A New Spin on Delphi

Delphi Graphics Programming

ON THE COVER

7 A New Spin on Delphi — Peter Dove and Don Peer
Misters Dove and Peer begin a series on graphics programming by developing the *TGMP* component. The series will follow it from inception, through development, to a fully-functional 3D rendering component.

FEATURES

14 Informant Spotlight — Danny Thorpe
A member of the Delphi R&D team, Mr Thorpe begins a two-part series that explores virtual methods and polymorphism.

19 DBNavigator — Cary Jensen, Ph.D.
INI files or the Windows 95 registry? Dr Jensen offers an overview of how to use either — or both.

25 OP Tech — Ray Lischner
Undocumented — until now; Mr Lischner shares some of the internal messages of the Delphi VCL.

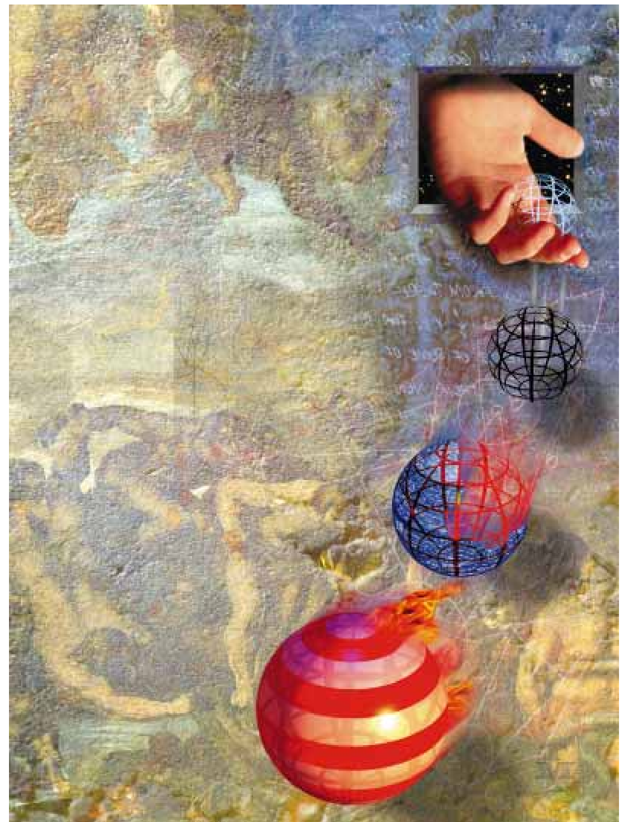
34 In Development — Mark Ostroff
Object names getting unwieldy? Mr Ostroff shares a set of Delphi naming conventions that can make your programming life easier.

38 Delphi at Work — David Faulkner
Offering his *TBarCode* component, Mr Faulkner demonstrates creating Code 39 bar codes.

42 Case Study — Don Bauer
How good is Mr Bauer's *TWorldMap* component? NASA found it useful.

REVIEWS

45 InfoPower 2.0 — Product Review by Bill Todd
49 SysTools for Delphi — Product Review by Alan C. Moore, Ph.D.



Cover Art By: *Tom McKeith*

DEPARTMENTS

2 Symposium by Jerry Coffey
3 Delphi Tools
6 Newsline
55 File | New by Richard Wagner

I will be leaving Borland by the end of [October 1996] to take a job at Microsoft. This has not been an easy decision to make, but I have now been with Borland for 13 years, and I feel that it is time for me to try some new challenges.

— Anders Hejlsberg, Delphi co-architect (from CompuServe forum, BDelphi32)

After Anders

No — you didn't miss anything; there just wasn't a December issue of *Delphi Informant*. And there was no gap in your monthly Delphi coverage; it was strictly a logistical move to help *DI* get wider distribution and appear sooner each month on newsstands. So relax; with any luck, we'll get this magazine thing down soon.

On to the main topic: As you've probably heard by now (this is written on October 21st, 1996), Anders Hejlsberg has left for Microsoft. Hejlsberg has been with Borland since its inception. He created the now-legendary Turbo Pascal product that revolutionized the desktop computer industry with the first integrated development environment.

News of Anders' departure hit me like a thunderbolt. On the several occasions I've met Anders or heard him speak, I was invariably struck by his affability and — well — brilliance. It was comforting that my all-time-favorite development environment was in his skillful hands. What now would happen to the Delphi R&D team? Would others follow Anders to Microsoft? Would they just scatter to the four winds?

From all accounts, Hejlsberg had personal reasons for moving on; he's not so much "leaving Borland" as beginning a new endeavor. According to Zack Urlocker, Director of Product Management for Borland and a member of the Delphi R&D team: "We're certainly sad to see Anders leave, but it was a personal decision on his front. After being here 13 years — since college — he's decided to try something different. The rest of the team is here to stay."

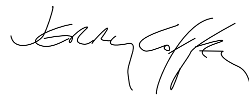
The most important thing to readers of this magazine, however, is the health of Delphi without Hejlsberg. Fortunately, the view of the product being in the hands of one man is naive. Again, from Anders: "For those of you worried about Delphi's future, I want to assure you that the product is in the hands of an incredibly competent team of people for whom I harbor the deepest respect. Back in the old Turbo Pascal days it was possible for one person to write and maintain an entire product. This is no longer the case. Delphi was built by a team, and I have full confidence in the team's ability to develop and deliver new versions of Delphi. In fact, the Delphi team at this point is almost twice the size it was when we shipped 1.0 in early '95. And Delphi97 is going to be a great product [with] multi-tier database access and COM/ActiveX support."

Urlocker is also encouraging: "The architectural work that Anders covers is complete for Delphi97 and we're in beta. Anders' departure won't affect the ship date or features going forward. Chuck Jazdzewski will move up from co-architect to chief architect. Chuck's been here longer than I have and worked closely with Anders for eight years. In fact, he designed the VCL and the UI builder that's such a powerful aspect of Delphi. He's also played an important role on Latté and our upcoming C++ product. So again, even though we'll miss Anders, I think the Delphi team is in very good shape to ship a very impressive release. There is a whole crew of folks working on Delphi97 many of whom have been involved since the very beginning of the project and have a strong vision for where we are taking it in this next release and beyond."

As you know by now from its debut at Comdex, Borland has given the "Delphi treatment" to C++. This one's been kept under tight wraps and doesn't have a shipping name as of this writing. It masquerades under various names such as Ebony and Pronto, but what's important is what the product represents to the industry and to Borland.

Delphi boosted academia-bound Pascal into the stratosphere. Given the preponderance of C++ in the programming world — as Urlocker says "some folks think more naturally in C++" — Ebony could make Delphi look like a moderate success. Things look bad for Borland right now, but I can remember a darker period. In the winter of 1994-95, Borland was relying on the flagging sales of Paradox and dBASE, and had pinned all hopes on a Pascal-based product no one had heard of.

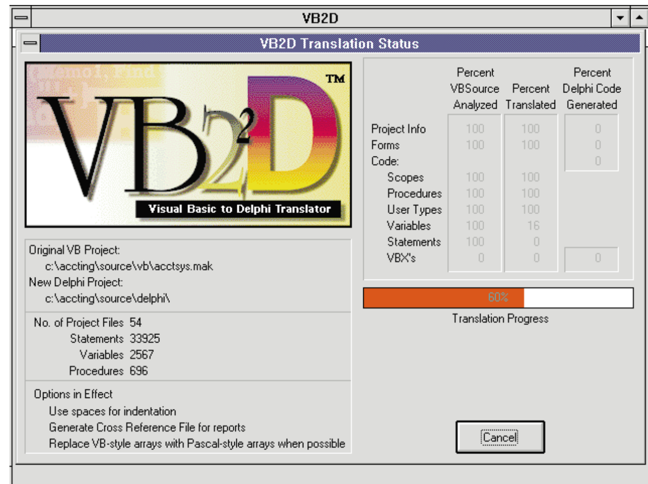
Delphi Informant wishes you well Anders; you've given us an incredible tool. Gotta run though — I have this Delphi project I'm working on ...



Jerry Coffey, Editor-in-Chief

Internet: jcoffey@informant.com
CompuServe: 70304,3633
Fax: (916) 686-8497
Snail: 10519 E. Stockton Blvd., Ste. 100, Elk Grove, CA 95624

Eagle Research Announces Version 2.0 of VB2D Translator



Eagle Research Inc. of San Francisco, CA has released version 2.0 of its Visual Basic (VB) to Delphi translator. ***VB2D 2.0*** translates applications created in VB 3.0 or 4.0 to 32-bit Delphi 2.

VB2D 2.0 also features online reporting, conversion of most database code, automatic replacement of common VBXes, and better code output.

According to Eagle Research, Delphi's new variant, string, and currency data types make it easier to create Delphi code from VB applications.

For VB projects that use Access or other databases via the JET Database Engine, VB2D replaces standard VB data-aware controls with Eagle's JETset controls for Delphi.

JETset controls are derived from standard Delphi data-aware controls, but instead of connecting to the BDE, they connect directly to Microsoft's JET Database Engine.

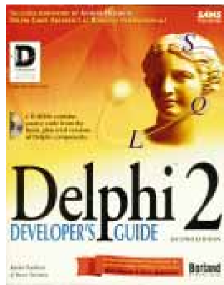
VB2D is offered in standard and professional editions. The Professional Edition includes a copy of Eagle's JETset product, additional reporting capabilities (such as side-by-side listing of VB and Delphi code), and source code.

Price: Standard Edition, US$150; and Professional Edition, US$450. Both versions include a 30-day money-back guarantee.

Contact: Eagle Research Inc., 360 Ritch St., Ste. 300, San Francisco, CA 94107
Phone: (415) 495-3136
Fax: (415) 495-3638
E-Mail: sales@xeaglex.com
Web Site: http://www.xeaglex.com

New Delphi Book

Delphi 2 Developer's Guide, 2nd Edition
Xavier Pacheco & Steve Teixeira
SAMS Publishing



ISBN: 0-672-30914-9
Price: US$59.99
(1,322 pages, CD-ROM)
Phone: (800) 428-5331

ExceleTel Releases TeleTools-Delphi 1.04

ExceleTel Inc. of Raleigh, NC has released ***TeleTools-Delphi 1.04***. TeleTools-Delphi is a set of VCL modules that provides native access to telephony functions for Delphi's IDE.

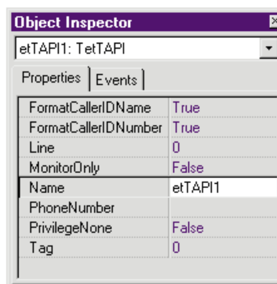
Using a component to access Windows telephony (TAPI) functions, TeleTools enables developers to add caller ID, dialing, call logging, and screen pops. After the TAPI component is placed on a form, a developer can access 22 properties, 9 events, 18 methods, and additional TAPI functions.

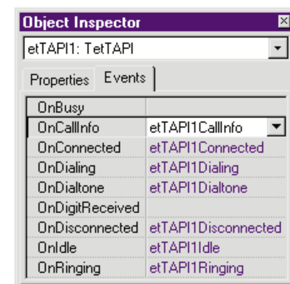
Future releases of ExceleTel products include TeleTools-OCX, as well as the second series of TeleTools products: TeleTools2-Delphi and TeleTools2-OCX. The TeleTools2 series will add .WAV functionality with access to telephony record and playback features.

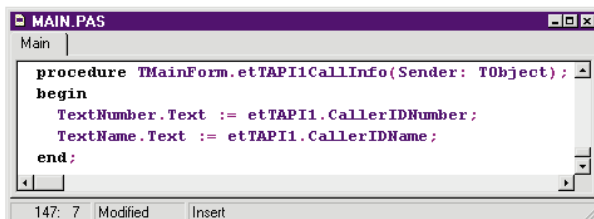
Price: TeleTools-Delphi, US$99.

Contact: ZAC Catalog, 1090 Kapp Drive, Clearwater, FL 34625
Phone: (800) 463-3574 or (813) 298-1181
Fax: (813) 461-5808
E-Mail: sales@zaccatalog.com
Web Site: http://www.zaccatalog.com

Contact: ExceleTel Inc., 5142 Simmons Branch Trail, Ste. 100, Raleigh, NC 27606
Phone: (919) 233-2232
Fax: (919) 233-2230
E-Mail: sales@exceletel.com







Tamarack Associates Releases Rubicon for Delphi

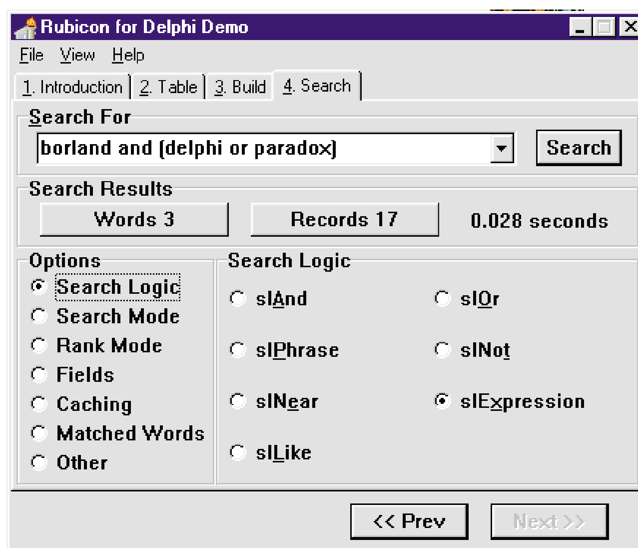
Tamarack Associates of Palo Alto, CA has released ***Rubicon for Delphi***, a database search engine.

The Rubicon technology allows the end-user of an application to perform searches using wildcards; apply AND, OR, and NOT logic to the search; and narrow the search without regard to the underlying database or field structure.

Rubicon encapsulates this search technology in a set of three native Delphi VCL components that build indexes, update indexes, and execute searches, respectively.

Rubicon performs all searches at keyed index-like speeds by building a single Rubicon table that indexes all the words in the source table(s) and their locations (most Rubicon searches never read the source table[s]).

Reads and writes against this table are minimized by built-in compression technology.



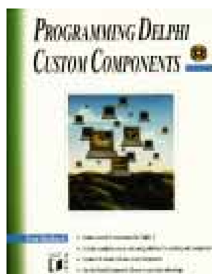
Rubicon reduces the complexity associated with searching normalized tables and tables containing BLOB data. All components are compatible with Delphi 1 and 2.

Trial versions of Rubicon are available from Tamarack's Web site, as well as the Delphi and BDelphi CompuServe forums (filenames: RUBICON.ZIP and RBCNDEMO.ZIP).

Price: US$99, includes free updates and support via e-mail. Rubicon may also be ordered through CompuServe shareware registration ID 11536.
Contact: Tamarack Associates, 868 Lincoln Ave., Palo Alto, CA 94301
Phone: (415) 322-2827
Fax: (415) 322-2827
E-Mail: 72365.46@compuserve.com
Web Site: http://www.tamaracka.com

New Delphi Book

Programming Delphi Custom Components
Fred Bulback
M&T Books



ISBN: 1-55851-457-0
Price: US$39.95
(420 pages, CD-ROM)
Phone: (212) 886-1068

IntegrationWare Ships Speed Daemon for Delphi v1.1

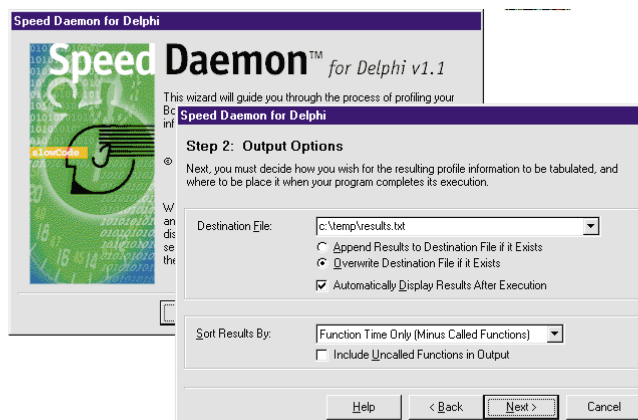
IntegrationWare, Inc. of Deerfield, IL has released ***Speed Daemon for Delphi v1.1***, a source code profiler. It provides an analysis engine for optimization and performance tuning.

Using Speed Daemon, developers can monitor the efficiency of key sections of code. Given any Delphi 1 or 2 project, the utility produces statistics for each function. These include the number of times a function is called, the total time spent executing the function relative to other pieces, and average time per call. Speed Daemon's output can be printed or viewed onscreen.

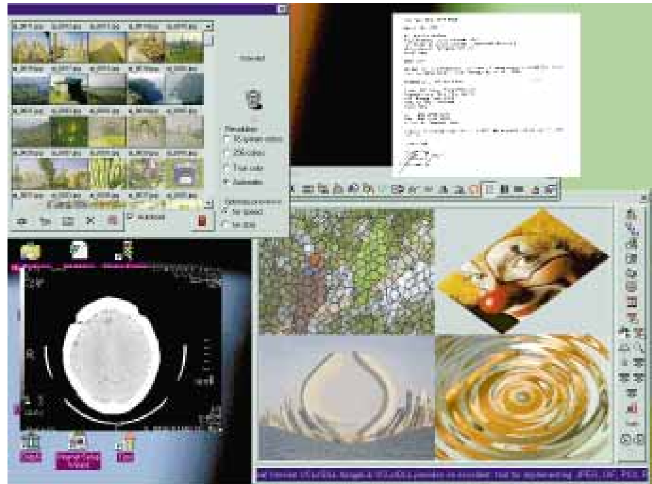
Speed Daemon doesn't require any changes to a project or source code. It parses the source code and adds any required constructs to generate function timings. This modified version can then be recompiled. The entire process is guided by a wizard-like interface.

Speed Daemon works with DLLs and OLE automation servers developed in Delphi, and can be used on single-threaded and multi-threaded applications.

Price: US$59
Contact: IntegrationWare, Inc., Deerfield Tech Center, 111 Deer Lake Road, Ste. 109, Deerfield, IL 60015
Phone: (888) 773-1133 or (847) 940-1133
Fax: (847) 940-1132
Web Site: http://www.integrationware.com



SkyLine Debuts ImageLib Corporate Suite Document Imaging Package



SkyLine Tools of North Hollywood, CA debuted ***ImageLib Corporate Suite Version 1.0***. This version of the ImageLib software includes the features of ImageLib's Combo Package and ImageLib@the edge, image manipulation, and correction package.

The ImageLib Corporate Suite package incorporates all the formats of the ImageLib Combo (.PCX, .PNG, .TIF [baseline], .JPG, .BMP, .WMF, et al.) plus ISIS and TWAIN scanning support, multimedia formats, a video frame grabber, thumbnails, and BLOB support.

Also included are point-and-click image correction effects such as brightening, contrast, gamma correction, color reduction, and rotation package.

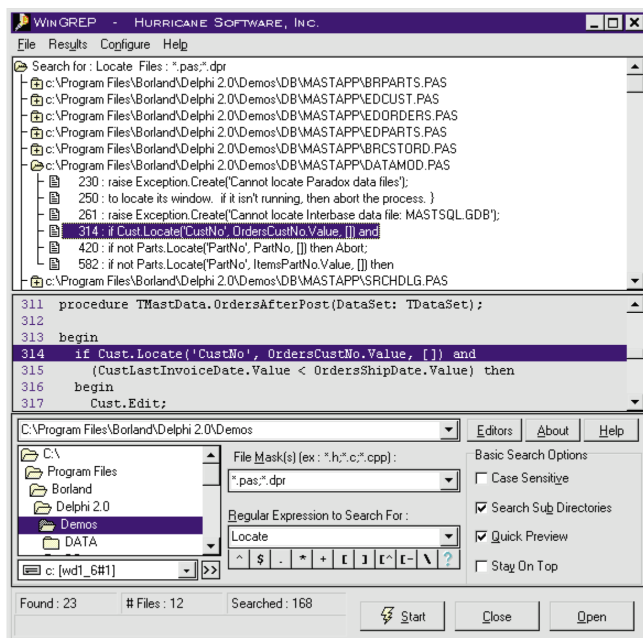
The ImageLib Corporate Suite features special effects such as mosaic, page curl, wave, and transition effects. To these features, the suite adds multipage scanning, low-level scanning, as well as enabling users to read and write the following TIFF formats: TIFF III, CITT; TIFF IV, CITT; Multipage TIFF; Packbits; and LZW. Other formats included are Photo-CD (Kodak), .IMG, in addition to .PCX and .EPS (read only).

Skyline Tools is planning to release a new Web kit that will contain a progressive display for .GIF, .PNG, and .JPG, as well as animated .GIFs.

Price: US$499
Contact: SkyLine Tools, 11956 Riverside Dr., Ste. 107, North Hollywood, CA 91607
Phone: (818) 766-4561
Fax: (818) 766-9027
E-Mail: 72130.353@compuserve.com
Web Site: http://imagelib.com

Hurricane Software Announces Multi-File Search Utility

In Blue Springs, MO, **Hurricane Software, Inc.** is shipping ***WinGREP***, a multi-file search utility that allows programmers to locate text strings in source code files. WinGREP features a collapsible tree of search results that expands to reveal the matches for a filename.



WinGREP also includes a Quick-Preview window. As results are highlighted in the tree, the Quick-Preview window shows several lines of text around the search match.

WinGREP can automatically synchronize an editor or IDE with the results of a search. It will open the file and position to the line of the match.

Other features include Windows 95 long filename support, the ability to save, load, and print search result sets, a button bar to help create regular expressions, UNIX text file support, online Help, and install/uninstall.

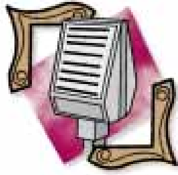
WinGREP also ships with the Hurricane Editor, a multi-file text editor that features word wrapping, and search and replace.

Price: US$39
Contact: Hurricane Software, Inc., 2401 SE 7th St., Blue Springs, MO 64014
Phone: (816) 373-9252
E-Mail: mail@hurricanesoft.com or 102470.2564@compuserve.com
Web Site: http://www.hurricanesoft.com

News

L I N E

January 1997



Borland Cuts Staff; Expects Second Quarter Loss

Scotts Valley, CA — Borland has reduced its corporate headcount by approximately 15 percent or 125 individuals, as part of its new worldwide restructuring and realigning program.

Paul Emery, vice president and chief financial officer, said Borland's cost reduction measures are expected to produce annual savings from US$15 million to US$17 million.

In addition, Borland announced it expects to report a loss of US$.32 to US$.36 per share on revenues of approximately US$36 million for the quarter ending Sept. 30, 1996.

According to Borland, the quarter's losses were due to slower than expected transition of its sales, marketing, and development efforts in moving from desktop markets into departmental and corporate technologies.

ICG Relocates, Discontinues CompuServe Forum

Elk Grove, CA — Informant Communications Group, Inc. (ICG) has relocated its corporate headquarters to 10519 E. Stockton Blvd., Ste. 100, Elk Grove, CA, 95624-9703, effective December 1, 1996. The ICG telephone number remains (916) 686-6610 and the company's fax number remains (916) 686-8497.

In addition, ICG has announced it will discontinue its CompuServe Forum (GO ICGFORUM) beginning January 1, 1997. The company will place all of its customer support services and efforts into its Web site at http://www.informant.com.

The ICG Web site contains code listings and applications referenced in all Informant publications, and will also contain subscription support and threaded discussion areas.

Borland Announces Interim President and CEO

Scotts Valley, CA — Borland has named Whitney G. Lynn acting President and Chief Executive Officer. Lynn succeeds William F. Miller, who returns to his role as Chairman of the Board. Miller had been the acting CEO of Borland since the resignation of Gary Wetsel.

Borland is continuing to search for a permanent CEO, and Lynn will serve until a new CEO is named. Lynn is a technology industry veteran with a broad range of executive and management experience. Most recently, Lynn has served as a consultant in the integration process surrounding Borland's pending acquisition of Open Environment Corp.

Additionally, Borland announced replacements for Paul Gross, departing Senior Vice President of Research and Development. Heading enterprise development is Jothy Rosenberg, a Borland development vice president. Jeff Rudy, also a Borland development vice president, leads the Scotts Valley research and development efforts.

Learning Tree International Offers Delphi Courses

Learning Tree International is hosting two Delphi courses in January. The first, Object Oriented Analysis and Design, will be held January 6-10 at the Learning Tree Education Center in Washington, D.C. Delphi Application Development, scheduled for January 21-24, will be held at the Learning Tree Education Center in Los Angeles, CA. For more information, contact Learning Tree International at (310) 417-9700 or http://www.learningtree.com.

ICG Announces *Delphi Informant* and *Oracle Informant* on CD-ROM

Elk Grove, CA — Informant Communications Group, Inc. (ICG) has announced the release of *Delphi Informant* and *Oracle Informant* magazines on CD-ROM. Both titles will offer duplicate versions of all articles as they appeared in print during 1996, with simplified keyword access through a word index.

Each CD includes the code, supporting files, screen shots, and graphics for the stories listed. Readers can print any page within the CD.

A 16-page color booklet ships with each title. It features installation instructions, contents, and additional information.

Orders for *Delphi Informant Works: 1996* and *Oracle Informant Works: 1996* may be placed with ICG by calling (800) 88-INFORM (in the US), or (916) 686-6610; or by faxing (916) 686-8497.

Each title is available for US$49.95, plus US$5 shipping and handling (US$15 for international customers).

Borland Ships Java-Enabled InterBase SQL Database Server

Scotts Valley, CA — Borland has released InterClient for the InterBase cross-platform SQL database server. InterClient, written in Java, provides JDBC-compliant connectivity for InterBase.

Built on JavaSoft's Java Enterprise-API set and its JDBC protocols, InterClient streamlines distributed transaction processing at both the Java client and database server. It eliminates installation and maintenance for companies committed to Web-centric technologies for information distribution and application development. InterClient also targets Web-oriented VARs and consultants creating Web sites and applications.

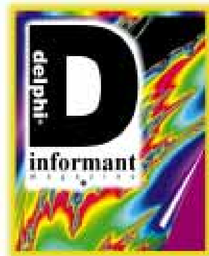
InterClient is one component of Borland's Internet and Intranet initiatives. Their RAD environment for Java, code-named Latté, will help developers extend the value of InterClient with its Java object component model.

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

Errors & Omissions

Reference to downloadable code accompanying Dan Miser's article, "The INISource Component," was inadvertently omitted from the October issue of *Delphi Informant*. This code is available for download from the Informant Web site at http://www.informant.com, file name DI9610DM.ZIP.

We apologize for any inconvenience this omission may have caused.



ON THE COVER

Object Pascal / Delphi 2

By Peter Dove and Don Peer

A New Spin on Delphi

Delphi Graphics Programming: Part I

Delphi is rapidly gaining respect as a games and graphics programming language. It offers a graphical interface, rapid application development (RAD), and the use of components at design time. More importantly, Delphi offers true compiler technology — *speed*.

This article begins a series about Delphi graphics programming. In this series, we'll develop a component, *TGMP*, following it from inception, through development, to a fully-functional 3D rendering component. Our journey will take us through some basic topics such as properties, events, and property editors. Eventually we'll visit more exotic and advanced topics, such as device-independent bitmaps, sprites, palettes, optimization, pointers, memory management, inline assembler, and reading polygon data files.

```
unit Unit1;

interface

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

type
  TGMP = class(TComponent)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Graphics', [TGMP]);
end;

end.
```

Figure 1: The template unit created by Delphi.

If you are a C or C++ programmer, you'll find that writing this kind of component illustrates how to get Delphi to do all those things you may have taken for granted. Regardless, this discussion will bring you closer to discovering what makes Delphi such a great product for games and graphics development.

To summarize what we'll create throughout this series, a brief description of the *TGMP rendering engine* is in order. *TGMP* will enable you to create 3D worlds based on polygonal data, allowing users to move through and observe the world from all positions and angles. (The game Doom is an example of a rendering engine.) The *TGMP* engine will also include shading based on light source positions, clipping, collision detection, texture mapping, and Gourard shading.

Now let's set our minds into 3D mode and dive into the project.

Let the Games Begin

Begin by selecting **File | Close All** from Delphi's main menu. This will ensure that all items currently open are closed. Next, select **Component | New**. In the Component Expert, enter `TGMP` for the **Class Name**, `TComponent` for the **Ancestor type**, and

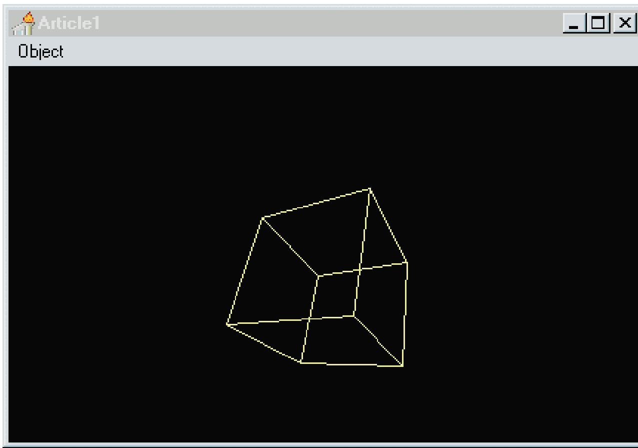


Figure 2: The cube rendered in wireframe.

Graphics for the **Palette Page**. (Because we want *TGMP* to be a non-visible component you can drop onto a form, it's derived from the *TComponent* class.)

After you have entered these values, select **OK**, and Delphi will create a template unit (see Figure 1). This is the template unit from which *TGMP* will evolve.

Let's start with something simple. We'll create *TGMP* to be a component capable of generating a wireframe cube and pyramid. Figure 2 shows how the cube will appear rendered in wireframe. The background color is black and is therefore the color of the canvas on which we are drawing. (Black is the usual choice for background color, although you can select any color.)

The main routines that must be developed for our wireframe rendering component are line-drawing and screen-clearing routines, 3D to 2D projection routines, and rotational calculations. The Graphic Design Interface (GDI) functions encapsulated within *TCanvas* make line-drawing and screen-clearing routines easy to write. (3D to 2D projection routines and rotational calculations are covered later in this article.) We'll begin by adding some data members to the component. At this point we need:

1) something to draw on and something to hold the background color;
2) variables to hold our viewing distance and variables for the screen height and width; and
3) a means to keep the handle of the window that receives the *TGMP* component.

Instead of using a device context, we'll use *TBitmap* as the backbuffer, because it will be compatible with the current color depth and it encapsulates the idea of a device context. This means we can set the height and the width of the bitmap, and leave *TBitmap* to deal with any necessary resource allocation. *TBitmap* also has a *Canvas* property, so copying between the bitmap canvas and form canvas will be easy. This backbuffer also allows us to draw on a hidden surface, which is then copied to the screen all at once, creating a smoother animation effect. To do this, place the following code in the **private** section of the *TGMP* component:

```
{ Place this code in the public section }
constructor Create(AOwner : TComponent) ; override;

{ Place this code in the implementation section }
constructor TGMP.Create(AOwner : TComponent);

begin
  { We must call the inherited Create of the TComponent
    from which we derived this class }
  inherited Create(AOwner);

  { Create the bitmap canvas }
  FBackBuffer := TBitmap.Create;

  { Get a pointer to the Form's canvas }
  FFrontBuffer := TForm(AOwner).Canvas;

  { Get the height and width of the window }
  ViewHeight := TForm(AOwner).Height;
  ViewWidth  := TForm(AOwner).Width;

  { Set the bitmap's height }
  FBackBuffer.Height := ViewHeight;
  FBackBuffer.Width  := ViewWidth;

  { Set up viewport }
  HalfScreenHeight := ViewHeight div 2;
  HalfScreenWidth  := ViewWidth div 2;

  { Get the handle of the window }
  FWindowHandle := TForm(AOwner).Handle;

  { Set the viewing distance }
  ViewingDistance := 200;

  { Set the Z distance }
  Z_Distance := -50;
end;
```

Figure 3: Overriding the *Create* constructor.

```
FBackBuffer: TBitmap;
FFrontBuffer: TCanvas;
ViewWidth, ViewHeight: Integer;
FWindowHandle: THandle;
HalfScreenWidth, HalfScreenHeight,
  ViewingDistance: Integer;
```

By placing this code in the **private** section, we can restrict access to data members through well-defined methods. (Think of the **private** section as the innards of a microwave. You wouldn't go fiddling with those innards unless you were a qualified microwave engineer. Access to the microwave is through a simple pushbutton user interface, equating to the **public** and **published** methods.)

Next we need to initialize the declared data members. To do this, we'll override the *Create* constructor as shown in Figure 3. The constructor is called to create the object. The *Create* method contains the code that allocates the memory and governs the look of the component. The **override** keyword indicates that we want to add to the inherited *Create* constructor. Note that Delphi allows you to cast *AOwner* as another object (*TForm*), an example of Delphi's polymorphism.

Note also that the parameter *AOwner* is in the *Create* constructor. *AOwner* is the owner of the component, the object respon-

sible for freeing the component before it frees itself. The owner will be the form on which the component is placed. This actually turns out to be quite useful because we can get the windows size and other information using Windows API calls, sending the window handle as a parameter.

We also want to destroy the *TBitmap* object, and release the device context when the object comes to the end of its scope. Thus, we need to override the destructor *Destroy*:

```
{ Place this in the public section of TGMP }
destructor Destroy; override;

{ Place this in the implementation section }
destructor TGMP.Destroy;
begin
{ Free the TBitmap }
  FBackBuffer.Free;
  inherited Destroy;
end;
```

Creating Properties

TGMP is now initializing and de-initializing correctly. Next, we should check if there are any variables that we want users to be able to set visually. Initially, two variables in *TGMP* fall into this category: *BackColor* and *ZDistance*. *BackColor* sets the background color and *ZDistance* sets the viewing distance from the screen to the object.

This article assumes the reader is familiar with Delphi properties. However, some discussion of basic implementation theory is appropriate. To do this, place the following code in the **private** section of the *TGMP* code:

```
FColor : TColor;
FDistance : Integer;
```

Then place the following code into the **published** section of the *TGMP* code:

```
property BackColor : TColor read FColor write FColor;
property ZDistance : Integer   read FDistance
                               write SetDistance default -50;
```

A property definition starts with the keyword **property** followed by the name of the property. This is followed by a colon, the type of the property, and the **read/write** keyword. The **read** keyword tells Delphi from where it should read the value for *BackColor*, and the **write** keyword tells Delphi how and where it should record the value a user may enter. The name of a method, rather than a variable, can follow the **read** and **write** keywords. This will allow some testing or screening of the data entered by the user.

The *ZDistance* property varies slightly in declaration from the *BackColor* property. *ZDistance* has **read** and **write** statements, but the **write** keyword uses a procedure, *SetDistance*, to save the value for the property. *ZDistance* also uses the keyword **default** with a value of minus 50. This is a bit of a misnomer — it doesn't really establish a default value for the property, but rather determines if the value is stored in the form file. If the default value is the same as the value in the form file, the value in the form file isn't altered.

Declaring a default value doesn't mean the value will be automatically assigned to the property; rather, you should provide an initial value for the property in the class' constructor. We will take this route with the *ZDistance* property to ensure that the default distance is at least minus 50. If this was not the case and the default value was zero, the user would be unable to see the object. The component would then behave as if users were viewing the program with their noses against the screen. The following code sets the property value and performs the initialization:

```
{ Set the FDistance  variable }
procedure TGMP.SetDistance(Distance : Integer);
begin
  FDistance := Distance;
end;

{ Initializing the ZDistance variable }
constructor TGMP.Create(AOwner : TComponent);
begin

   ...

  { Set the Z distance }
  ZDistance := -50;
end;
```

Basic 3D Procedures

All our variables are created and initialized, but they are of little use until we create some methods to do the 3D work for our component. Here are the methods we'll use:

```
{ Place in the private section }
procedure DrawLine3D(x1,y1,z1, x2,y2,z2 : Single);
procedure DrawLine2D(x1, y1, x2,y2 : Integer);
procedure SetDistance(Distance : Integer);

{ Place in the public section of TGMP }
procedure ClearBackPage;
procedure RenderNow(var Object3D : TObject3D);
procedure FlipBackPage;
procedure Rotate(x,y,z, angle);
procedure ChangeObjectColor(var Object3D : TObject3D;
                            Color : TColor);
```

Most of the methods are self-explanatory; however some require a little explanation. *FlipBackPage* copies the picture you were rendering in memory (on our temporary bitmap) and paints it on the form that holds the component. Use the *DrawLine3D* method to pass it the 3D coordinates of your line; it works out what that should resemble on a 2D screen, which is then drawn using the *DrawLine2D* method. The last method in the **public** section has a type that we have not yet addressed. The *TObject3D* is going to be our object structure. Its declaration is:

```
TPoint3D = record
  x,y,z : Single;
end;

TLine3D = record
  Start, End : TPoint3D;
end;

TObject3D = record
  LineStore   : array [0..100] of TLine3D;
  NumberLine : Integer;
  Color : Tcolor;
end;
```

Some 3D Theory

Currently, the object structure is simplistic. This will change radically over the next few articles. First, however, let's go over a few 3D math theories.

The first mathematical concept we'll cover is rotation of a point in 3D space. In this component, all rotations are performed around the point 0,0,0. This means that if you wanted to rotate a sphere on that point, the center of the sphere must be at 0,0,0. If not, the sphere would orbit around the point 0,0,0. Sometimes this is the desired effect.

For now, however, the object is defined and rotated about its local coordinate system. Be aware that more than one coordinate system exists:

- Local Coordinate System refers to the coordinates of the objects' vertices themselves.
- World Coordinate System refers to the position of the objects' vertices in a virtual world.
- Camera Coordinate System is the final position of the objects' vertices after being transformed through the camera matrix. This transformation positions all the objects as they would be if they were seen through a camera at a given position and angle of rotation.

These coordinate systems will be covered in greater detail in future articles.

Here are the formulas for rotation of an object through the `X`, `Y`, and `Z` axis:

Rotation around `Z`:

```
NewX := X * Cos(Angle) - y * Sin(Angle)
NewY := X * Sin(Angle) + y * Cos(Angle)
```

Rotation around `X`:

```
NewY := Y * Cos(Angle) - Z * Sin(Angle)
NewZ := Y * Sin(Angle) + Z * Cos(Angle)
```

Rotation around `Y`:

```
NewZ := Z * Cos(Angle) - X * Sin(Angle)
NewX := X * Cos(Angle) + Z * Sin(Angle)
```

Next we need to know how to convert a 3D line onto a 2D screen. This is a simple problem to solve. Simply divide `X` and `Y` by the `Z` value. However, doing this leads to a "zoomed" perspective, so we need to add the idea of viewing distance. You get the extreme zoomed effect because the algorithm assumes your nose is pressed against the screen.

Try holding an object close to your face and you'll see the zooming effect. By increasing the viewing distance, you reduce the zoom.

The following code converts from 3D to 2D with respect to viewing distance:

```
procedure TGMP.DrawLine3D(x1,y1,z1, x2,y2,z2 : Single);
var
  ScreenX1, ScreenX2, ScreenY1, ScreenY2 : Integer;
begin
  ScreenX1 :=
    HalfScreenWidth + Round(X1 * ViewingDistance / Z1);
  ScreenY1 :=
    Round(HalfScreenHeight - Y1 * ViewingDistance / Z1);
  ScreenX2 :=
    HalfScreenWidth + round(X2 * ViewingDistance / Z2);
  ScreenY2 :=
    Round(HalfScreenHeight - Y2 * ViewingDistance / Z2);
  DrawLine2D(ScreenX1, ScreenY1, ScreenX2, ScreenY2);
end;
```

You'll also notice we have two constants: *HalfScreenHeight* and *HalfScreenWidth*. Because we want objects to appear at the center of the screen, we need to add half the screen height to the `Y` value and half the screen width to the `X` value.

Finally, we'll create a variable of type *TObject3D*, fill it with 3D data, assign it a color, and we're ready to go. Listing One, beginning on page 11, shows the code for the *TGMP* unit, with some added data members. These members are filled in the *Create* constructor to determine the size of the window to render to. After this code has been added to the initial unit template, save the unit as GMP.PAS.

You are now ready to install the component into Delphi's component library. To do this, select **Component | Install** to display the Install Components dialog box. Select the GMP.PAS file, add it to the component list, and select **OK** to recompile. Delphi will return the Component Palette with a new Graphics page containing the *TGMP* component.

Our First Application

Now that you have installed the component, we can put it to work and create our wireframe application.

In our first application, we've created two arrays that hold the data for the cube and pyramid objects. This is to set the foundation for reading polygon data files to create the 3D objects. The arrays are declared in the **type** section of the application code:

```
TPyramidArray = array [0..47] of Integer;
TCubeArray    = array [0..71] of Integer;
```

In the **public** section of the source code for the wireframe application, notice the two variables of type *TObject3D* and a pointer to *TObject3D*, *CurrentObject*. We are using *CurrentObject* to keep track of the object the user selected. When the user selects a new object, we assign the current object pointer to the object selected.

You'll notice in the *Timer1.Timer* procedure that we use the pointer in the following format: `CurrentObject^`. By placing the carat after the pointer *CurrentObject*, we are referring to the object rather than its address. This enables us to have a generic place holder. Otherwise, we would need to test which object is selected, and have two sets of the code you see in the *Timer1.Timer* procedure (one set of code for each object we create). Typically, you would have to use a **case**

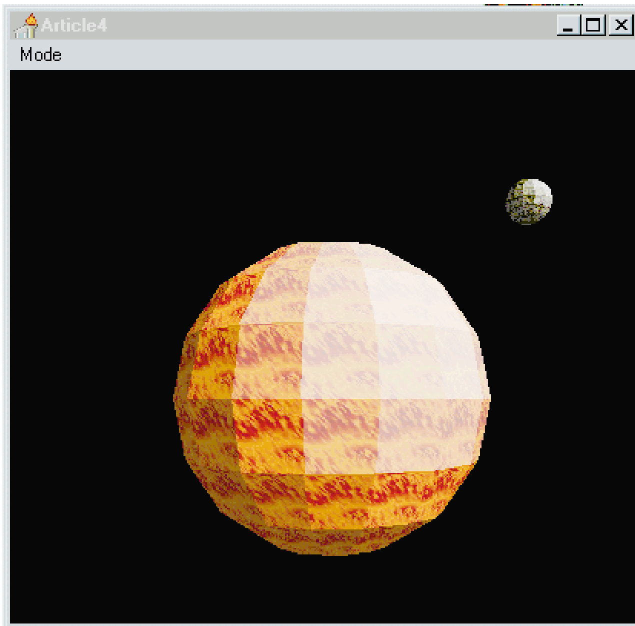


Figure 4: The type of graphic your *TGMP* rendering engine will be capable of generating by the fourth article of this series.

statement to accomplish this, but as you can see, pointers can be an eloquent solution to long-winded programming.

Listing Two on page 13 shows the first program that demonstrates the use of *TGMP*. The application loads a *TObject3D* variable and animates the object in wireframe mode.

Conclusion

Our initial development of the *TGMP* rendering component provides a solid foundation on which to add additional 3D math and rendering procedures. To give you an indication of where we're headed with this series, Figure 4 shows the type of graphic your *TGMP* rendering engine will be capable of generating later in this series.

We'll also be covering the issue of code optimization in some depth later in this series. In fact, some of those optimizations will include code that has been presented this month. It might be a good mental exercise to look over the example program and determine where you would optimize this code.

Although our first application is basic, the component has been initialized in such a manner that future articles can build upon it rather than re-design it from the ground up with each new rendering process. This will become apparent in our next article as we add solid matter to the wireframe objects created here. ∆

References:

LaMothe, A., *Black Art of 3D Game Programming* [Waite Group Press, 1995].
Lampton, Christopher, *Flights of Fantasy* [Waite Group Press, 1993].
Lyons, Eric R., *Black Art of Windows Game Programming* [Waite Group Press, 1995].

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JAN\DI9701DP.

Peter Dove is a Technical Associate with Link Associates Limited and a partner in Graphical Magick Productions. He can be reached via the Internet at peterd@graphicalmagick.com.

Don Peer is a Technical Associate for Greenway Group Holdings Inc. (GGHI) and a partner in Graphical Magick Productions. He can be reached via the Internet at dpeer@graphicalmagick.com.

Begin Listing One — The GMP Unit

```
unit gmp;

interface

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

type
  TPoint3D = record
    x,y,z : single;
  end;

  TLine3D = record
    StartPoint, EndPoint : TPoint3D;
  end;

  TObject3D = record
    LineStore  : array [0..100] of TLine3D;
    NumberLine : Integer;
    Color : Tcolor;
  end;

TPObject3D = ^TObject3D;

TGMP = class(TComponent)
  private
    { Private declarations }
    FBackBuffer : TBitmap;
    FFrontBuffer : TCanvas;
    FColor : TColor;
    ViewWidth, ViewHeight : Integer;
    FWindowHandle : THandle;
    HalfScreenWidth, HalfScreenHeight,
      ViewingDistance : Integer;
    FDistance : Integer;
    procedure DrawLine3D(x1,y1,z1, x2,y2,z2 : Single);
    procedure DrawLine2D(x1, y1, x2,y2 : Integer);
    procedure SetDistance(Distance : Integer);
  protected
    { Protected declarations }
  public
    { Public declarations }
    constructor Create(AOwner : TComponent) ; override;
    destructor Destroy; override;
    procedure ClearBackPage;
    procedure RenderNow(var Object3D : TObject3D);
    procedure FlipBackPage;
    procedure Rotate(x,y,z, angle : Single;
                var Object3D : TObject3D);
    procedure ChangeObjectColor(var Object3D : TObject3D;
                                Color : TColor);
  published
    { Published declarations }
    property BackColor : TColor read FColor write FColor;
    property ZDistance : Integer read FDistance write
      SetDistance default -50;
  end;
```

```
procedure Register;

implementation

procedure TGMP.SetDistance(Distance : Integer);
begin
  FDistance := Distance;
end;

procedure TGMP.RenderNow(var Object3D : TObject3D);
var
  A : Integer;
begin
  FBackBuffer.Canvas.Pen.Color := Object3D.Color;
  for A := 0 to Object3D.NumberLine - 1 do
    with Object3D.LineStore[A] do
      DrawLine3D(StartPoint.x, StartPoint.y, StartPoint.z,
                 EndPoint.x, EndPoint.y, EndPoint.z );
end;

procedure TGMP.ChangeObjectColor(var Object3D : TObject3D;
                                 Color : TColor);
begin
  Object3D.Color := Color;
end;

procedure TGMP.DrawLine3D(x1,y1,z1, x2,y2,z2 : Single);
var
  Screenx1, Screenx2, Screeny1, Screeny2 : Integer;
begin
  Screenx1 := HalfScreenWidth +
              Round(X1*ViewingDistance/(Z1+(ZDistance)));

  Screeny1 := Round(HalfScreenHeight-Y1*ViewingDistance /
                    (Z1+(ZDistance)));


  Screenx2 := HalfScreenWidth +
              Round(X2*ViewingDistance/(Z2+(ZDistance)));
  Screeny2 := Round(HalfScreenHeight-Y2 *
                    ViewingDistance/(Z2+(ZDistance)));
  DrawLine2D(Screenx1, Screeny1, Screenx2, Screeny2);
end;

procedure TGMP.Rotate(x,y,z, angle : Single; var
                      Object3D : TObject3D);
var
  P : Integer;
  NewX, NewY, NewZ : Single;
begin
  for P := 0 to Object3D.NumberLine - 1 do
    with Object3D.LineStore[P] do begin
      if Z <> 0 then
        begin
          NewX := StartPoint.X * Cos(Angle) -
                  StartPoint.y * Sin(Angle);
          NewY := StartPoint.X * Sin(Angle) +
                  StartPoint.y * Cos(Angle);
          StartPoint.X := NewX;
          StartPoint.y := NewY;
          NewX := EndPoint.X * Cos(Angle) -
                  EndPoint.y * Sin(Angle);
          NewY := EndPoint.X * Sin(Angle) +
                  EndPoint.y * Cos(Angle);
          EndPoint.X := NewX;
          EndPoint.y := NewY;
        end;
      if X <> 0 then
        begin
          NewY := StartPoint.Y * Cos(Angle) -
                  StartPoint.Z * Sin(Angle);
          NewZ := StartPoint.Y * Sin(Angle) +
                  StartPoint.Z * Cos(Angle);
          StartPoint.y := Newy;
          StartPoint.z := Newz;
          NewY := EndPoint.Y * Cos(Angle) -
                  EndPoint.Z * Sin(Angle);
          NewZ := EndPoint.Y * Sin(Angle) +
                  EndPoint.Z * Cos(Angle);
          EndPoint.y := Newy;
          EndPoint.z := Newz;
        end;
      if Y <> 0 then
        begin
          NewZ := StartPoint.Z * Cos(Angle) -
                  StartPoint.X * Sin(Angle);
          NewX := StartPoint.X * Cos(Angle) +
                  StartPoint.Z * Sin(Angle);
          StartPoint.z := NewZ;
          StartPoint.x := NewX;
          NewZ := EndPoint.Z * Cos(Angle) -
                  EndPoint.X * Sin(Angle);
          NewX := EndPoint.X * Cos(Angle) +
                  EndPoint.Z * Sin(Angle);
          EndPoint.z := NewZ;
          EndPoint.x := NewX;
        end;
    end;
end;

procedure TGMP.DrawLine2D(x1, y1, x2,y2 : Integer);
begin
  FBackBuffer.Canvas.MoveTo(x1, y1);
  FBackBuffer.Canvas.LineTo(x2, y2);
end;

procedure TGMP.ClearBackPage;
begin
  FBackBuffer.Canvas.Brush.Color := FColor;
  FBackBuffer.Canvas.FillRect(Rect(0,0,ViewWidth,
                              ViewHeight));
end;



procedure TGMP.FlipBackPage;
var
  ARect : TRect;
begin
  ARect := Rect(0,0,ViewWidth,ViewHeight);
  FFrontBuffer.CopyRect(ARect, FBackBuffer.Canvas, ARect);
end;

constructor TGMP.Create(AOwner : TComponent);
begin
  { We must call the inherited create of the Tcomponent
    from which we derived this class }
  inherited Create(AOwner);

  { Create the bitmap canvas }
  FBackBuffer := TBitmap.Create;

  { Get a pointer to the Form's canvas }
  FFrontBuffer := TForm(AOwner).Canvas;

  { Get the height and width of the window }
  ViewHeight  := TForm(AOwner).Height;
  ViewWidth   := TForm(AOwner).Width;

  { Set the bitmaps height }
  FBackBuffer.Height  := ViewHeight;
  FBackBuffer.Width   := ViewWidth;

  { Set up viewport }
  HalfScreenHeight := ViewHeight div 2;
  HalfScreenWidth  := ViewWidth div 2;

  { Get the handle of the window }
  FWindowHandle := TForm(AOwner).Handle;

  { Set the viewing distance }
  ViewingDistance := 200;
```

```
  { Set the Z distance }
  ZDistance := -50;
end;

destructor TGMP.Destroy;
begin
  { Free the Tbitmap }
  Fbackbuffer.Free;
  inherited;
end;

procedure Register;
begin
  RegisterComponents('Graphics', [TGMP]);
end;

end.
```

End Listing One

Begin Listing Two — The Article1 Unit

```
unit Article1;

interface

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

type
  TPyramidArray = array [0..47] of Integer;
  TCubeArray    = array [0..71] of Integer;

  TForm1 = class(TForm)
    GMP1      : TGMP;
    Timer1    : TTimer;
    MainMenu1 : TMainMenu;
    Object1   : TMenuItem;
    Cube1     : TMenuItem;
    Pyramid1  : TMenuItem;

    procedure Timer1Timer(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure Pyramid1Click(Sender: TObject);
    procedure Cube1Click(Sender: TObject);
  public
    Pyramid       : TObject3D;
    Cube          : TObject3D;
    CurrentObject : TPObject3D;
  end;

const
  PyramidPoints : TPyramidArray =
    (  0, -10,   0,   10,   10,   10,
      10,  10,  10,  -10,   10,   10,
     -10,  10,  10,    0,  -10,    0,
       0, -10,   0,   10,   10,  -10,
      10,  10, -10,  -10,   10,  -10,
     -10,  10, -10,    0,  -10,    0,
     -10,  10,  10,  -10,   10,   10,
      10,  10, -10,   10,   10,   10   );

  CubePoints : TCubeArray =
    (-10,  10,  10,   10,   10,   10,
      10,  10,  10,   10,  -10,   10,
      10, -10,  10,  -10,  -10,   10,
     -10, -10,  10,  -10,   10,   10,
     -10,  10, -10,   10,   10,  -10,
      10,  10, -10,   10,  -10,  -10,
      10, -10, -10,  -10,  -10,  -10,
     -10, -10, -10,  -10,   10,  -10,
      10,  10,  10,   10,   10,  -10,
      10, -10,  10,   10,  -10,  -10,
     -10,  10,  10,  -10,   10,  -10,
     -10, -10,  10,  -10,  -10,  -10   );

var
  Form1: TForm1;
implementation

{$R *.DFM}

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  with GMP1 do begin
    ClearBackPage;
    Rotate(1,1,0,0.1, CurrentObject^);
    RenderNow(CurrentObject^);
    FlipBackPage;
  end;
end;

procedure TForm1.FormShow(Sender: TObject);
var
  LineCount, ArrayCount : Integer;
begin
  for LineCount := 0 to 7 do begin
    ArrayCount := LineCount * 6;
    Pyramid.LineStore[LineCount].StartPoint.x :=
      PyramidPoints[ArrayCount];
    Pyramid.LineStore[LineCount].StartPoint.y :=
      PyramidPoints[ArrayCount + 1];
    Pyramid.LineStore[LineCount].StartPoint.z :=
      PyramidPoints[ArrayCount + 2];
    Pyramid.LineStore[LineCount].EndPoint.x :=
      PyramidPoints[ArrayCount + 3];
    Pyramid.LineStore[LineCount].EndPoint.y :=
      PyramidPoints[ArrayCount + 4];
    Pyramid.LineStore[LineCount].EndPoint.z :=
    PyramidPoints[ArrayCount + 5];
  end;
  Pyramid.NumberLine  := 8;
  Pyramid.Color       := clYellow;
  for LineCount := 0 to 11 do begin
    ArrayCount := LineCount * 6;
    Cube.LineStore[LineCount].StartPoint.x :=
      CubePoints[ArrayCount];
    Cube.LineStore[LineCount].StartPoint.y :=
      CubePoints[ArrayCount + 1];
    Cube.LineStore[LineCount].StartPoint.z :=
      CubePoints[ArrayCount + 2];
    Cube.LineStore[LineCount].EndPoint.x :=
      CubePoints[ArrayCount + 3];
    Cube.LineStore[LineCount].EndPoint.y :=
      CubePoints[ArrayCount + 4];
    Cube.LineStore[LineCount].EndPoint.z :=
      CubePoints[ArrayCount + 5];
  end;
  Cube.NumberLine := 12;
  Cube.Color      := clYellow;
  CurrentObject := @Cube;
  Cube1.Checked := True;
end;

procedure TForm1.Cube1Click(Sender: TObject);
begin
  if (Cube1.Checked = True) then
    Exit;
  Cube1.Checked    := True;
  Pyramid1.Checked := False;
  CurrentObject    := @Cube;
end;

procedure TForm1.Pyramid1Click(Sender: TObject);
begin
  if (Pyramid1.Checked = True) then
    Exit;
  Pyramid1.Checked := True;
  Cube1.Checked    := False;
  CurrentObject    := @Pyramid;
end;

end.
```

End Listing Two

By Danny Thorpe

Virtual Methods, Inside Out

Virtual Methods and Polymorphism: Part I

Polymorphism is perhaps *the* cornerstone of object-oriented programming (OOP). Without it, OOP would have only encapsulation and inheritance — data buckets and hierarchical families of data buckets — but no way to uniformly manipulate related objects.

Polymorphism is the key to leveraging your programming investments to enable a relatively small amount of code to drive a wide variety of behaviors, without requiring carnal knowledge of the implementation details of those behaviors. However, before you can extend existing Delphi components, or design new, extensible component classes, you must have a firm understanding of how polymorphism works and the opportunities it provides.

True to its name, polymorphism allows objects to have "many forms" in Delphi, and a component writer typically uses a mix of all these forms to implement a new component. In this article, we'll closely review the implementation and use of one of Delphi's polymorphism providers, the virtual method, and some of its more peculiar sand traps and exotic applications, e.g. its part in making .EXEs smaller. (Dynamic methods, message methods, and class reference types are Delphi's other polymorphism providers, but are outside the scope of this article.)

This article assumes you are familiar with Delphi class declaration syntax and general OOP principles. If you're a bit rusty with these concepts, you should first refer to the *Delphi Language Reference*. Also note that in this article, "virtual" denotes the *general* term that applies to all forms of virtual methods (i.e. methods declared with **virtual**, **dynamic**, or **override**), and "**virtual**" denotes the *specific* term that refers only to methods declared with the **virtual** directive. For example, most polymorphism concepts and issues apply to all virtual methods, but there are a few noteworthy items that apply only to **virtual** methods.

Review: Syntax of Virtual Methods

Here's a review of the two kinds of virtual methods and four language directives used to declare them:

- Virtual methods come in two flavors: **virtual** and **dynamic**. The only difference between them is their internal implementations; that is, they use different techniques to achieve the same results.
- Calls to **virtual** methods are dispatched more quickly than calls to **dynamic** methods.
- Seldom-overridden **virtual** methods require much more storage space for their compiler-generated tables than **dynamic** methods.
- The keywords, **virtual** and **dynamic**, always introduce a new method name into a class' name space.
- The **override** directive redefines the implementation of an existing virtual method (**virtual** or **dynamic**) that a class inherits from an ancestor.

- The **override** method uses the same dispatch mechanism (**virtual** or **dynamic**) as the inherited virtual method it replaces.
- The **abstract** directive indicates that no method body is associated with that virtual method declaration. Abstract declarations are useful for defining a purely conceptual interface, which is in turn useful for maintaining absolute separation between the user of a class and its implementation.
- The **abstract** directive can only be used in the declaration of new virtual (**virtual** or **dynamic**) methods; you can't make an implemented method abstract after the fact.
- A class type that contains one or more abstract methods is an *abstract class*.
- A class type that contains nothing but abstract methods (no static methods, no virtual methods, no data fields) is called an *abstract interface* (or, in C++ circles, a *pure virtual interface*).

Polymorphism in Action

What do **virtual** methods do? In general, they allow a method call to be directed, at run time, to the appropriate piece of code, appropriate for the type of the object instance used to make the call. For this to be interesting, you must have more than one class type, and the class types must be related by inheritance from a common ancestor.

Figure 1 shows the three classes we'll use to explore the execution characteristics of polymorphism: a simple base class named *TBaseGadget* that defines a static method named *NotVirtual* and a virtual method, *ThisIsVirtual*; and two descendant classes, *TKitchenGadget* and *TOfficeGadget*, that override the *ThisIsVirtual* method they inherit from *TBaseGadget*. *TOfficeGadget* also introduces a new static method named *NotVirtual* and a new **virtual** method named *NewMethod*.

```
type
  TBaseGadget = class
    procedure NotVirtual(X: Integer);
    procedure ThisIsVirtual(Y: Integer); virtual;
  end;

  TKitchenGadget = class(TBaseGadget)
    procedure ThisIsVirtual(Y: Integer); override;
  end;

  TOfficeGadget = class(TBaseGadget);
    function NewMethod: Longint; virtual;
    procedure NotVirtual(X,Y,Z: Integer);
    procedure ThisIsVirtual(Y: Integer); override;
 end;
```

Figure 1: Three classes to explore polymorphism.

Identical names in different classes aren't related. Declaring a static method in a descendant that happens to have the same name as a static method in an ancestor is not a true override. Other than same-name similarity, no relationship exists between static methods declared in a descendant and static methods declared in an ancestor class. Your brain makes an association, but the compiler does not. For instance, *TBaseGadget* has a *NotVirtual* method, and *TOfficeGadget* has a disparate method, also named *NotVirtual*.

If we start with a variable *P* of type *TBaseGadget*, we can assign to it an instance of a *TBaseGadget*; or an instance of one of its descendants, such as a *TKitchenGadget* or *TOfficeGadget*. Recall that Delphi object instance variables are pointers to the instance data allocated from the global heap, and that pointers of a class type are type compatible with all descendants of that type. We can then call methods using the instance variable *P*:

```
var
  P : TBaseGadget;
begin
  P := TBaseGadget.Create;
  P.NotVirtual(10);    { Call TBaseGadget.NotVirtual }
  P.ThisIsVirtual(5);  { Call TBaseGadget.ThisIsVirtual }
  P.Free;
end;
```

(In the interest of brevity, I'll fold the execution traces into comments in the source code. You can step through the sample code to verify the execution trace.)

If *P* refers to an instance of *TKitchenGadget*, the execution trace would resemble the code in Figure 2. Nothing remarkable here; we have one call to a static method going to the version defined in the ancestor type, and one call to a virtual method going to the version of the method associated with the object instance type.

You may deduce that the inherited static method, *NotVirtual*, is called because *TKitchenGadget* doesn't override it. This observation is correct, but the explanation is flawed, as Figure 3 shows. If *P* refers to an instance of *TOfficeGadget*, you may be a little puzzled by the result.

Static method calls are resolved by variable type. Although *TOfficeGadget* has its own *NotVirtual* method, and *P* refers to an instance of *TOfficeGadget*, why does *TBaseGadget.NotVirtual* get called instead? This occurs because static (non-virtual)

```
var
  P : TBaseGadget;
begin
  P := TKitchenGadget.Create;
  P.NotVirtual(10);    { Call TBaseGadget.NotVirtual }
  P.ThisIsVirtual(5);  { Call TKitchenGadget.ThisIsVirtual }
  P.Free;
end;
```

Figure 2: Execution with an instance of *TKitchenGadget*.

```
var
  P : TBaseGadget;
begin
  P := TOfficeGadget.Create;
  P.NotVirtual(10);    { Call TBaseGadget.NotVirtual }

  { The compiler will not allow the following two lines:
    P.NotVirtual(1,2,3);  "Too many parameters"
    P.NewMethod;          "Method identifier expected" }

  P.ThisIsVirtual(5); { Call TOfficeGadget.ThisIsVirtual }
  P.Free;
end;
```

Figure 3: Execution with an instance of *TOfficeGadget*.

method calls are resolved at compile time according to the type of the variable used to make the call. For static methods, what the variable refers to is immaterial. In this case, *P*'s type is *TBaseGadget*, meaning the *NotVirtual* method associated with *P*'s declared type is *TBaseGadget.NotVirtual*.

Notice that *NewMethod* defined in *TOfficeGadget* is out of reach of a *TBaseGadget* variable. *P* can only access fields and methods defined in its *TBaseGadget* object type.

New names obscure inherited names. Let's say *P* is declared as a variable of type *TOfficeGadget*. The following method call would be allowed:

```
P.NotVirtual(1,2,3)
```

However, this method call:

```
P.NotVirtual(1)
```

would not be allowed, because *TOfficeGadget.NotVirtual* requires three parameters.

TOfficeGadget.NotVirtual obscures the *TBaseGadget.NotVirtual* method name in all instances and descendants of *TOfficeGadget*. The inherited method is still a part of *TOfficeGadget* (proven by the code in Figure 3); you just can't get to it directly from *TOfficeGadget* and descendant types.

To get past this, you must typecast the instance variable:

```
TBaseGadget(P).NotVirtual(1)
```

If *P* were declared as a *TOfficeGadget* variable, *P.NewMethod* would also be allowed, because the compiler can "see" *NewMethod* in a *TOfficeGadget* variable.

Descendant >= ancestor. An instance of a descendant type could be greater than its ancestor type in both services and data. However, the descendant-type instance can never be less than what its ancestors define. This makes it possible for you to use a variable of an ancestral type (e.g. *TBaseGadget*) to refer to an instance of a descendant type without loss of information.

Inheritance is a one-way street. With a variable of a particular class type, you can access any **public** symbol (field, property, or method) defined in any of that class' ancestors. You can assign an instance of a descendant class into that variable, but cannot access any new fields or methods defined by the descendant class. The fields of the descendant class are certainly in the instance data that the variable refers to, yet the compiler has no way of knowing that run-time situation at compile time.

There are two ways around this "nearsightedness" of ancestral class types:

- Typecasting — The programmer assumes a lot and forces the compiler to treat the variable as a descendant type.
- **Virtual** methods — The magic of **virtual** will call the method appropriate to the type of the associated instance, determined at run time.

Ancestors set the standard. Why do we care about the nearsightedness of ancestral classes? Why not simply use the matching variable type when you create or manipulate an object instance? Sometimes this is the simplest thing to do. However, this "simplest" solution falls apart when you begin talking about manipulating multiple classes that do almost the same things.

Ancestral class types set the minimum interface standard through which we can access a set of related objects. Polymorphism is the use of virtual methods to make one verb (method name) produce one of many possible actions depending on the context (the instance). To have multiple, possible actions, you must have multiple class types (e.g. *TKitchenGadget* and *TOfficeGadget*) each potentially defining a different implementation of a particular method.

To be able to make one call that could cover those multiple class types, the method must be defined in a class from which all the multiple class types descend — in an ancestral class such as *TBaseGadget*. The ancestral class, then, is the least common denominator for behavior across a set of related classes.

For polymorphism to work, all the actions common to the group of classes need to at least be named in a common ancestor. If every descendant is required to override the ancestor's method, the ancestral method doesn't need to do anything at all; it can be declared **abstract**.

If there is a behavior that is common to most of the classes in the group, the ancestor class can pick up that default behavior and leave the descendants to override the defaults only when necessary. This consolidates code higher in the class hierarchy, for greater code reuse and smaller total code size. However, providing default behaviors in an ancestor class can also complicate the design issues of creating flexible, extensible classes, since what is done by ancestors usually cannot be entirely undone.

Polymorphism lets ancestors reach into descendants. Another aspect of polymorphism doesn't appear to involve instance pointer types at all — at least not explicitly.

Consider the code fragment in Figure 4. The *TBaseGadget.NotVirtual* method contains an unqualified call to *ThisIsVirtual*. When *P* refers to an instance of *TKitchenGadget*,

```
procedure TBaseGadget.NotVirtual;
begin
  ThisIsVirtual(17);
end;

var
  P: TBaseGadget;

begin
  P := TKitchenGadget.Create;
  P.NotVirtual(10);   { Call TBaseGadget.NotVirtual }
  P.Free;
end.
```

Figure 4: Polymorphism allows ancestors to call into descendants.

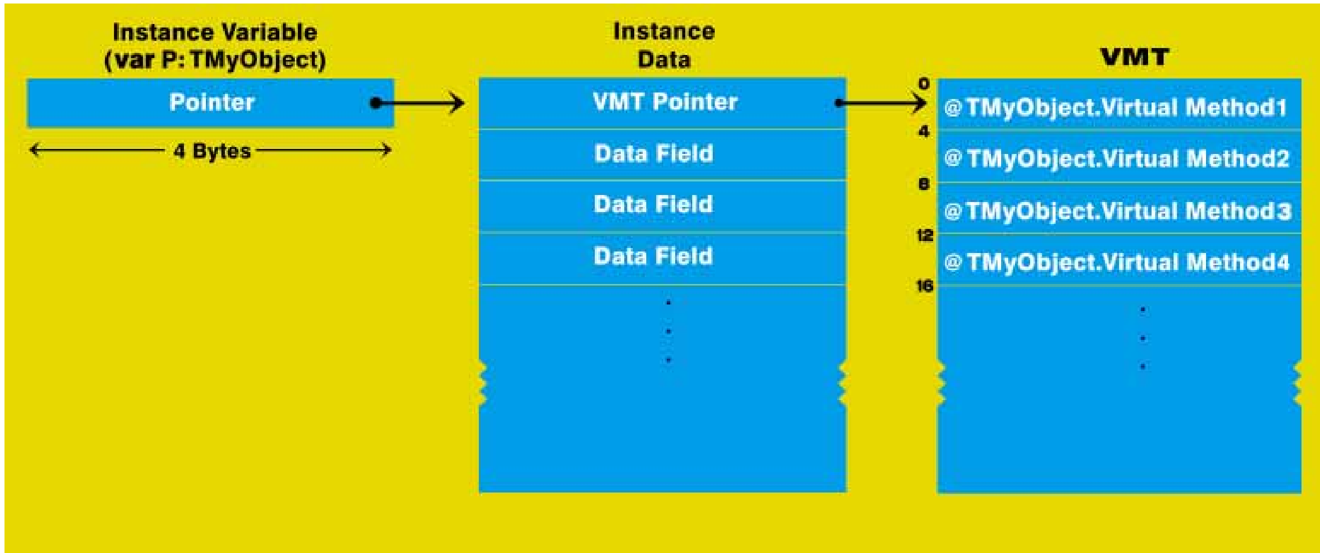


Figure 5: The structure of the Virtual Method Table, and its relationship to the object instance.

P.NotVirtual will call *TBaseGadget.NotVirtual*. Nothing new, so far. However, when that code calls *ThisIsVirtual*, it will execute *TKitchenGadget.ThisIsVirtual*. Surprise! Even within the depths of *TBaseGadget*, a non-virtual method, a virtual method call is directed to the appropriate code.

How can this be? The resolution of virtual method calls depends on the object instance associated with the call. A pointer to the object instance is secretly passed into all method calls, surfacing inside methods as the *Self* identifier. Inside *TBaseGadget.NotVirtual*, a call to *ThisIsVirtual* is actually a call to *Self.ThisIsVirtual*. *Self*, in this context, operates like a variable of type *TBaseGadget* that refers to an instance of type *TKitchenGadget*. Thus, when the instance type is *TKitchenGadget*, the virtual method call resolves, at run time, to *TKitchenGadget.ThisIsVirtual*.

How is this useful? An ancestral method — virtual or not — can call a sequence of virtual methods. The descendants can determine the specific behavior of one or more of those virtual methods. The ancestor determines the sequence in which the methods are called, plus miscellaneous setup and cleanup code. The ancestor, however, does not completely determine the final behavior of the descendants. The descendants inherit the sequence logic from the ancestor, and can override one or more of the steps in that sequence. But, the descendants *don't* have to reproduce the entire sequence logic. This is one of the ways OOP promotes code reuse.

Fully-qualified method calls are reduced to static calls. As a footnote, consider what happens if *TBaseGadget.NotVirtual* contains a qualified call to *TBaseGadget.ThisIsVirtual*:

```
procedure TBaseGadget.NotVirtual;
begin
  TBaseGadget.ThisIsVirtual(17);
end;
```

Although *ThisIsVirtual* is a virtual method, a fully-qualified method call will compile down to a regular static method call. You've specified that you want only the *TBaseGadget.ThisIsVirtual* method called, so the compiler does exactly what you tell it to do. Dispatching this as a virtual method call may call some other version of that method, which would violate your explicit instructions. Except in special circumstances, you don't want this in your code because it defeats the whole purpose of making *ThisIsVirtual* virtual.

The Virtual Method Table

A Virtual Method Table (VMT) is an array of pointers to all the **virtual** methods defined in a class *and* all the **virtual** methods the class inherits from its ancestors. A VMT is created by the compiler for every class type, because all classes descend from *TObject* and *TObject* has a **virtual** destructor named *Destroy*. In Delphi, VMTs are stored in the program's code space. Only one VMT exists per class type; multiple instances of the same class type refer to the same VMT. At run time, the VMT is a read-only lookup table.

Structure of the VMT. As shown in Figure 5, the first four bytes of data in an object instance are a pointer to that class type's VMT. The VMT pointer points to the first entry in the VMT's list of four-byte pointers to the entry points of the class' **virtual** methods. Since methods can never be deleted in descendant classes, the location of a **virtual** method in the VMT is the same throughout all descendant classes. Thus, the compiler can view a **virtual** method simply as a unique entry in the class' VMT. As we'll see shortly, this is exactly how **virtual** method calls are dispatched. Thinking of **virtual** methods as indexes into an array of code pointers will also help us visualize how method name conflicts are resolved by the compiler.

The VMT does not contain information indicating how many **virtual** methods are stored in it or where the VMT ends. The VMT is constructed by the compiler and accessed by compiler-generated code, so it doesn't need to make notes to itself about size or number of entries. (This does, however,

make it difficult for BASM code to call **virtual** methods.)
Optimization note. A descendant of a class with **virtual** methods gets a new copy of the ancestor's VMT table. The descendant can then add new **virtual** methods or override inherited **virtual** methods without affecting the ancestor's VMT. For example, if the ancestor has a 12-entry VMT, the descendant has at least a 12-entry VMT. Every descendant class type of that ancestor, and all descendants of those descendants, will have at least 12 entries in their individual VMTs.

All these VMTs occupy memory. For most programs, this won't be a problem, but extraordinarily large class types with thousands of **virtual** methods and/or thousands of descendants could consume quite a bit of memory, both in RAM and .EXE file size; **dynamic** methods are much more space efficient, but incur a slight execution speed penalty.

Now let's examine the mechanics behind the magic of **virtual** method calls.

Inside a virtual method call. When the compiler is compiling your source code and encounters a call to a **virtual** method identifier, it generates a special sequence of machine instructions that will unravel the appropriate call destination at run time. The following machine code snippets assume compiler optimizations are enabled, and stack frames are disabled:

```
// Machine code for statement P.SomeVirtualMethod;

{ Move instance data address (P^) into EAX }
MOV  EAX, [EBP+4]
{ Move instance's VMT address into ECX }
MOV  ECX, [EAX]
{ Call address stored at VMT index 2 }
CALL [ECX + 08]
```

The VMT pointer is always stored at offset 0 (zero) in the instance data. In this example, the method being called is the third **virtual** method of a class, including inherited **virtual** methods. The first **virtual** method is at offset 0, the second at offset 4, and the third at offset 8.

Conclusion

That's it — all the magic of **virtual** methods and polymorphism boils down to this: the indicator of which **virtual** method to invoke on the instance data is stored in the instance data itself.

Next month, we'll conclude our series with a discussion of **abstract** interfaces and how **virtual** methods can defeat and enhance "smart linking." See you then. ∆

This article is adapted from material for Danny Thorpe's book, Delphi Component Design *[Addison-Wesley, 1996].*

Danny Thorpe is a Delphi R&D engineer at Borland. He has also served as technical editor and advisor for dozens of Delphi programming books, and recently completed his book, *Delphi Component Design,* on advanced topics in Delphi programming. When he happens upon some spare time, he rewrites his to-do list manager to ensure that it doesn't happen again.

By Cary Jensen, Ph.D.

INI, the Registry, or Both?

Three Approaches to Creating MRU Lists

Many applications require configuration and initialization information to be stored on a per-user basis; for example, to retain a list of the files a user has accessed most recently. Similarly, some applications permit a user to select a custom bitmap to display as a background.

The standard technique for saving this type of information is to use INI files in Windows 3.1x, and the Registry in Windows 95 and Windows NT. This month's installment offers an overview of how to save this information, focusing on the creation of a "most recently used" (MRU) file list as an example.

Using INI Files

In Windows 3.1x, the standard technique for storing user-specific information is to employ an INI (initialization) file. While typically stored in the Windows directory, it can also be stored in the same directory as the EXE file (as long as the EXE isn't stored in a shared directory). Delphi 1, for example, stores information concerning its installation, as well as user preferences and settings, in a file named DELPHI.INI. A portion of this file, located in the \WINDOWS directory, is shown in Figure 1.

```
[Library]
SearchPath=C:\DELPHI\LIB
ComponentLibrary=C:\DELPHI\BIN\COMPLIB.DCL
SaveLibrarySource=0
MapFile=0
LinkBuffer=0
DebugInfo=0

[Gallery]
BaseDir=C:\DELPHI\GALLERY
GalleryProjects=1
GalleryForms=1
```

Figure 1: A portion of the DELPHI.INI file.

The structure of an INI file is simple. Each contains one or more sections. The name of each section appears within brackets. Each section contains zero, one, or multiple keys. The name of each key appears on a separate line, and is separated from its value by the "equal" sign (=). In Figure 1, for example, the INI file contains a section named `Library`. Within this section is a key named `SearchPath`. The value of this key is `C:\DELPHI\LIB`.

Delphi provides a unit named IniFiles that defines the *TIniFile* class. This class encapsulates all the basic calls you need to work with INI files. You create an instance of the *TIniFile* class (i.e. an object of type *IniFile*) by calling its *Create* constructor. This method has the following syntax:

```
constructor Create(const FileName: string);
```

You pass a single string parameter when you call *Create*. This parameter identifies the name of an INI file that will be either opened or created (e.g. `'TestINI.INI'`). If the specified INI file does not exist, calling *Create* creates it. Otherwise, *Create* opens the INI file. If the file name you supply includes a DOS path, the INI file will be created or opened in the directory you specify. If you omit the path, the Windows directory is used by default.

Once you've created an INI file, you're ready to read and write keys to it, using the

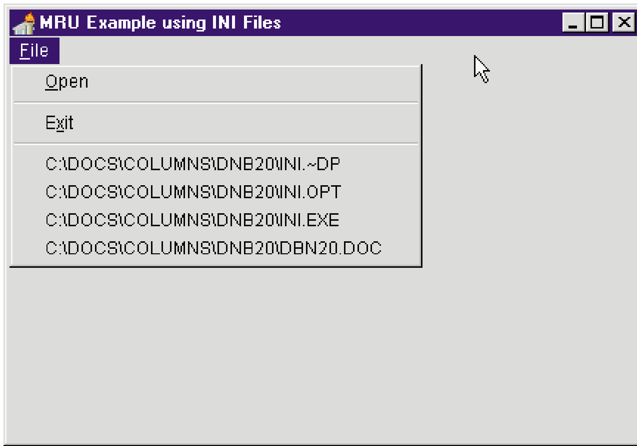


Figure 2: This application's MRU list is stored in an INI file.

WriteString, *WriteBool*, and *WriteInteger* methods. This is the syntax of the *WriteString* method:

```
procedure WriteString(const Section, Ident, Value: string);
```

When calling *WriteString*, you pass three string parameters. The first is the name of the section you're writing to, the second is the name of the key, and the third is the value you're assigning to the key.

The corresponding read methods are *ReadString*, *ReadBool*, and *ReadInteger*. The following is the syntax of *ReadString*:

```
function ReadString(const Section, Ident,
                    Default: string): string;
```

When you call *ReadString*, you pass the name of the section and the key that you want to read. In addition, you pass a third argument whose value *ReadString* retains if the specified section and key do not exist.

These methods are employed in the project named INI.DPR, shown in Figure 2. This project demonstrates how to implement a generic MRU list for a Delphi menu. Using this example code, however, requires some preparation. Specifically, you must perform the following steps:

1) Add the IniFiles unit to the **uses** clause for the form. This clause can appear in either the **interface** or the **implementation** section of the unit.

2) Create a *MainMenu* object that includes one *TMenuItem* object for each of the MRU values in the list. These *TMenuItem* objects must use the naming convention *MRU1*, *MRU2*, *MRU3*, and so forth. For example, if you plan to permit a maximum of four items in your MRU list, your *MainMenu* must include *TMenuItem* objects with the names *MRU1*, *MRU2*, *MRU3*, and *MRU4*.

3) You must add a separator to the menu immediately preceding the MRU-related menu items. Furthermore, this *TMenuItem* must be named *MRUDiv*.

4) You must add four methods to your form's **type** definition:

```
procedure LoadMRU;
procedure UpdateMRU(const FileName: string);
procedure WriteMRU;
procedure DisplayMRU;
```

5) You must add a constant named MaxMRUS to your form's unit. Assign to this constant the integer associated with the maximum number of items in your MRU list. For example, if you permit a maximum of four items, your **const** statement will look like this:

```
const
  MaxMRUS = 4;
```

6) Add a **var** declaration to your form's unit, as follows:

```
var
  MRUS: TStringList;
  MRUSChanged: Boolean;
  Ini: TIniFile;
```

7) Implement the new methods described in step 4. An example of these methods implemented for a *TForm1* class is shown in Listing Three on page 23.

Using these methods is straightforward. Call *LoadMRU* and *DisplayMRU* from the form's *OnCreate* event handler. Each time a new file is opened, call *UpdateMRU*. Finally, call *WriteMRU* from the form's *OnDestroy* event handler. You should also explicitly free both the *IniFile* object and the *StringList* object from within the *OnDestroy* event handler. Figure 3 shows examples of how these event handlers might look.

Using the Registry

While Windows 3.1x makes extensive use of INI files for storing client-specific information, Windows 95 and Windows NT encourage the use of the Registry for this purpose. The Registry is a centralized information database used as a repository for all client information by Windows 95 and Windows NT. You can

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MRUS := TStringList.Create;
  LoadMRU;
  DisplayMRU;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  WriteMRU;
  MRUS.Free;
end;

procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    begin
      Form2 := TForm2.Create(Self);
      Form2.Caption := OpenDialog1.FileName;
      UpdateMRU(OpenDialog1.FileName);
    end;
end;
```

Figure 3: Event handlers for the example MRU application.

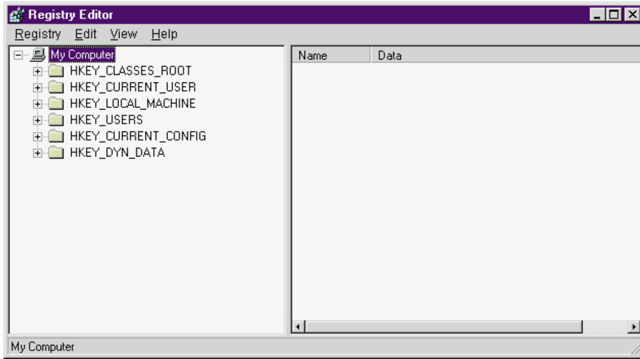


Figure 4: REGEDIT.EXE allows you to view the Registry Editor.

view the Registry Editor using REGEDIT.EXE (see Figure 4), an application located in the \WINDOWS folder.

The Registry consists of keys, subkeys, values, and data. This information is depicted hierarchically, meaning that a key may contain subkeys, and those subkeys may also contain subkeys. Typically, the lowest subkey on a branch will have one or more values displayed in the right panel of the Registry Editor dialog box. Within this right panel, the name of the value appears in the left column, and the data associated with that value appears in the right column.

The Registry offers many advantages over INI files:

- The Registry does not have an inherent size limit; INI files are limited to 64KB.
- The Registry is structured hierarchically, giving you more flexibility in how information is stored, and making specific values easier to locate.
- The Registry can be administered remotely by a system administrator (or your Delphi code, for that matter).
- A single Registry can store information about multiple users.

Delphi 2 has two Object Pascal classes for working with the Registry. These are *TRegIniFile* and *TRegistry*. Both are defined in the Registry unit.

Using the *TRegIniFile* Class

The *TRegIniFile* class is designed to permit applications written with the *TIniFile* class to be quickly and easily converted to Registry use. All methods and properties of the *TIniFile* class are present in the *TRegIniFile* class. This permits you to upgrade an application simply by changing all *TIniFile* class references to *TRegIniFile*.

The *Create* method of *TRegIniFile* either opens or creates a subkey in the HKEY_CURRENT_USER root key, using the file name as the key name. For example, the code in Listing Three creates an INI file named MRU.INI. If you change the *TIniFile* references to *TRegIniFile*, and replace the IniFiles unit with the *TRegistry* unit, this code will create a key named MRU.INI under the HKEY_CURRENT_USER root key. Furthermore, instead of creating sections using the write methods, subkeys to MRU.INI will be created. In addition, keys written to a section become values within the subkey.

One of the major advantages of the *TRegIniFile* unit is that it permits you to create a single source file compilable by either Delphi 1 or 2 — if you're willing to include a few conditional compiler directives. For example, you can replace each *TIniFile* variable declaration with a conditional compiler directive that will create a *TIniFile* variable under Delphi 1, or a *TRegIniFile* variable under Delphi 2.

The following code segment demonstrates how this should look:

```
var
{$IFDEF Win32}
  Ini: TRegIniFile;
{$ELSE}
  Ini: TIniFile;
{$ENDIF}
```

Furthermore, you must also use conditional compiler directives when calling the *Create* constructor for the declared variable, as well as for the **uses** clause. For example, the following code calls the *TIniFile* constructor under Delphi 1, and the *TRegIniFile* constructor under Delphi 2:

```
{$IFDEF Win32}
  Ini := TRegIniFile.Create('mru.ini');
{$ELSE}
  Ini := TIniFile.Create('mru.ini');
{$ENDIF}
```

An example of a project that uses these techniques is shown in Figure 5. This project is named INIREG.DPR. Figure 6 shows how the Registry appears at run time after compiling the INIREG application under Delphi 2.

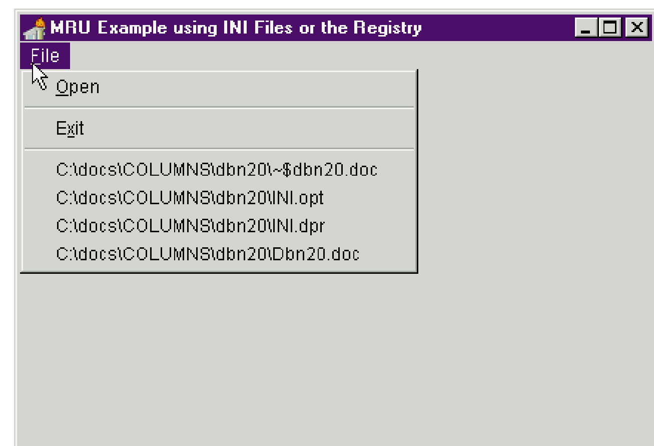


Figure 5: Though nearly identical to INI.DPR, INIREG.DPR uses an INI file when compiled in Delphi 1, and the Registry when compiled by Delphi 2.

Using the *TRegistry* Class

The *TRegIniFile* class is extremely useful when creating applications that must be used in both 16- and 32-bit environments, as well as for quickly porting applications to a 32-bit environment. It's limited, however, in that it permits you to add subkeys only directly under the HKEY_CURRENT_USER root key.

For more complete control, you should use the *TRegistry* class, which permits you to add subkeys and values to any key within the Registry. While the *TRegistry* class contains a large number

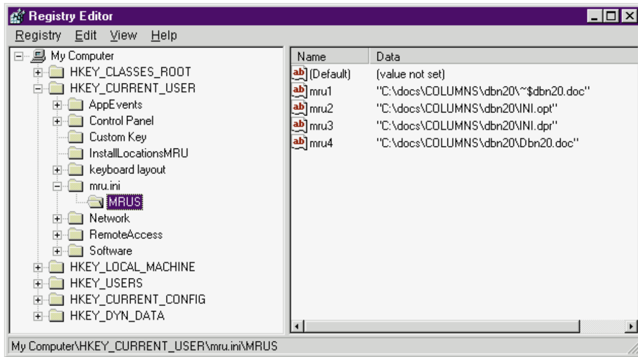


Figure 6: This is how the Registry Editor appears after running INIREG.DPR compiled with Delphi 2.

of methods, most of the work you'll do can be accomplished with a select few. Among the most valuable are *Create*, *KeyExists*, *OpenKey*, *CreateKey*, and *ValueExists*. The write and read methods (such as *WriteString* and *ReadString*) are essential as well.

Create is used to initialize a *TRegistry* descendant. Unlike the *Create* constructor for the *TIniFile* and *TRegIniFile* classes, *TRegistry.Create* requires no arguments. By default, the root key when you create a Registry object is HKEY_CURRENT_USER. You can change the root key by using the *RootKey* property of the *TRegistry* class.

KeyExists tests whether a particular key exists. The key whose existence you're testing is relative to the root key when you use an absolute address, and is relative to the current key when using a relative address. Absolute addresses begin with a backslash; relative addresses do not. To test whether HKEY_CURRENT_USER\MRU.INI\MRUS exists, you can use the following statement, regardless of which is the current key:

```
if RegVar.KeyExists('\MRU.INI\MRUS') then
  ...
```

However, the statement:

```
RegVar.KeyExists('MRU.INI\MRUS')
```

evaluates to *True* only when a key named MRU.INI\MRUS exists under the current key.

OpenKey makes the named key the current key; also, if you pass a Boolean *True* as the second parameter in the call to *OpenKey*, it will create the specified key if that key doesn't already exist. When using *OpenKey*, just as when using *KeyExists*, you can give an absolute key name that begins with a backslash, or a relative key name. For example, calling:

```
OpenKey('\Software\YourCompany\ThisApp',True)
```

opens or creates a key named HKEY_CURRENT_USER\-SOFTWARE\YOURCOMPANY\THISAPP, and makes it the current key. By comparison, calling:

```
OpenKey('MRUS',True)
```

will open or create a key named MRUS under the current key, and will make it the new current key.

CreateKey is used (again, as you might guess) to create new keys. *CreateKey* is different from *OpenKey* in that it doesn't make the specified key the current key. As with the preceding statements, you can use either an absolute or a relative address to specify the key.

Finally, *ValueExists* tests whether a specified value exists under the current key. Unlike *TIniFile* or *TRegIniFile* objects, which permit you to read from a specified section and key even if the key doesn't exist, the *TRegistry* class will generate an exception. Consequently, you'll generally use *ValueExists* to test the existence of a value before attempting to read from it.

The read and write methods for *TRegistry* are also different from the associated methods of the *TIniFile* class. For example, this is the syntax of the *TRegistry.ReadString* method:

```
function ReadString(const Name: string): string;
```

To use this method, pass the name of a value under the current key. *ReadString* returns the data associated with that value. Again, if the specified value doesn't exist, an exception is generated.

The write methods in the Registry class are somewhat more similar to their *TIniFile* counterparts than are the read methods. For example, this is the syntax of the *TRegistry.WriteString* method:

```
procedure WriteString(const Name, Value: string);
```

The first parameter you pass to *WriteString* is the name of a value under the current key. If the named value doesn't already exist, it will be created. The second is the data to write to the specified key. In addition to the syntactical differences between the read and write methods of *TRegistry* and those in the *TIniFile* class, the *TRegistry* class supports methods for reading and writing data types other than just string, Boolean, and integer. Read and write methods are included for binary, datetime, currency, and float data, among others.

The use of the *TRegistry* class is demonstrated in the project REGISTRY.DPR. This project is quite similar to the INI.DPR and INIREG.DPR projects discussed earlier. The primary difference revolves around the declaration of the Registry variable, as well as the implementation of the four MRU methods. This code is shown in Listing Four, beginning on page 23.

Figure 7 shows how the Registry looks after running REGISTRY.DPR.

Conclusion

Delphi provides a number of options for saving user-specific information from one execution of your application to the next. For Windows 3.1x applications, use the *TIniFile* class to

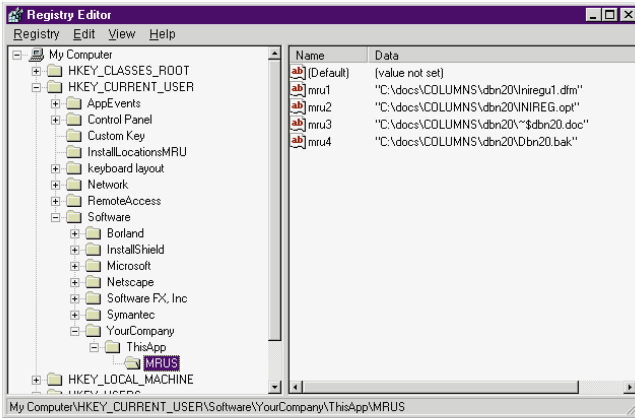


Figure 7: The REGISTRY.DPR project writes to the Registry key HKEY_CURRENT_USER\SOFTWARE\YOURCOMPANY\THISAPP\MRUS.

write to INI files. For applications that must be compiled under both Delphi 1 and 2, you can use the *TRegIniFile* class. Finally, when creating 32-bit applications with Delphi 2, use the *TRegistry* class. ∆

The files referenced in this article are *available on the Delphi Informant Works CD located in INFORM\97\JAN\DI9701CJ.*

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]. He is also Contributing Editor to *Delphi Informant*. You can reach Jensen Data Systems at (713) 359-3311, or via CompuServe at 76307,1533.

Begin Listing Three — Example MRU Implementation

```
procedure TForm1.LoadMRU;
var
  Ini:   TIniFile;
  i:     Integer;
  Value: string;
begin
  Ini := TIniFile.Create('mru.ini');
  try
    { Load MRUs into Tstrings }
    for i := 1 to MaxMRUS do
      begin
        Value := Ini.ReadString(
                   'MRUS',Concat('MRU',IntToStr(i)),'');
        MRUS.Add(value);
      end;
    { Remove blank MRUs }
    while MRUS.IndexOf('') <> -1 do
      MRUS.Delete(MRUS.IndexOf(''));
  finally
    Ini.Free;
  end;
end;

procedure TForm1.DisplayMRU;
var
  MI: TMenuItem;
  i: Integer;
begin
  i := 0;
  while (i < (MRUS.Count)) and (i < MaxMRUS) do
    begin
      MI := TMenuItem(FindComponent(Concat(
              'mru',inttoStr(i+1))));
      MI.Visible := True;
      MI.Caption := MRUS.Strings[i];
      Inc(i);
    end;
  MRUDiv.Visible := MRU1.Visible;
end;

procedure TForm1.UpdateMRU(const filename: string);
var
  i: Integer;
begin
  MRUSChanged := True;
  if MRUS.IndexOf(filename) = -1 then
    { Add filename to top, move all down }
    begin
      MRUS.Insert(0,filename);
      { Remove last item if MRU list already full }
      if MRUS.Count > MAXMRUS then
        MRUS.Delete(MAXMRUS-1);
    end
  else
    MRUS.Move(MRUS.IndexOf(filename),0);
  DisplayMRU;
end;

procedure TForm1.WriteMRU;
var
  Ini: TIniFile;
  i: Integer;
begin
  if MRUSChanged then
    begin
      Ini := TIniFile.Create('mru.ini');
      try
        i := 0;
        while (i < (MRUS.Count)) and (i < MaxMRUS) do begin
          Ini.WriteString('MRUS',Concat('mru',
            inttoStr(i+1)),MRUS.Strings[i]);
          Inc(i);
        end;
      finally
        Ini.Free;
      end;
    end;
end;
```

End Listing Three

Begin Listing Four — Demonstrating the *TRegistry* Class

```
var
  { Hold the MRUs during the application }
  MRUS: TStringList;
  { Flag for writing MRUs from OnDestroy for form }
  MRUSChanged: Boolean;
  Reg: TRegistry;
procedure TForm1.LoadMRU;
var
  i: Integer;
  Value: string;
begin
  Reg := TRegistry.Create;
  try
    Reg.OpenKey('\Software\YourCompany\ThisApp\MRUS',True);
    { Load MRUs into Tstrings }
    for i := 1 to MaxMRUS do begin
      if Reg.ValueExists(Concat('MRU',IntToStr(i))) then
        begin
          Value := Reg.ReadString(
                     Concat('MRU',IntToStr(i)));
          MRUS.Add(value);
        end
      else
        Break;
    end;
```

```
  finally
    Reg.Free;
  end;
end;

procedure TForm1.DisplayMRU;
var
  MI: TMenuItem;
  i: Integer;
begin
  i := 0;
  while (i < (MRUS.Count)) and (i < MaxMRUS) do begin
    MI := TMenuItem(FindComponent(Concat('mru',
                                    inttoStr(i+1))));
    MI.Visible := True;
    MI.Caption := MRUS.Strings[i];
    Inc(i);
  end;
  MRUDiv.Visible := MRU1.Visible;
end;

procedure TForm1.UpdateMRU(const filename: string);
var
  i: Integer;
begin
  MRUSChanged := True;
  if MRUS.IndexOf(filename) = -1 then
    { Add filename to top, move all down }
    begin
      MRUS.Insert(0,filename);
      { Remove last item if MRU list already full }
      if MRUS.Count > MAXMRUS then
        MRUS.Delete(MAXMRUS-1);
    end
  else
    MRUS.Move(MRUS.IndexOf(filename),0);
  DisplayMRU;
end;

procedure TForm1.WriteMRU;
var
  i: Integer;
begin
  if MRUSChanged then
    begin
      Reg := TRegistry.Create;
      try
        Reg.OpenKey('\Software\YourCompany\ThisApp\MRUS',
                    True);
        i := 0;
        while (i < (MRUS.Count)) and (i < MaxMRUS) do begin
          Reg.WriteString(Concat('mru',
            inttoStr(i+1)),MRUS.Strings[i]);
          Inc(i);
        end;
      finally
        Reg.Free;
      end;
    end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  MRUS := TStringList.Create;
  LoadMRU;
  DisplayMRU;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  WriteMRU;
  MRUS.Free;
end;
```

End Listing Four

OP TECH

Delphi 1 / Delphi 2

By Ray Lischner

Stream of Consciousness

The VCL's Internal Component Messages

The components in the VCL send messages among themselves to communicate events, changes in state, and other information. When you write a custom component, you can usually inherit the correct message handlers from one of Delphi's control classes, such as *TCustomControl* or *TGraphicControl*. In some cases, however, you might need to handle these internal messages differently. This article describes some of the internal messages the VCL uses.

The component messages are generated by the VCL in response to Windows messages and user actions. The default action of a component when it receives many of the VCL component messages is to broadcast that message to all its child components. These messages originate with a Windows control that receives a standard Windows message.

For example, a form receives a *Wm_WinIniChange* message, and broadcasts a *Cm_WinIniChange* message to its children, which in turn broadcast that message to their children, until it's been sent to every child on the form (see Figure 1).

This enables you to write components that can respond to standard Windows messages without interfering with the form, which is important when you're writing a reusable component. The component can silently do its work, responding to messages the form automatically sends; the application programmer never needs to be concerned with these details.

Other *Cm_* messages are generated by VCL components to broadcast to their child components, informing the children of a change in the parent's state. For example, when you set the *Color* property of a control, it broadcasts the *Cm_ParentColorChanged* message to its children so they can respond to the change, if needed. Delphi's pre-defined components, for example, check the *ParentColor* property, and if it is *True*, change the component's *Color* to match the *Parent* control's new *Color*.

The messages described here are only some of the internal component messages that Delphi defines. Note that not all Windows messages have corresponding component messages.

If you want your component to intercept, say, a *Wm_Compacting* message sent to the form, you cannot use a component message, but must set up an application message hook, which is a completely different topic.

Many of the component messages don't have specific argument types, so you can use *TMessage*. Others do have specific types, such as *TCmDialogChar* for the *Cm_DialogChar* message. If a message type is defined, the descriptions here show the message type.

Some messages require that a result be stored in the message argument's *Result* field. The value of the result depends on the message.

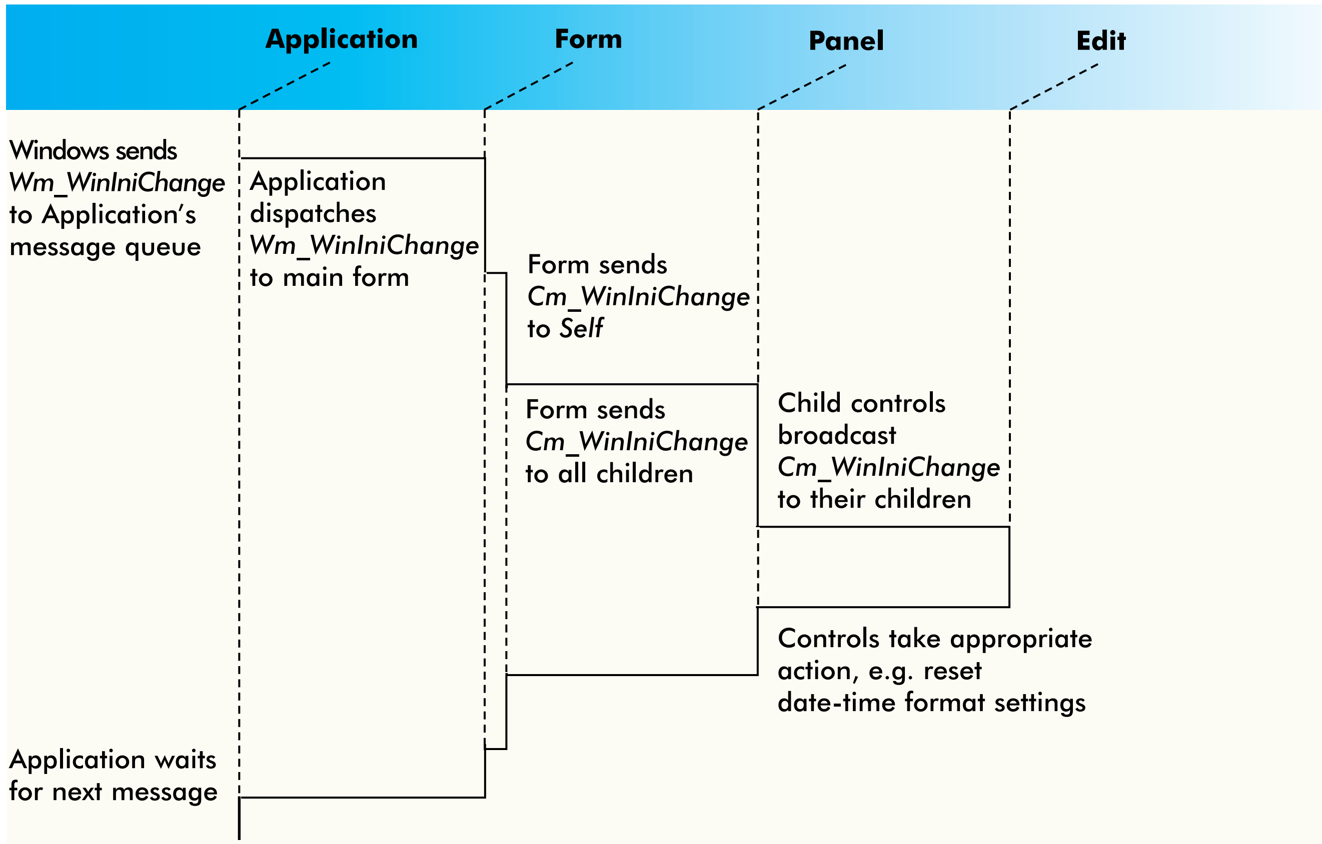


Figure 1: Sending the *Cm_WinIniChange* message.

Cm_AppKeyDown Message

The *Cm_AppKeyDown* message is sent from a control to the *Application* object in response to a *Cn_KeyDown* or *Cn_SysKeyDown* event. The control first checks whether the key event is a shortcut for a pop-up menu. If not, the control sends the *Cm_AppKeyDown* event to the application, and the application handles the message by checking whether the key-down event is a shortcut key for the main form. If so, the main form handles the event, and the message *Result* is set to 1. If *Result* is zero, the main form didn't handle the event, which leaves the control that originated the message to handle the *Cn_KeyDown* event. In other words, this message is how Delphi forwards a menu shortcut key from a child form to the main form.

For this message, a shortcut key is the value of a menu item's *ShortCut* property. When the menu item *Caption* contains an ampersand (&), the letter after the ampersand is also a shortcut key, but it's treated differently. In this article, the latter shortcuts are called *caption shortcuts* and the former are called *accelerator shortcuts*.

In Delphi 2, you don't need to handle or send the *Cm_AppKeyDown* message. If a child form has a menu bar, it handles its own menu shortcuts, and if it doesn't, it forwards the shortcut keys to the main form. In Delphi 1, however, *you* must handle this message if you want a child form to have its own menu bar, separate from the main form's menu bar.
In Delphi 1, a child form forwards its menu shortcut keys to the main form, even if the child has its own menu. If you went to the trouble of adding a menu bar to a child form, you probably want the child form to handle its own menu shortcut messages. To do this, you must intercept the *Cm_AppKeyDown* and the *Cm_AppSysCommand* messages. The former handles the accelerator shortcuts and the latter handles the caption shortcuts. Because the *Cm_AppKeyDown* message is sent only to the *Application* object, you cannot write a handler for it.

You can, however, write a message hook to intercept the message. The message hook can pretend the message has been handled, thereby preventing the main form's menu from ever seeing the shortcut key. That lets each form handle its own key stroke events. Figure 2 shows one way to hook the *Cm_AppKeyDown* and *Cm_AppSysCommand* messages to prevent the main form from receiving them.

The *Cm_AppKeyDown* message, like other key stroke messages, uses the *TWmKeyDown* message type. The *CharCode* field is the virtual key code, and the *KeyData* field holds the modifier keys, repeat count, and other key data. Figure 3 shows a description of the constituent parts of the *KeyData* field.

Cm_AppSysCommand Message

The *Cm_AppSysCommand* message is sent in response to a *Wm_SysCommand* message, but only when the sending form is not the main form, the form is not minimized, the command is *Sc_KeyMenu*, and the key is not a space or a hyphen

```
function TMainForm.AppKeyDownHook(var Msg: TMessage):
Boolean;
begin
  case Msg.Msg of
    Cm_AppkeyDown     : Result := True;
    Cm_AppSysCommand  : Result := True;
  else
    Result := False;
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.HookMainWindow(AppKeyDownHook)
end;
```

Figure 2: How to intercept and discard the *Cm_AppKeyDown* and *Cm_AppSysCommand* events.

```
type
  TKeyDataFlag = ( kdExtended,  kdUnused1,   kdUnused2,
                   kdInternal1, kdInternal2, kdAltDown,
                   kdWasDown,   kdReleased);
  TKeyDataFlags = set of TKeyDataFlag;
  TKeyData = record
    RepeatCount:  Word;
    ScanCode:     Byte;
    Flags:        TKeyDataFlags;
  end;
```

Figure 3: *TKeyData* record for interpreting key event data.

(-). In other words, this message is sent when the user presses a caption shortcut key.

In Delphi 1, you must intercept this message when you want a child form to have its own menu bar, separate from the menu of the main form. Figure 2 shows how to do this. Otherwise, there is no reason to handle or send this message.

In Delphi 1, a form sends the *Cm_AppSysCommand* to the application, which forwards it to the main menu. The message type is *TWmSysCommand*. In Delphi 2, a control first sends the *Cm_AppSysCommand* message to its parent form; the parent form forwards it to the application only if the form is an MDI child, has no menu, or has a main menu whose *AutoMerge* property is set to *True*. This lets a child with its own menu handle the shortcut key. The *lParam* argument is a pointer to the original message's *TMessage* record. The parent form passes this *TMessage* record to the *Application* object, which forwards it to the main form.

Cm_ButtonPressed Message

A speed button sends this message to its siblings to enforce the group semantics; that is, to ensure that only one button in a group is pressed at any given time. When a speed button is pressed, it broadcasts the *Cm_ButtonPressed* message to all its sibling controls (not just the *TSpeedButton* components).

By default, *TSpeedButton* components handle this message so that all buttons in the same group can set themselves to the Up state. The message arguments are the group index

```
type
  TCmButtonPressed = record
    Msg: Cardinal;
    GroupIndex: Integer;
    Sender: TSpeedButton;
    Result: LongInt;
  end;
```

Figure 4: *TCmButtonPressed* type, for use when handling *Cm_ButtonPressed* events.

(*wParam*) and the sender button (*lParam*), but no message type is defined. For your convenience, Figure 4 shows the *TCmButtonPressed* type, which you can use when handling this message.

If you derive a new class from *TSpeedButton*, you can handle this message if you want to implement different behavior when a speed button is pressed. You can also handle this event for non–speed button components, perhaps to reflect in a different manner which speed button is currently depressed.

The result is ignored for this message.

Cm_ColorChanged Message

When a control's *Color* property changes, it sends the *Cm_ColorChanged* message to itself. The default message handler invalidates the control, so Windows will repaint it. A control with children broadcasts the *Cm_ParentColorChanged* message to its children so they can respond to the color change, if needed.

TWinControl also copies the color to its *Brush.Color* property, and *TForm* copies the *Color* to its *Canvas.Brush.Color* property. If you derive a class from a different base class (such as *TGraphicControl*) that publishes the *Color* property, you might want to handle the *Cm_ColorChanged* message. Remember to call the inherited message handler to ensure the control is refreshed and all child controls are properly notified.

Because this message takes no arguments and returns no result, you should use *TWmNoParams* for the message type.

Cm_ControlListChange Message

A window control sends the *Cm_ControlListChange* message to itself when its control list changes; that is, when a control is added or removed. A *TDBCtrlPanel* component updates its database links when it receives this message.

The *Cm_ControlListChange* message is sent before a control is added to the list, so a message handler can examine the control and raise an exception to prevent it from being added. When a control is removed, the *Cm_ControlListChange* message is sent after the control is removed.

The message type is *TWmControlListChange*. The *Control* field refers to the control being added or removed. The *Inserting* field is *True* when the control is to be added and *False* when the control is being removed.

The *Cm_ControlListChange* message and the *TDBCtrlPanel* component are not included in Delphi 1.

Cm_Ctl3DChanged Message

The *Cm_Ctl3DChanged* message is similar to *Cm_ColorChanged*, but corresponds to the *Ctl3D* property.

Cm_CursorChanged Message

When a control's *Cursor* property changes, it sends the *Cm_CursorChanged* message to itself. The default implementation, in *TWinControl*, checks whether the cursor is currently over the control, and if so, immediately changes the cursor. You probably don't need to change this behavior for any Windows control. For a component derived from *TGraphicControl*, however, you might decide to implement similar behavior.

This message takes no arguments and returns no result, so you should use *TWmNoParams* as the message type.

Cm_DesignHitTest Message

As the name implies, Delphi sends the *Cm_DesignHitTest* message to a component at design time to determine whether the cursor position is in a design area. If your component needs to handle mouse events at design time, you can set the *Result* to a non-zero value.

For example, the various grid components handle column and row resize events at design time. When the mouse is over a resize bar, the grid component sets the *Result* to 1 in response to a *Cm_DesignHitTest* event. This tells Delphi to pass mouse events to the component, rather than handling these events itself (for resizing and moving components, and for using the component editor).

In most cases, your component doesn't need to handle this message. If you're writing a component that must respond to mouse events at design time, you can supply a message handler. If you decide that the mouse cursor is not in a design area, call the inherited message handler. The inherited handler checks for child controls that might need to handle this message.

The message type is *TCmDesignHitTest*. The cursor position is given by the *XPos* and *YPos* fields, or by the *Pos* field, which is of type *TPoint*. The *Keys* field gives the state of the Shift and Ctrl modifiers, and tells you which mouse button is pressed, in the same manner as *TWmMouseMove* (as a bit mask). Call the *KeysToShiftState* function to convert this bit mask into a *TShiftState* set, which is easier to use.

Cm_DialogHandle Message

In Delphi 2, an *Application* object in a DLL sends the *Cm_DialogHandle* message to its counterpart in the application, which responds by setting or returning its *DialogHandle* property. When *wParam* equals 1, the *Application* object returns the current value of *DialogHandle*; otherwise, it sets *DialogHandle* to the value of the *lParam* field. This lets Delphi keep a single *DialogHandle* for an application and all DLLs.

```
library Demo;

uses WinTypes, Forms;

{ The application passes its Application.Handle to
  Initialize, so the DLL can communicate with the main
  application. }
procedure Initialize(Handle: HWnd); export;
begin
  Application.Handle := Handle;
end;

...

exports
  Initialize;

end.
```

Figure 5: Initializing an *Application* object in a DLL.

```
uses Forms;

procedure Initialize(Handle: Hwnd); external DemoDLL;

...

  Initialize(Application.Handle);

...
```

Figure 6: Initializing a DLL from the main application.

The *DialogHandle* property lets you use non-Delphi modeless dialog boxes with Delphi. You assign the window handle of the dialog box to the application's *DialogHandle* property, and the *Application* forwards dialog messages to the dialog box. In particular, the *TFindDialog* and *TReplaceDialog* components rely on the *DialogHandle* property.

When using the *Application* object in a DLL, be sure to assign the main application's handle to the *Application.Handle* property. This allows the *Application* object in the DLL to forward the *Cm_DialogHandle* message to the *Application* object in the main application. Figure 5 shows an example of an *Initialize* procedure in a DLL, which takes a window handle as an argument, assigning it to *Application.Handle*.

To use the DLL's *Initialize* procedure, you must pass *Application.Handle* as its argument. This is shown in Figure 6.

Because the *Cm_DialogHandle* message is used internally by the *TApplication* class, there is no reason for you to send or handle it. Instead, use the *Application.DialogHandle* property. This message is defined only in Delphi 2.

Cm_Drag Message

The *Cm_Drag* message communicates drag events between controls. The *DragMessage* field specifies what kind of drag event is taking place (Enter, Leave, Move, Drop, Cancel, or Find Target), and the *DragRec* field points to a *TDragRec* record, which contains a reference to the *Source* and *Target* objects and the cursor position.

The *TCmDrag* message type declares the *DragMessage* and *DragRec* fields. The *Cm_Drag* message is defined only in Delphi 2. In Delphi 1, you can override the *DragDrop* and *DragCanceled* methods, but to customize any other drag-and-drop behavior, you must handle the mouse events explicitly. Borland has addressed this limitation by adding the *Cm_Drag* message and by adding virtual methods to *TControl* in Delphi 2.

By inheriting from the standard Delphi components, your components automatically inherit the proper behavior for the *Cm_Drag* message. In most cases, you can customize any specific behavior by overriding the drag-and-drop methods: *DragOver*, *DragCanceled*, *DoStartDrag*, *DoEndDrag*, and *DragDrop*. In rare cases when you need greater control over drag events, you can override the *Cm_Drag* message handler. The following sections describe the appropriate response for each drag message.

***dmDragCancel* Drag Message.** If the user finishes a drag-and-drop operation by dropping on a target that does not accept the drag source, the intended target receives a *Cm_Drag* message with the *dmDragCancel* drag message. The cursor position in the drag record is set to (0,0). The default behavior is to do nothing when receiving the *dmDragCancel* drag message. This message returns no result.

***dmDragDrop* Drag Message.** When the intended target of a drop operation accepts the drop, it receives a *Cm_Drag* message with the *dmDragDrop* drag message. The default behavior, upon receiving the *dmDragDrop* message, is to call the *DragDrop* method. The preferred way to handle a *dmDragDrop* message is to override the *DragDrop* method. If you do so, remember to call the inherited method, which calls the *OnDragDrop* event handler. This message returns no result.

***dmDragEnter* Drag Message.** During a drag operation, when the mouse cursor enters a control, the control receives a *Cm_Drag* message with the *dmDragEnter* drag message. The default response is to treat the *dmDragEnter* message as a normal *dmDragMove* message, i.e. to call the *DragOver* method. The preferred way to handle the *dmDragEnter* message is to override the *DragOver* method. If you do so, remember to call the inherited method, which calls the *OnDragOver* event handler.

The result is non-zero if the target accepts the drag source, or zero if rejected.

***dmDragLeave* Drag Message.** When the mouse cursor leaves a control during a drag operation, the control receives a *Cm_Drag* message with the *dmDragLeave* drag message. The default response is to treat the *dmDragLeave* message as a normal *dmDragMove* message, i.e. to call the *DragOver* method. The preferred way to handle the *dmDragLeave* message is to override the *DragOver* method. If you do so, remember to call the inherited method, which calls the *OnDragOver* event handler.

The result is non-zero if the target accepts the drag source, or zero if rejected.

***dmDragMove* Drag Message.** During a drag operation, when the mouse cursor moves over a control, the control receives a *Cm_Drag* message with the *dmDragMove* drag message. The default response is to call the *DragOver* method. The preferred way to handle the *dmDragMove* message is to override the *DragOver* method. If you do so, remember to call the inherited method, which calls the *OnDragOver* event handler.

The result is non-zero if the target accepts the drag source, or zero if rejected.

***dmFindTarget* Drag Message.** When the mouse moves during a drag operation, Delphi must identify the component under the mouse cursor. It starts with the window control under the mouse cursor, and sends it the *Cm_Drag* message with the *dmFindTarget* drag message. The control responds by returning an object reference, i.e. the component that is the drag target. Your custom component can handle this message by forwarding drag operations to a different control or by returning **nil** to refuse the drag operation. In most cases, however, you will want to keep the default behavior. When setting the result, there is no convenient message type defined, but you can use *TCmDragFindTarget*, as shown in Figure 7.

```
type
  TCmDragFindTarget = record
    Msg: Cardinal;
    Source: TObject;
    Unused: LongInt;
    Result: TControl;
  end;
```

Figure 7: *TCmDragFindTarget* type, for use when handling *Cm_Drag* events.

Cm_EnabledChanged Message

When a control's *Enabled* property changes, it sends the *Cm_EnabledChanged* message to itself. Upon receiving this message, most components invalidate themselves to force a repaint. A Windows control enables or disables its window and gives up the input focus when disabled. If you write a component that needs to do something different when the *Enabled* property changes value, then you can handle this message. Remember to call the inherited event handler.

This message takes no arguments and returns no result, so you should use *TWmNoParams* as the message type.

Cm_FocusChanged Message

After changing the focused control, a form sends the *Cm_FocusChanged* message to itself, which broadcasts the message to all child components. The control that received the input focus is stored in the *Sender* parameter, which is of type *TWinControl*. The message type is *TCmFocusChanged*, which declares the *Sender* field. The message handler returns no result.

Most likely, your component would override the *DoEnter* and *DoExit* methods to handle focus changes, but there might be situations in which you want your component to reflect focus changes that occur elsewhere on the form. In that case, you can handle the *Cm_FocusChanged* message. Remember to call the inherited method to ensure that child components are notified of the focus change, too.

Cm_FontChange Message

When a form receives the *Wm_FontChange* message, it sends *Cm_FontChange* to itself. Upon receiving *Cm_FontChange*, a window broadcasts the message to its children. A control handles *Cm_FontChange* by updating its font and broadcasting the *Cm_ParentFontChanged* message. There is rarely any reason to handle this message yourself, unless your component manages a list of installed fonts.

Note that *Cm_FontChange* is very different from *Cm_FontChanged*. Windows sends the *Wm_FontChange* message when the list of installed fonts changes, usually because the user installed or removed a font.

This message takes no arguments and returns no result, so you should use the *TWmNoParams* message type.

Cm_FontChanged Message

The *Cm_FontChanged* message is similar to *Cm_ColorChanged*, but corresponds to the *Font* property. Notice the difference between *Cm_FontChanged* and *Cm_FontChange*.

Cm_GetDataLink Message

When a *TDBCtrlPanel* updates its data links, it sends the *Cm_GetDataLink* message to each of its child database controls. A database component, in response to this message, returns a reference to its data link object.

Although there is no class defined for this message, you can use the *TCmGetDataLink* record shown in Figure 8. Notice that the *Result* field is not an integer, but a *TFieldDataLink* object. The *wParam* and *lParam* fields are not used. The *Cm_GetDataLink* message is specific to Delphi 2.

Cm_HintShow Message

In Delphi 2, the *Application* object sends the *Cm_HintShow* message to a control before displaying its hint. If the control returns a *True* response, then the application doesn't display the hint window. The default response is to return a *False* result, thereby letting the application display the hint window. Note that if the control's *ShowHint* property is *False*, the application doesn't attempt to show a hint, and the control never receives the *Cm_HintShow* message.

Although there is no message type, Figure 9 shows the *TCmHintShow* record, which you can use. The *HintInfo* field points to a *THintInfo* record, which contains the

```
type
  TCmGetDataLink = record
    Msg: Cardinal;
    Unused1: Integer;
    Unused2: LongInt;
    Result: TFieldDataLink;
  end;
```

Figure 8: *TCmGetDataLink* type, for use when handling *Cm_GetDataLink* events.

```
type
  TCmHintShow = record
    Msg: Cardinal;
    Unused: Integer;
    HintInfo: PHintInfo;
    Result: LongBool;
  end;
```

Figure 9: *TCmHintShow* type, for use when handling *Cm_HintShow* events.

position, size, and color of the hint window. When responding to the *Cm_HintShow* message, the control can change any of these fields.

Cm_InvokeHelp Message

The *Cm_InvokeHelp* message is sent by an *Application* object in a DLL when it has no associated Help file. The main application handles this message by calling its *InvokeHelp* method, which asks Windows to open a Help file. In other words, Delphi uses the *Cm_InvokeHelp* message to forward Help requests from a DLL to an application. The *wParam* and *lParam* arguments are passed to *WinHelp* as the *Command* and *Data* arguments. No results are returned, so you can use *TMessage* for the message type.

When using the *Application* object in a DLL, be sure to assign the main application's handle to the *Application.Handle* property. This allows the *Application* object in the DLL to forward the *Cm_InvokeHelp* message to the *Application* object in the main application, as illustrated in Figures 5 and 6.

Ordinarily, your components, applications, and libraries don't need to send or handle this message, because you can call the *TApplication.InvokeHelp* method.

Cm_IsToolControl Message

In Delphi 2, an OLE container sends the *Cm_IsToolControl* message to a control to learn whether it's a toolbar or similar control that must remain when an OLE object is activated in place. This message is sent only to controls whose *Align* property is *alLeft*, *alRight*, *alTop*, or *alBottom*. If the control returns a zero result, then it remains; if a non-zero result is returned, the control is hidden while the OLE object is active.

By default, a control returns a zero result. A *TCustomPanel* object returns a non-zero result if its *Locked* property is *False*. In other words, most aligned controls remain visible during an OLE in-place activation, but to keep a panel visible, you must set its *Locked* property to *True*.

```
type
  TCmIsToolControl = record
    Msg: Cardinal;
    Unused1: Integer;
    Unused2: LongInt;
    Result: LongBool;
  end;
```

Figure 10: *TCmIsToolControl* type, for use when handling *Cm_IsToolControl* events.

```
type
  TCmMouseEnter = record
    Msg: Cardinal;
    Unused: Integer;
    Sender: TControl;
    Result: LongInt;
  end;
  TCmMouseLeave = TCmMouseEnter;
```

Figure 11: *TCmMouseEnter* type, for use when handling *Cm_MouseEnter* events.

There is no message type defined for this message. You can use the *TCmIsToolControl* record, shown in Figure 10. This message is not defined for Delphi 1.

Cm_MouseEnter Message

The *Application* object sends the *Cm_MouseEnter* message to a control when the mouse enters that control's bounds. If a control is capturing all mouse input, then the *Cm_MouseEnter* message is sent only when the mouse enters the capture control. The *Application* object doesn't send *Cm_MouseEnter* unless it has no other messages to process, so your component must not rely on receiving this message in a timely manner.

Delphi controls do not handle this message, except to forward it to the parent control. The *lParam* argument is the message sender, or **nil** if the sender is the *Application* object. This message returns no result. For a diagram of the *Cm_MouseEnter* message, see *Cm_MouseLeave*.
There is no message type, so you can use the *TCmMouseEnter* type shown in Figure 11.

Cm_MouseLeave Message

The *Application* object sends the *Cm_MouseLeave* message to a control when the mouse leaves that control's bounds. If a control is capturing all mouse input, then the *Cm_MouseLeave* message is sent only when the mouse leaves the capture control. The *Application* object doesn't send *Cm_MouseLeave* unless it has no other messages to process, so your component must not rely on receiving this message in a timely manner.

Delphi controls do not handle this message except to forward it to the parent control. The *lParam* argument is the message sender, or **nil** if the sender is the *Application* object. This message returns no result. Figure 12 depicts a simple form with the mouse moving across it. Figure 13 shows the resulting event diagram.

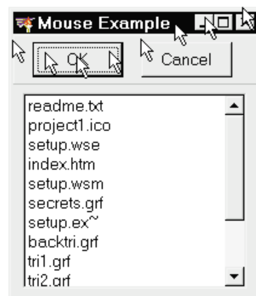


Figure 12: Moving the mouse on a form.

There is no message type, but you can declare *TCmMouseLeave* to be the same as *TCmMouseEnter*, as shown in Figure 11.

Cm_ParentColorChanged Message

When a control's *Color* property changes, it notifies its child controls by sending the *Cm_ParentColorChanged* message. When a control's *ParentColor* property becomes *True*, it sends the *Cm_ParentColorChanged* message to itself to update its *Color* property to match that of its parent. Upon receiving this message, a control checks its *ParentColor* property, and if *True*, sets its *Color* property to match that of its *Parent*. When a component is added to a form, Delphi sends this message to ensure the component's *Color* property is initialized correctly.

In most cases, you don't need to handle or send this message, because Delphi does so automatically. This message takes no arguments and returns no result, so you should use *TWmNoParams* as the message type.

Cm_ParentCtl3DChanged Message

The *Cm_ParentCtl3DChanged* message is similar to *Cm_ParentColorChanged*, but corresponds to the *Ctl3D* property.

Cm_ParentFontChanged Message

The *Cm_ParentFontChanged* message is similar to *Cm_ParentColorChanged*, but corresponds to the *Font* property.

Cm_ParentShowHintChanged Message

The *Cm_ParentShowHintChanged* message is similar to *Cm_ParentColorChanged*, but corresponds to the *ShowHint* property.

Cm_Release Message

A form's *Release* method posts the *Cm_Release* message to itself. When the form receives this message, it frees itself by calling the *Free* method. Note that by *posting* the message instead of *sending* it, the form ensures all pending messages will be handled properly. This ensures an orderly cleanup before closing the window.

There is little reason to handle this message. Instead, you can write an *OnClose* or *OnDestroy* event handler, or override the form's destructor, whichever is more convenient. There is also no reason to send this message. Instead, call the *Release* method.

This message takes no arguments and returns no result, so you should use *TWmNoParams* as the message type.

Cm_ShowHintChanged Message

The *Cm_ShowHintChanged* message is similar to *Cm_ColorChanged*, but corresponds to the *ShowHint* property.

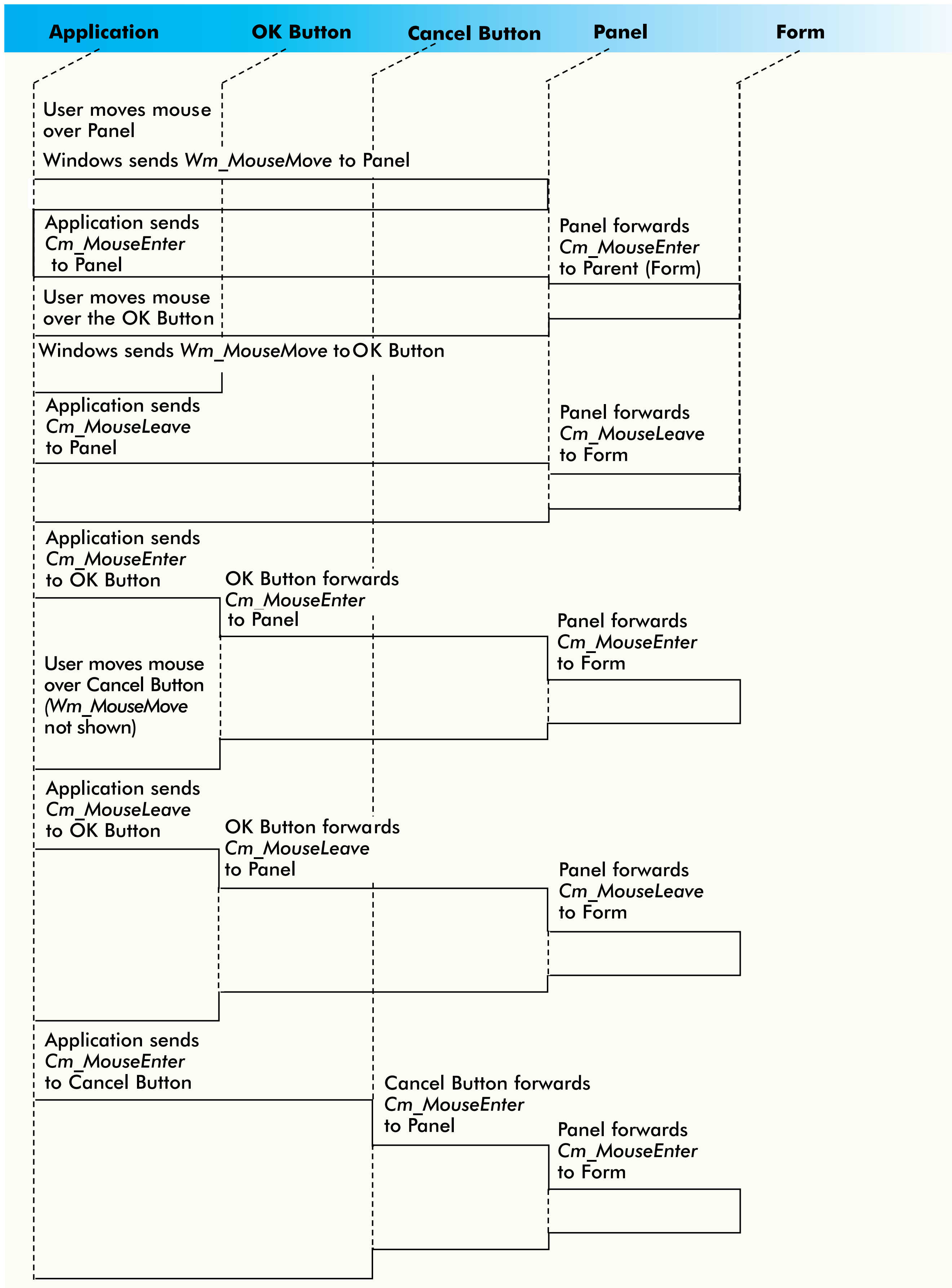


Figure 13: Event diagram for *Cm_MouseEnter* and *Cm_MouseLeave*.

Cm_SysColorChange Message

When a form receives a *Wm_SysColorChange* message, it sends the *Cm_SysColorChange* message to itself and clears the graphics caches (pens, brushes, and canvases). The default behavior of a control when it receives the *Cm_SysColorChange* message is to broadcast the message to its children.

Remember that Windows sends the *Wm_SysColorChange* message when the user changes the system color scheme. Every form repaints itself with the new color scheme, but if your component saves any color information (such as a bitmap background), you should handle this message to refresh your component's information.

This message takes no arguments and returns no result, but the message type *TCmSysColorChange* is defined, which is the same as *TWmNoParams*.

Cm_TextChanged Message

When a component's *Text* or *Caption* property changes, it sends the *Cm_TextChanged* message to itself. Many controls invalidate themselves upon receiving this message. If you derive your component from a custom component, you can let the base class handle this message.

However, if you create a new component that publishes the *Text* or *Caption* property, you might need to handle this message to call *Invalidate*. If your component takes additional measures when its text changes, you can write an event handler, but remember to call the inherited handler, too.

This messages takes no arguments and returns no result, so you should use *TWmNoParams* as the message type.

Cm_TimeChange Message

When a form receives a *Wm_TimeChange* message, it sends the *Cm_TimeChange* message to itself. The default behavior of a control when it receives the *Cm_TimeChange* message is to broadcast the message to its children. If your component uses the system date or time, you must handle this message.

This message takes no arguments and returns no result, but the message type *TWmTimeChange* is defined, which is the same as *TWmNoParams*.

Cm_WinIniChange Message

When a form receives a *Wm_WinIniChange* message, it sends *Cm_WinIniChange* to itself. The default behavior of a control when it receives the *Cm_WinIniChange* message is to broadcast the message to its children. Your component can handle this message if it needs to respond to changes in the user's environment.

The message type is *TWmWinIniChange*, which defines the *Section* field as a *PChar* variable. The *Section* is the name of the section that changed in WIN.INI, or **nil** if more than one section changed.

Conclusion

In this article, you learned about some of the messages that Delphi components send to each other. These messages are an integral part of the VCL; when you write custom components, you need to be aware of these messages so you know which ones the component must handle and which messages it must send.

In many cases, your component inherits the correct behavior from the standard Delphi control classes, such as *TCustomControl*. There are times, however, when a component must implement non-standard or uncommon behavior. For example, a clock component might need to be notified when the system time changes. Windows sends the *Wm_TimeChange* message to the form, and the form broadcasts the *Cm_TimeChange* message to all its controls. Thus, when you implement a clock component, you must know how to handle the *Cm_TimeChange* message, which was covered in this article.

Some messages are sent to or from the *Application* object, such as the *Cm_AppKeyDown* message. In Delphi 1, this message is especially important when you are writing an application where each form has its own menu bar. Unless you handle this and the *Cm_AppSysCommand* messages, all keyboard shortcuts are forwarded to the main form's menu bar, so the individual forms can't implement their own menu bars correctly. You learned how to intercept these messages, so any form can have its own menu bar. ∆

This article is adapted from material for Ray Lischner's Secrets of Delphi 2 *[Waite Group Press, 1996], available for $49.99 from your local bookstore or by calling (800) 428-5331.* Secrets of Delphi 2 *contains a description of every internal VCL message.*

Ray Lischner is a software author and consultant for Tempest Software (http://www.tempest-sw.com). His current focus is object technology, especially in Delphi, Java, and Smalltalk. You can reach him at delphi@tempest-sw.com.

Delphi 1 / Delphi 2

By Mark Ostroff

What's in a Name?

Naming Conventions for Delphi Projects

Delphi's focus on RAD development enables a developer to drop a few components onto a form, write a few lines of code, and have a working prototype of the application with little or no effort.

In fact, Delphi might make it too easy. Soon you can have a bewildering number of components in your application with useful names such as *Edit1*, *Query2*, and *DataSource5*. Delphi's default object names are only good as place holders until you develop a workable naming convention.

Why Design a Naming Convention?

Without some forethought, the list of component names you choose could become as confusing as the names Delphi assigns. Many of us begin naming our Delphi objects based on their functionality — names such as *ExitButton*, *DeptField*, and *DateSpin*.

This convention seems to make sense for a while. However, as you use your objects, it soon becomes apparent that you need a better way of organizing their names.

Organization becomes important. Delphi's Object Inspector lists your components in alphabetical order. This feature is only useful in large projects when you have an effective naming system. The larger the number of objects in your project, the more important the naming of those objects becomes. Otherwise, you'll have more and more difficulty finding the correct object to select in the Object Inspector's pull-down lists. A workable naming convention is also important when you start working on a project with a team of developers, because you might not be the only developer creating object names.

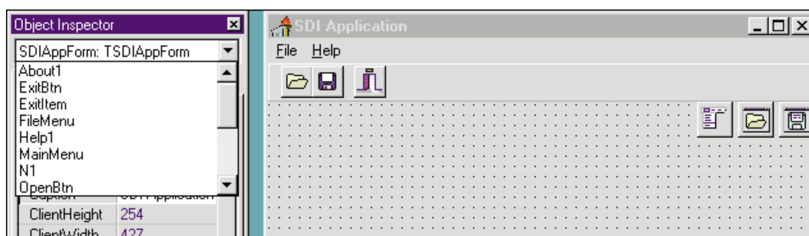


Figure 1: Lack of a workable naming convention causes an unorganized list of objects in the Object Inspector.

Figure 1 shows the result of creating a new project using the SDIApp template from Delphi's Object Repository. Notice that no apparent organization exists for this list of objects. Because the names are ambiguous and inconsistent, it's hard to tell which object some of the names refer to.

Do FileMenu and MainMenu refer to separate objects of type *TMainMenu*? Do ExitItem and FileMenu both refer to *TMenuItem* objects? These aren't idle questions. As your projects grow, you will remember what *type* of component you want to work with, but you might not remember its name.

Side-effects of poor object naming. The lack of a good object naming convention affects more than selecting objects from the Object Inspector. Delphi automatically names the event handlers that respond to user events. These method names are derived from the name of the target object

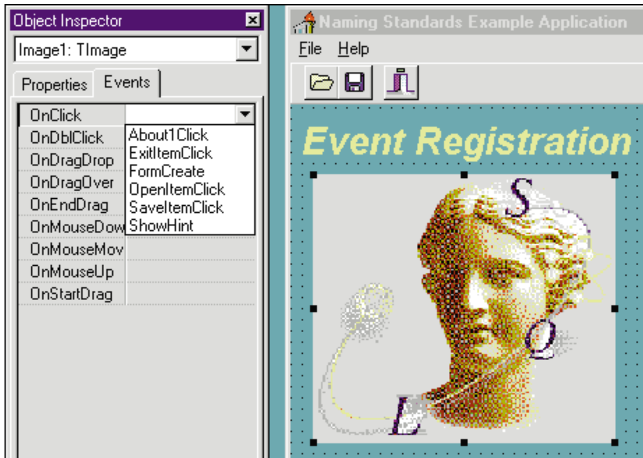


Figure 2: Side-effects of assigning an existing method to the *OnClick* event of another object.

for which they are defined. Therefore, disorganized object names also make it difficult to select the proper event handler to reuse in other objects.

Figure 2 shows what happens when you try to assign an existing method to the *OnClick* event of another object. The naming inconsistency of the objects carries over to the naming of the event handlers. While *FormCreate* is fairly clear, what object association does the *ShowHint* method have? You'd have to look at the code to find out.

Why doesn't a name such as ExitButton work? Most people start by using names that begin with the functionality and end with the component type. This convention eliminates event handler naming inconsistencies. Unfortunately, it doesn't help when you're looking for an object of a certain type, but can't remember what abbreviation you used for its functionality. You need to use something that leverages the Object Inspector's auto-sorting of names.

A System That Works

What we need is a naming convention that follows the natural thought processes of the human mind, and then takes advantage of Delphi's features to make even large projects easier to work with. Because most people have trouble remembering names, but not *types* of names, the object naming convention should result in a list that is sorted by type, then description. We also want some visual method whereby the developer can easily focus on the functional description after the desired type has been found.

Pre-pending the object type. Getting Delphi to sort the object names by type is easy. Simply place the type first — rather than last — in the object's name. For example, a collection of buttons might be named *btnClose*, *btnDirChange*, and *btnFileOpen*. The pre-pending of the object type places all objects of the same type along with their object-related event handlers in any Object Inspector pull-down list. The functional descriptions that follow the type abbreviation make it clear what action each button is meant to perform.

Figure 3: Use of two-letter prefixes to designate legal values for color schemes.

This type abbreviation prefix scheme is actually used in Delphi, and all enumerated types use this prefix convention. For example, Figure 3 shows the use of a two-letter prefix to designate legal values for color selections.

Delphi provides over 100 types of components. The use of a three-letter prefix for object names seems to provide enough information for meaningful type abbreviations.

Why case is important. The above examples all use a lower-case object type abbreviation. The type abbreviation is followed by the object's functional description in proper case (i.e. the first letter of each word or phrase is capitalized). This particular case selection is deliberate. Once again, it follows the convention used within Delphi.

The initial goal of sorting object names is achieved by placing the type abbreviation first. The use of lower-case followed by proper-case enables the developer to ignore the type and focus on the object's description. This case organization helps when selecting a *specific* object from a grouped list of similar objects.

The type prefixes. The table in Figure 4 is a suggested list of object type prefixes you can apply to your object names. It covers all the components that ship with Delphi. This list is, of course, not cast in concrete. Adjust the prefixes to suit your tastes and add prefixes for new object types that you create or purchase. The only hard and fast rule is to keep your object type prefixes consistent.

Explaining Some of the Suggestions

Some of the distinctions between the prefixes listed may seem a bit arbitrary. Why lump some components under the same prefix, while others are out on their own? In general, components used in similar ways are placed under the same prefix. Both the Edit and DBEdit components are employed in much the same way, so both use the `edt` prefix. The Memo, DBMemo, and RichEdit objects also have similar functions, yet they differ quite a bit from Edit objects. So they get their own prefix, `mmo`.

Some of the more "colorful" prefixes were created to avoid collisions. The QuickReport components are an example. Sometimes you must be imaginative to create a set of unique three-letter combinations.

Component types you don't often use might be good candidates for prefix consolidation. You could lump all the

Prefix	Object Type
bar	ProgressBar, StatusBar, TrackBar
bat	BatchMove
btn	Button, BitBtn, SpeedButton
bvl	Bevel
cal	Calendar
cbx	ComboBox, DBComboBox
chk	CheckBox, DBCheckBox
dbs	Database
dcc	DDEClientConv
dci	DDEClientItem
dir	DirectoryOutline
dlg	Dialogs (Open, Save, Font, Color, Print, PrinterSetup, Find, Replace)
drv	DriveComboBox
dsc	DDEServerConv
dsi	DDEServerItem
dtm	DataModule
dts	DataSource
edt	Edit, DBEdit, SpinEdit
fcb	FilterComboBox
frm	Form
ftp	FTP (Internet File Transfer Protocol)
gge	Gauge
grd	ColorGrid, DrawGrid, DBGrid, DBControlGrid, StringGrid
grf	ChartFX, VCFirstImpression, GraphicsServer
grp	GroupBox
hdr	HeaderControl, Header
hot	Hotkey
htm	HTML (Internet Web Browser)
htt	HTTP (Send, Receive or Search HTML Documents
ibe	IBEventAlerter
img	Image, DBImage
lbl	Label, DBText
lst	ListBox, DBListBox, DirectoryListBox, FileListBox, ImageList
luc	DBLookupComboBox, DBLookupCombo
lul	DBLookupListBox, DBLookupList
lvw	ListView
med	MediaPlayer
mmo	Memo, DBMemo, RichEdit
mni	MenuItem
mnu	MainMenu
msk	MaskEdit
nav	DBNavigator
nbk	Notebook, TabbedNotebook
nnt	NNTP (Internet Network Newsgroup Access)
ole	OLEContainer
out	Outline
pbx	PaintBox
pge	PageControl
pnl	Panel
pop	POP (Internet Post Office Protocol to receive E-Mail)
pum	PopUpMenu
qrb	QRBand
qrc	QRCalc
qrd	QRSysData
qrg	QRGroup
qrl	QRDetailLink
qrm	QRMemo
qrp	QuickReport
qrv	QRPreview
qrs	QRShape
qrt	QRLabel, QRDBText
qry	Query
rad	RadioButton
rgp	RadioGroup, DBRadioGroup
rpt	Report
sbr	Scrollbar
sbx	Scrollbox
shp	Shape
sht	VCFormulaOne
smt	SMTP (Internet Simple Mail Transport)
spl	VCSpeller
spn	SpinButton, UpDown
ssn	Session
stp	StoredProc
tab	TabControl, Tabset
tbl	Table
tcp	TCP (Internet Data Exchange … like a telephone)
tmr	Timer
tre	TreeView
udp	UDP (Internet Data Broadcast … like a radio)
ups	UpdateSQL

Figure 4: Suggested object type prefixes.

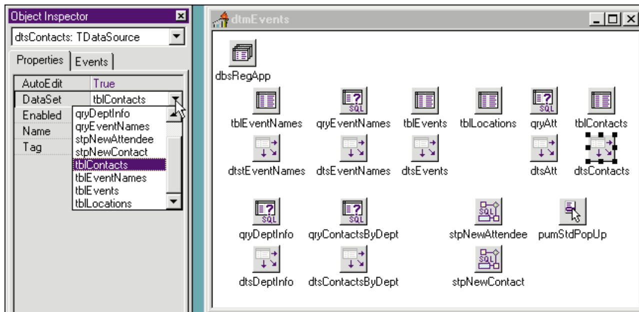
Internet Solution Pack components under the prefix `web`. If you rarely use DDE, you might want to use the prefix `dde` for all four DDE component types.

Some Examples

The effects on the ease of development are best seen in an example. In Figure 2, we saw that poor object naming can make method selection difficult.

Data object naming. Look at the DataModule displayed in Figure 5. Assigning a DataSet to the *dtsContacts* DataSource object is simplified by this naming convention. It's easy to see that this DataModule contains a number of Query, Stored Procedure, and Table objects.

Menu object naming. Note that this DataModule also contains an application standard PopUpMenu named



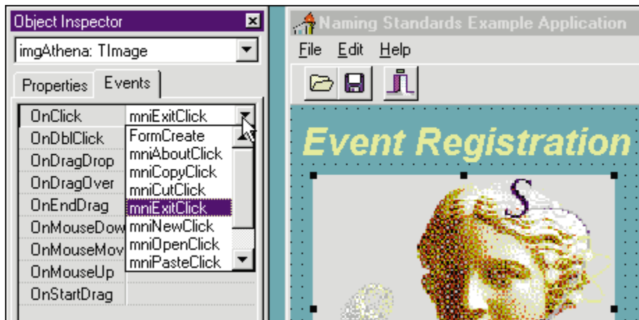


Figure 5 (Top): An example of a workable naming convention.
Figure 6 (Bottom): Easing method selection.

pumStdPopUp. Because PopUpMenus are more amenable to reuse, it makes sense to include standard ones in a DataModule.

It also makes sense to give them their own type prefix. PopUpMenus are used differently than MainMenus, and their type prefix should reflect this, rather than lumping them together with other menus.

Menus also have unique naming convention requirements. Typically you'll want to perform one set of actions on a menu object itself. You'll want to execute another kind of logic in response to events that occur for MenuItems *within* that menu object.

Thus, the prefix list in Figure 4 suggests `mnu` and `pum` for MainMenu and PopUpMenu objects, respectively, and `mni` for use with MenuItems themselves. The clarity this adds to selecting event handlers is shown in Figure 6.

What about file names? File naming conventions weren't much of an issue in Delphi 1. Nearly every .PAS file was a form. However, Delphi 2 added the ability to create DataModules.

At a glance, this issue seems easy to resolve. Simply use the same three-letter prefixes you used for Form and DataModule objects. However, Delphi doesn't allow you to save a unit with the same name as an object within that unit.

Prefix	File Type
fm	Form
dm	DataModule

Figure 7: File type prefixes.

You only need two designations for file names, one for Forms and another for DataModules. Simply use a two-letter prefix when you save the file (see Figure 7).

Conclusion

Try this naming convention in your next project. After an adjustment period, I think you'll find your Delphi development will proceed even more rapidly than before. As your project grows, you'll also realize more efficiencies.

This article builds on the work of two groups. The initial effort was started by Borland. Many of the concepts and naming conventions covered here are used by Borland for developing its in-house applications.

Additional work was performed by the Delphi Study Group, sponsored by Informant Communications Group on its CompuServe forum. Additional thanks go to Tom Arnold and the other members of the group for consolidating much of the discussion about type prefixes. ∆

Mark Ostroff has over 18 years experience in the computer industry. He began by programming minicomputer medical research data acquisition systems, device interfaces, and process control database systems in a variety of 3GL computer languages. He then moved to PCs using dBASE and Clipper to create systems for the US Navy, as well as for IBM's COS Division. He also volunteered to help create the original Paradox-based "InTouch System" for the Friends of the Vietnam Veterans' Memorial. Mark has worked for Borland for the past six years as a Systems Engineer, specializing in database applications.

Delphi at Work

Delphi / Object Pascal

By *David Faulkner*

TBarCode

A Custom Bar Code Component

Ubiquitous, even annoying, bar codes are nevertheless extremely handy, and can relieve much data entry tedium. Their Delphi implementation was therefore just a matter of time. And — well — it's time!

This article presents *TBarCode*, a component that displays bar codes which can be read into a computer using an optical scanner as the input device. *TBarCode* implements the Code 39 specification, but you should be able to extend it to display any bar code format. Figure 1 shows *TBarCode* in action.

Before creating this component, I experimented with bar code fonts that display coded symbols for each character, i.e. setting *TLabel*'s *Font* property to a bar code font effectively creates a bar code. While this approach worked, it meant I had to worry about licensing the font and installing it on users' machines. This was potentially expensive, and a lot of trouble compared to the ease with which *TBarCode* was assembled.

Request for Verification of Deposit

Island Community Lending

254X3760

tion of Deposit

ation is to be used by the agency collecting it or its assignees in determining whether you qualify as a prospective mortgagor under its program. It will not
ept as required and permitted by law. You do not have to provide this information, but if you do not, your application for approval as a prospective
l or rejected. The information requested in this form is authorized by Title 38, USC, Chapter 37 (if VA); by 12 USC, Section 1701 et. seq. (if HUD/FHA); by 42
Title 42 USC. 1471 et. seq., or 7 USC. 1921 et. seq. (if USDA, FmHA).

lete Items 1 through 8. Have applicant complete Item 9. Forward directly to depository named in Item 1.
ase complete either Part II or Part III as applicable. Complete Part IV and return directly to lender named in Item 2.
be transmitted directly to the lender and is not to be transmitted through the applicant(s) or any other

epository)	2. From (Name and address of lender)
nolulu Street HI, 96813 43-3700	Island Community Lending 201 Merchant Street Honolulu, HI, 96813 Phone (808) 545-1000

been sent directly to the bank or depository and has passed through the hands of the applicant or any other party.

	4. Title (Loan Processor/Underwriter)	5. Date	6. Loan Number
		8/7/96	123

FIED

	Account in the Name of	Account Number	Balance
		123	$45,000.00

Figure 1: The *TBarCode* component at work.

The Specification

Code 39 maps each of a set of pre-defined printable characters to a set of nine bars. A bar can be wide or narrow, black or white, meaning both the black bars and the spaces between are meaningful.

There's no specific width assigned to a bar; instead, only the ratio of the bars is specified. The wide bars must be 2.2 to 3.0 times as wide as the narrow bars. The larger ratio is usually favored, as it results in fewer read errors.

Code 39 supports 44 printable characters:

- upper-case letters A through Z
- the numbers 1 through 9
- the symbols -, ., $, /, +, %, and the blank character

(The * character is also supported, but it's used as a start and stop character, and cannot be used as a data item.)

Each character is assigned a unique series of nine bars. The first and last bar are always black; the remaining bars alternate between black and white. For example, the series for "A" is `WNNNNWNNW`; that is, a wide black bar followed by a narrow white bar followed by a narrow black bar, etc. Figure 2 shows the bar code for this character.

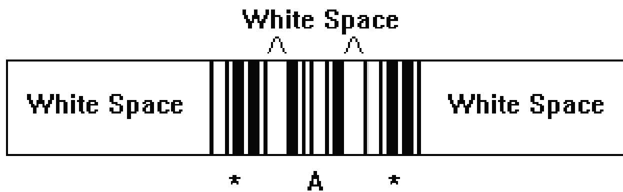


Figure 2: The series of bars for the letter "A".

The series for the rest of the character set is in the source code listing as *BarCodeTable*. Every character is encoded with nine bars, three of which are wide, thus the name *Code 39* (Code 3 of 9).

A white space is inserted between each character so the reader can distinguish the end of one character and the beginning of another. The white space's width isn't specified, but it's generally set to the width of a narrow bar.

A legal Code 39 bar code requires a start and stop character — the * character is encoded as NWNNWNWNN. Thus the shortest Code 39 bar code is three characters, although the bar code-reading hardware usually strips off the * characters. This requirement also makes it easy to visually check if a bar code is Code 39. If the first nine bars match the last nine bars and the bars are two distinct widths, you can be fairly certain you are looking at a Code 39 bar code.

The left and right edges of the entire bar code must be surrounded by enough white space for the optical scanner to distinguish between the end of one bar code and the beginning of another. The white space must be at least .25 inches, or 10 times the width of a narrow bar, whichever is greater. The *TBarCode* component doesn't consider this; instead, it expects the user to place the component in an area with sufficient white space.

The Component

Many excellent articles on component building are available. So here we'll concentrate on the code unique to *TBarCode*.

TBarCode's component declaration is one of the simplest you'll see:

```
type
  TBarCode = class(TCustomLabel)
  private
    procedure CaptionChanged(var Message: TMessage);
      message CM_TEXTCHANGED;

  protected
    procedure Paint; override;

  public
    constructor Create(AOwner: TComponent); override;

  published
    property Caption;
    property Alignment;
end;
```

Descend

The first step in creating a component is to decide from which component to descend. In *TBarCode*'s case, *TCustomLabel* is the perfect choice. *TBarCode* is simply a label that paints itself in an encoded format. *TCustomLabel* provides a canvas on which to draw the bar code, and has published properties, such as *Left*, *Top*, *Width*, and *Height*, that we don't have to code.

Additionally, *TCustomLabel* has a number of fully functional properties that are hidden from the user because they are protected. By simply publishing the *Alignment* and *Caption* properties, we round out the required properties.

Create

Next, the *Create* method is overridden to provide some reasonable defaults:

```
constructor TBarCode.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ControlStyle := ControlStyle - [csOpaque];
  Alignment    := taCenter;
  AutoSize     := False;
  Height       := 50;
  Width        := 175;
end;
```

If the *Height* and *Width* properties are not set, they take on the default *Height* and *Width* of their ancestor, *TCustomLabel*. This causes problems, because the default *Width* of a label is not wide enough for *TBarCode* to display the bar code interpretation of its default caption, BARCODE1. The *Paint* method won't print a bar code if the whole caption doesn't fit, so the user would have an invisible component. Setting the *Width* property to 175 assures that a default bar code will be visible on the screen when the user places *TBarCode* on a form.

A minor disadvantage to deriving *TBarCode* from *TCustomLabel* (compared to *TGraphicControl*) is that *TCustomLabel* implements the *AutoSize* feature. Of course, *TCustomLabel* gets the width wrong for *TBarCode* because it expects the caption to be painted in the current font. Because we aren't publishing the *AutoSize* property, the *Create* method sets *AutoSize* to *False*, so we can forget about it.

Removing the *csOpaque ControlStyle* is also important to the drawing of the bar code. According to Delphi's online Help, adding *csOpaque* to the *ControlStyle* property means that the control hides any items behind it, making it unnecessary to draw them.

By removing *csOpaque*, we're telling Windows to redraw the background before calling *TBarCode*'s *Paint* procedure. Redrawing the background erases any existing bar codes from previous paints, so we don't need to code an erase routine. Additionally, when printing white bars, the *Paint* routine doesn't need to print anything at all; instead, it relies on the background color to show through.

Auto Update

You have probably noticed that when editing the *Caption* property of a label in the Object Inspector, the Label on the form is updated with each key stroke. Two mechanisms can make this happen.

The first mechanism is through the creation of a custom Property Editor, say *TMyCaptionEditor*, with the *paAutoUpdate* attribute:

```
type TMyCaptionEditor = class(TStringProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure SetValue(const Value: string); override;
  end;

...

function TMyCaptionProperty.GetAttributes:
  TPropertyAttributes;
begin
  Result := [paMultiSelect,paAutoUpdate];
end;
```

paAutoUpdate tells Delphi to call the *SetValue* procedure after each change is made to the *MyCaption* property, instead of waiting until an entire edit session is posted. Within the *SetValue* procedure, you can then cause a component to be repainted. This works fine, but the second mechanism is simpler.

Delphi defines a set of Component Messages that are listed in the online Help. One of the messages, *Cm_TextChanged*, is triggered whenever the *Text* property of a component is changed (regardless of the Property Editor associated with that property). By attaching code to this message to repaint the bar code, the *TBarCode* will exhibit the same *AutoUpdate* behavior as a *TLabel*.

What's puzzling about this mechanism is that the user is changing the *Caption* property of *TBarCode*, not the *Text* property. In fact, *TBarCode* doesn't even appear to have a *Text* property. Because it's descended from *TCustomLabel*, *TBarCode* has all the properties of *TCustomLabel*, including the *Text* property. This is a **protected** property, so the Object Inspector and your code don't have access to it. But Delphi does have access to it, and magically keeps "in sync" with the *Caption* property.

You can prove this by descending a component from *TCustomLabel* and publishing the *Text* property. As you edit the *Caption* property in the Object Inspector, the *Text* property is automatically updated (the reverse is also true). Thus, whenever the *Caption* property is changed, a *Cm_TextChanged* message is triggered, which in the case of *TBarCode*, calls the *CaptionChanged* procedure. *CaptionChanged* causes the bar code to repaint, effectively implementing the *AutoUpdate* feature.

Validating Input

As mentioned, the Code 39 specification only supports 44 characters, so it's necessary to validate any input the user supplies. It would be reasonable to give the user an error message when an invalid message is entered, and raise an exception if it happens at run time. However, it's easier and less intrusive to convert any lower-case characters to upper case and filter out any illegal characters.

Since the *CaptionChanged* procedure is called each time the *Caption* is changed, the validation is done within the *CaptionChanged* procedure. Note that although some characters will be discarded, they still appear in the Object Inspector's in-place editor until the user exits. This is because the in-place editor does not refresh itself from the *Caption* property after each change.

Painting Concerns

As you can see from the component definition, the *Paint* procedure is overridden so we can paint a bar code instead of a text caption. The code for the *Paint* procedure is:

```
procedure TBarCode.Paint;
begin
  BarCodePaint(Caption,Canvas,0,0,Width,Height,Alignment);
end;
```

There's not much to it; in fact, all it does is call *BarCodePaint*. While this appears inefficient, it's always a good idea to design visual components in this manner. By doing so, the component can be "painted" on a printer as well as on the screen.

Because *BarCodePaint* is declared in the **interface** section of the Barcode unit, it can be called by other units. For example, the *PrintClick* event prints the form's bar code:

```
procedure TForm1.PrintClick(Sender: TObject);
var
  ScaleX,ScaleY : Integer;
begin
  Printer.BeginDoc;

  ScaleX := WinProcs.GetDeviceCaps(
             Printer.Handle,LOGPIXELSX) div 96;
  ScaleY := WinProcs.GetDeviceCaps(
             Printer.Handle, LOGPIXELSY) div 96;

  BarCodePaint(BarCode1.Caption,Printer.Canvas,
              BarCode1.Left*ScaleX,
              BarCode1.Top*ScaleY,
              BarCode1.Width*ScaleX,
              BarCode1.Height*ScaleY,
              BarCode1.Alignment);

  Printer.EndDoc;
end;
```

I find it simplifies component development if I make *Paint* routines available to other units and have those routines accept a *Canvas* parameter so the components can be easily printed. Because *BarCodePaint* is coded in this way, only one line of code was necessary to implement a QuickReport version of this component.

BarcodePaint

BarcodePaint is the workhorse of the component, but it's still quite simple. It begins by checking the input, setting up the canvas, then adding the following code:

```
for x := 1 to Length(Caption) do begin
  Index := Pos(Caption[x],cValidCode39Characters);
  if Index = 0 then
    Continue;
  for y := 1 to 9 do
    PaintABar(BarCodeTable[Index][y],odd(y),Canvas,
             LeftEdge,Top,Height,NarrowWidth,WideWidth);
  Inc(LeftEdge,NarrowWidth);
end;
```

The interesting part here is that the *cValidCode39Characters* constant is not only used to validate the input; it's also a map into *BarCodeTable*. For example, the character "A" is the 11th character in the *cValidCode39Characters* string and corresponds to the 11th element in the *BarCodeTable* array.

BarCodeTable is a two-dimensional array whose first element is 1,1 for conveyance (as compared to 0,0). Each row of *BarCodeTable* contains a series of nine Ws and Ns which specify the pattern of wide and narrow bars for the corresponding character.

PaintABar

PaintABar's job is to paint a single bar at the passed coordinates. The code that paints the bar is:

```
if ThisBarIsBlack then
  aCanvas.Rectangle(LeftEdge,Top,
                    LeftEdge+CurrentWidth,Top+Height);
Inc(LeftEdge,CurrentWidth);
```

If the bar being printed is white, *PaintABar* does not paint anything, but still advances the left edge. *TBarCode* can get away with not painting anything because it removed *csOpaque* from the *ControlStyle*. This means that when painting on a screen, Windows will erase the previously painted bar code, and when painting on a printer's canvas, the white spaces will let the paper show through (see Figure 3).

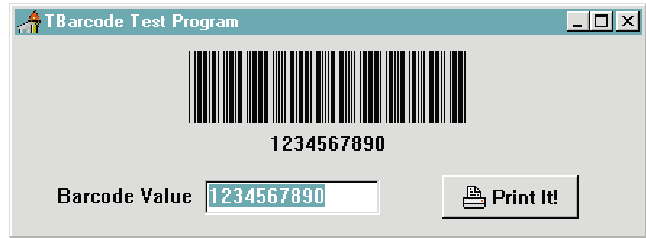


Figure 3: A Code 39 bar code created by *TBarCode*.

Other Goodies

In addition to *TBarCode*, you should check out the download file. It contains *TDBBarCode*, a data-aware version of *TBarCode*; *TQRBarcode*, a QuickReport compatible version; and *TQRDBBarCode*, a QuickReport data-aware version. Both 16- and 32-bit resource files are included, so it can be compiled in Delphi 1 or 2.

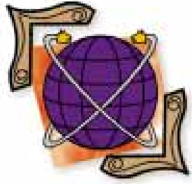
Conclusion

I've included the sample program, shown in Figure 3. It lets you enter a value at run time and see the resulting bar code. The sample program also prints the bar code for you so you can test it against your optical scanner.

For more information and specifications for other bar codes, check out http://www.hp.com:80/AccessGuide/TI_ahp.html and follow the link to Bar Code Symbologies. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JAN\DI9701DF.

David R. Faulkner is President of Software Development at Island Community Lending in Honolulu, HI. He is co-author of *Using Delphi: Special Edition* (QUE, 1995) and can be reached at (808) 572-5524 or davef@maui.net.



Case Study

By Don Bauer

TWorldMap

Seeing the World in a New Way

While creating an application for the United States Marine Corps, Stanley Associates' Senior Technical Manager, Don Bauer, and Senior Analyst, Dan Adams, wanted to add a means to graphically view training schedules relative to training location. Unfortunately, time and money constraints wouldn't allow Stanley Associates to simply buy conventional map software. Additionally, the Marine Corps Training Exercise and Employment Planning (MCTEEP) project didn't warrant developing the map at the government's expense.

With this in mind, Don read *Programming Windows with Borland C++* by William Roetzheim [Ziff-Davis Press, 1992] which contained, as an example, a primitive map system — no other functionality was provided. Don decided to build a map component on his own, and if successful, market it for future release.

The Development Process

Don's ultimate goal was to create a low-cost, royalty-free, easy-to-use GEO interface that wasn't encumbered with the complexities of typical GIS software. His goals, as he began development in September 1995, were straightforward — he wanted to create a component in which users only needed to know the longitude and latitude, and the online Help would walk them through the rest. The original features would include a map, zooming, and points.

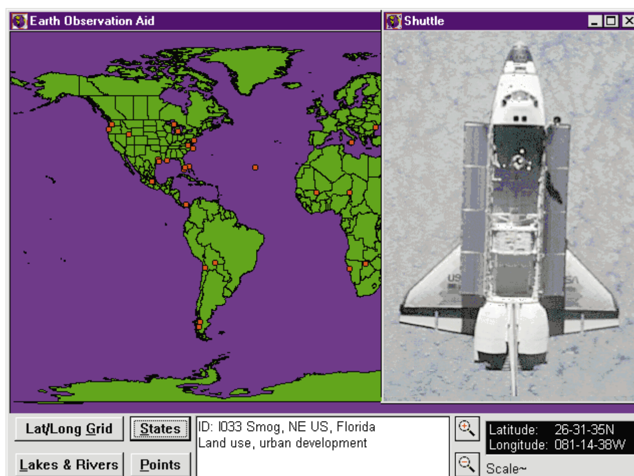


Figure 1: A capture from the Earth Observation Aid being used on the Mir space station.

Delphi seemed the natural choice to Don, as earlier that year, he, along with four other programmers, converted MCTEEP v1.0 from PowerBuilder to Delphi in less than four months. The switch to Delphi resulted in much-improved capability, tremendous performance gains, and the ability to easily modify and improve the code.

Don approached the map development process incrementally, finalizing each step of the process to ensure that the original goals were met. As development proceeded, he added functionality that fit in well with the design intent of the component.

The development process wasn't always easy. The biggest setbacks came from working with the Windows 3.1 graphic design interface (GDI). It was a constant struggle to keep within the bounds of performance and resource usage with the 16-bit version, because the GDI not only affected display, but printing as well. When working with

Application Profile

The *TWorldMap* component is totally configurable by the developer. Its functionality includes built-in zoom, built-in adjustable latitude/longitude grid, a custom point object, custom line object, clickable polygons, custom printing, and creating bitmaps. Available in both 16- and 32-bit versions, *TWorldMap* is a royalty-free, easy-to-use solution for many day-to-day geo-presentaion needs.

Target Audience: Anyone needing a simple, fast, easy-to-use, customizable map interface added to their application.

Contact: Don Bauer
Phone: (202) 396-6970
Fax: (202) 396-6914
Voice Mail/Pager: (703) 616-6166
Web Site: http://www.stanleyassoc.com/worldmap/wmap.htm
E-Mail: donbauer@illuminet.net

large polygons in the Windows 3.1 GDI, things stopped working for no apparent reason, making development difficult. To overcome this, Don and David Steel, a Senior Windows Programmer, systematically determined the limitations of the GDI and built the map component's display and printing capabilities to work within them.

The Beta Cycle

Despite the setbacks, the beta cycle began moving the component toward its current form. The beta testers gave Don feedback, and wherever possible, the changes were implemented. This crucial process resulted in a stable, feature-rich component. *TWorldMap* version 1.0 began shipping in March of 1996 as a 16-bit component. Currently, *TWorldMap* is available in both 16- and 32-bit versions with parallel capabilities under a combined codebase.

Capabilities

TWorldMap's capabilities include:

- Built-in Zoom — The three ways to control the map are: set the *EnableZoom* Property to *True* and click your mouse button, drag a rectangle on the map, and it resizes automatically; use the *ZoomByFactor* method and zoom the map, centered on the latitude/longitude value sent; or set custom coordinates and redraw the map.
- Custom Point Object — The Custom Point object allows the developer to add, delete, update, or find points on the map. An *OnPointClicked* event passes all point information to the calling program, enabling actions to be based upon clicking a point on the map. Points support various geometric shapes as well as bitmaps to be displayed.
- Live Clickable Polygons — Each polygon (based upon level of detail) can be systematically included or excluded for use in the *OnPolyClick* event. The component passes the polygon object to the calling program. The description can be extracted and displayed or used to query a database to provide amplifying information.

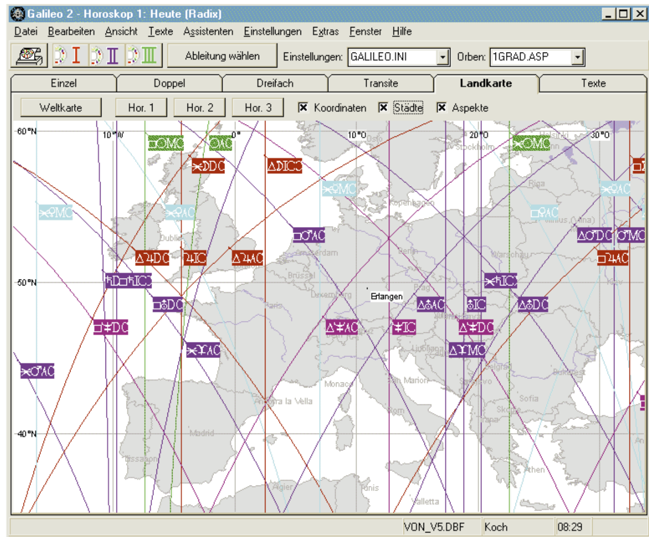


Figure 2: *Galileo*, an astrology program written by Dirk Paessler.

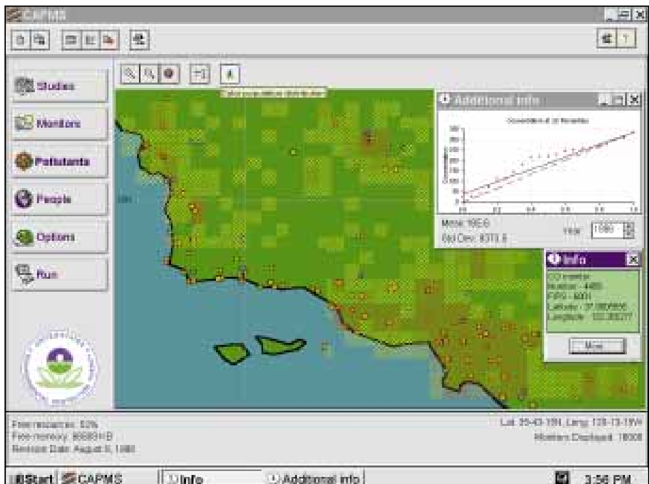


Figure 3: The EPA Clean Air Act model using *TWorldMap*.

- Custom Line Object — The component maintains an internal list of lines that are redrawn anytime the map is redrawn. Support for lines has also been included to allow connecting points on the map to show routes, service webs, etc. If a line passes through the current display area, it is drawn.
- Custom Printing — The component supports two types of printing — full page or user-defined location and size — on an existing print job. Additionally, the map will print in color and supports plotters.
- Create Bitmaps — Create bitmaps of your map screens for inclusion in your presentations and word processing documents. Simply pick a file name and dimensions, and the component will create a bitmap representation of your current map, including all details.

Customers

TWorldMap has many customers who use the component in different ways: NASA and the Earth Observation Program, as illustrated in Figure 1, are currently using the component onboard the Mir space station in software that uses the world map to display points of interest and link them to satellite photographs taken on previous missions (see the sidebar "Earth Observation Aid" on page 44); a trucking company in

Earth Observation Aid

The Earth Observation Group approached the Field Deployable Trainer (FDT) project, a joint team of NASA and United Space Alliance employees, to add an application to a laptop computer already on the Mir space station. Currently, the Earth Observation Group is conducting a scientific experiment on Mir that involves taking a series of pictures over time of many locations on Earth from Mir. The group asked for a map in which you could select one of their sites and see a picture of that site previously taken from space.

The FDT team planned on using a bitmap image for the map. However, while waiting for the pictures to use in the application, the team saw an advertisment for *TWorldMap*. The team thought it would provide a better interface, if the component could be obtained in time; only three weeks remained until the flight software needed to be delivered, and it isn't normally possible to purchase software that quickly for a government project. They called Don to determine if NASA had a license, or if there was some way to use the code. It turned out that another group at the Johnson Space Center, working on a Shuttle emergency landing site program, had the software, and Don let the team tag on.

It took less than a week to write the program using the *TWorldMap* component. The program includes a scale bar, variably spaced latitude/longitude grid bars, and site information that displays when the cursor approaches a marked site, which is why it took as long as it did. *TWorldMap* made the application much better.

The FDT team has received nothing but favorable comments from NASA staffers, including astronauts, who have seen the application. Additional features are planned for the next version.

— *David B. Chiquelin*
Lead Programmer, Field Deployable Trainer project
United Space Alliance

Canada uses the component to show truck routes; three commercial astrology systems use *TWorldMap* (see Figure 2); in Newport Beach, CA, a statistics company uses *TWorldMap* to show salary and benefit distribution within North America;

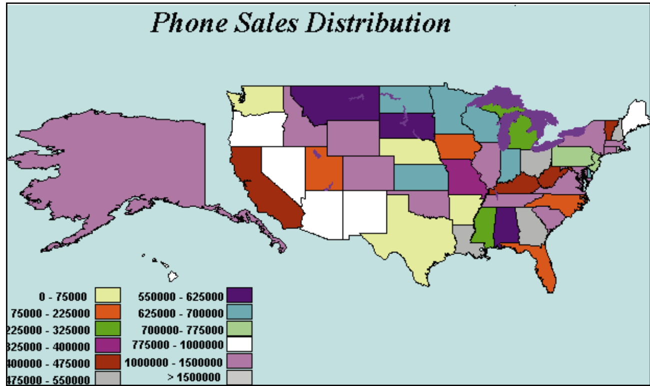


Figure 4: A phone sales distribution system written by Gary Holbrook.

the Environmental Protection Agency (EPA) uses *TWorldMap* for its Clean Air Act health benefit model (see Figure 3); the US Marine Corps uses it for its scheduling system; a commercial marketing company is using the component to color states by sales revenue (see Figure 4); and the US Army is integrating *TWorldMap* into the Automated BattleBook system to show the world's reserve equipment depots.

Looking Ahead

Custom systems are created by Don for customers who need specific capabilities, and all added capabilities are made available to registered users.

Don currently has a few major clients that are waiting for the map to support different projections, live scrolling, infinite zoom-in, and, of course, a native link to database tables; these are his current areas of focus. ∆

Don Bauer is Director of Software Development, East Coast for Marotz, Inc. (a Delphi Development Company), and is currently converting the Navy's Fleet Modernization Program Management Information System to Delphi 2. He can be reached at donbauer@illuminet.net.

NEW & USED

By Bill Todd

InfoPower 2.0

Making Life Easier for Database Developers

A lot of great new features have been added to Woll2Woll Software's InfoPower 2.0, but one thing hasn't changed: InfoPower is still the "must have" addition to Delphi for anyone developing database applications. If you've never used InfoPower, you may want to browse the sidebar "An InfoPower Primer" (on page 47) before reading on.

Picture Masks

One of the more exciting and useful new features in version 2.0 is the use of Paradox-style picture masks for controlling what users can enter in a field. This is a welcome replacement for the very limited edit mask capability built into Delphi. Figure 1 shows the picture characters used with InfoPower.

There are several ways to enter pictures when you use InfoPower components, but perhaps the easiest is by clicking on the *PictureMask* property of a *TwwTable* component to display the dialog box shown in Figure 2.

Character	Description
#	Any digit.
?	Any letter, either upper or lower case.
&	Any letter. Lower-case letters are converted to upper case.
~	Any letter. Upper-case letters are converted to lower case.
@	Any character.
!	Any character. Letters are converted to upper case.
;	Treat the next character as a literal, not a mask character.
*	Repeat count. *# means any number of numbers. *5# means five numbers.
[ ]	Anything in square brackets is optional.
{}	Group of alternatives. {Y, N, U} means the user must enter either Y, N, or U.

Figure 1: The picture characters used with InfoPower.

Clicking the ellipsis button in the **Picture Mask** edit control of this dialog box leads to the Lookup Picture Mask dialog box shown in Figure 3. This dialog box lists a number of useful pictures that have already been built for you; just select the one you want.

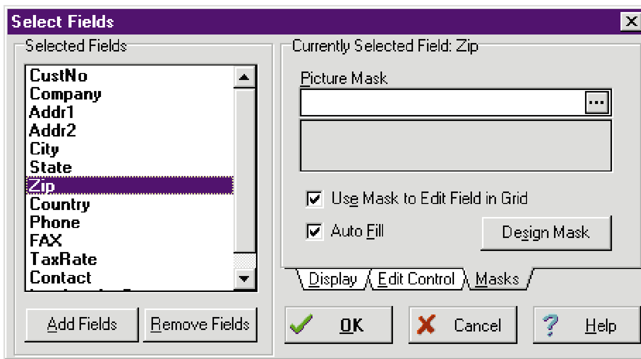
If you need to design a custom picture, click the **Design Mask** button in the Select Fields dialog box (again, see Figure 2) to display the Design Picture Mask dialog box in Figure 4.

For example, suppose you need a picture that will let users enter either a valid US five-digit or nine-digit ZIP code or a Canadian postal code in a field. Start by typing the picture:

```
{#&# &#&,#####[-####]}
```

into the **Picture Mask** field. Now click the **Verify Syntax** button to verify that your picture is syntactically correct, then type sample values into the **Sample Value** field to ensure the picture works as you intended.

If you will use this picture again, click the **Save Mask** button to add it to the database of pictures displayed in the Lookup Picture Mask dialog box shown in Figure 3. This is just one example of a mask that's impossible to create in Delphi without InfoPower.



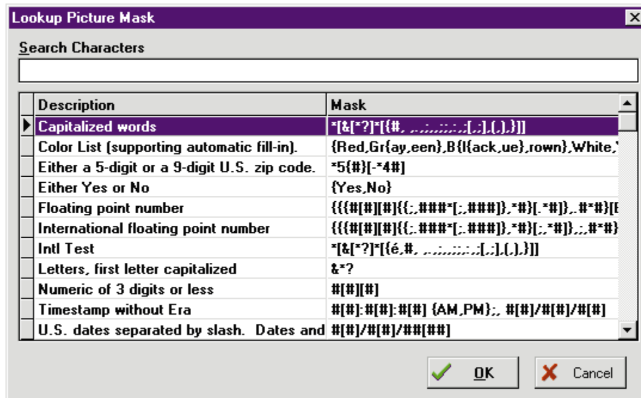


Figure 2 (Top): The Select Fields dialog box provides an easy way to enter pictures. **Figure 3 (Bottom):** Simply select the picture that best fits your needs.

Another good example of the superiority of InfoPower's pictures over Delphi's edit masks is the use of the picture:

```
#[#]/#[#]/##[##]
```

for a simple date. Not only does this picture allow you to enter either a one- or two-digit month or day, it also lets you enter either a two- or four-digit year.

If you've ever tried to use a Delphi edit mask for a date, you have probably discovered that it puts the two literal characters (the slashes) into the field as soon as it gets focus; you can't eliminate them if you then decide to leave the field blank. InfoPower doesn't insert the literals until, when typing your value, you reach the point at which they appear. If you highlight the field and delete the value, the literal characters disappear as well, allowing you to easily leave a field blank.

As with all other InfoPower features, pictures work identically in both Delphi 1 and 2. Also, they apply whether you assign a value to a field in code, or the data is entered by the user through a form.

By setting the *AllowInvalidExit* property to *True*, you can allow a user to exit a field that contains an invalid value. You can also control whether the pictures are used for interactive editing by setting the *UsePictureMask* property.

InfoPower controls also include *OnCheckValue* and *OnInvalidValue* events to let you control what happens when

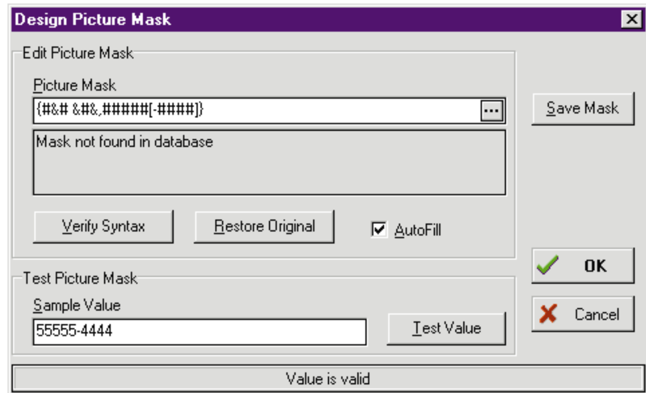


Figure 4: If no picture fits the bill, InfoPower 2.0 lets you "roll your own."

the user enters an invalid value. Using these events, you can determine if the value entered by the user is indeed invalid. If the value is invalid, you can display an error message that identifies the invalid field when the user tries to post the record. Because the *OnInvalidValue* event identifies the offending field, you can also move focus to that field and change its color.

Filter Dialog

TwwFilterDialog gives end users an easy way to filter or query a dataset on multiple fields. To use it, just drop the *FilterDialog* component on your form, set its *DataSource* property, and provide a button or menu choice to call its *Execute* method. Calling *Execute* displays the dialog box shown in Figure 5.

You can specify one of two ways to select records. If the dataset being searched is a *TwwTable* or *TwwQBE*, then a BDE filter is applied. If the dataset is a *TwwQuery*, you can either filter the query result set, or let the *FilterDialog* modify the WHERE clause of the query and re-execute it to select the records. You can set the caption of the dialog box, as well as choose the fields for which the user can enter selection criteria.

The **View Summary** button allows users to see the field or fields for which they have entered selection criteria. The By Value and By Range tabs let users enter a single value to search for, or a range of values. Entering a single value lets users choose to search for an exact match, a record that starts with the search value, or a record that contains the search value anywhere in the field. Users can also select a case-sensitive search.

Wait — There's More

InfoPower 2.0 includes three other new components:

1) The *TwwIntl* component is a non-visual control that allows easy internationalizing of applications that use InfoPower. It provides a central location where you can change the captions, hints, and button styles used by all of InfoPower's built-in, end-user dialog boxes.
2) The *TwwDBSpinEdit* lets users easily increment or decrement the value in a numeric or date field using the mouse or the up- and down-arrow keys. You can set the

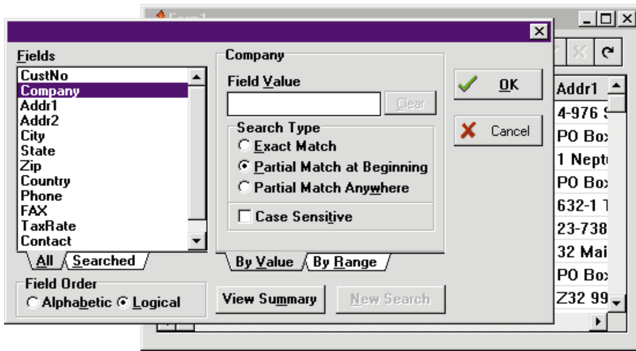


Figure 5: Calling *Execute* displays this dialog box.

minimum, maximum, and increment values. Also, you can use this control without binding it to a data source.

3) *TwwDBEdit* is a replacement for Delphi's *TDBEdit* control. The *TwwDBEdit* provides support for InfoPower picture masks. In addition, it automatically detects date fields and lets users enter the current date by pressing Spacebar.

An Even More Powerful Grid

One of the stars of InfoPower 1.0 was the *TwwDBGrid* component, which provided many features not found in the Delphi *TDBGrid* component. InfoPower 2.0 enhances the grid by letting you embed a checkbox, combo box, spin edit, bitmap, lookup combo box, or custom edit box in a field of the grid. Handling long text strings is now easier because you can scale a cell to double or triple its normal height and word-wrap text in the cell.

Picture edit masks are fully supported in a grid, as is selecting multiple records. You can also cause Enter to behave as Tab and move focus to the next cell. Another great enhancement is the *PerfectFit* property. If set to *True*, it automatically sizes the grid so that no space is left between the last row and the bottom of the grid.

Lookup Combo Power for Client/Server Developers

The use of lookup combo boxes presented performance problems for client/server developers because the lookup dataset had to be a Table component; as a result, opening each table and filling the BDE cache could go slowly. The only solution was to load a combo box from a query's result set. InfoPower 2.0 solves this problem by letting you use a SQL or QBE query as the source for the lookup data; you don't have to write a single line of code.

The new *TwwDBLookupCombo* not only can be placed in a cell in the InfoPower grid, but can also be placed in the new *TDBCtrlGrid* in Delphi 2. Also, you can use a detail table as the lookup source, and multiple lookup combo components can now be attached to a single lookup table.

No More Codes

Some of the most annoying things to deal with when displaying data in a grid are fields that contain codes such as

An InfoPower Primer

For those who haven't had the pleasure, here's a short rundown of existing features in InfoPower version 1.0. All are available to users of Delphi 1 or 2.

A Table component that fully supports Borland Database Engine (BDE) filters. This includes the ability to change the filter criteria on-the-fly at run time; a *Pack* method for both Paradox and dBASE tables; and a *wwFindKey* method that works faster with SQL tables than the Delphi 1 *FindKey* method.

A QBE component that fully supports QBE queries. This includes an *AnswerTable* property to let you easily save your result set to disk, as well as an *AuxiliaryTables* property so you can have the query create Paradox-style KeyViol, Changed, Inserted, and Deleted tables in the user's private directory.

An enhanced data-aware grid component. This lets you:

- display a cell as a checkbox, combo box, lookup combo box, or custom dialog box,
- display the text of a memo field in the grid,
- double-click a memo field in the grid and display a pop-up memo editor, and
- display multiple tables in a single grid and define non-scrollable columns in the grid.

High-performance search controls. These include incremental, exact match, "starts with," and sub-string searching.

A customizable pop-up memo field editor. This is used to edit memo fields displayed in any control.

A DBComboBox component. This is equivalent to the Delphi DBComboBox component except that Tab, Shift+Tab, Enter, and Esc work as expected.

A DBComboDialog component. This features an ellipsis button and an event triggered when the user clicks the button, allowing you to display custom dialog boxes to help users edit a field in a table.

A DBLookupCombo component. This lets you:

- display any number of fields in the drop-down list,
- display column separators in the drop-down list,
- display column headings in the drop-down list,
- control whether the drop-down list grows to the left or to the right,
- control which column sorts the drop-down list, and
- allow users to incrementally search the drop-down list by typing — in the field display — a description instead of a code, even though the code is stored in the table.

You can use this component without binding it to a data table.

A DBLookupComboDialog component. This includes all the features of the DBLookupCombo component except that, instead of a drop-down list, this control displays a dialog box with the lookup table displayed in a customizable grid.

— *Bill Todd*

part numbers, job codes, and customer numbers. In Delphi 1, the only solution is to create calculated fields, then write code to look up the values and display them next to the codes in the grid.

Unfortunately, this does not help users who must enter data and may not remember the codes. The enhanced *TwwDBComboBox* comes to the rescue by allowing you to display a description instead of the code in both the field itself and in the drop-down list, while still storing the code in the underlying table.

TwwDBLookupCombo has done this since version 1.0 in instances when the lookup data is stored in a table; but now you have the option to show descriptions instead of codes when the lookup information is not in a table.

Other Enhancements

If you deal with memo fields, one of the handiest components in InfoPower is the *TwwMemoDlg*. This provides a pop-up memo field editor that you can call from your code at any time to let users edit the text in a memo field. Now you can attach additional buttons to the memo editor dialog box to call a spell checker or perform any other function you need.

The *TwwLocateDialog* component, which lets users search for a value in any field of a dataset, now allows users to search on calculated and lookup fields.

Conclusion

If you develop database applications and don't use InfoPower, you're making your life a lot harder than it needs to be. Not only does InfoPower offer a wide array of features unavailable in any version of Delphi, it also offers the same feature set for both Delphi 1 and 2. This makes maintaining a common code base for both the Win16 and Win32 platforms much easier. ∆

INFORMANT FACT FILE

InfoPower 2.0 includes, among others, these welcome additions to Delphi: Paradox-style picture masks, simplified dataset filtering and querying, and an even more powerful data-aware grid component. If you're a database developer, InfoPower 2.0 is a must-have tool.

Woll2Woll Software
2217 Rhone Dr.,
Livermore, CA 94550
Phone: (800) 965-2965 or (510) 371-1663
Fax: (510) 371-1664
E-Mail: Internet: woll2woll@woll2woll.com or CIS: 76207,2541
Web Site: http://www.woll2woll.com
Price: InfoPower 2.0, US$199; source code is available for an additional US$99. Upgrade from previous version, US$99.

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

NEW & USED

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

SysTools for Delphi

A Low-Level Delphi Supermarket

Most third-party Delphi products fall into one of two categories: component libraries or utilities. The first is self-explanatory; the latter helps programmers develop, debug, and profile applications, and create custom components.

SysTools for Delphi is different; it contains no components or utilities. It is however, a large library of low-level routines that can help with the inner workings of applications. Produced by TurboPower Software Company, SysTools is based on an earlier TurboPower product, Win/Sys Library for Borland Pascal and C++. While most tools in the earlier library are included, some differences exist (see Figure 1).

A Comparison of Win/Sys and SysTools

In transforming the Win/Sys Library into SysTools for Delphi, TurboPower had to decide whether to provide full reverse compatibility with Win/Sys or take full advantage of Delphi's new features. TurboPower opted for the latter, writing SysTools almost from scratch in Delphi (with a significant amount of assembly code). SysTools works with either Delphi 1 or 2. While many of its methods and properties have the same names and parameters as in Win/Sys Library, some do not.

SysTools works with either 16- or 32-bit operating systems. In 16-bit programs, most SysTools routines are similar to those in Win/Sys; however, this isn't the case with 32-bit programs. Some Win/Sys units aren't available, including WsTimer (timing and delay procedures), WsDPMI (DPMI access), and WsHeap (Heap analysis). TurboPower also has a new utility, Memory Sleuth (available separately), that includes tools to analyze and report various problems such as memory leaks and resource allocation errors in 32-bit programs.

The classes comprising the SysTools library can be divided into three groups: string and numerical data manipulation, container classes, and a low-level interface to the Windows operating system. With the exception of the container classes (which are discussed in one chapter), each class is explained in its own chapter of the manual which defines all the key properties and methods using clear examples. Each chapter also includes a sample application demonstrating its features and capabilities. Now we'll look at the features, uses, and limitations of the main class.

Purpose of Unit	Win/Sys Library	SysTools
String Manipulation	WSString	STString STStr? (S/L/Z)
Date/Time Manipulation	WSDate	STDate
Low Level Operating System Routines	WSInLine WSDos	STUtils
Bit Set Manipulation	WSBitSet	STBits
String Dictionary	WSPchDct	STDict
Double Ended Queue	WSQueue	STDQue
Large Array and Large Matrix (2 dim)	WSLarray	STLarr STMatrix
Collection Classes	WSColl	STColl
Linked List	WSList	STList
Virtual Array	(not implemented)	STVarr
Tree Class	WSTree	STTree
Sorting Engine	WSSort	STSort
Timer	WSTimer	(not implemented)
BCD Arithmetic Methods	(not implemented)	STBCD
Registry/INI File Access	(not implemented)	STRegIni

Figure 1: Differences between the tools in Win/Sys Library and SysTools for Delphi.

String Methods

SysTools fully supports the three types of strings used in Delphi 2: length-byte (old-style Pascal), null-terminated, and ANSI strings. Each string-type has a separate unit (i.e. STStrS, STStrZ, and STStrL) with all methods duplicated in each unit. There are routines to parse and manipulate filenames such as *AddBackSlash*, *DefaultExtention*, and *JustName*; and string parsing/formatting routines to count, wrap, and locate words (a substring surrounded by delimiters) in a string.

The Boyer-Moore algorithm, a powerful search method, is fully implemented; searches can be case sensitive or insensitive. SysTools' Soundex algorithm enables you to test for words that sound alike. Also included are text formatting primitives to trim blanks, center text, and so on. When you combine these methods with those in Delphi, the string-handling tools are well represented.

Date/Time Routines

If you own or have seen TurboPower's component library, Orpheus, you're probably familiar with many of the date and time manipulation methods. SysTools and Orpheus share a common unit, STDate, which includes the basic date and time manipulation tools while using additional, autonomous units to handle international issues and date/time string manipulation. The unit provides support for Julian dates (a compact numeric representation used for any date from 1/1/1600 to 12/31/3999). Dates can be represented in a variety of ways using picture masks such as `hh:mm:ss`. There are also useful date/time testing functions, including *ValidDate* and *IsLeapYear*, along with many conversion methods such as *DayOfWeek*, *MonthToString*, *TimeStringToSTTime*, and *RoundToNearestMinute*. Many contain error checking. *InternationalDate* and *InternationalLongDate* are useful functions that return the appropriate picture masks as stored in the `[intl]` section of WIN.INI.

BCD Math

Binary Coded Data (BCD) is a high-precision, floating-point class useful in financial or accounting applications. As explained in the manual, BCD was "originally defined in Turbo Pascal 3.0 [as] an array of bytes providing 18 significant digit accuracy and a range of 1E-63 to 1E+63." Both the original and SysTools' BCD implementation store amounts directly in the familiar base, avoiding conversion to and from binary floating point types. Thus, the class is intuitive to work with. This new BCD class can have as many as 36 significant digits and range from 1E-64 to slightly less than 1E+64. Most of the math functions required for financial applications are included (up to power functions); however some of those that might be needed in scientific applications (trigonometric sine, cos, etc.) are not.

The BCD class includes methods to convert to and from Delphi strings, integers, and real types. Reverse compatibility with TurboPower's Object Professional (DOS Turbo Pascal) Library is also provided, as long as the latter's transcendental functions are not used. A BCD Calculator application is included as an example (see Figure 2).

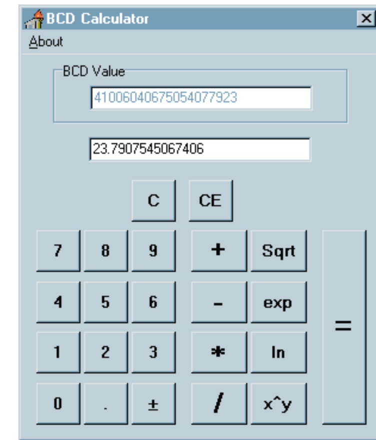


Figure 2: Accepting input in decimal format, SysTools' BCD Calculator shows the result in BCD's internal format.

Container Classes

All SysTools' container classes descend from *TPersistent*, so all are streamable. Overriding *TPersistent's Assign* method, many of the classes provide a means to convert data to another container class, including *TSTList*, *TSTDQue*, *TSTLArray*, *TSTLMatrix*, and the two collections. As you would expect, methods to read from and write to files or streams of different types are included.

The container classes fall into two general groups: those based on pointers, such as linked lists, trees, and collections; and those based on untyped variables (i.e. arrays). Therefore you can store any kind of data in each. You can also adjust data element size and the number of elements at run time. All of these classes descend from/or use one or two base classes, *TSTContainer* and sometimes *TSTNode*. You can use these classes to derive your own container classes.

As with many other classes in this library, all the container classes are "thread-safe." In other words, if you want to use them in Delphi 2 applications which take advantage of Win32's multi-tasking capabilities, a safety net is provided. This safety net includes critical sections, which must finish their work before another thread can become active. Among other scenarios, this prevents two threads from attempting to read from and write to the same data within too close a time-frame, potentially corrupting that data. Altogether there are 10 container classes. Let's take a brief look at each.

Bit Sets, Lists, and Queues

If you're writing an application that uses a large number of Boolean switches, and/or memory is a critical factor, the *TSTBits* class may fulfill your requirements. Its resizable bit sets can be set, cleared, toggled, or tested. Both of the list classes, *TSTList* and *TSTDQue*, provide methods for adding, deleting, moving, or testing for items. Among other unusual capabilities, you can step through (iterate) a list and join or separate lists.

Arrays of Global Proportions: Sorted and Unsorted Collections

Using the Windows global heap, SysTools' two large array classes expand beyond the dimensional limits of Delphi's arrays (particularly in 16-bit mode.) There is, however, a caveat in 32-bit mode: operations involving large arrays will be slower than Delphi's native arrays, despite the former's optimization. While no separate class for 3D arrays is included, an example program

is included that demonstrates how to create one by having each element of a large array point to a large matrix. An example program employing this technique is shown in Figure 3.

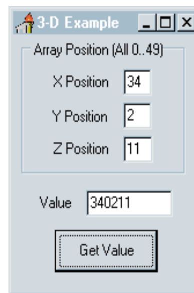


Figure 3: While not supported directly, you can create 3D arrays with SysTools.

SysTools' two collections are sparse arrays of pointers in which most of the pointers are assumed to be empty. When you instantiate a collection, you set a page size that determines the initial size of the collection. During this process you are faced with the speed versus size dilemma. Large pages enable faster access while small pages are bulkier, becoming closer in character to linked lists.

Much of the functionality of the Borland Pascal 7 *TCollection* is included in these classes with the exception of the *FirstThat* and *LastThat* functions. Interestingly, these methods are also absent from the new incarnation of *TCollection* in Delphi 2. These collection classes include new methods to test the efficiency of a collection, to pack it, to iterate through it, and to clear it with a single call. The familiar methods *At*, *AtInsert*, and *IndexOf* are present, along with the *Count* and *Items* properties.

A Dictionary and a Tree

TSTDictionary is a string dictionary, and is not all that different from a dictionary on your book shelf. You look up objects by referring to their key strings, analogous to the terms or words in a language dictionary. The objects themselves can be anything: strings, data records, or numerical data. The class uses hashing algorithms to search for data and is particularly efficient when all the data fits into RAM. *TSTDictionary* shares many basic methods with other container classes including add, delete, clear, find, and iterate. While you can join two string dictionaries in this class and in *TSTList*, you can join — but not split — a single string dictionary into two (as you can with *TSTList*).

TSTTree provides a balanced, binary search tree. Its size is only limited by available memory. The documentation emphasizes that *TSTTree* includes capabilities of both a string dictionary and the sorting engine (which we'll discuss soon). The nodes of the tree are stored in sorted order; order and balance are maintained during inserts and deletes. Figure 4 shows a sample application that demonstrates the capabilities of the *TSTTree* class. Having discussed each container class in some detail, let's look at SysTools' Sorting Engine.

The Sorting Engine and ASCII Text File Handling

SysTools' sorting unit can be applied to a variety of data types, including records and arrays. It uses a non-recursive quicksort algorithm, along with some characteristics of a merge sort. As with the container classes, it's thread-safe. It can theoretically hold up to two billion elements; however, the total size used by the elements is limited to available memory. A sample sorting application creates a list of random strings, enabling you to sort them (see Figure 5).

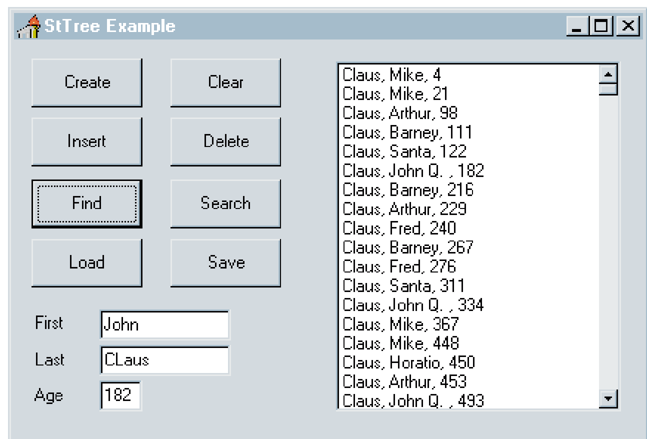


Figure 4: SysTools' *TSTTree* class has many inserting, deleting, and searching capabilities.

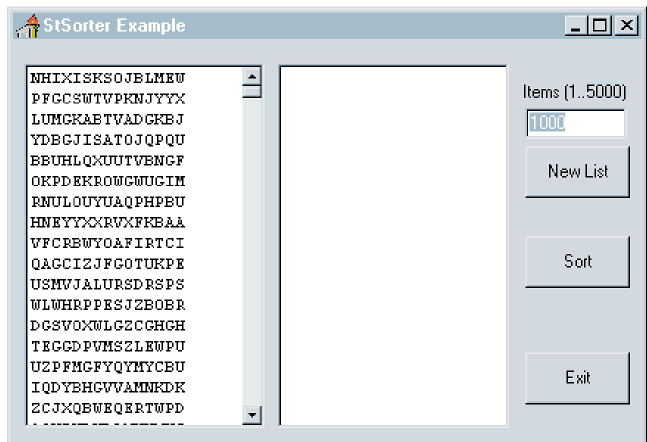


Figure 5: The sample sorting application with randomly generated strings before being sorted.

Now let's discuss input and output of text files. The STText unit provides a high-level interface to ASCII (text) files. The four procedures and functions are *TextFileSize*, *TextFlush*, *TextPos*, and *TextSeek*. These provide a convenient set of methods to facilitate working with text files.

Windows Operating System Interface

In the operating system domain, SysTools has considerably fewer procedures and functions than Win/Sys. TurboPower omitted some because they didn't apply to all the operating systems on which SysTools can be used. However, new routines were added to provide better access to the operating system.

The STUtils unit includes routines for setting, filling, clearing, testing, or exchanging values in various data types, including *SetLongFlag* (sets one or more bits in the parameter *Flags*) and *FillStruct* (fills a given region of memory with values of a specified size, i.e. not just byte-size as in *FillChar*). Others include *ClearByteFlag*, *LongFlagIsSet*, *ExchangeLongInts*, and *SetMediaID*.

Low-level file handling routines are included to retrieve the number of file handles remaining; read, write, or delete volume labels; and check valid drive or directory status. In the process of conversion, many Win/Sys routines were not implemented, including those relating to DOS. Other functions have been superseded by the func-

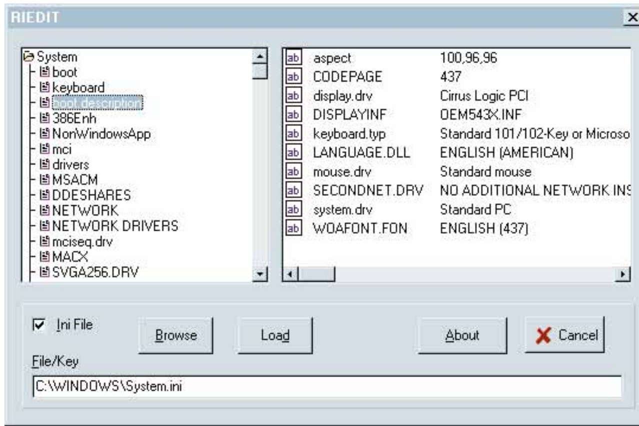


Figure 6: The example program, RIEdit, allows you to view and edit .INI files or the Windows Registry.

tionality of Delphi components (e.g. Number of Drives by the *TDriveCombo* component). Win/Sys' *ExistFile* function now has a Delphi equivalent (*FileExists* in the SysUtils unit); and various methods such as *ExtractFileExt* and *ExtractFileName* functions have been replaced by Delphi's filename parsing functions.

Registry and .INI Files

The Windows Registry (.REG) and .INI files are used in Windows 3.x, Windows 95, and Windows NT. While .INI files were the preferred way to store information in Windows 3.x, the Registry has assumed this role in Windows 95.

Since .INI files are ASCII text files, they are easy to edit. .REG files, which are binary, pose more problems. However, SysTools' *TRegIni* class provides all the tools necessary to write to or read from .REG and .INI files. You can use practically the same code to work with both types. The example application for this unit, RIEdit, mimics the Windows 95 RegEdit utility. It transparently opens and shows either the Registry or a particular .INI file (see Figure 6). Data types you can write to or read from a .REG or .INI file include strings, Boolean, datetime types, integers, floating point, and so on.

A configuration example. To demonstrate the power of this class, I've written a small demonstration program that sets a few configuration options after testing for a serial number. It consists of a project file and four unit files for each of the four forms: CGFTest, SerialDg (a Serial Number testing dialog box), EntDlg (Entry Dialog Box), and ShowCFG (which allows both viewing and editing of existing .REG or .INI files). Admittedly, the program is much simpler than SysTools' example program. However, it demonstrates the power of these tools.

Following is the method that writes to a .REG or .INI file. *CFG* is the variable that refers to the file being written to. *InstallPath*, *UserName*, *CreateTime*, and *ProgOptEnabled* hold the application's configuration data. Note the compiler directives that enable the appropriate changes for Delphi 1 or 2 (see Figure 7).

```
procedure CreateConfigEntry;
begin
  CFG.Free;
  CFG := nil;
  if not DirectoryExists(ConfigTest.InstallPath) then
    MkDir(ConfigTest.InstallPath);
{$IfDef Win32}
  CFG := TSTRegIni.Create(RICUser, False);
  CFG.CurSubKey := 'Software';
  CFG.CreateKey('MyApp');
{$Else}
  if FileExists('C:\Windows\MyApp.Ini') then
    DeleteFile('C:\Windows\MyApp.Ini');
  CFG := TSTRegIni.Create('C:\Windows\MyApp.Ini',True);
{$EndIf}
  CFG.CreateKey('User Info');
  CFG.CurSubKey := 'User Info';
  CFG.WriteString('User', ConfigTest.UserName);
  CFG.WriteDateTime('Config_Changed',
                    ConfigTest.CreateTime);
  CFG.CreateKey('Program Options');
  CFG.CurSubKey := 'Program Options';
  CFG.WriteString('Prog_Directory',
                  ConfigTest.InstallPath);
  CFG.WriteBoolean('Program Option Enabled',
                   ConfigTest.ProgOptEnabled);
  CFG.Free;
end;
```

Figure 7: *InstallPath*, *UserName*, *CreateTime*, and *ProgOptEnabled* hold the application's configuration data.

Assembly Code in Delphi

One of the bonuses in SysTools is a large collection of routines demonstrating the different ways to use assembly code. Two units have the bulk of ASM code. The STBase unit contains a number of strict assembler functions. The other unit, STUtils, includes assembly code within Delphi methods and functions. There is even inline code for some of the 16-bit primitives (inline is no longer supported in Delphi 2, but was used in Delphi 1). Let's look at two examples.

At the start of the STUtils unit is a series of 16-bit procedures and functions written in inline code for Delphi 1. Most are low level. How low? Here is the 16-bit version of *MakeWord*, a function to construct a word out of two bytes:

```
function MakeWord(H, L : Byte) : Word;
{ Construct a word from two bytes }
{$IFNDEF OS32}
  inline(
    $58/       { pop ax    ;low byte into AL }
    $5B/       { pop bx    ;high byte into BL }
    $88/$DC);  { mov ah,bl ;high byte into AH }
{$ENDIF}
```

Now let's take a look at the 32-bit version for Delphi 2:

```
function MakeWord(H, L : Byte) : Word;
begin
  Result := (Word(H) shl 8) or L;
end;
```

Among other things, this comparison demonstrates the optimization built into Delphi 2. It's no longer necessary to write as much assembly code to optimize. Few of the 32-bit examples contain assembly code. The Delphi 2 version of the *WriteVolumeLabel* function is lean compared to Delphi 1:

```
function WriteVolumeLabel (const VolName: string;
  Drive: AnsiChar) : Cardinal;
const
  RootMask = 'x:\';
var
  Temp : string;
  Root : string;
begin
  Temp := VolName;
  Root := Drive + ':\';
  if Length(Temp) > 11 then
    SetLength(Temp, 11);
  if Windows.SetVolumeLabel(PAnsiChar(Root),
                            PAnsiChar(Temp)) then
    Result := 0
  else
    Result := GetLastError;
end;
```

While this example contains five lines of code, the 16-bit version contains 16 lines of Pascal code and 46 lines of assembly code (see Listing Five on page 54). With these excellent examples of source code, this library is a great resource and learning tool.

The TurboPower Legacy

Typical of the written documentation accompanying other TurboPower libraries, the SysTools manual is excellent: all key properties and methods are fully explained; carefully selected examples of code are included in each chapter (concentrating on the issues raised by the classes themselves); and many insights into Windows programming in general, and 32-bit development in particular, are provided.

As mentioned earlier, the company's tradition of including full source code at no additional cost continues. In addition to the many example programs, TurboPower has included the console programs used to test these classes. Known problems are always articulated, and excellent support is provided via e-mail, online, and phone. You can also download minor upgrades free of charge. Having used TurboPower tools for the past 10 years, I can state with conviction that SysTools for Delphi continues the TurboPower tradition. I wish every programming tool vendor could reach this level of excellence.

Conclusion

SysTools for Delphi is a truly multi-faceted library. Compared to its predecessor, Win/Sys Library, it's a major step forward, as Delphi was from Turbo Pascal. One of the more important new features in SysTools is its ability to handle the three types of strings in Delphi: length-byte, ANSI string (introduced in Delphi 2), and null-terminated. It has an impressive set of container classes, a marvelous sorting engine, a suite of low-level tools, and a BCD math class. It also provides a class that can directly read from and write to any .INI file or the Windows Registry (16- and 32-bit). Best of all, SysTools is compatible with both 16- and 32-bit programs compiled in Delphi 1 and 2 respectively. No serious Delphi programmer should be without this marvelous library. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\JAN\DI9701AM.

INFORMANT FACT FILE

SysTools for Delphi is a large library of Delphi classes for manipulating strings of all kinds, 10 container classes, and low-level routines to interface the Windows operating environment, including the Registry. It includes example programs, full source code, and additional console test programs. The manual is well organized, informative, and very well written. SysTools is an excellent collection of Delphi building tools and an excellent learning resource.

TurboPower Software Company
P.O. Box 49009
Colorado Springs, CO 80949
Phone: (800) 333-4160 or (719) 260-9136
Fax: (719) 260-7151
E-Mail: Internet: info@tpower.com or CIS: 76004,2611
Web Site: http://www.tpower.com
Price: SysTools for Delphi, US$149; upgrade from Win/Sys Library, US$59; upgrade from any other TurboPower product, US$119.

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.

Begin Listing Five — STUTILS.PAS 1.00

From SysTools for Delphi, Copyright (c) TurboPower Software, Co. 1996

```
{ 16-bit version of function for Delphi 1 }
function WriteVolumeLabel(const VolName : string;
  Drive : AnsiChar) : Cardinal;
var
  XFCB : XFCBRec;
  DTA : DTABuf;
  SaveDTA : Pointer;
  MediaID : MediaIDType;
  DriveNo : Byte;
  VName : array[0..10] of AnsiChar;
  MediaBlockExists : Boolean;
  ErrorCode : Byte;
begin
  SaveDTA := GetDTA;
  SetDTA(DTA);
  FillChar(XFCB, SizeOf(XFCB), 0);
  FillChar(XFCB.FileSpec, 11, '?');
  XFCB.Flag        := $FF;
  XFCB.AttrByte    := 8;
  DriveNo          := Ord(UpCase(Drive)) - (Ord('A') - 1);
  XFCB.DriveCode   := DriveNo;
  MediaBlockExists := (GetMediaID(Drive,MediaID) = 0);
  FillChar(VName, SizeOf(VName), ' ');
  Move(VolName[1],VName,
       MinWord(Length(VolName),SizeOf(VName)));
  asm
    push ds
    mov  ax,ss
    mov  ds,ax
    mov  es,ax
    lea  dx,XFCB
    mov  ah,11h
    int  21h
    or   al,al
    jnz  @@NoLabel

    lea  di,DTA
    mov  dx,di
    add  di,18h
    cld
    mov  cx,ss
    mov  ds,cx
    lea  si,VName
    mov  cx,11
    rep  movsb
    mov  ax,ss
    mov  ds,ax
    mov  ah,17h
    int  21h
    mov  ErrorCode,al
    jmp  @@ExitPoint

  @@NoLabel:
    lea  di,XFCB
    push ss
    push di
    add  di,8
    mov  cx,ss
    mov  ds,cx
    mov  es,cx
    lea  si,VName
    cld
    mov  cx,11
    rep  movsb


    call DosFCBCreate
    mov  ErrorCode,al
    cmp  al,0
    ja   @@ExitPoint

    lea  di,XFCB
    push ss
    push di
    call DosFCBClose
    mov  ErrorCode,al

  @@ExitPoint:
    pop  ds
  end;

  if MediaBlockExists and (ErrorCode = 0) then
    begin
      Move(VName[0], MediaID.VolumeLabel,
           SizeOf(MediaID.VolumeLabel));
      ErrorCode := SetMediaID(Drive, MediaID);
    end;

  SetDTA(SaveDTA^);
  Result := ErrorCode;
end;
```

End Listing Five

Business Objects: Moving beyond Vaporware?

The term business object has a certain ethereal quality to it. Although touted for years by object-oriented (OO) proponents, questioning its utility isn't necessarily wrong. Most developers have found it hard to sink their teeth into business objects and use them in real-life applications. Last year, I talked about emerging trends in object-oriented programming. This month, we'll take a closer look at one of those trends — business objects — and determine to what extent you can take advantage of them in Delphi.

What are business objects? Perhaps the simplest way to understand what business objects are is to contrast them with what I call *system objects*. System objects are reusable components that can be used across a variety of business applications. The *TTable* component, along with all other components in the Delphi Visual Component Library, is a good example of a system object. In contrast, business objects encapsulate business logic specific to a business entity (such as a customer, student, or invoice) or a process (such as a purchase or ATM withdrawal). System objects are *horizontal*, whereas business objects are — by their nature — vertical creatures, specific to a particular line of business, e.g. a business object developed for a bank probably won't have tangible benefits for an HMO.

Why use business objects? When developing applications, the greatest amount of energy is usually spent on the business-specific parts of the application. Given this investment, it's clear that business rules are the heart of most applications. Thus, encapsulating these rules into business objects has several advantages:

Business objects provide the ability to build a reusable component architecture. Reuse is touted as one of the fundamental motivations behind OO methodologies. As with any Delphi component you create, a well-built business object can be reused in multiple applications. Therefore, after a company has developed business objects, it should be able to gain a return on the investment in future applications.

Business objects simplify the process of making rule changes. After developing a user interface component (such as an edit box), how often will you need to make changes to it? While some changes are expected, it's unlikely that many properties or methods of this visual component will need to be modified or added. Contrast this edit box with a typical business object. As the companies these business objects imitate are constantly changing, there will be an ongoing need to make changes to a business object. Because of encapsulation, a properly designed business object can be modified rather quickly, without causing a ripple effect across an application — as long as its external interface remains constant.

Business objects separate business rules from the rest of the application. In client/server systems, a fundamental issue has always been determining where the business rules of the application should be stored: within the client's user interface, or as stored procedures on the server. From an OO standpoint, the solution is to partition the application, adding a business logic layer between the presentation and database levels. Such an application architecture becomes more extensible and easier to maintain.

Can you use business objects in Delphi? Delphi features an object-oriented language, so the ability to create business object classes is not an issue. However, objects instantiated at run time are stored only in memory; all information related to an object is lost when you close an application. Thus, in order for business objects to be useful, you must be able to store them persistently between sessions. Unfortunately, out of the box, Delphi has little to offer when it comes to object storage. Because Delphi is primarily targeted as a client/server development tool, its database access is focused exclusively on being a good client for back-end relational database systems. Also, Delphi data classes don't support persistent storage of objects, so you really have no choice but to do it on your own. In an upcoming *File | New*, we'll look at the four options you have for creating a custom business object storage solution.

Are you using business objects in Delphi? If so, contact me at rwagner@acadians.com and let me know how you're using them. I'll compile this information and share it with all of you. ∆

— Richard Wagner

Richard Wagner is the Chief Technology Officer of Acadia Software in the Boston, MA area. He welcomes your comments at rwagner@acadians.com or on the File | New home page at http://www.acadians.com/filenew.htm.