created and populated with data by calling the *TBarSeries* method *AddBar*. Finally, all series associated with the TeeChart are made inactive; then the new one is made active. The result is shown in Figure 11.

If you inspect the *OnClick* event handler in Figure 10, you'll notice that it's possible to control series either individually

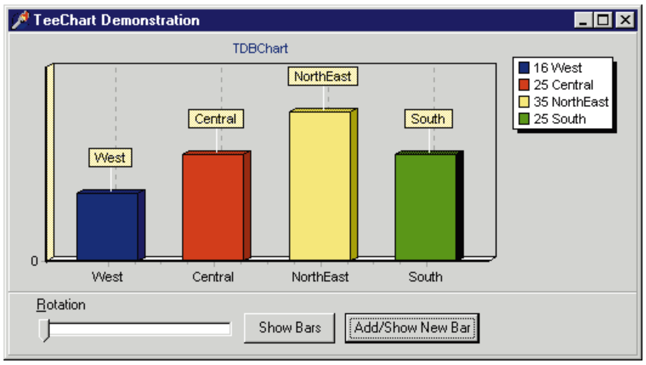


Figure 11: A bar chart created at run time, using *TChartSeries* methods.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  i: Integer;
begin
  if DBChart1.SeriesCount < 3 then
    begin
      DBChart1.AddSeries(TBarSeries.Create(Self));
      TBarSeries(DBChart1.Series[2]).
         AddBar(16,'West',clBlue);
      TBarSeries(DBChart1.Series[2]).
         AddBar(25,'Central',clRed);
      TBarSeries(DBChart1.Series[2]).
         AddBar(35,'NorthEast',clYellow);
      TBarSeries(DBChart1.Series[2]).
         AddBar(25,'South',clGreen);
    end;

  for i := 0 to DBChart1.SeriesCount -1 do
    DBChart1.Series[i].Active := False;

  DBChart1.Series[2].Active := True;

end;
```

Figure 12: This alternative requires no instance variable.

though an instance variable, or through the *Series* property of the TeeChart (an array of pointers to the series owned by the TeeChart). Although a *TBarSeries* variable was declared in the preceding example, there are alternatives. For example, the code in Figure 12 would produce the same effect without the need for an instance variable.

As mentioned previously, each *TChartSeries* descendent has different methods, depending on the type of series it represents. For example, when creating a point chart, you can't call *AddBar*; you would use *AddXY* instead. This is demonstrated in Figure 13, which is associated with the button labeled **Add/Show New Point** in the TEECHART project.

This code creates a new series each time, using Delphi's built-in random-number generator. This example shows that you can create a chart using any data, not just that stored in a Dataset. In fact, with the appropriate *Add* method for the series you're creating, you can define the contents of a chart based on any valid data, regardless of origin. Figure 14 shows a chart produced by this code.

Conclusion

Using one of the TeeChart components, you can quickly create business graphs and charts, and include them in your

```
procedure TForm1.Button3Click(Sender: TObject);
var
  i: Integer;
begin
  if Series4 <> nil then
    Series4.Free;

  Randomize;
  Series4 := TPointSeries.Create(Self);
  DBChart1.AddSeries(Series4);
  Series4.ShowInLegend := False;

  for i := 1 to 100 do
    Series4.AddXY(i,i*2 + Random(i),IntToStr(i),clRed);

  for i := 1 to 100 do
    Series4.AddXY(i,i*2,IntToStr(i),clBlue);

  for i := 0 to DBChart1.SeriesCount -1 do
    DBChart1.Series[i].Active := False;

  Series4.Active := True;

end;
```

Figure 13: The code for a point chart.

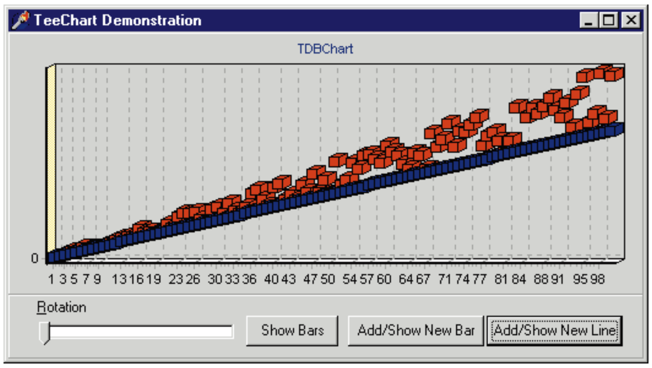


Figure 14: A point chart created at run time, using random numbers.

Delphi applications. Furthermore, TeeChart is flexible enough to permit your charts to be built at either design time or run time, from any data available to your application.

Next month, DBNavigator takes a look at the Decision Cube components available in the Client/Server and Enterprise editions of Delphi 3. Among these components is the DecisionGraph, which is itself a TeeChart descendant. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\OCT\DI9710CJ.

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

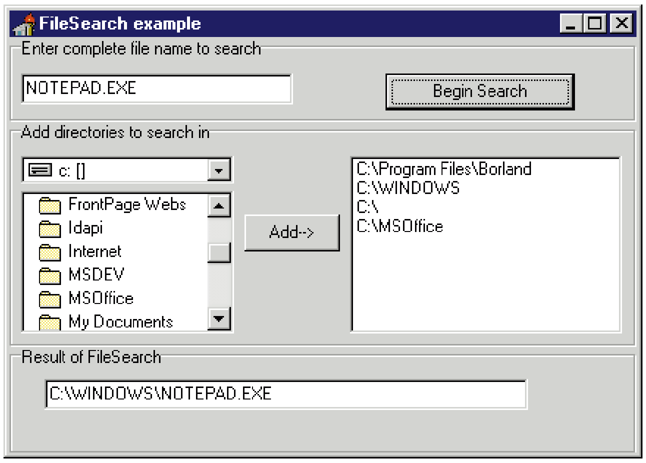
By John P. Gambrell

Searching Multiple Directories, etc.

Delphi Tips and Techniques

What's the best way to search for a file in multiple directories? Delphi's *FileSearch* function seeks out the pathname of a specified file in one or many directories with a single call. If the file is found, it returns the fully-qualified path name of the file; otherwise, it returns an empty string. "Fully qualified" means the string contains enough path information for you to access the file directly.

If the file is in the current directory, no path information will be included; if it isn't, the appropriate path will be prepended to the file-name. *FileSearch* requires two arguments: the name of the file and the list of directories to search. The directory list is of type **string** and contains directory names separated by semi-colons (e.g. C:\Windows;C:\Windows\System).



NOTEPAD.EXE

C:\Program Files\Borland
C:\WINDOWS
C:\
C:\MSOffice

C:\WINDOWS\NOTEPAD.EXE

Figure 1: The *FileSearch* example.

```
procedure TForm1.btnSearchClick(Sender: TObject);
var
  Result, DirList :string;
  x : Integer;
begin
  DirList := '';
  for x := 0 to ListBox1.Items.Count-1 do
    DirList := DirList + ListBox1.Items[x] + ';';

  Result := FileSearch(Edit1.Text,DirList);
  if Result = '' then
    ShowMessage('File ' + Edit1.Text + ' was not found.')
  else
    Edit2.Text := Result;
end;

procedure TForm1.btnAddClick(Sender: TObject);
begin
  Listbox1.Items.Add(DirectoryListBox1.Directory);
end;
```

Figure 2: The *OnClick* event handlers for the **Add** and **Begin Search** buttons.

Figure 1 shows an example application that demonstrates *FileSearch*. To use it, enter the file-name to search, then use the DirectoryListBox component to add directories to the ListBox, then press the **Begin Search** button. Figure 2 shows the code for the *OnClick* event handlers for the **Add** and **Begin Search** buttons. The search first builds a directory list by concatenating the items in the ListBox and calling the *FileSearch* function. If the file is found, the second Edit component displays the fully-qualified path name of the file (otherwise a message box alerts the user).

How does the parent/child relationship work for Delphi windows controls?

Every windows control includes the property *Parent* that specifies its parent control. Controls assigned to another control via the *Parent* property become child controls to that parent. For example, RadioButton components placed inside a GroupBox are the children of that GroupBox; thus, the RadioButtons relate to each other, so only one can be checked at a time. When a parent control is destroyed, all the child controls assigned to that parent are destroyed as well.

This example demonstrates the relationship between parent and child controls with the use of two GroupBoxes and a few RadioButtons, as shown in Figure 3. Clicking on the **Change Parent of RadioButton3** button reassigns *RadioButton3*'s *Parent* property to the other

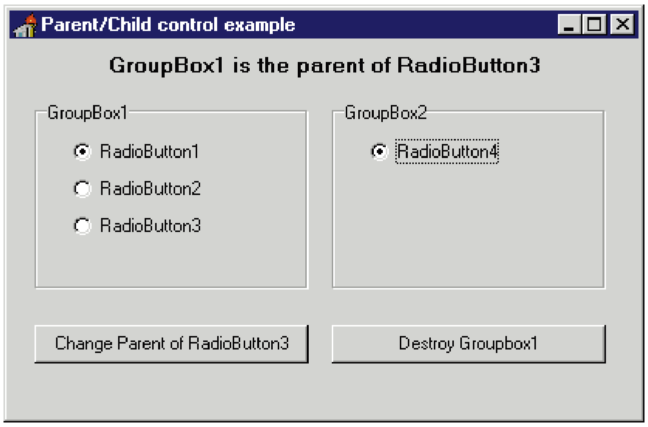

Parent/Child control example
GroupBox1 is the parent of RadioButton3
GroupBox1
RadioButton1
RadioButton2
RadioButton3
GroupBox2
RadioButton4
Change Parent of RadioButton3
Destroy Groupbox1

Figure 3: The parent/child control example.

GroupBox; it then relates to the RadioButtons in the other GroupBox (not something you want to practice, obviously). Clicking on the **Destroy Groupbox1** button calls the *Destroy* method for *GroupBox1*, which in turn destroys its children RadioButtons. See Figure 4 for the code assigned to the *OnClick* event handlers for the **Change Parent of RadioButton3** and **Destroy Groupbox1** buttons. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\OCT\ DI9710JG.

John Gambrell is a consultant with Ensemble Corporation, specializing in client/server and Internet application development. John can be reached at jgambrell@ensemble.net.

```
procedure TForm1.btnChangeParentClick(Sender: TObject);
begin
  try
    with RadioButton3 do begin
      if Parent.Name = 'GroupBox2' then
        begin
          if Checked then Checked := False;
          Parent := Groupbox1;
        end
      else
        begin
          if Checked then Checked := False;
          Parent := GroupBox2;
        end;
      Label1.Caption :=
        Parent.Name + ' is the parent of RadioButton3';
    end;
  except
    ShowMessage('GroupBox1 does not exist.');
  end;
end;

procedure TForm1.btnDestroyClick(Sender: TObject);
begin
  Groupbox1.Free;
end;

end.
```

Figure 4: The *OnClick* event handlers for the **Change Parent of RadioButton3** and **Destroy Groupbox1** buttons.

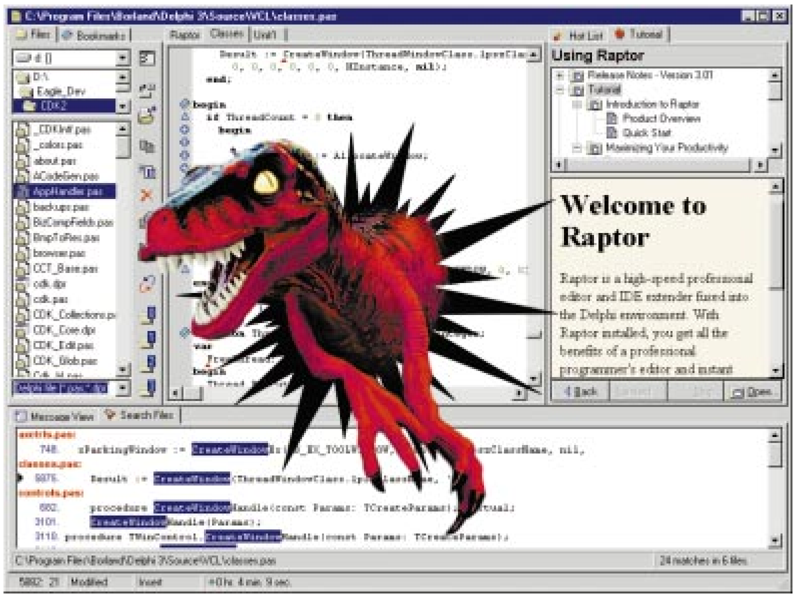
By *Alan C. Moore, Ph.D.*

The Birth of Raptor

RADifying Delphi

I know, I know — Delphi is already RAD. But before you get too complacent with the Delphi IDE and the various built-in tools Borland provides, you'd better check out Raptor, the latest in a series of powerful development tools from Eagle Software. If you've used the Component Development Kit (see Robin Karlin's review in the May 1997 *Delphi Informant*) or the component testing tool, reAct, you are already familiar with Eagle Software's commitment to excellence and innovation. Their latest creation, Raptor, is much more than simply another useful tool — it has the potential to revolutionize how you work in Delphi 3, and to considerably enhance your productivity. Let's find out how by examining its major features.

Raptor has two main components: a greatly enhanced Object Pascal editor, and a series of panels that surround the editor, providing many powerful utilities. Figure 1 shows how Raptor and its panels integrate with the Delphi IDE. Both the enhanced editor and the panels deserve a good deal of attention; let's begin with the editor.



```
      Result := CreateWindow(ThreadWindowClass.lpszCla
        0, 0, 0, 0, 0, 0, HInstance, nil);
    end;

begin
  if ThreadCount = 0 then
    begin
```

Welcome to Raptor

Raptor is a high-speed professional editor and IDE extender fused into the Delphi environment. With Raptor installed, you get all the benefits of a professional programmer's editor and instant

```
extctrls.pas:
  748.   xParkingWindow := CreateWindowEx(WS_EX_TOOLW...
classes.pas:
  5875.    Result := CreateWindow(ThreadWindowClass.lpszClassName, ...
controls.pas:
  882.    procedure CreateWindowHandle(const Params: TCreateParams); virtual;
  3101.    CreateWindowHandle(Params);
  3110. procedure TWinControl.CreateWindowHandle(const Params: TCreateParams);
```

C:\Program Files\Borland\Delphi 3\Source\VCL\classes.pas

5882: 21   +0 hr. 4 min. 9 sec.

The Editor of Your Dreams

If you were to sit down and compile a wish-list of features you wanted in your Delphi editor, what would it include? Of course, you'd want complete integration with the IDE. Your wish-list might also include a powerful and extendable set of keyboard templates, bookmarks, and other code-navigational tools, a means of keeping track of code changes, and a Clipboard with recent entries (history) to paste into your code. Believe it or not, every one of these features is part of the Raptor editor!

And while other third-party code editors, such as Multi-Edit for Windows by American Cybernetics (see Moore's review in the April 1997 *Delphi Informant*), provide some of these features, none of them provide full integration with the Delphi IDE. So, while you can save a lot of keystrokes by using such products, when it's time to compile and debug, you must switch back to Delphi. With Raptor, those hassles are history. Because you're working within Delphi, you have access to Code Insight, the integrated debugger, and other essential tools that are otherwise unavailable outside the IDE.

One of the features I liked about Multi-Edit was its keyboard templates — who wants to type the same patterns over and over, day after day? Here's an area where Raptor really shines, providing over 600 highly intuitive keyboard templates. They're so intuitive, in fact, that I was able to figure out most of them without even checking the Help screens. That's not surprising, because all of them are geared towards Object Pascal and the kind of code we write in Delphi every day. Let's look at one example.

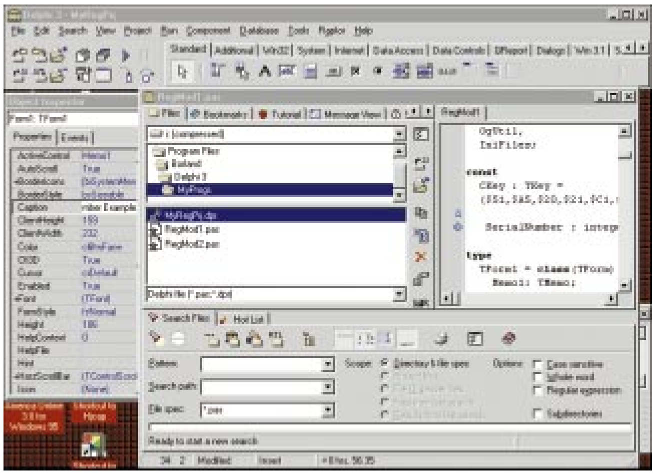
Type `v`, then hit Space; then type `bl` and hit Space; then type `MyBooleanVariable`. Finally, with the cursor over the last name ("MyBooleanVariable"), hit Ctrl C (or your hot key for copying to the Clipboard). Move down a line, then type `b` and hit Space; move down another line, and type `sf`. These few keystrokes produce the following code snippet:

```
var
  MyBooleanVariable : Boolean;
begin
  MyBooleanVariable := False;
```

As you've probably figured out, Space is the trigger key. If you want, you can reconfigure it to Shift Space (a request I made as a beta-tester.) Most of the templates are static and obvious (they always expand the same way each time). The last one ("sf") deserves special mention, as it is dynamic (my term). Like many other templates, "sf" (or "set to false") inserts the current contents of the Clipboard before the assignment, " := False;". Figure 2 lists these and other templates with their expansions. Note that {paste} indicates the point in a dynamic template where text will be inserted from the Clipboard; note also that many templates, including "bl," appropriately place the cursor before or within the expanded text.

If you don't like some of the keyboard assignments, you can easily change them; and with just a little more effort, you can create your own. As previously mentioned, these templates can be either static or dynamic. Figure 3 shows the template editor, where you can view, edit, or create new templates. Note the special shortcuts that are provided to add a Clipboard paste area to your template ({paste}, as shown in the previous examples), to insert the editing cursor at any location within the expanded code, or to add backspaces (hard and soft), tabs, character deletion, and date/time information. Raptor templates also allow you to place special markers in the code, to indicate insertion points. (After you've finished entering code at one insertion point, you can press Esc to jump to the next one.)

Templates are also smart: They expand in your particular code style, which is configurable. They're also smart enough not to expand within comments or strings. You can bring up your own dialog box from within a Raptor template, or call specially created string functions. Clearly, these templates are a tremendous help in accelerating coding speed, but the good news doesn't stop here.

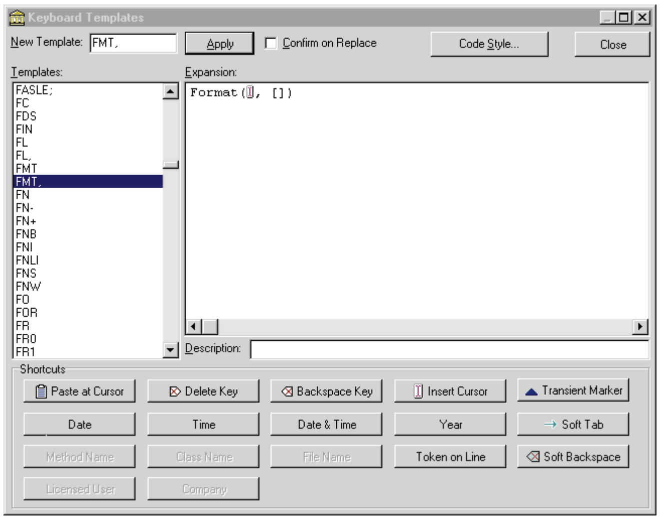


```
OgUtil,
IniFiles;

const
  CKey : TKey =
  ($51,$A5,$D0,$21,$C1,

  SerialNumber : integ

type
  TForm1 = class(TForm)
    Memo1: TMemo;
```

Figure 1: Raptor adds considerable functionality to Delphi's editor (center right) and powerful panels (center and bottom).

Template	Expansion
v	var
bl	: boolean;
b	begin
sf	{paste} := false;
e	end;
sm	ShowMessage
acf	Application.CreateForm(T{paste}, {paste}
rs	ResourceString
ifnax	if not assigned({paste}) then exit
tf	a try..finally block
lb	{paste}.Handle := LoadBitmap(HInstance, PChar(' '));

Figure 2: A small sampling of Raptor's comprehensive set of keyboard templates.



New Template: FMT,

Templates:

FASLE;
FC
FDS
FIN
FL
FL,
FMT
FMT,
FN
FN-
FN+
FNB
FNI
FNLI
FNS
FNW
FO
FOR
FR
FR0
FR1

Expansion:

```
Format(|, [])
```

Description:

Figure 3: "FMT" (one of over 600 keyboard templates) expands to "Format (X, [])" placing the cursor at the X.

How many of you have experienced this: While developing a new class, you find yourself constantly jumping back and forth between declarations in the **interface** section and the enabling code in the **implementation** section. Wouldn't it be great to be able to drop multiple markers wherever you are, so you can quickly jump from one active section of code to another? With Raptor's editor, you can.

Similarly, when you're in the heat of creative energy, coding like a maniac, it's easy to forget where you've made changes to your code. If you add too many lines in different locations and forget what you've done, you might be in trouble. I've done this myself. With Raptor, you'll never face this nightmare again. It monitors every change you make, leaving a descriptive marker in the gutter to the right of your code (the place that Delphi 3 reserves for breakpoint markers). You can instantly jump through all your newly added and modified lines using a hot key or the command menu.

Another particularly powerful and useful feature is Raptor's Clipboard history. Every time you copy or cut text to the Clipboard from inside the editor, that text is also saved to Raptor's special Clipboard, which is subdivided into nine boxes. As you cut and copy text, the most recent text placed in the Windows Clipboard is also inserted in the beginning of the Raptor Clipboard (buffer one). All other Clipboard buffers are shifted to the back of the queue by one, and the last Clipboard buffer (buffer nine) is discarded. It gets even better. You can lock buffers so that often-used entries won't be erased. Best of all, your settings are persistent from one Delphi session to the next. Conveniently, Raptor's Clipboard history displays the selected text in the same font and syntax highlight settings as in Delphi.

Raptor includes other useful tools that enhance your navigation within the editor. I have always been a big fan of bookmarks; they just never went quite far enough. Raptor provides bookmarks that are named, persistent (yes!), and compatible with version control for team programming. Its stack-based code markers are also persistent; they enable jumping through your code in the order they are set, or back and forth between two points. You can also retrace your steps. It's a lot like leaving florescent markers as you explore a large cave with many branches. You can work with large unit files over half a meg as easily as with files one-tenth the size.

Finally, Raptor provides a sophisticated set of searching, selecting, and formatting options available from a pop-up menu. You can quickly jump to the next occurrence of the current word. If you've selected a block of text, you can quickly perform a number of common operations, including indenting/unindenting, commenting/uncommenting (in any of the styles), changing case in all of the usual ways, and even sorting the block. Remarkably, Raptor changes Delphi's command menu depending on the amount of text selected, with appropriate responses for no selection, multi-line selection, or selection within a single line. We've taken a detailed look at Raptor's enhanced editor, but we've barely scratched the surface. Now, let's take a look at its panels.

Panels, Panels Everywhere

Raptor comes with a number of built-in panels to address many special programming needs. It provides an open architecture that allows you to create your own panels and add-ons. At the time of this writing, Raptor was still in beta, and included the following panels: Files, Bookmarks, Tutorial, Message View (Delphi's compiler messages), Statistics, Search Files, and Hot List. Some of these, such as Bookmarks, are closely related to the editor. Others, such as the Files and the Hot List, allow for quick and easy file imports into your current project (as well as other useful tasks). Let's briefly examine each.

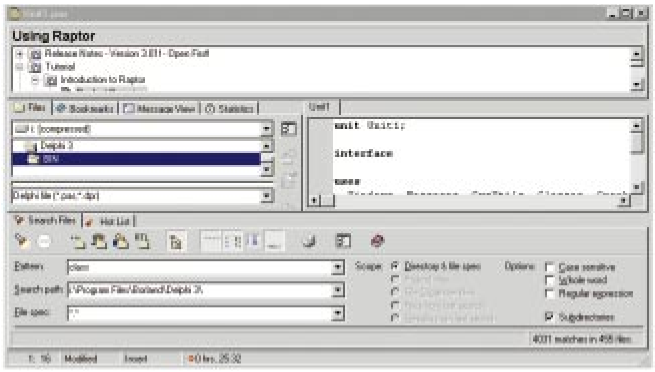
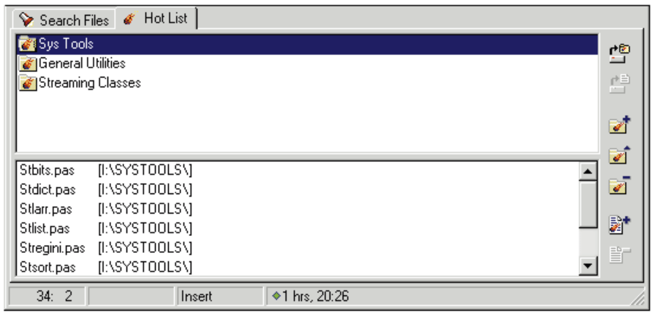


Figure 4: Raptor's powerful file searching tool provides speed buttons to access Delphi's main folders.



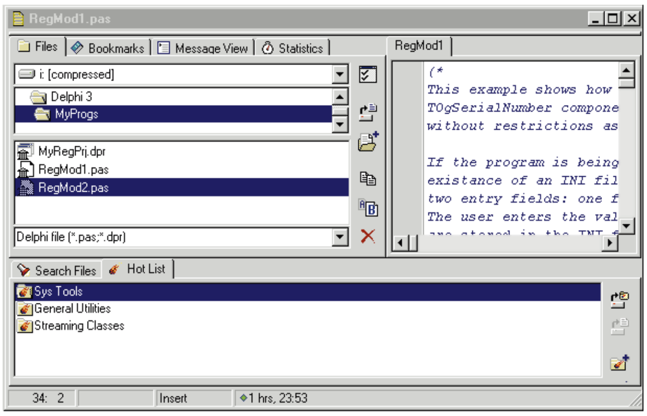
```
Stbits.pas    [I:\SYSTOOLS\]
Stdict.pas    [I:\SYSTOOLS\]
Stlarr.pas    [I:\SYSTOOLS\]
Stlist.pas    [I:\SYSTOOLS\]
Stregini.pas  [I:\SYSTOOLS\]
Stsort.pas    [I:\SYSTOOLS\]
```

Figure 5: Raptor's Hot List provides easy access to non-component units and classes.

Raptor's Search Files panel (see Figure 4) is considerably faster than Delphi's. All the expected options — search pattern, search path, file spec, case sensitivity, and subdirectory search — are included. There are also speed buttons to quickly set the path, or search within the current project directory, the VCL directory, or the run-time library (RTL) directory. Other buttons provide a means to toggle various elements of the display, print results, get help, or examine the File Grep properties dialog box. The latter provides control over the visual display of the search results, the maximum number of entries, and the thread priority (the default is Highest, just short of Time Critical.)

The Hot List panel (see Figure 5) allows you to organize and instantly access frequently used groups of files. You could have one hot folder for all your include files, another for all your utility files, and another for your packages. I use this feature for easy access to TurboPower's SysTools library with its many container classes, string functions, mathematical functions, and sorting routines. The Hot List panel includes speed buttons to create, delete, or rename folders, to add or remove files, and to open a single file or all the files in the current folder.

In addition to the two specialized file-manipulation panels we've just examined, Raptor includes a full-featured Files panel (see Figure 6) that allows easy navigation through folders and files. Again, there are speed buttons to open,



RegMod1.pas

Files | Bookmarks | Message View | Statistics

i: [compressed]

Delphi 3
MyProgs

MyRegPrj.dpr
RegMod1.pas
RegMod2.pas

Delphi file (*.pas;*.dpr)

RegMod1

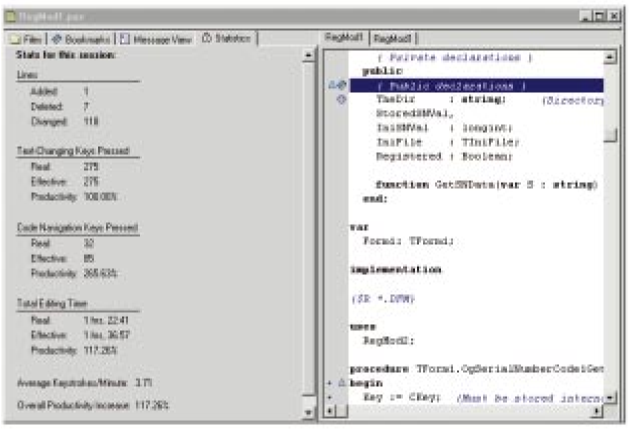
```
(*
This example shows how
TOgSerialNumber compone
without restrictions as

If the program is being
existance of an INI fil
two entry fields: one f
The user enters the val
```

Search Files | Hot List

Sys Tools
General Utilities
Streaming Classes

34: 2 | Insert | 1 hrs, 23:53

Figure 6: Raptor's Files panel includes all of the common operations.



Files | Bookmarks | Message View | Statistics

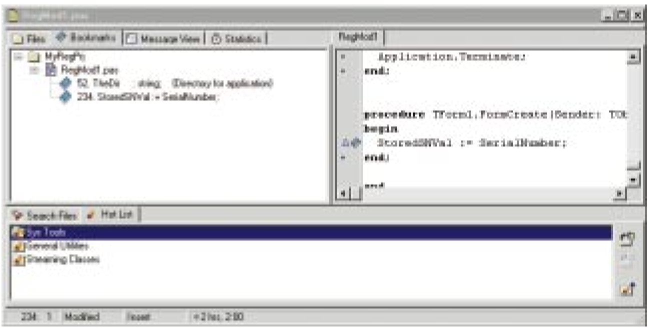
Stats for this session:

```
Lines
    Added        1
    Deleted      7
    Changed      118

Text-Changing Keys Pressed
    Real         275
    Effective    275
    Productivity 100.00%

Code Navigation Keys Pressed
    Real         32
    Effective    85
    Productivity 265.63%

Total Editing Time
    Real         1 hrs, 22:41
    Effective    1 hrs, 36:57
    Productivity 117.26%

Average Keystrokes/Minute: 3.71

Overall Productivity Increase: 117.26%
```

RegMod1 | RegMod2

```
    { Private declarations }
  public
    { Public declarations }
    TheDir       : string;      {Directory
    StoredSNVal,
    IniSNVal     : longint;
    IniFile      : TIniFile;
    Registered   : Boolean;

    function GetSNData(var S : string)
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses
  RegMod2;

procedure TForm1.OgSerialNumberCode1Get
begin
  Key := CKey;   {Must be stored interna
```

Figure 8: The Statistics panel provides detailed time-tracking and productivity information.



Files | Bookmarks | Message View | Statistics

MyRegPrj
RegMod1.pas
52. TheDir : string; {Directory for application}
234. StoredSNVal := SerialNumber;

RegMod1

```
  Application.Terminate;
end;


procedure TForm1.FormCreate(Sender: TOb
begin
  StoredSNVal := SerialNumber;
end;

end.
```

Search Files | Hot List

Sys Tools
General Utilities
Streaming Classes

234: 1 | Modified | Insert | 2 hrs, 2:00

Figure 7: The special Bookmarks panel.

copy, rename, touch, and yes, delete files. It keeps track of your favorite directories, so if you want to add files to your current project from several directories, you can do so easily. You can even bring up a DOS prompt from this panel. As with other panels, there is a properties dialog box that gives you complete control over which buttons are displayed, and which actions require confirmation.

When I described the enhanced code editor, I mentioned its bookmarks. Within the editor, you can manage and use bookmarks easily. In addition, there is a Bookmarks panel that provides a summary of all the bookmarks set in the current project and its files (see Figure 7). By double-clicking any of these bookmarks, you can easily jump to that location. Has source-file navigation ever been easier than this?

The Statistics panel (see Figure 8) is fascinating and highly useful. In and of itself, it provides a powerful means of marketing this product to Delphi developers. This panel monitors your keystrokes and your use of Raptor's features as you work on your project, monitoring and storing all kinds of time-tracking and editing statistics, including:

- lines of code added, deleted, or changed;
- actual keystrokes;
- code navigation keys, real and effective (based on shortcuts);
- total editing times (with or without Raptor's features); and
- a summary of average keystrokes per minute and the increase in productivity by using Raptor.

In case you're concerned about my apparent lack of typing skills, please be aware that I spent much more time in my word processor than in the Raptor/Delphi editor this session! Later I learned that I could adjust Raptor's Statistics panel settings to pause after 0.1 minutes of inactivity (the default is three minutes). This produces a more accurate keystroke/minute average.

Configuring and Adding New Panels

The one panel we haven't discussed is the Tutorial panel. This panel provides detailed information about every aspect of Raptor: its editor, panels, and the Raptor API (RAPI). How customizable is Raptor? Let's see.

By right-clicking on the editor panel and choosing **Panels**, you can open or close any panel, or maximize the code-editing panel. Similarly, by right-clicking on the various panels, you can move them to the top, bottom, right, or left; you can also hide them or float them independent of the other panels. This gives you tremendous control over the appearance of the expanded IDE. Just as the main editing panel has an expanded pop-up menu, each of the other panels has its own pop-up menu that provides quick and easy access to common tasks. The good news doesn't stop here. With Raptor, you can easily add your own tools and panels. (Several third-party tool makers are currently working on placing their utilities within Raptor panels for easy access.) Let's take a brief look at RAPI.

RAPI: A Quick Overview

Raptor's API supports two kinds of add-ons: standard and panel. Standard plug-ins allow you to add new menu commands either to Delphi's menu system or to its editor's pop-up menu. You can also implement special string functions, or call up dialog boxes to perform any task you like. String functions are added to Raptor's keyboard templates. This way, templates can be smarter about the code they manipulate. You can provide even more power and control with "Method" string functions. These allow you to display a special dialog box, enter method names and parameters, and return an appropriate value.

At first glance, the process for adding plug-ins seems rather involved, consisting of 14 steps for standard plug-ins, and nine for panel plug-ins. However, these steps are all documented and bookmarked in template files available from the Tutorial panel. Many of the steps are as simple as replacing placeholder text with something meaningful for your plug-in. Soon this will be even easier; Eagle Software plans to develop a Raptor API wizard, so many of these steps will appear as choices in a dialog box.

Again, each of the steps needed to create your own panels and install them in Delphi is explained in the tutorial. The main steps include the following:
- Create a Delphi package to contain your panel.
- Create a unit to implement the functionality of your new panel.
- Install and test your new panel.

Looking to the Future

Raptor's goal, as explained by its chief architect, Mark Miller, is to allow developers to code faster than they can think. While such a goal may be unattainable, I think you'll be amazed at how closely Raptor allows you to approach the speed of thought. And there's more to come. At the time of this writing (one week after the Borland Developers Conference), Raptor is still in beta; the main focus of the beta process continues to be adding and enhancing features, rather than fixing bugs. Therefore, I am certain this product will be even more impressive when it is released. (At this writing, Eagle Software had planned to release Raptor in late September.) Let's take a peek at some of the projected enhancements.

The next beta version will provide better support for high-speed navigation, with several new features. These will include cursor toggling between a method's declaration and its implementation, the ability to jump to any method in the current file using a combo-box pick list, and more. There will be a customizable speedbar at the top of the edit window. New directory and file comparison tools are also planned. The Statistics panel will be expanded to track time spent compiling (so if you spend a large portion of your day watching the compiler's hourglass, you can finally provide evidence to your boss — and get that new 200+ MHz Pentium you deserve.) In addition, Raptor's interface will be significantly expanded.

Can you tell how excited I am about this product? I am not alone. Of all the vendor exhibits at the Borland Developers Conference in Nashville, none drew more consistently large crowds than that of Eagle Software. As outrageous as it sounds, Raptor was being expanded as it was being demonstrated — with new templates and with a new panel contributed by a beta tester. I expect this is only the first of a number of articles I will write about Raptor. I plan to fully explore its API, and add a panel or add-on of my own, not to mention some new templates. Watch for detailed information on exploiting Raptor's open interface in future issues.

I would not be at all surprised to see Raptor win many awards after its release, including Product of the Year. Those of you who have recently upgraded from Delphi 2 to Delphi 3 are no doubt pleased by the many enhancements. Well, you'd better sit down. When you add Raptor to Delphi 3 (the only version for which it is available), you may feel — as I did — that you have moved from Delphi 3 to Delphi 4! There's that much of a difference.

My recommendation is simple: If you do any amount of coding at all, get Raptor. If you work for a company that supplies all your Delphi tools, don't wait for your boss to get this for you — get it yourself! (At the introductory price of US$149, you'll have to search far and wide to find a better deal.) With the immediate and significant increase in productivity, it may not be long before every Delphi 3 developer in your shop has Raptor. You may even find yourself promoted because of your forward vision. With this powerful and revolutionary tool, this is truly an exciting time to be programming in Delphi. Raptor definitely RADifies Delphi. ∆

INFORMANT FACT FILE

Raptor is the ultimate Delphi add-on. It enhances Delphi's editor with powerful navigational tools and a comprehensive set of over 600 keyboard templates. These templates are customizable and extendable, and provide quick and easy access to common Object Pascal structures. Raptor also includes powerful panels that surround the IDE editor and provide many useful tools, including a Hot List to access commonly used files, a file search panel, a file manager panel, etc. With the Raptor API, you can add your own panels, or customize Delphi's IDE — without writing an expert.

At this writing, Eagle Software planned to release Raptor in late September, 1997.

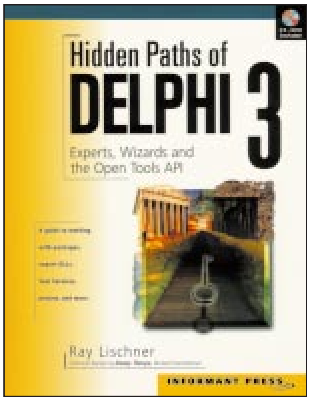
Eagle Software, Inc.
12021 Wilshire Blvd., Suite 655
Los Angeles, CA 90025
Phone: (310) 441-4096
Fax: (310) 441-1996
Web Site: http://www.eagle-software.com
Price: Introductory price (valid for 2 months after release) US$149; after this period, US$199.

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

table of CONTENTS

Exploring Delphi's Hidden Paths

The release of Ray Lischner's eagerly anticipated book coincided with this year's Borland Developers Conference. Although an ample supply was available — several hundred at least — all of them sold out before the end of the conference. This is just the beginning. I can't think of any other new Delphi author who created more of a sensation than this author did with his *Secrets of Delphi 2* (see my review in the February, 1997 issue of *Delphi Informant*). If we compare this latest journey into undocumented territory with the earlier one, we find many similarities and a few important differences. I'll begin with a general comparison; then, through a series of questions, I'll discuss some of the major topics Lischner discusses.



Both *Secrets of Delphi 2* and the newest book, *Hidden Paths of Delphi 3*, concentrate on aspects of Delphi that are either poorly documented or not documented at all in Borland's manuals. Both contain information you would have a hard time finding elsewhere. While *Secrets* explores many aspects of Delphi programming — particularly some of the more advanced aspects of component writing — *Hidden* focuses exclusively on Delphi's Open Tools API. In a practical sense, you learn how to write Delphi experts — or wizards, as we call them these days. While *Secrets* provides excellent discussions of the differences between Delphi 1 and 2, this current volume contains only minimal information on the earlier Delphi versions. Still, as Lischner points out, many of the function calls are applicable to Delphi 2, at least.

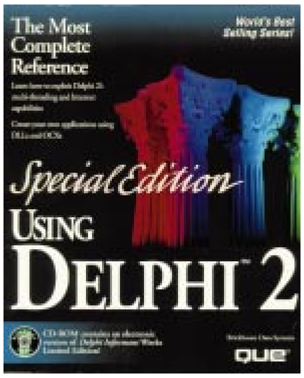
Both books share a similar writing style. There are quite a few authors who write about advanced Delphi topics. However, I am not aware of anyone who writes more clearly about such topics, making them accessible to a large number of Delphi developers. Some might criticize Lischner's use of repetition in his exposition of topics. However, his well organized and methodical approach increases his potential audience. In fact, anyone with a solid foundation in Delphi could read this book with much benefit. At the very least, anyone who adds this reference to their Delphi library will end up with a nice collection of new Delphi experts, including an enhanced Project Browser and

"Exploring Delphi's Hidden Paths" continued on page 38

Special Edition Using Delphi 2 and *Special Edition Using Delphi 3*

Most readers are probably familiar with QUE's popular *Using* series. The particular software covered can be anything in a broad spectrum, from Fortran to PageMaker. One of my first Pascal books was *Using Turbo Pascal*, long since passed on to my son. In going through my library, I found three remaining titles: *Using Turbo Prolog*, *Using Reflex*, and *Using WordPerfect for Windows*. The common thread that runs through all these books is the publisher's aim to provide a complete reference in a single volume. This goal is evident in the volumes I'll be discussing in this review: *Special Edition Using Delphi 2* and *Special Edition Using Delphi 3*.

While *Using Delphi 2* can't boast a well-known author among its nine contributors, it hardly suffers. In fact, all the authors are active developers rather than professional writers, giving the volume a "real world" and



practical flavor. This will delight those who are put off by too much theory and text that falls into the nether regions of advanced topics. By contrast, the cadre of authors who produced *Using Delphi 3* includes many Borland engineers among its numbers.

The organization and content of the two books are similar, but they maintain their individuality. Both provide a very comprehensive and thorough introduction to Delphi programming, best suited for developers well versed in programming but still learning Delphi. However, if you've mastered Delphi's essentials, and are looking for something more esoteric, these books will probably disappoint you. Before you make any decisions, I'll discuss some of the specific topics covered in each.

Using Delphi 2. Like most Delphi books that aspire to be comprehensive, *Using Delphi 2* begins by presenting some of the basic principles of Windows programming using Delphi. Here you'll find all the expected topics, from an introduction to Object-

"Using Delphi 2 & 3" continued on page 38

Exploring Delphi's Hidden Paths (cont.)

some useful components I'll be discussing later. So, what are the topics?

Quiz Time! This is new material for most of us — lots of new terms and concepts, and dozens of new methods. Therefore, I'll introduce many of the topical discussions with a question or a series of questions. (If you can answer all of them completely, then you should think about writing a book yourself.) All these are questions Lischner answers and expounds upon.

Working with memory and saving and retrieving files are always important topics. Internally, how does Delphi use read-write memory? Under what circumstances is read-only memory used? What problems can occur if we are unaware or ignore the differences? What kinds of streams can best be used in Delphi experts? What are interface streams, and how can we use them? What internal methods does Delphi make available to us, so we can edit a form definition (*.DFM) file as text? These questions certainly imply that we have access to a lot of hidden power, and we're just getting started.

Communication is also important in a Delphi application, or in a Delphi tool. The Open Tools API provides a lot of control here, as well. Consider these questions: How can we use drag-and-drop functionality in our expert DLLs? How do we get information about, and access, Delphi's menu system? (One chapter focuses exclusively on menu interfaces in add-in experts.)

The user interface is a crucial part of Delphi communication; but there's an internal, event-driven system that kicks into gear as soon as an event is invoked. Question: What are notifiers, and how are they used in Delphi experts? Lischner shows us how notifiers provide information about major occurrences in the Delphi IDE, such as opening or closing units, opening or closing projects, and so on. He explains that add-in experts have a special notifier that we can use to keep track of events and file manipulations.

To make working with notifiers easier, Lischner creates *TProjectNotifier* and *TModuleNotifier*. These classes encapsulate the structure of these notifier classes in a way that makes them easily available when we're writing experts. The various settings and events are available as properties, and many of the internal details are handled for us. In fact, there are event handlers for each notification in these classes.

Have you heard about or worked with editor interfaces? If your expert will be modifying code, whether at the project or unit level, you must be concerned about text editing. Editor interfaces provide easy access to Delphi's Code Editor. In demonstrating the uses of this interface, Lischner identifies some serious bugs in the first release of Delphi 3 (indicated by a bug graphic and the word "warning" in the margin). This alone should make this a worthwhile investment for anyone working with Delphi experts. No doubt, Lischner spent many hours crashing his system so we wouldn't have to.

This new book by Ray Lischner is right at the top of my list of favorite Delphi books, next to his *Secrets of Delphi 2*. There's very little I can find to complain about. I would have liked to have seen information about which functions are available in which Delphi versions. However, the book clearly identifies itself as a "Delphi 3" book, so I shouldn't necessarily have expected that. In conclusion, I recommend this book very highly to every serious Delphi programmer. Even if you've never considered writing a Delphi expert, this book will show you why and when you should, and how to go about it.

— *Alan C. Moore, Ph.D.*

Hidden Paths of Delphi 3 by Ray Lischner, Informant Press, 10519 E. Stockton Blvd., Ste. 100, Elk Grove, CA 95624, (916) 686-6610.
ISBN: 0-9657366-0-1
Price: US$39.99
(350 pages, CD-ROM)

Using Delphi 2 & 3 (cont.)

Oriented Programming to a tour of Delphi's IDE. The section on Object Pascal is brief but cogent; and all these topics are covered in detail.

If you haven't explored all the tools that ship with Delphi, both volumes include useful discussions on ReportSmith, the Database Desktop, the Local InterBase Server, and Delphi's built-in debugging tools. While most of the topics are handled in a succinct (some would say cursory) manner, there are exceptions. One of the better discussions I've seen is that of threads, as well as the discussion on writing and using DLLs. The examples provided are very good.

There are chapters on using Delphi for database and Internet programming, as well as a chapter on component writing. While they provide an excellent foundation, they don't often venture into more advanced topics. So, if you want to get heavily involved in component writing, you should read Ray Konopka's *Developing Custom Delphi 3 Components* [Coriolis Group Books, 1997], and Ray Lischner's *Secrets of Delphi 2* [Waite Group Press, 1996].

Using Delphi 3. Compared to its predecessor, *Using Delphi 3* is expanded considerably. It's considerably longer, and most of the topics are discussed in greater depth. It also includes additional chapters on advanced Object Pascal topics, such as Run-Time Type Information and the seldom-discussed Math unit. There are hidden gems in both volumes, but several are expanded quite a bit in the latter version. For example, I was delighted to find information about using the TDump utility to gather information about a DLL; I was also pleased to see a discussion of thunking, and the various methods used to enable communication between 16- and 32-bit DLLs.

"Using Delphi 2 & 3" continued on page 39

Using Delphi 2 & 3 (cont.)

Eric Uber, the only contributor to both volumes, contributed an excellent chapter in *Using Delphi 3* about using C++ with Delphi. Among other things, he shows the reader how to introduce C++ code into a Delphi project by using object (*.OBJ) files or DLLs, concluding with a discussion of type conversions (with a comprehensive chart).

Similar to *Using Delphi 2*'s excellent discussion of threads, *Using Delphi 3* also provides an excellent introduction to this topic — one of the major capabilities introduced with Delphi 2. Starting with an overview of multitasking, and covering the *Synchronize* procedure, thread priorities, and thread cooperation, both volumes will give you a good start in writing multithreaded applications. *Using Delphi 3* goes further, including a lengthy discourse on Critical Sections, or how you can protect vital data or processes.

I would recommend either of these volumes to any programmer who is looking for a complete reference for either of the Delphi 32-bit environments. Of the two, I prefer *Using Delphi 3*, because it provides additional information and is written at a more advanced level. *Using Delphi 2*, on the other hand, provides an excellent, practical introduction to this rich programming environment.

— *Alan C. Moore, Ph.D.*

Special Edition Using Delphi 2 by Jonathan Matcho, et al., QUE, 201 W. 103rd St., Indianapolis, IN 46290, (800) 428-5331 or (317) 228-4231.
ISBN: 0-7897-0591-5
Price: US$49.99
(891 pages, CD-ROM)

Special Edition Using Delphi 3 by Todd Miller and David Powell, et al., QUE, 201 W. 103rd St., Indianapolis, IN 46290, (800) 428-5331 or (317) 228-4231.
ISBN: 0-7897-1118-4
Price: US$49.99
(1,043 pages, CD-ROM)

Nobody's Perfect ... Not Even Delphi 3

A "must" upgrade for Delphi developers, Delphi 3 provides a host of new features and functionality that make it, perhaps, the best environment ever for developing Win32 applications. Yet, because no "perfect" development tool has been developed, it behooves us to note some of Delphi 3's shortcomings as well; in doing so, we'll be better informed about what to use, avoid, or work around.

The VCL has always been a centerpiece of the Delphi architecture. Therefore, the robustness and flexibility of these components are critical factors in the product's success. Delphi 3 introduced several new components to the VCL, but my experience has shown that some of these new controls don't meet Delphi's usual standards. With that background, I offer some lessons learned from working with Coolbars and Decision Cube controls.

Coolbars. My experience using *TCoolbar* and its related components (*TToolbar*, *TSpeedbutton*, and *TToolbutton*) has been quite frustrating. If you intend to use them, there are several issues to note. Using a *TSpeedbutton* on a *TToolbar* when the *Flat* property is set to *True* creates a vertical line up the center of the button when your system has a recent version of ctrl3d32.dll. You can avoid this *TSpeedbutton* glitch by using *TToolbuttons*, but these also have problems, such as confusing and sometimes inconsistent resizing.

Another negative of the *TToolbutton* control is its reliance on using a *TImageList* for images. While there are some advantages to using imagelists, I find it much easier to work with single images for a toolbar button. For example, the seemingly simple task of moving a button between toolbars will usually force you to edit the *.res* file that that imagelist references. Adding additional controls (such as a *TComboBox*) to a *TToolbar* can cause problems if the toolbar already has *TToolbuttons* contained in it. I found changing the positioning of the combobox would often cause "Out of Index" and "Access Violation" errors. Then the toolbar became unstable at design time.

I haven't seen a freeware or shareware "Coolbar" replacement, but for "Office97-like" toolbars, a great solution is to use Jordan Russell's *TToolbar97* component suite. These freeware VCLs are my newest entries to the "All-Wagner Team" (see the August 1997 issue of *Delphi Informant*). Download them at http://members.aol.com/jordanr7/index.htm.

Decision Cube controls. A second type of native VCL component to approach cautiously is the Decision Cube controls (*TDecisionCube*, *TDecisionGrid*, etc.). I've determined, through company experience, that these controls work well for basic displaying of data, but if you intend to do much more with them — watch out. As with the Coolbars, there are issues you should be aware of. For instance, accessing data and dimension cells, and working with data inside them, can be difficult; and much-needed events, such as *OnSelectCell*, aren't fully implemented. Also, Help files and printed documentation, in some cases, aren't correct in listing methods or properties supported. Thus, be sure to use the .INT files to get correct parameters for methods and functions. The Decision Cube controls are the lone group of VCLs created by Borland that don't have source code included with any Delphi package. Therefore, you don't have the same ability to correct or modify these controls as you do with other native VCLs.

Aside: revisiting business objects. On a different note, I wanted to briefly touch on the subject of business objects. Earlier this year, I did a two-part series on creating and using business objects in Delphi. These two columns generated more interest on behalf of readers than anything else I've written. Most people responding to me via e-mail have been interested in employing business objects, but would like further information on the specific techniques of doing so. Other readers have been using business objects with Delphi, and wanted to share their ideas with others. I am in the process of trying to link together these Delphi developers interested in discussing business objects, so if you have an interest, drop me a note at rwagner@acadians.com, and I'll put you in touch with this group of people. Δ

— Richard Wagner

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