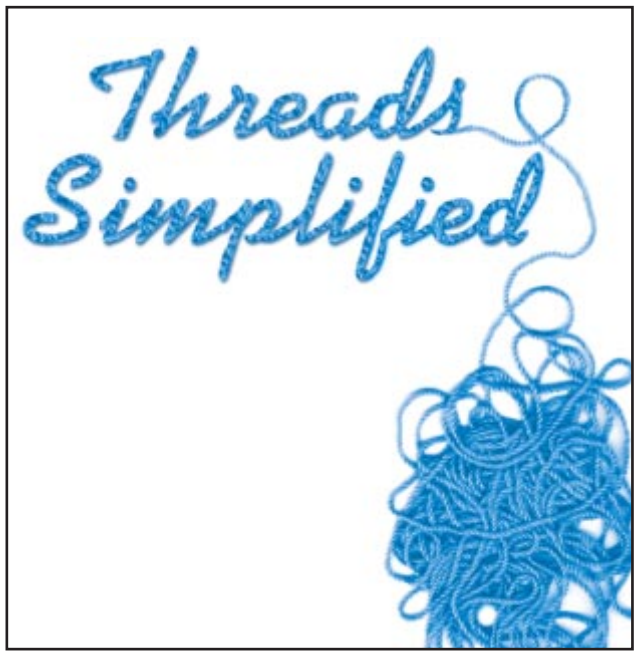
December 1997, Volume 3, Number 12

Threads Simplified

Untangling a Gnarly Topic



Cover Art By: *Tom McKeith*

ON THE COVER

FEATURES

REVIEWS

DEPARTMENTS

Project JEDI

To a large extent, Windows is built on a series of dynamic-link libraries (DLLs), including old workhorses like USER.DLL and GUI.DLL. But Windows doesn't stand still. Even as work is proceeding on the successor to Windows 95, new functionality is being introduced. Microsoft makes this functionality available to programmers through various application programming interfaces (APIs) to these DLLs. Generally, these are presented in various software developer kits (SDKs).

Unfortunately, most of this new functionality is initially available only in C, and must be converted to Pascal for use by Delphi programmers. While Borland has done its best, the period from the initial appearance of a new API/SDK to its translation into Delphi seems interminable for many developers, particularly those who must stay on the cutting edge of new technologies.

It's time for a revolution. Great revolutions can begin with a small complaint — particularly if that complaint resonates with a larger group. What about the JEDI movement?

In the Beginning ...

It was a quiet Friday at the end of September, and I was engaged in a familiar activity: reading through posts to the COBB DDJ-Thread (*Delphi Developers Journal*), which I help moderate. I didn't pay much attention to the initial message that spoke of the long waiting period until a new API becomes available in Delphi, but soon quite a few others joined in and echoed the concerns. One or two spoke of deserting Delphi for C++ to have quicker access, suggesting that Borland needed to wake up, etc.

Then the focus of the thread turned to what we could do. At that point I had to jump in. I suggested that it was "pretty much up to us, as developers, to fill in these gaps ... Perhaps if more of us would join in, and then make the translations ... available on the Web, we could solve the problem ourselves." Others agreed this was the way to go. As a result, the DDJ-Thread was pretty much taken over for the weekend by a zealous group of developers determined to do something about the problem.

At the end of the weekend it became clear that we would need our own list server. Someone involved in the discussion volunteered to set the whole thing up. Another person, Tim Hayes, emerged as the main coordinator. He kept everyone informed, provided daily updates to the whole group, and was eventually elected to head the Administrative Group (of which I too became a member). After much debate we agreed upon a name: JEDI (for Joint Endeavor of Delphi Innovators). There were other debates during the first week regarding the organization of the project, legal issues, Borland's possible involvement, and style guides for code and Help files.

On Saturday an IRC (Internet Relay Chat) channel was made available and a dozen or more JEDI enthusiasts joined in from all over the world. An organization was beginning to emerge; there would be a conversion team who would convert the C headers to Delphi and possibly develop classes and components, testers to catch the bugs, Help-file creators to provide the needed documentation, and Web-site developers to make sure the fruits of Project JEDI could be presented to the Delphi community in an exciting and usable fashion.

At this point (some ten days later) Project JEDI is still in its infancy. It could die or flourish. Its beginnings, however, remind us of some important lessons: no single company, not even Borland International, can provide everything we might need or desire in programming tools; developers can be a particularly resourceful and generous group who can accomplish great tasks when they put their minds to it; and finally, sometimes it does make sense to lodge a complaint!

On the Horizon

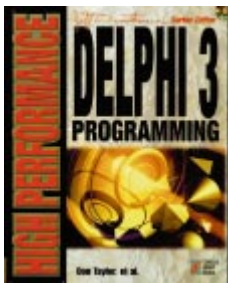
Next month we'll examine Delphi 3 packages. In March, we'll return to a further discussion of Delphi and the APIs. To find out more about Project JEDI, you can subscribe to the JEDI list-server by sending e-mail to listserv@pure-science.com with the message SUBSCRIBE JEDI in the main body. Or you can write me at acmdoc@aol.com. Δ

— Alan C. Moore, Ph.D.

Delphi TOOLS

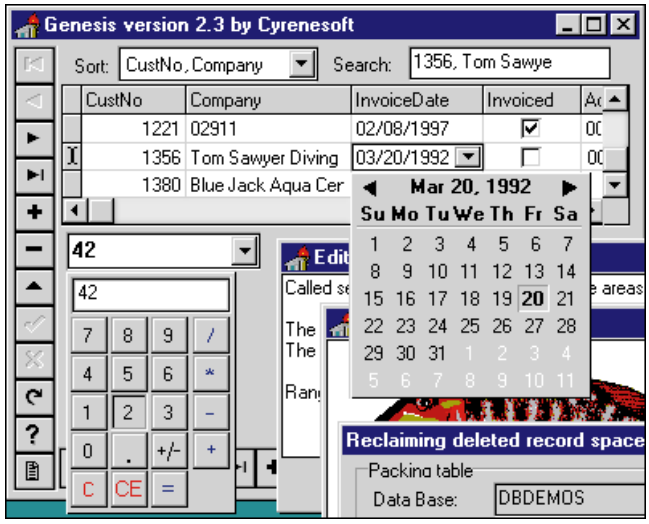
New Products and Solutions

High Performance Delphi 3 Programming
Don Taylor, et al.
Coriolis Group Books



ISBN: 1-57610-179-7
Price: US$49.99
(635 pages, CD-ROM)
Phone: (602) 483-0193

Cyrenesoft Announces Database Component Set for Delphi



Cyrenesoft has released *Genesis 2.3* for Delphi, a suite of database components such as the Data Doctor, enabling developers to program database applications faster and with greater ease.

Developers can drop the Data Doctor into their application, where it will automatically run at intervals. It automatically packs deleted records from tables and recreates indexes, saving disk space and increasing application-index performance.

Genesis also features components such as the Intelligrid component, a data-aware grid that allows developers to embed any data-aware component, even custom components — such as special combo boxes.

Working in conjunction with the Intelligrid is a sort-edit box and search-combo box that changes the sort order of any data-aware grid, and has a special feature that allows the end user to specify more than one key.

Genesis also features csNavigator, a replacement navigator component that can be oriented vertically or horizontally. csNavigator allows developers to change the glyphs, and has two added buttons that permit the end user to find a record through a dialog box and print the current record in an automatically formatted report.

Other components include csDateTimeCombo, a data-aware calendar that works with date and time fields; csCalculatorCombo, a drop-down calculator; several lookup combo boxes that correct bugs in the Delphi VCL; dialog components, including memo and graphic dialog boxes; and more. Both data-aware and non data-aware versions of several components are included.

Cyrenesoft
Price: US$149 for source code; US$99 for DCU; US$45 for mini-packs.
Phone: (770) 838-0404
Web Site: http://www.cyrenesoft.com

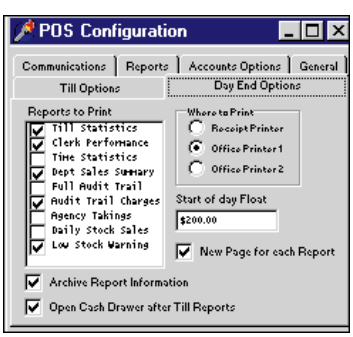
Parity Software Announces ChatterBocx/SAPI

Parity Software announced *ChatterBocx/SAPI*, an ActiveX control that can be used in Delphi, Visual C++, and Visual Basic. Text-to-speech (TTS) technologies based on Microsoft's Speech API are also supported.

ChatterBocx/SAPI allows users to build telephony applications based on VoiceBocx control and desktop applications that use multimedia loudspeakers for speech output. It synthesizes speech from text presented as a string of ASCII characters, either in a text file or from a string passed as an argument from a user's program code. In a telephony application, the speech is generated on a telephone line and heard by the human caller; in a desktop application, the speech is generated through loudspeakers.

Parity Software
Price: Call for pricing.
Phone: (415) 332-5656
Web Site: http://www.paritysw.com

Mountaintop Systems Releases Remember All Suite Comprising



Mountaintop Systems has released *Remember All Suite Comprising*, which includes two components aimed at simplifying the entry and storage of user-preference and program-Configuration information.

TRememPanel, a descendant of *TPanel*, allows for the quick preparation for editing and automatic storage of a large range of user-preference options. It will also hold a PageControl or tabbed notebook, which in turn holds other setup recording controls (edit boxes, checkboxes, etc.). *TRememPanel* automatically saves changes made by the user to an .INI file, restoring them at startup. Multiple configurations saved to different .INI files are handled with a single line of code.

The EditFields component adds flexibility by allowing users to enter text, dates, times, integers, floats, currency, etc., automatically validating and storing their entries.

Mountaintop Systems
Price: US$38; US$65 with source code.
Phone: 011 61 2 9541 1348
Web Site: http://www.ozemail.com.-au/~mtntop

Delphi TOOLS

New Products and Solutions

Digital Zinnia Studios and Modern Medium Inc. Announce Conduit

Digital Zinnia Studios and **Modern Medium Inc.** announced ***Conduit***, a DLL that sends and receive files to a location on the Internet using FTP.

Using Conduit, developers can Internet-enable Windows applications such as Delphi and Microsoft Access, Excel, and Word. Additional features include automatic updates of Web content controlled from an application, integration of file communication on the Internet in an application, sharing of data over the Internet from within an application, distribution of data on the Internet controlled from an application, and creation of Windows-based order-entry systems or Internet sales systems. API functions provided by Conduit include sending, downloading, and removing Internet files, and getting a file list and adding and removing Internet directories. Conduit will communicate with servers running Windows NT, UNIX, Novell, or Mac OS.

Digital Zinnia Studios and **Modern Medium Inc.**
Price: Single-user license, US$295; 20-user license within one application, US$495; 50-user license within one application, US$795; unlimited users within one organization, US$995; unlimited distribution within one application, US$1,495.
Phone: (541) 343-4281
Web Site: http://www.modmed.com/-conduit/conduit.shtml

HyperAct Ships WebApp 1.0

HyperAct, Inc. released ***WebApp 1.0***, the framework for Web-server applications development using Delphi 2 and 3 and C++Builder.

Features in WebApp include server independence; automatic session-management; HTML template support; transparent, cookies-based, session-state management; and Delphi 3 WebModules compatibility.

WebApp ships with native support for Microsoft Internet Information Server (ISAPI), Microsoft Personal Web Server (ISAPI), Netscape FastTrack (NSAPI), O'Reilly WebSite and WebSite Pro (WSAPI), and CGI and Win-CGI interfaces, which will work with any Web server.

WebApp also ships with browser capabilities, an ad rotator, SMTP components, and data-aware HTML-generation components and functions.

HyperAct, Inc.
Price: Standard, US$199; Professional, US$495.
Phone: (515) 987-2910
Web Site: http://www.hyperact.com

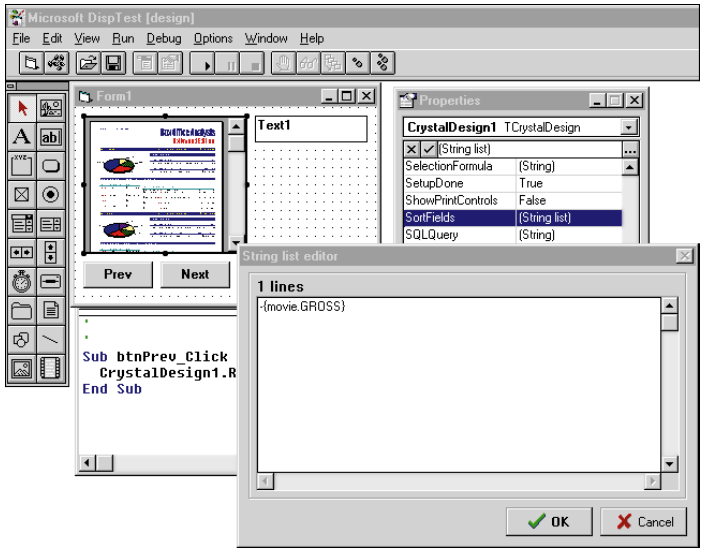
CoStar Introduces Developers Kit

CoStar Corp. introduced the ***Software Developers Kit*** (SDK), for the Easy Software Suite, that solves label-printing issues for application developers, including bar-code printing, label design, label formatting, and shrinking text to fit specific label sizes. Developers can run programs invisibly to provide printing support; open label files by name; print labels and control the number of copies to be printed; change a label's text, address, or bar-code contents; and specify image files to be printed and paste pictures into badge templates. The SDK includes software, application notes illustrating label-printing functions, a CoStar LabelWriter XL Plus, and a roll of address labels; it supports Delphi, Visual Basic, Access, and FoxPro. For more information, call (800) 426-7827 or visit http://www.costar.com.

SupraSoft Offers Crystal Reports Support

SupraSoft Ltd. released ***Crystal Design Component***, an ActiveX control allowing users of Crystal Reports to embed a report into an application.

Crystal Design Component allows users to specify custom report windows, add print-preview capabilities to an application, and provide international language localization features to a database report. The component doesn't restrict developers to prebuilt, report-preview interfaces; rather, they can create user interfaces using the standard user-interface building tools particular development environments provide. The component also provides properties to control and modify run-time report parameters, and options to ensure end users' ability to modify report parameters as intended.



Crystal Design Component is available in 16- and 32-bit Delphi, 32-bit OCX, and 16-bit VBX formats.

SupraSoft Ltd.
Price: US$149
Phone: 358 14 218 500
Web Site: http://www.-suprasoft.com

News LINE

December 1997

Borland, MicroEdge Announce Visual SlickEdit - Delphi Edition

Scotts Valley, CA — Borland and MicroEdge, Inc. announced the MicroEdge SlickEdit - Delphi Edition, an automated program editing solution for the Delphi 3 development environment.

Visual SlickEdit - Delphi Edition provides development tools for building enterprise-wide, client/server, or Web-enabled applications. Delphi developers have access to Visual SlickEdit's editing features without having to save or reload files when switching between the editor and Delphi; files are saved only when the user prompts Delphi to do so. Other features include procedure tagging, difference editing, selective display, and syntax expansion.

Visual SlickEdit - Delphi Edition is available for US$149 and includes licenses for Windows NT, Windows 95, and Windows 3.1. Delphi 1, 2, and 3 are supported. For more information, call (800) 934-3348 or visit http://www.slickedit.com.

Borland Strategies Announced at Microsoft Professional Developers Conference

San Diego, CA — At September's Microsoft Professional Developers Conference, Borland CTO Rick LeFaivre demonstrated Borland's support for key Microsoft distributed-computing technologies, such as the ActiveX Template Library (ATL) and DCOM (Distributed Component Object Model).

LeFaivre also highlighted the importance of the distributed computing model and outlined Borland's strategy for helping developers create multi-tier applications. These applications allow the transfer of business logic and configuration instructions to a middle tier, thereby providing easier, more cost-effective maintenance and support of applications.

In addition to ATL and DCOM, LeFaivre demonstrated Borland's support of Microsoft's Windows and systems technology, such as Microsoft Internet Information Server and Internet Explorer. Using Borland's C++Builder, Delphi 3, and MIDAS (Multi-Tier Distributed Application Services Suite), LeFaivre demonstrated how to build a distributed decision-support application, leveraging data stored in a Microsoft Access database, making this information available to users over the Web. LeFaivre also demonstrated how developers can build Java applications that connect to Microsoft SQL Server or Microsoft Access using Borland's new JBuilder pure Java development tool and Borland's DataGateway for Java middleware.

MCBA, Inc. Joins Borland's Partner/400 Program

San Antonio, TX — Borland announced that MCBA, Inc., an AS/400 solution provider and IBM hardware distributor, has joined the Borland Partner/-400 program to deliver client/server development expertise to the AS/400 community.

This puts MCBA in good position to provide clients with the next generation of hardware and software solutions for the IBM mid-range platform. Additionally, MCBA's experience in delivering AS/400 business solutions worldwide will allow Borland to continue to reach more corporations interested in building advanced client/server software applications.

For a complete list of Borland/400 partners visit http://www.borland.com/-borland400/partner.html.

Borland Announces Borland DataGateway for Java

San Antonio, TX — Borland announced Borland DataGateway for Java, a database connectivity middleware solution providing 100% Pure Java client access to corporate databases through industry-standard JDBC interfaces.

Borland DataGateway provides JDBC database access, using native drivers on a middle tier, to corporate information residing on Oracle, Sybase, Informix, Microsoft SQL Server, DB2, InterBase servers, and desktop databases and ODBC data sources. Features include a Zero-Configuration/Zero-Install Client, Pure Java/JDBC Client, and native database connectivity.

Borland DataGateway is available in JBuilder Client/Server Suite, and as a stand-alone solution for any JDBC-compliant Java application, including those developed with JBuilder Professional, Microsoft Visual J++, IBM VisualAge for Java, and Symantec Visual Café.

The Borland DataGateway Enterprise server is available for US$1,499. The Borland DataGateway Professional server supports dBASE, Paradox, Microsoft Access, and FoxPro, and is available for US$299.95. For more information, visit http://www.-borland.com/datagateway/.

By Jon Jacobs

Threads Simplified

Untangling a Gnarly Topic

A *thread* is the basic executable code sequence to which CPU time is allocated by the preemptive multitasking operating system (OS). Typically, each thread gets 20 milliseconds before the OS switches to another thread. In general, each thread has access to all the code, variables, and resources of the application (called a *process* in this context). Every process has at least one thread, but may have more. No thread "owns" any part of the application's code; several threads can execute the same code, as appropriate.

You may be wondering why you would want to use threads. If you're running applications on a 32-bit Windows platform, you're already using threads. Each process (application) has at least one thread vying for the CPU's attention. From this point on, I'll refer to the main thread of a given application simply as "the main thread." If you're using Delphi 2, 3, or another 32-bit compiler for a 32-bit Windows platform, you're already programming threads. So a better version of the previous question would be: "Why explicitly use any threads other than the main thread?" This is harder to answer, because there are many possible reasons. Here are a few:

- You want to change the priority of the thread.
- You want several sequences to run simultaneously and do not want to manage them yourself.
- You want the thread to continue running while you have a modal window open.

A War Story

I had a series of instructions controlling some equipment. The loop was started by clicking a **Start** button and stopped by clicking a **Stop** button. At the bottom of the loop it used *Application.ProcessMessages*, so other things could be done, including pressing the **Stop** button. Unfortunately, many of the things I wanted to do required menu selections. What's wrong with that? Nothing important was wrong, except that the control loop stopped controlling. The same thing happened when I opened a modal form. None of this was a big deal; it simply defeated the purpose of the application!

As a temporary protection, I had the **Start** button disable all menu items, leaving the **Stop** button to re-enable them. This preserved the main purpose of the application, which was to control the equipment reliably, but it made things rather inconvenient. It would have been nice to be able to do other things while remaining in control.

The long-term solution was to make the control loop an independent thread. I began reading various thread-related topics in the Help system, which made the task seem daunting. Finally, when I distilled the subject to its essence, it turned out to be simple. Of course, if you want to exploit a lot of fancy features, you can make the use of threads as complicated as you want. Personally, I like simple, so what I present in this article is very simple.

Keep It Simple

Other than creating the thread, if you take advantage of the facilities that Delphi provides, there is only one thing you must do to run an independent thread: You must define a thread class as a descendant of the

```
program Thrd;

uses
  Forms,
  uMain in 'uMain.pas' {frmMain},
  uModal in 'uModal.pas' {frmModal};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TfrmMain, frmMain);
  Application.CreateForm(TfrmModal, frmModal);
  Application.Run;
end.
```

Figure 1: The project file, Thrd.dpr, for our sample application.

TThread class, and override the *Execute* method. It's in the *Execute* method that you place the code you want to execute. Now that's simple!

Well, few things in life are really that simple. There are a few details to consider. The first is that the *Create* constructor has a Boolean parameter. Pass it a value of *False* to start the thread running as soon as you create it. If you pass *True* to the constructor, the thread will be created in a suspended state, and you'll have to do something else to get it running.

The next detail to consider is whether you want to manage disposal of the thread object yourself, or if you want that to happen automatically when the thread execution terminates. For the automatic approach, set the *FreeOnTerminate* property to *True*. I prefer to do that in the constructor. Yes, I know: Now you have two things to override; but once you've gone to the small effort of deriving a class in the first place, overriding the constructor is trivial.

The third detail to consider is that once you start a thread running, you'll usually want the ability to stop it. If it stops on its own after a short time, that's great. Normally, you'll need to include code to check the *Terminated* property periodically (to make sure it actually does terminate). That way you can tell the thread to terminate from the outside. You can have any other thread (such as the main thread) call its *Terminate* method.

An Illustration

In spite of these extra details, the subject of threads is still rather simple. I will present a small application that illustrates the simplicity of using threads, as well as some of the benefits. The source code for the project is shown in Figure 1. Listing One, beginning on page 10, shows the main unit, named uMain, in its first version.

As usual, I started with a blank form. I named it `frmMain`, set its *Caption* to `Main (Thread Test)`, and set its *Width* to just enough to show the *Caption*. Then I added two buttons (BitBtn components) to the top of the form, side by side. The first button's purpose is to perform a simple count, so I creatively named it *btnCount* with a *Caption* of `Count`. The second button is named *btnThread*, with a *Caption* of `Thread Count`. Below those buttons are *btnStop* and *btnAuto*, with *Captions* of `Stop` and `&Auto`, respectively. Below them are two edit boxes, *edCount* and *edThread*, whose initial *Text* properties are blank. The next two controls are checkboxes, *cbCount* and *cbThread*, with *Captions* of `Count` and `Thread`.

The next item I placed on the form was for later use. It's a radio group, *rgPriorities*, with the *Caption* `Thread Priorities`. I set its *Columns* property to `2`, its *ItemIndex* property to `3`, and its *Items* property to `Idle`, `Lowest`, `Lower`, `Normal`, `Higher`, `Highest`, and `TimeCritical`. Below it I placed another button, named *btnModal*, with a *Caption* of `&Modal Form`. Finally, I added a Timer component, *tmrCount*, and set its *Interval* to `10000` (for ten seconds), and its *Enabled* property to *False*. The result is shown in Figure 2.

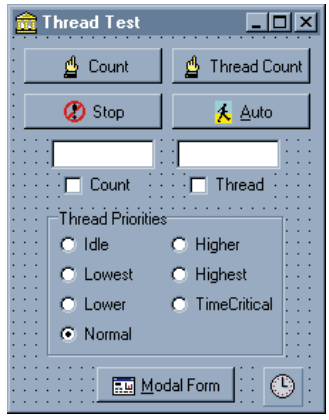


Figure 2: The sample application at design time.

In the **private** section of the *TfrmMain*, I declared a Boolean variable named *Done*; in the global **var** declaration are two Longints: *Count* and *ThreadCount*.

Of course the buttons and Timer need event handlers. The handler for the **Count** button's *OnClick* event is not the solution to one of humanity's great problems. It does, however, provide a basis for comparing the effects of using an independent thread. The essential code is:

```
repeat
  Inc(Count);
until Done;
```

Done is set to *True* by the **Stop** button's *OnClick* handler. As shown here, the loop for *btnCountClick* has a serious problem: There is no opportunity for *Done* to be set. The application will freeze as the loop continues forever, or until you press Ctrl Alt Delete. I'll shortly demonstrate a loop that's even tougher to stop, but for now let's tame the one we have. The loop needs to call *Application.ProcessMessages* so the rest of the application can go about its business — especially the **Stop** button. There's a little more administrative work needed. Initialize *Done*, *Count*, and *edCount.Text* before the loop. And, because it would be nice to see the result, put a string representation of *Count* into *edCount.Text* after the loop.

Even without the other event handlers, you can compile and run the application. You can press the **Count** button, wait a while, then press the **Stop** button. The ending value of the counter will appear in *edCount*. For later comparisons, I found it convenient to count off 10 seconds. You know: "One thousand-one, one thousand-two," etc. On a 486 DX-40 my results were about 225,000, but to paraphrase an automobile advertisement disclaimer, "Your

```
unit uModal;

interface

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

type
  TfrmModal = class(TForm)
    btnClose: TBitBtn;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmModal: TfrmModal;

implementation

{$R *.DFM}

end.
```

Figure 3: Source for the modal dialog box, uModal.pas.

mileage may vary with road conditions, CPU speed, accuracy of your timing estimate, etc."

Modal Oops

To see the effect of a modal form on the counting, let's add an *OnClick* event handler to *btnModal*. Put this in *btnModalClick*:

```
ModalForm.ShowModal;
```

Of course, we should create a form, name it `ModalForm`, and set its *Caption* to `Modal Form`. Its unit, called uModal, is shown in Figure 3. Be sure to put *uModal* in *uMain*'s **implementation uses** clause. Add a BitBtn component to *ModalForm*, name it `btnClose`, and set its *Kind* property to *bkClose*. It's shown in Figure 4.

Compile and run the program. While you count off ten seconds, press the **Modal** button on the main form. After a few seconds, press the **Close** button on *ModalForm* well before you press the **Stop** button. In another run, press the **Modal** button as soon as you can after the **Count** button. Count most of the 10 seconds with *ModalForm* open, then click its **Close** button, then the main-form **Stop** button when the full 10 seconds have elapsed. The results are inescapable: While *ModalForm* is open, the count halts!

Counting the time myself became a little tedious, and the difficulty of obtaining consistent results is obvious. For better tests, I used the Timer component and the **Auto** button, which starts the Timer and optionally starts the count. The *OnClick* handler for **Auto** is:

```
procedure TfrmMain.btnAutoClick(Sender: TObject);
begin
  tmrCount.Enabled := True;
  if cbCount.Checked then
    btnCountClick(nil);
end;
```

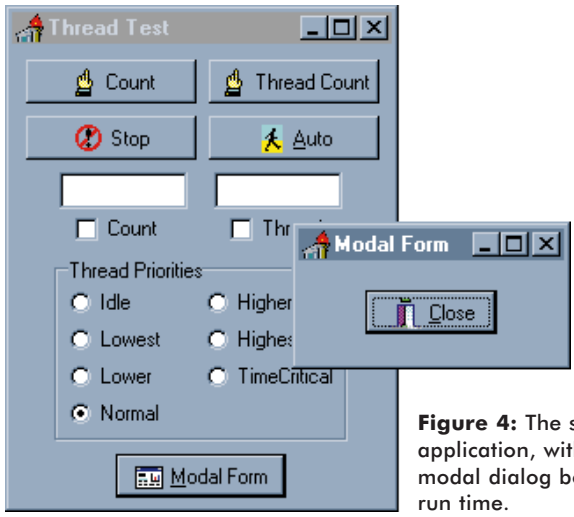


Figure 4: The sample application, with modal dialog box, at run time.

The Timer's *OnTimer* handler effectively "presses" the **Stop** button for us with:

```
procedure TfrmMain.tmrCountTimer(Sender: TObject);
begin
  TTimer(Sender).Enabled := False;
  btnStopClick(nil);
  frmModal.Close;
end;
```

Consider the importance of starting the Timer before starting the count. If we started the count first, the Timer would not be enabled until the count was stopped, which would defeat the purpose of having the Timer. With two successive statements in the same (in this case the main) thread, the first must complete before the second starts. I just snuck in a potential use for coding an independent thread. With preemptive multitasking, you can launch a thread, then execute subsequent statements long before the thread finishes.

At this point you can compile and run the program. Be sure to check the **Count** checkbox, then click the **Auto** button. After 10 seconds you'll see the result. I gave hot keys to the **Auto** button and the **Modal** button so I could click the **Modal** button very quickly after the **Auto** button. If I held down the Alt key, then pressed A then M as quickly as possible, I could see counts under 3000. I let the Timer take care of closing *ModalForm* for me. After you have done that at least once, with and without opening *ModalForm*, you're ready for something a little more interesting.

A New Thread

The *TCountThread* class overrides the *Create* constructor and the *Execute* method. *Create* calls the inherited *Create*, passing *False* for its *CreateSuspended* parameter. Then it sets the *FreeOnTerminate* property to *True*. The *Execute* method is similar to the loop in *btnCountClick*:

```
procedure TCountThread.Execute;
begin
  ThreadCount := 0;
  repeat
    Inc(ThreadCount);
  until Terminated;
end;
```

The *Terminated* property will be set from outside the thread, by calling the thread's existing *Terminate* method. Because it's a separate thread, it is handled by Windows' preemptive multitasking. Therefore, it can omit *Application.ProcessMessages*, which speeds the loop substantially. Of course, if the task were more complicated, the savings from that one call would be relatively less significant.

The repeat loop stops when *Terminated* is *True*, but other (more useful) *Execute* methods may require different approaches. The key fact is that your *Execute* code must monitor the state of the *Terminated* property, and bail out of the method when *Terminated* is *True*. If you assign a handler for the *OnTerminate* event, it will be executed when you call the thread's *Terminate* method. The *TThread* class handles synchronization with the main thread for you.

The Thread Count button's event handler, *btnThreadClick*, creates a thread of class *TCountThread* and automatically starts its execution:

```
procedure TfrmMain.btnThreadClick(Sender: TObject);
begin
  edThread.Text := '';
  ct := TCountThread.Create;
end;
```

We'll use the thread's *OnTerminate* event handler, *ShowResult*, to put text into the *edThread* edit box. Notice that I assigned *ShowResult* inside the *TMainForm* method instead of within the *TCountThread* constructor. *ShowResult* itself is a *TMainForm* method that uses *TMainForm* objects, so I found it convenient to do it this way. Go ahead and add the *ShowResult* method:

```
procedure TfrmMain.ShowResult(Sender:TObject);
begin
  edThread.Text := IntToStr(ThreadCount);
end;
```

and a call to it in the *btnThreadClick* procedure:

```
procedure TfrmMain.btnThreadClick(Sender: TObject);
begin
  edThread.Text := '';
  ct := TCountThread.Create;
  ct.onTerminate := ShowResult;
end;
```

Of course, the *ShowResult* method must be declared as well; Listing Two (on page 11) shows the uMain module's code in its final version.

The two Edit components will give us an obvious comparison of the two ways to accomplish the task. Although the task used in this illustration is a trivial counting job, it represents important real-world tasks that could be used instead.

Now, before we do anything rash, let's add some code to *btnStopClick*, so it can stop our new thread as well as the old count routine. It should now look like this:

```
procedure TfrmMain.btnStopClick(Sender: TObject);
begin
  if (ct <> nil) then
    begin
      ct.Terminate;
      ct := nil;
    end;
  Done := True;
end;
```

The program also needs an **initialization** section for the unit in which we put:

```
ct := nil;
```

We're insuring that *ct* is valid when not **nil**, and **nil** when not valid.

Action

Now that we have something for the Thread Count button to do, let's do it! Compile and run the program, click the Thread Count button, count off 10 seconds and click the Stop button. You'll see quite a different result. On my machine, the count that appears in *edThread* is about 38,000,000, which means the loop went about 169 times faster. There certainly seems to be a lot less overhead in checking the *Terminated* property than there is in calling *Application.ProcessMessages* and checking the *Done* variable. Score one more point in the independent thread column.

Now let's automate. Put the following new code in *btnAutoClick*. Make sure you put it before the call to *btnCountClick*, for the same reasons as previously mentioned. The added code is:

```
if cbThread.Checked then
  btnThreadClick(nil);
```

so the procedure should look like this:

```
procedure TfrmMain.btnAutoClick(Sender: TObject);
begin
  tmrCount.Enabled := True;
  if cbThread.Checked then
    btnThreadClick(nil);
  if cbCount.Checked then
    btnCountClick(nil);
end;
```

This will enable you to start one count or the other, or both, and then stop after 10 seconds. When you compile and run now, check both checkboxes and click the Auto button. Notice that both counts come out smaller now that the CPU's attention is divided between two active threads, but the new thread still gets much further in its count. Of course the main thread was active all along, but when we ran the other thread before, the main thread was doing little more than waiting for user input.

Now try the same thing, but open *ModalForm* right afterwards and let the Timer close it. As expected, it stopped the main count cold for the duration, but look what it did to the new thread. The new thread got almost as far as if it were the only count activated.

Set Your Priorities

A thread can have priorities. With a lower priority it gets less of the CPU's time, and with a higher priority it gets more of the CPU's time. Delphi gives us priority values from an enumerated type, whose ordinal values are sequential and zero-based. The radio group placed on the form, but not yet used, has a zero-based *ItemIndex* property corresponding to which radio button is on. With a little typecasting, we can set the priority of the thread by setting the appropriate radio button. The *btnThreadClick* procedure should look like this:

```
procedure TfrmMain.btnThreadClick(Sender: TObject);
begin
  edThread.Text := '';
  ct := TCountThread.Create;
  ct.OnTerminate := ShowResult;
  ct.Priority := TThreadPriority(rgPriorities.ItemIndex);
end;
```

Now you can experiment with different priorities for the new thread. The priority we were using before by default was *tpNormal*. Notice that if you have *ModalForm* open most of the time, the priority setting has little effect; it's most meaningful when the thread has to compete for attention.

Unless you like re-booting, do not use *tpTimeCritical* in this application. Use it only with threads that are quickly self-terminating, and that truly are "time critical." On the other hand, if you're prepared to shut off the computer's power, go ahead and have fun. *But you've been warned.*

Last Thread

I trust this little illustration has encouraged you that implementing independent threads can be quite easy, and I hope it's given you some ideas for using independent threads.

Incidentally, I did all this in Delphi's IDE. It occurred to me that Delphi itself was an application with at least one thread, so I shut down Delphi and launched THRD.EXE as the only obvious application running. I saw numbers that were noticeably higher, but not startlingly so. Evidently, because Delphi had been doing nothing but awaiting user input, it took very little of the CPU's time. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\DEC\DI9712JJ.

Jon Jacobs works for USLife converting Turbo Pascal software to Delphi. He lives in Dallas with his wife and son.

Begin Listing One — uMain.pas (Version 1)

```
unit uMain;

interface

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

type
  TfrmMain = class(TForm)
    btnCount: TBitBtn;
    btnThread: TBitBtn;
    btnStop: TBitBtn;
    btnAuto: TBitBtn;
    btnModal: TBitBtn;
    edCount: TEdit;
    edThread: TEdit;
    cbCount: TCheckBox;
    cbThread: TCheckBox;
    tmrCount: TTimer;
    rgPriorities: TRadioGroup;
    procedure btnCountClick(Sender: TObject);
    procedure btnStopClick(Sender: TObject);
    procedure btnAutoClick(Sender: TObject);
    procedure btnModalClick(Sender: TObject);
    procedure tmrCountTimer(Sender: TObject);
  private
    Done : Boolean;
  public
    { Public declarations }
  end;

var
  frmMain: TfrmMain;
  Count,
  ThreadCount : Longint;

implementation

uses uModal;

{$R *.DFM}

procedure TfrmMain.btnCountClick(Sender: TObject);
begin
  Done := False;
  Count := 0;
  edCount.Text := '';

  repeat
    Inc(Count);
    Application.ProcessMessages;
  until Done;

  edCount.Text := IntToStr(Count);
end;

procedure TfrmMain.btnStopClick(Sender: TObject);
begin
  Done := True;
  frmModal.Close;
end;

procedure TfrmMain.btnAutoClick(Sender: TObject);
begin
  tmrCount.Enabled := True;

  if cbCount.Checked then
    btnCountClick(nil);
end;

procedure TfrmMain.tmrCountTimer(Sender: TObject);
begin
  tmrCount.Enabled := False;
  btnStopClick(nil);
```

```
  frmModal.Close;
end;

procedure TfrmMain.btnModalClick(Sender: TObject);
begin
  frmModal.ShowModal;
end;

end.
```

End Listing One

Begin Listing Two — uMain.pas (Version 2)

```
unit uMain;

interface

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

type
  TfrmMain = class(TForm)
    btnCount: TBitBtn;
    btnThread: TBitBtn;
    btnStop: TBitBtn;
    btnAuto: TBitBtn;
    btnModal: TBitBtn;
    cbCount: TCheckBox;
    cbThread: TCheckBox;
    edCount: TEdit;
    edThread: TEdit;
    rgPriorities: TRadioGroup;
    tmrCount: TTimer;
    procedure btnCountClick(Sender: TObject);
    procedure btnStopClick(Sender: TObject);
    procedure btnAutoClick(Sender: TObject);
    procedure tmrCountTimer(Sender: TObject);
    procedure btnModalClick(Sender: TObject);
    procedure btnThreadClick(Sender: TObject);
  private
    Done : Boolean;
    procedure ShowResult(Sender:TObject);
  public
    { Public declarations }
  end;

  TCountThread = class(TThread)
    protected
      procedure Execute; override;
    public
      constructor Create;
    end;

var
  frmMain : TfrmMain;

  Count,
  ThreadCount : Longint;
  ct : TCountThread;

implementation

uses uModal;

{$R *.DFM}

procedure TfrmMain.btnCountClick(Sender: TObject);
begin
  Done := False;
  Count := 0;
  edCount.Text := '';

  repeat
    Inc(Count);
    Application.ProcessMessages;
  until Done;

  edCount.Text := IntToStr(Count);
end;

procedure TfrmMain.btnStopClick(Sender: TObject);
begin
  if (ct <> nil) then
    begin
      ct.Terminate;
      ct := nil;
    end;
  Done := True;
end;

procedure TfrmMain.btnAutoClick(Sender: TObject);
begin
  tmrCount.Enabled := True;

  if cbThread.Checked then
    btnThreadClick(nil);
  if cbCount.Checked then
    btnCountClick(nil);
end;

procedure TfrmMain.tmrCountTimer(Sender: TObject);
begin
  TTimer(Sender).Enabled := False;
  btnStopClick(nil);

  frmModal.Close;
end;

procedure TfrmMain.btnModalClick(Sender: TObject);
begin
  frmModal.ShowModal;
end;

procedure TfrmMain.btnThreadClick(Sender: TObject);
begin
  edThread.Text := '';
  ct := TCountThread.Create;
  ct.OnTerminate := ShowResult;
  ct.Priority := TThreadPriority(rgPriorities.ItemIndex);
end;

procedure TfrmMain.ShowResult(Sender:TObject);
begin
  { Used for thread's OnTerminate }
  edThread.Text := IntToStr(ThreadCount);
end;

constructor TCountThread.Create;
begin
  inherited Create(False);
  FreeOnTerminate := True;
end;

procedure TCountThread.Execute;
begin
  ThreadCount := 0;

  repeat
    Inc(ThreadCount);
  until Terminated;
end;

initialization

  ct := nil;

end.
```

End Listing Two

By Howard Schutzman

FTP Programming

Creating File Transfer Protocol Programs with Windows Sockets

The purpose of this article is to get "under the hood" of Internet TCP/IP socket programming. You will learn to write code that can communicate with a program running on another computer. The other program doesn't need to be running on Windows; it can be running on a UNIX workstation, a Macintosh, or a mainframe. As long as the remote computer is connected to your computer with a TCP/IP network (such as the Internet), the two programs can send and receive messages.

Real-World Applications

It sounds interesting, but you may be wondering what kinds of problems you can solve with this technology. To spark your imagination, I will describe two socket-programming projects I helped implement.

The first project involved verifying and transferring Electronic Data Interchange (EDI) files, such as invoices, between different companies with different kinds of computers. For example, Joe at Company A might want to retrieve a file from Company B's computer, verify its contents, and send it to Fran at Company C. When done manually, this is a tedious process. First, Joe needs to connect to Company B's computer using some type of FTP (File Transfer Protocol) program. Assuming he understands how to use FTP commands (using FTP isn't easy, especially for less technical users), he downloads the file to his computer. Second, he runs the verification program. Third, if successful, he starts his e-mail program, attaches the file, and sends it to Fran.

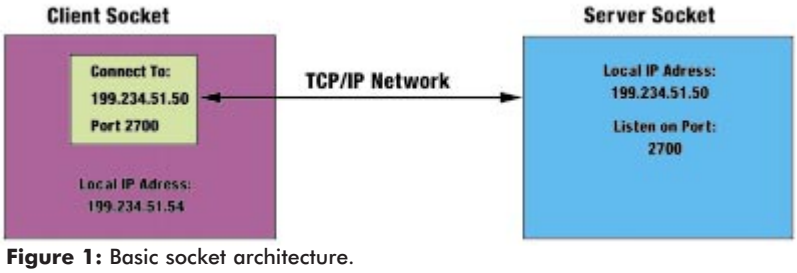


Figure 1: Basic socket architecture.

Using socket programming, I implemented a program that reduces this to a button click. The program automatically logs on to Company B's computer, downloads the appropriate file, verifies it, and composes and sends an e-mail to Fran. The code presented in this article is derived from the FTP portion of the EDI project (see end of article for download details).

The second project involved implementing a high-performance link between a server and many remote clients. At peak times, the server must handle more than 100,000 requests per hour, and still respond within a few seconds. To solve this problem, we designed a compact proprietary messaging scheme and used socket programming to handle the communications. As a side project, we implemented an automatic updating system, whereby the server notifies the client

if it's in need of updating, and schedules a file transfer to provide the update.

Overview of Sockets

A socket is the object that handles the sending and receiving of messages over a TCP/IP network, including the Internet. TCP/IP is a set of protocols that ensures the reliable delivery of messages. TCP stands for Transmission Control Protocol; IP stands for Internet Protocol. Using sockets, you're only concerned with message contents. TCP/IP administers delivery; it breaks the message into packets, routes each packet to the correct destination, requests retransmission if a packet has been corrupted, and reassembles the message.

Figure 1 provides an overview of how sockets work. One socket is the Server; it listens on the network to see if anyone wants to communicate with it. The other socket is the Client; it attempts to connect to a socket that is listening. The Server socket notifies the Client socket if it accepts the connection. Once a connection is established, either the Client or Server can send messages. Additionally, either socket can terminate the connection.

If you are writing the programs on both sides of the connection, you may want to implement your own communications protocol. You can also implement one of the standard Internet protocols. This article will use standard FTP to illustrate how to write a socket program.

A socket has two properties that are used to identify it: the IP Address and the Port. You are probably familiar with the IP Address. This is the "dot" address used on the Internet. It consists of four eight-bit numbers separated by periods, such as 199.234.51.54. Each computer on a TCP/IP network must have a unique IP Address.

You may be wondering how to determine a machine's IP Address. Typically, when you are browsing on the Internet, you use a name, such as "www.borland.com", to connect to a machine. This name must be translated to a 32-bit IP Address before using a socket. Name-to-IP Address translation is taken care of by the Domain Name Server (DNS) running at your Internet Service Provider (ISP). If you are running on a local intranet, there is usually a local process that takes care of name resolution. In any case, as you will see, you need not worry about this problem because name resolution is part of the socket-programming interface.

The Port is similar in concept to a serial or parallel port. A computer may have several server sockets running simultaneously, each listening on a different Port. For example, if a computer has both a Web Server and an FTP server running, the Web Server is listening on Port 80 and the FTP server is listening on Port 21. Figure 2 lists several common Internet protocols and the ports they use. Port names can also be symbolic. For example, when you type `http:` in your browser, you are referring to Port 80; when you type `ftp:`, you are referring to Port 21.

The Socket Programming Interface

The standard network API for Windows TCP/IP programming is known as WinSock. The WinSock implementation on a machine is often referred to as the "TCP/IP stack." Most machines running Windows 95 and Windows NT 4 use the WinSock implementation provided by Microsoft as part of the operating system. The situation under Windows 3.1 (and 3.11) is more chaotic. Because Microsoft did not include a WinSock implementation with the operating system, a Windows 3.1 machine may contain any of a number of WinSock implementations, each slightly different from the others. Fortunately, you should still be able to support most flavors of Windows 3.1 TCP/IP stacks.

The WinSock API is complex. Therefore, a number of vendors have written components that provide an object-oriented wrapper around the API. I've tried several — some with limited success. I found the dWinsock VCL components to be very reliable on both Windows 3.1 and Windows 95. Additionally, the dWinsock authors provide excellent support for their product. A shareware version has been included with the code accompanying this article. The dWinsock Web site is located at http://www.aait.com/dwinsock.

Windows Sockets Network Programming by Bob Quinn and Dave Shute [Addison-Wesley, 1995] is considered one of the best books on WinSock programming. If you are using one of the wrapper components, however, you probably don't need to read it. But if you are one of those people who likes to know all the gory technical details, it's the book for you.

The rest of this article is concerned with implementing HsSocket, an object that allows you to add FTP client capabilities to an application. You can connect to an FTP server, change directories, obtain a directory listing, download a file from the server, upload a file to the server, delete a file, send site-specific commands, and disconnect. Because I needed to support Windows 3.1, the code was implemented using 16-bit Delphi, but it can easily be ported to 32-bit Delphi.

The code in this article was implemented using the dWinsock control mentioned previously. Therefore, if you intend to run this code within Delphi, you must install dWinsock as a VCL component by following the instructions in the installation file.

To test the code, you need to connect to an FTP server. You can certainly connect to any number of servers on the Internet. However, you will probably find it useful to have a small FTP server running on your machine. (WFTPD, an inexpensive FTP server, is available from Texas Imperial Software; contact alun@texis.com for further information.)

Protocol	Port
HTTP (Web)	80
FTP	21
Gopher	70
SMTP (e-mail)	25
POP3 (e-mail)	110
Telnet	23

Figure 2: Some common Internet protocol ports.

Before proceeding, we'll look at some code that shows how to

```
{ Connect command socket to FTP server. }
function THsSocket.FtpConnect(Address : string;
                              Port : Integer) : Boolean;
begin
  Result := False
  CmdSkt.Address := Address;
  CmdSkt.Port := IntToStr(Port);

  try
    CloseSkt(CmdSkt);
    CmdSkt.Open(TStreamSocket);
  except
    on E : Exception do
      DisplayError('Open error: ' + E.Message);
  end;

  Result := True;

end;
```

Figure 3: A simplified version of the *FtpConnect* function.

```
{ Command connection OnDisconnect handler. }
procedure THsSocket.CmdSktDisconnect(Sender: TObject;
                                     Socket: TSocketBase);
begin
  if (FActionCompleted) then
    Exit;

  case WaitState of
    wsQuit : FActionCompleted := True;
  end;
end;
```

Figure 4: The *OnSocketDisconnect* event handler.

establish a connection with a client socket. The code in Figure 3 shows a simplified version of the *FtpConnect* function. The two arguments are the *Address* of the FTP server and the *Port*. *Address* can be either an IP address (e.g. 199.234.51.54), or a domain name (e.g. world.std.com); the dWinsock component takes care of name-to-address resolution. *Port* is normally 21, which is the standard FTP server port. To connect to the server, set the client socket's *Address* and *Port* properties, and call the *Open* method.

The *Open* method takes one argument: the socket type. In our case, this is *TStreamSocket*, which represents reliable TCP transport. dWinsock also supports the *TDatagramSocket* type, which implements the faster — but less reliable — UDP (User Datagram Protocol).

When the server accepts the connection, the *OnConnect* event is fired. A flag is set in the event handler, indicating the connection was successful. This illustrates the asynchronous nature of socket programming. The next section presents a technique for simplifying handling of asynchronous events.

Serializing Sockets

Socket programming is *asynchronous*; after sending a request, an event must fire before an application knows whether the request succeeded. If the network goes down, the event will never fire. The advantage of an asynchronous approach is a more efficient use of computer resources. It can take a long time to receive a response over the network. Rather than sitting idle and waiting, the computer can perform other useful work until the response is received.

```
{ ActionCompleted is the key routine for serializing
  asynchronous behavior. It's called after each command is
  sent to the FTP server. It waits until the server
  returns a successful status code, or until it times out.
  It returns True on success, False on failure. }
function THsSocket.ActionCompleted : Boolean;
var
  EndTime : TDateTime;
begin
  { Wait until TimeOut seconds have elapsed. }
  EndTime := Now + CmdSkt.TimeOut / SECONDS_PER_DAY;

  while (Now < EndTime) do begin
    { Exit successfully if event handler detected
      success status. }
    if (FActionCompleted) then
      begin
        WaitState := wsNone;
        Result := True;
        Exit;
      end;

    { Reset wait time if requested by event handler. }
    if (ResetTime) then
      begin
        EndTime := Now + CmdSkt.TimeOut / SECONDS_PER_DAY;
        ResetTime := False;
      end;

    { Let other processes work. }
    Application.ProcessMessages;
  end;  { while }

   { Failure if we time out. }
   Result := False;
end;
```

Figure 5: The *ActionCompleted* function.

The disadvantage is that more complex logic is needed to handle asynchronous events. The program cannot proceed sequentially. It must remember its state so that when an event fires, it knows what to do.

To simplify matters, HsSocket implements a technique that allows requests to be made in a serial fashion. A simple state machine is used. Before making a request, the code sets a state variable indicating which request is pending. Then the request is made. A call is made to the *ActionCompleted* function. *ActionCompleted* will not return control until the appropriate event has fired, or the request times out. Therefore, each request is synchronous, waiting for *ActionCompleted* to return before proceeding.

ActionCompleted works with the event handlers. When an event is fired, the handler looks at the state variable to see which request is pending. If the event is a response to the pending request, the event handler sets the *FActionCompleted* variable to *True*, to indicate the request is no longer pending. The code in Figure 4 is the *OnDisconnect* event handler. If the state is *wsQuit*, indicating a Quit request has been sent to disconnect from the server, the *FActionCompleted* variable is set to *True*.

The code in Figure 5 shows the *ActionCompleted* function. The code loops until either the *FActionCompleted* flag has been set by one of the event handlers, or looping has continued for longer than the time-out value, *EndTime*. The

```
{ Connect command socket to FTP server. }
function THsSocket.FtpConnect(Address : string;
                              Port : Integer) : Boolean;
begin
  Result := False;
  SetWaitState(wsConnect);
  CmdSkt.Address := Address;
  CmdSkt.Port := IntToStr(Port);

  try
    CloseSkt(CmdSkt);
    CmdSkt.Open(TStreamSocket);
  except
    on E : Exception do
      DisplayError('Open error: ' + E.Message);
  end;

  if (not ActionCompleted) then begin
    DisplayError('Unable to connect to server ' + Address);
    Exit;
  end;

  Result := True;
end;
```

Figure 6: The full version of FtpConnect.

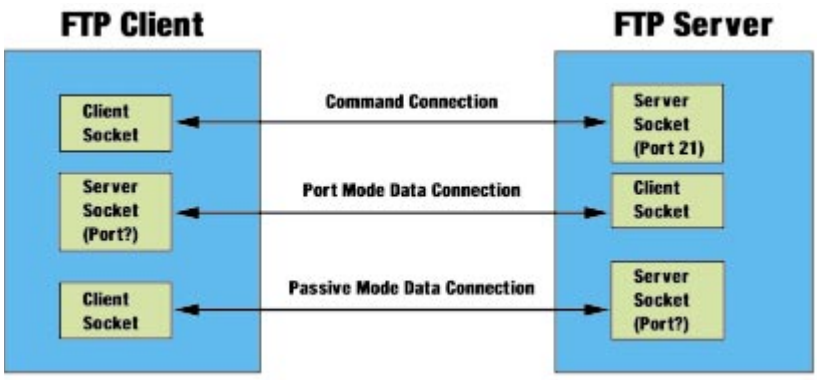


Figure 7: The FTP architecture.

Function	Command Syntax	Success Status
set user	USER <user name>	230, 331
set password	PASS <password>	230
change directory	CWD <path>	250
set data type	TYPE <type code>	200
port-mode data	PORT <address, port>	200
passive-mode data	PASV	227
list directory	LIST <path name>	(data port closed)
retrieve file	RETR <file name>	(data port closed)
store file	STOR <file name>	226, 250
delete file	DELE <file name>	250
site specific	SITE <string>	200
disconnect	QUIT	(cmd port closed)

Figure 8: An FTP command summary.

ResetTime flag is used for requests that fire more than one event. If an intermediate event is fired, the *ResetTime* flag indicates the time-out value should be reset so a time out doesn't occur before the final event is fired. The primary use of this flag is for retrieving a file. The file is received in blocks, with each block firing an intermediate event that sets the *ResetTime* flag. When the event is fired that indicates the entire file has been retrieved, the *FActionCompleted* flag is set. If there was no *ResetTime* flag, a time-out could easily occur before the entire file is received.

Figure 6 is a full listing of the *FtpConnect* function. The state variable is set to *wsConnect* to indicate a connection is pending. The *Open* method is called. *ActionCompleted* is called and will not return until the connection is accepted, or the wait exceeds the time-out value.

FTP Overview

Figure 7 illustrates the architecture of the FTP. The main connection from the FTP client to the FTP server occurs on Port 21. This is known as the *command connection*. All messages over this connection are ASCII text ending in a carriage return/line feed. The client sends commands to the server. *Commands* are defined as three- or four-character mnemonics, and some commands include parameters. The server responds to the commands with status messages, which consist of a three-digit code followed by a text message. The value of the three-digit code indicates whether the request was successful.

Certain commands, such as those for retrieving or sending files, require another connection to be opened; this is known as the *data connection*. The file data is sent over the data connection, and the connection is closed once the data has been sent. There are two modes for establishing data connections. In *port mode*, the FTP client listens on a server socket and sends the port number to the FTP server. The server socket must listen on Port 20, or on the next available port after 1024. (Ports 1-1024 are reserved for standard protocols.) In *passive mode*, the FTP server listens and sends the port number in response to a passive request. The choice of data-connection modes is up to you. Most commercial FTP clients default to port mode. If you are concerned about security, or must pass through a firewall, passive mode is a better choice because you connect to the server rather than having the server connect to you.

HsSocket uses three socket components to implement the FTP: *CmdSkt* is a client socket used for the command connection; *ListenSkt* is a server socket used for the port-mode data connection; and *DataSkt* is a client socket used for the passive-mode data connection.

The table in Figure 8 summarizes the FTP commands implemented in this article, as well as the return codes indicating success. These commands represent a subset of the entire FTP command set.

As with all Internet protocols, FTP is described in a Request for Comment (RFC) document, RFC #929, available at http://www.internic.net. Many books on Internet programming also contain a good description of the FTP.

Command Processing

The code in Listing Three (beginning on page 18) shows the routines used to send commands to the FTP server. The client socket named *CmdSkt* is used for the command connection.

The procedure *SendCommand* is called each time a command is sent. It merely adds a carriage return/line feed and calls *SendSktData*. The *SendSktData* procedure handles the sending of all data (commands and files). It attempts to send data using the *Send* method, which returns the number of bytes sent. If all the data is not sent, the *OnWrite* event will fire when the socket is ready to send more data. The *OnWrite* event handler sets the *OnWriteFlag* to *True*. Therefore, *SendSktData* will loop until either the *OnWrite* event fires (as indicated by *OnWriteFlag*), or the time-out value is exceeded.

When the FTP server sends a reply, it will trigger an *OnRead* event. The *CmdSktRead* procedure is the event handler. It's possible the FTP server will send several reply lines. Therefore, *CmdSktRead* builds a string list with one entry for each reply line. *CmdSktRead* can determine where each reply line ends by searching for the terminating carriage-return/line-feed characters. *CmdSktRead* also handles the case where a partial line is sent (although this rarely happens), by saving the data until the next *OnRead* event is fired with the rest of the line.

For each reply line, *CmdSktRead* strips off the status code. Based on the wait state, which indicates the pending command, the *FActionCompleted* is set when the status code indicates success. This allows the *ActionCompleted* function, as discussed above, to return a success indicator, and the application can proceed to the next command.

Note that this approach has minimal error reporting. If the FTP server returns an error status code, it is simply ignored. Eventually, *ActionCompleted* will time out and return a failure indicator. The program will fail gracefully, but no indication is given as to why the failure occurred. This is acceptable for my automated applications because, once a connection is established, things almost always proceed smoothly. Depending on your application, however, you may want to expand the *CmdSktRead* routine to improve error reporting by processing failure statuses as well as success statuses.

Data Connections

As indicated earlier, certain commands require that a data connection be established. These commands include *retrieve file* (from the server), *store file* (on the server), and *list directory* (the directory listing is treated as a file). The code in Listing Four (beginning on page 20) is used to establish a data connection.

The *FtpSetType* function is used to establish the type of data to be transferred. The code in Listing Four supports ASCII and binary data. The FTP also supports EBCDIC data.

The *FtpPort* function is used to establish a port-mode data connection. The *ListenSkt* port property is set to 0; this allows WinSock to choose any port above 1024 for listening. The *Listen* method is called to place *ListenSkt* in listen mode. The FTP PORT command string is constructed. The syntax is the word "PORT", followed by six comma-separated values. The first four values are the four parts of the IP Address of the computer running the FTP client. The last two values are the upper and lower bytes of the port number. The command string is then sent to the FTP server so it knows where to connect.

Alternatively, the *FtpPassive* function is used to establish a passive-mode data connection. The PASV command is sent to ask the FTP server to set up a server socket. The reply from the server contains connection information about the server socket. As with the PORT command, it consists of six comma-separated values. The first four values are the server IP address. The last two values are the upper and lower bytes of the port number. *FtpPassive* decodes these values and stores them in the *DataSkt Address* and *Port* properties. The *Open* method is then called to establish a data connection with the server.

The *DoDataSktRead* event-handler procedure is used to read data from the FTP server over the data connection. It calls the socket's *Recv* method to fill a buffer with the data. It uses the *BlockWrite* routine to transfer this data to a file. It sets the *ResetTime* flag to inform the *ActionCompleted* routine to reset its time-out value.

The *SendSktData* procedure is used to send data to the FTP server over the data connection. This code was discussed in the previous section.

Testing HsSocket

Figure 9 lists the public methods in HsSocket. There is one method for each command you can send to the FTP server. There is a utility routine to set the time-out value. There are also three debugging routines to allow you to log information to a log file or to the screen.

```
{ FTP methods. }
  function FtpChangeDir(Directory : string) : Boolean;
  function FtpConnect(Address : string;
                      Port : Integer) : Boolean;
  function FtpDelete(RemoteFile : string) : Boolean;
  function FtpDisconnect : Boolean;
  function FtpListDir(Filter : string;
                      ListFile : string) : Boolean;
  function FtpPassive : Boolean;
  function FtpPassword(Password : string) : Boolean;
  function FtpPort : Boolean;
  function FtpRetrieveFile(LocalFile : string;
                           RemoteFile : string) : Boolean;
  function FtpSetType(FtpType : TFtpType) : Boolean;
  function FtpSite(Command : string) : Boolean;
  function FtpStoreFile(LocalFile : string;
                        RemoteFile : string) : Boolean;
  function FtpUser(User : string) : Boolean;

  { Other methods. }
  procedure SetTimeOut(Seconds : Integer);

  { Debugging methods. }
  procedure OpenLog(LogFileName : string);
  procedure CloseLog;
  procedure DisplayDebugForm(ShowFlag : Boolean);
```

Figure 9: HsSocket public methods.

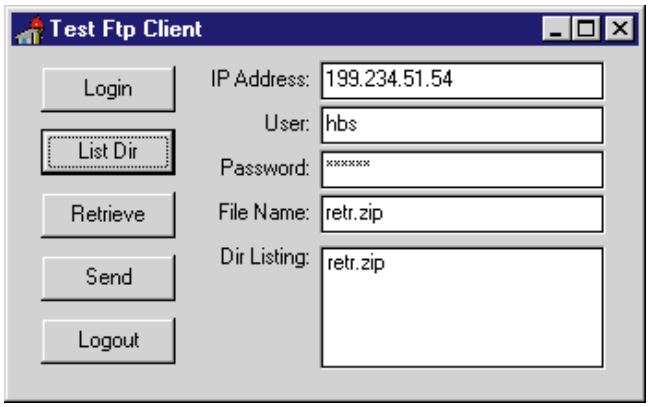


Figure 10: The Ftptest.exe main form.

```
procedure TMainFrm.LoginBtnClick(Sender: TObject);
var
  Success : Boolean;
begin
  { IP address, user name, and password must be filled. }
  if ((IpAddrEdit.Text = '') or
      (UserEdit.Text = '') or
      (PassEdit.Text = '')) then
    begin
      MessageDlg(
        'Please enter ip address, user name, and password',
        mtWarning, [mbOk], 0);
    end;

  { Open debug log. }
  HsSocket.OpenLog('log.txt');

  { Connect, send user command, send password command. }
  with HsSocket do begin
    Success := FtpConnect(IpAddrEdit.Text, 21);
    if (Success) then
      Success := FtpUser(UserEdit.Text);
    if (Success) then
      Success := FtpPassword(PassEdit.Text);
  end;

  { Indicate if successful. }
  if (Success) then
    MessageDlg('Login successful', mtInformation, [mbOk], 0);
  else
    MessageDlg('Unable to log in', mtWarning, [mbOk], 0);

end;
```

Figure 11: The *OnClick* event handler for the **Login** button.

I chose to implement HsSocket as an invisible form. This gave me a container for the socket components, and provided me with a debugging form that can be made visible upon request. Alternatively, I could have implemented HsSocket as a plain Delphi object, or as a non-visual VCL component.

The form in Figure 10 is the main form for Ftptest.exe, a test application for HsSocket. Each of the buttons exercise different commands that can be sent to the FTP server. The complete source for this executable is available; see end of article for details.

The **Login** button code connects to the server, and sends the user name and the password. Its *OnClick* event handler is shown in Figure 11.

```
procedure TMainFrm.ListBtnClick(Sender: TObject);
var
  Success : Boolean;
  ListFile : TextFile;
  FileStr : string;
begin
  { Set ASCII data mode, set up port data connection,
    get directory listing. }
  with HsSocket do begin
    Success := FtpSetType(ftAscii);
    if (Success) then
      Success := FtpPort;
    if (Success) then
      Success := FtpListDir('', 'listdir.txt');
  end;

  if (not Success) then
    begin
      MessageDlg('Unable to obtain directory listing',
                 mtWarning, [mbOk], 0);
      Exit;
    end;

  { Fill list box with directory listing passed back
    in Listdir.txt. }
  DirList.Clear;
  AssignFile(ListFile, 'listdir.txt');
  Reset(ListFile);
  while (not Eof(ListFile)) do begin
    Readln(ListFile, FileStr);
    DirList.Items.Add(StripFileName(FileStr));
  end;

  CloseFile(ListFile);
end;
```

Figure 12: The *OnClick* event handler for the **List Dir** button.

```
procedure TMainFrm.RetrieveBtnClick(Sender: TObject);
var
  Success : Boolean;
begin
  { Make sure file name has been entered. }
  if (FileEdit.Text = '') then
    begin
      MessageDlg('Please enter a file name',
                 mtWarning, [mbOk], 0);
      Exit;
    end;

  { Set binary data mode; set up passive data connection;
    retrieve file. }
  with HsSocket do begin
    Success := FtpSetType(ftBinary);
    if (Success) then
      Success := FtpPassive;
    if (Success) then
      Success := FtpRetrieveFile(FileEdit.Text,
                                 FileEdit.Text);
  end;

  { Indicate success or failure. }
  if (Success) then
    MessageDlg(FileEdit.Text + ' retrieved successfully',
               mtInformation, [mbOk], 0);
  else
    MessageDlg('Unable to retrieve ' + FileEdit.Text,
               mtWarning, [mbOk], 0);
end;
```

Figure 13: The *OnClick* event handler for the **Retrieve** button.

The **List Dir** button code (see Figure 12) sets the data-connection type to ASCII, sets up a port mode data connection, and sends the list-directory command. It reads the file returned by the server, stripping out the file names and loading the list box.

The **Retrieve** button (see Figure 13) sets the data connection to ASCII, sets up a passive-mode data connection, and sends the "retrieve" file command. Remember, the mode choice for data connections is up to you. I simply wanted to test both modes. The **Send** button is similar to the **Retrieve** button, except it sends the "store" file command instead of the "retrieve" file command.

The **Logout** button code simply sends the "quit" command:

```
procedure TMainFrm.LogoutBtnClick(Sender: TObject);
begin
  HsSocket.FtpDisconnect;
  HsSocket.CloseLog;
end;
```

Figure 14 is a listing of the log file generated by pushing each button. Figure 15 shows the FTP server log, demonstrating how the protocol is perceived from the server's perspective. I recommend you take a look at these figures, because they will quickly give you a good understanding of how the FTP works.

```
Log started at 4/27/97 1:54:19 PM
RECV: 220 WFTPD 2.30 service (by Texas Imperial Software) ready for new user
SEND: USER hbs
RECV: 331 Give me your password, please
SEND: PASS
RECV: 230 Logged in successfully
SEND: TYPE A
RECV: 200 Type is ASCII
SEND: PORT 199,234,51,54,4,88
RECV: 200 PORT command okay
SEND: LIST
RECV: 150 File Listing Follows in ASCII mode
RECV: 226 Transfer finished successfully.
SEND: TYPE I
RECV: 200 Type is Image (Binary)
SEND: PASV
RECV: 227 Entering Passive Mode (199,234,51,54,4,89)
SEND: RETR retr.zip
RECV: 150 "D:/TEMP/retr.zip" file ready to send (144714 bytes) in IMAGE / Binary mode
RECV: 226 Transfer finished successfully.
SEND: TYPE I
RECV: 200 Type is Image (Binary)
SEND: PORT 199,234,51,54,4,91
RECV: 200 PORT command okay
SEND: STOR send.zip
RECV: 150 "D:/TEMP/send.zip" file ready to receive in IMAGE / Binary mode
RECV: 226 Transfer finished successfully.
File store successfully completed
SEND: QUIT
Log file closed at 4/27/97 1:54:43 PM
```

Figure 14: The log file listing.

Figure 15: The server log listing.

Conclusion

This article gives you a solid introduction on how to program using TCP/IP sockets. FTP was chosen as an example because it exposes most of the issues related to socket programming. You might want to choose a simpler protocol for your first socket program, such as the SMTP for sending mail. Alternatively, you can design your own protocol and implement both the client and the server. You need not be on a network to test your implementation; sockets can be used to communicate between two programs running on the same machine. Sockets are a powerful technology; combined with Delphi, you can solve some very interesting and important problems. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\DEC\-DI9712HS.

Howard Schutzman is the principal consultant at Computing Concepts, Inc. in Westford, MA. He develops custom software solutions, including Internet and multimedia applications. He prefers programming with Delphi, but can be coerced into using C++ or Visual Basic if necessary. Howard can be reached at hbs@world.std.com.

Begin Listing Three — Command Processing Routines

```
{ Append cr/lf prior to sending a command. }
procedure THsSocket.SendCommand(Msg : string);
const
  CrLf = #13#10;
var
  SendMsg : array [0..258] of Char;
begin
  StrPCopy(SendMsg, Msg + CrLf);
  AddDiagnostic('SEND: ' + Msg);
  SendSktData(CmdSkt.Conn, SendMsg, StrLen(SendMsg));
end;
```

```
{ Send data to FTP server. }
function THsSocket.SendSktData(Socket : TSocketBase;
  Buffer : PChar; Count : Integer) : Boolean;
var
  Remaining : Integer;
  Ptr : PChar;
  EndTime : TDateTime;
begin
  Result := False;
  if (Count <= 0) then
    Exit;
  Remaining := Count;
  Ptr := Buffer;

  { Keep trying to send until all data sent or timeout. }
  while (True) do begin
    OnWriteFlag := False; { Reset by OnWrite. }

    { If all bytes have been sent, we're done. }
    Count := Remaining;
    Remaining := Count - Socket.Send(Ptr^, Count);
    if (Remaining <= 0) then
      Break;

    { Otherwise, wait until OnWrite event is fired,
      or we time out. }
    EndTime := Now + CmdSkt.TimeOut / SECONDS_PER_DAY;
    while (not OnWriteFlag and (Now < EndTime)) do begin
      Application.ProcessMessages;
    end;
    if (not OnWriteFlag) then
      begin
        DisplayError('Unable to complete send');
        Exit;
      end;

    { Reset data ptr past bytes that have been sent. }
    Ptr := Ptr + (Count - Remaining);
  end;
  Result := True;
end;

{ OnWrite handler for all 3 sockets... }
procedure THsSocket.SktWrite(Sender: TObject;
                             Socket: TSocketBase);
begin
  OnWriteFlag := True; { ...so SendSktData can continue. }
end;


{ Command connection OnRead event handler. }
procedure THsSocket.CmdSktRead(Sender: TObject;
                               Socket: TSocketBase);
const
  { Success statuses for all ftp commands. }
  CONNECT_OK = '220';
  USER_OK = '230';
  USER_OK_PASS = '331';
  PASS_OK = '230';
  PASV_OK = '227';
  TYPE_OK = '200';
  CWD_OK  = '250';
  PORT_OK = '200';
  STOR_OK = '125';
  STOR_OK_OPEN = '150';
  SITE_OK = '200';
  STOR_DONE_OK = '250';
  STOR_DONE_OK_CLOSE = '226';
  DELE_OK = '250';

  CrLf = #13#10;
var
  Buffer : array [0..2047] of Char;
  BuffNdx : Integer;
  RecvCnt : Integer;
  Remaining : Integer;
  BuffPtr, CrLfPtr : PChar;
  i : Integer;
  Status : string [3];
begin
  { Build a string list; one entry per server reply line. }

  { If a previous read contained a partial reply,
    add it to the receive buffer. }
  StrPCopy(Buffer, PartialRecvBuffer);
  PartialRecvBuffer := '';

  { Get the data using the Recv method. }
  RecvList.Clear;
  BuffNdx := StrLen(Buffer);
  RecvCnt := (Socket as TStreamSocket).Recv(
               Buffer[BuffNdx], 2047 - BuffNdx);

  { Loop until all bytes received have been processed. }
  Remaining := RecvCnt;
  BuffPtr := Buffer;

  while (Remaining > 0) do begin
    { All server replies end in cr/lf. }
    CrLfPtr := StrPos(BuffPtr, CrLf);

    { If there is no cr/lf, we have a partial reply;
      save it until next OnRead. }
    if (CrLfPtr = nil) then
      begin
        (BuffPtr + Remaining)^ := #0;
        PartialRecvBuffer := StrPas(BuffPtr);
        Remaining := 0;
      end
    else  { If a cr/lf, create a string list entry. }
      begin
        CrLfPtr^ := #0;
        RecvList.Add(StrPas(BuffPtr));
        Remaining := Remaining - (StrLen(BuffPtr) + 2);
        BuffPtr := CrLfPtr + 2;
      end;
  end;

  { Process each of the replies in the string list. }
  for i := 0 to RecvList.Count - 1 do begin
    { Status is first 3 bytes. }
    Status := Copy(RecvList.Strings[i], 1, 3);
    AddDiagnostic('RECV: ' + RecvList.Strings[i]);
    if (FActionCompleted) then
      Continue;  { Ignore if nothing in process. }

    { For each type of command in process, set
      FActionCompleted if status code indicates success. }
    case WaitState of
      wsConnect :
        FActionCompleted := (Status = CONNECT_OK);
      wsUser :
        FActionCompleted := ((Status = USER_OK) or
                             (Status = USER_OK_PASS));
      wsPass :
        FActionCompleted := (Status = PASS_OK);
      wsPasv :
        if (Status = PASV_OK) then
          begin
            { Save reply for FtpPassive. }
            RecvData := RecvList.Strings[i];
            FActionCompleted := True;
          end;
      wsType :
        FActionCompleted := (Status = TYPE_OK);
      wsCwd :
        FActionCompleted := (Status = CWD_OK);
      wsPort :
        FActionCompleted := (Status = PORT_OK);
      wsStor :
        FActionCompleted := ((Status = STOR_OK) or
                             (Status = STOR_OK_OPEN));
      wsSite :
        FActionCompleted := (Status = SITE_OK);
```

```
      wsStorDone :
        FActionCompleted := ((Status = STOR_DONE_OK) or
                             (Status = STOR_DONE_OK_CLOSE));
      wsDele :
        FActionCompleted := (Status = DELE_OK);
    end;
  end;

end;

{ OnWrite handler for all 3 sockets. }
procedure THsSocket.SktWrite(Sender: TObject;
                             Socket: TSocketBase);
begin
  OnWriteFlag := True;  { So SendSktData can continue. }
end;
```

End Listing Three

Begin Listing Four — Data Connection Routines

```
{ Set data connection type to ASCII or binary. }
function THsSocket.FtpSetType(FtpType: TFtpType) : Boolean;
begin
  Result := False;
  SetWaitState(wsType);

  case FtpType of
    ftBinary : SendCommand('TYPE I');
    ftAscii : SendCommand('TYPE A');
  end;

  if (not ActionCompleted) then
    begin
      DisplayError('Unable to set type');
      Exit;
    end;

  Result := True;
end;

{ Set up port-mode data connection. }
function THsSocket.FtpPort : Boolean;
var
  Addr : string;
  DotPos : Integer;
  Msg : string;
  i : Integer;
begin
  { Begin listening. }
  Result := False;
  try
    { Close socket if it is open. }
    if (ListenSkt.ClientCount > 0) then
      ListenSkt.CloseDown;
  except
  end;

  try
    ListenSkt.Port := '0';  { Any port. }
    ListenSkt.Listen(TStreamSocket);
  except
    on E : Exception do begin
      DisplayError('Listen port error: ' + E.Message);
      Exit;
    end;
  end;

  { Build port data as 6 comma-separated fields. }
  SetWaitState(wsPort);
  Addr := CmdSkt.Conn.LocalAddress;
  Msg := '';

  { First 4 fields are dot address fields. }
  for i := 1 to 3 do begin
    DotPos := Pos('.', Addr);
    Msg := Msg + Copy(Addr, 1, DotPos - 1) + ',';
    Addr := Copy(Addr, DotPos + 1, Length(Addr));
  end;

  { Last 2 fields are upper and lower bytes of port. }
  Msg := Msg + Addr + ',';
  Msg := Msg + IntToStr(ListenSkt.Conn.LocalPort div 256) +
         ',' + IntToStr(ListenSkt.Conn.LocalPort mod 256);

  { Send PORT command to tell server where to connect. }
  SendCommand('PORT ' + Msg);
  if (not ActionCompleted) then begin
    DisplayError('Server did not accept PORT command');
    Exit;
  end;

  PasvFlag := False;
  Result := True;
end;

{ Set up passive-mode data connection. }
function THsSocket.FtpPassive : Boolean;
var
  ChrPos : Integer;
  Addr : string;
  Port : Integer;
  i : Integer;
begin
  { Request data socket address/port from server. }
  Result := False;
  SetWaitState(wsPasv);
  SendCommand('PASV');
  if (not ActionCompleted) then
    begin
      DisplayError('Unable to obtain data port using PASV');
      Exit;
    end;

  { IP address is first 4 comma-separated fields
    in reply after open paren. }
  ChrPos := Pos('(', RecvData);
  if (ChrPos = 0) then
    Exit;

  RecvData := Copy(RecvData,ChrPos + 1,Length(RecvData));
  Addr := '';
  for i := 0 to 3 do begin
    ChrPos := Pos(',', RecvData);
    if (ChrPos = 0) then
      Exit;
    Addr := Addr + Copy(RecvData, 1, ChrPos - 1);
    if (i < 3) then
      Addr := Addr + '.';
    RecvData := Copy(RecvData,ChrPos + 1,Length(RecvData));
  end;

  { Port is last two fields. }
  ChrPos := Pos(',', RecvData);
  if (ChrPos = 0) then
    Exit;
  Port := StrToInt(Copy(RecvData, 1, ChrPos - 1)) * 256;
  RecvData := Copy(RecvData, ChrPos + 1, Length(RecvData));
  ChrPos := Pos(')', RecvData);
  if (ChrPos = 0) then
    Exit;
  Port := Port + StrToInt(Copy(RecvData, 1, ChrPos - 1));

  { Open data connection. }
  SetWaitState(wsDataConnect);
  DataSkt.Address := Addr;
  DataSkt.Port := IntToStr(Port);
  try
    CloseSkt(DataSkt);
    DataSkt.Open(TStreamSocket);
  except
    on E : Exception do
```

```
      DisplayError('Data port open error: ' + E.Message);
  end;

  if (not ActionCompleted) then
    begin
      DisplayError('Unable to connect to data port');
      Exit;
    end;
  PasvFlag := True;
  Result := True;
end;

{ Event handler for OnRead of
  passive and port data sockets. }
procedure THsSocket.DoDataSktRead(Socket: TSocketBase);
const
  MAXIDX = 2047;
var
  Buffer : array[0..MAXIDX] of Char;
  Count : Integer;
begin

  case WaitState of
    { List directory/retrieve file command in process. }
    wsList, wsRetr :
      { Write the data to the output file. Reset
        ActionCompleted wait to avoid timeout. }
      try
        Count := Socket.Recv(Buffer, MAXIDX + 1);
        BlockWrite(OutputFile, Buffer, Count);
        ResetTime := True;
      except
        AddDiagnostic('Read Failed');
      end;
  end;

end;
```

End Listing Four

Delphi 1, 2, 3

By Robert Vivrette

Quick on the Draw

A Drawing Program Shows Off Delphi's RAD Abilities

Recently, a reader of *The Unofficial Newsletter of Delphi Users* e-mailed: "How do I create a program that allows me to draw objects on the form, move them around, etc.?" I started thinking about it, then sat down to see what I could design. In just a shade over an hour, I completed a functional drawing program. It can draw, move, and alter shapes, as well as save and load drawings to disk. The entire project was simple to accomplish, primarily because it took advantage of the power already in Delphi.

Basically, the application I came up with is a drawing program. However, I am not speaking of a paint-type program that manages bitmap files. Instead, this program allows you to draw squares, rectangles, circles, etc., that live on the surface of a drawing document. If you don't like where a square is, drag it to a new location. If you want to change its fill pattern, color, or border, you can do that as well.

```
TDrawShape = class(TShape)
  private
    Grabbed : Boolean;
    XX, YY : Integer;
    procedure MouseUp(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
    procedure MouseDown(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
    procedure MouseMove(Shift: TShiftState;
                        X, Y: Integer); override;
  end;

procedure TDrawShape.MouseUp(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Grabbed := False;
end;

procedure TDrawShape.MouseDown(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  BringToFront;
  SelectedShape := Self;
  XX := X; YY := Y;
  Grabbed := True;
end;

procedure TDrawShape.MouseMove(Shift: TShiftState;
                               X, Y: Integer);
begin
  if Grabbed then
    SetBounds(Left+X-XX, Top+Y-YY, Width, Height);
end;
```

Figure 1: Deriving *TDrawShape* from *TShape*.

First, let's analyze what we'll be drawing. As it turns out, Delphi has a graphic object called *TShape*, which is a direct descendent of *TGraphicControl*. A *TShape* object can appear in six shapes:

- a square
- a rounded square
- a rectangle
- a rounded rectangle
- a circle
- an ellipse

Delphi developers needing a graphic shape in their application can take a Shape component, drop it on a form, and set various properties defining its style, fill pattern, color, line pattern, and width.

Enhancing *TShape*

Much of the behavior we want in our draw program is available in *TShape*. First, it knows how to exist on a form, paint itself, change size, manage mouse actions, and modify its appearance. All we really need to do is build a system around *TShape* that allows the user (at run time) to add, move, and delete copies of these *TShape* objects. To add features to *TShape*, I decided to create a new descendant called *TDrawShape*. This descendant will add one new ability to *TShape* — the ability to be dragged on the form at run time. A definition of the *TDrawShape* class is shown in Figure 1.

As you can see, we didn't do much to *TShape* to get *TDrawShape*. We added a private variable

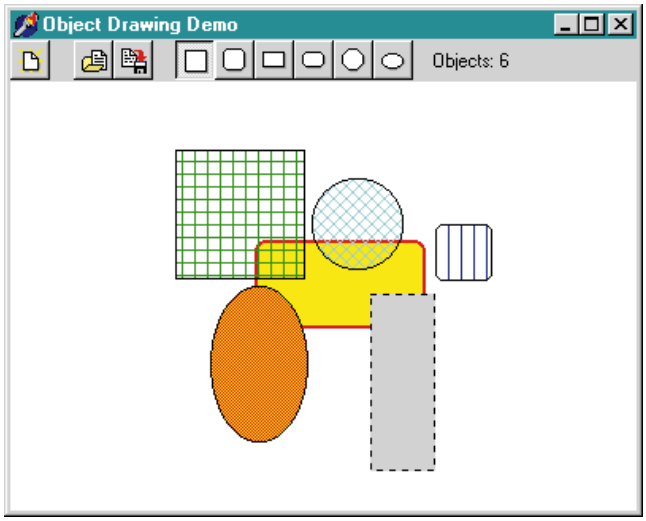


Figure 2: The finished ObjDraw program.

called *Grabbed* that indicates when the user has clicked the mouse over the object and is trying to move it across the form. It also has two additional private variables, *XX* and *YY*, that mark the location where the user grabbed the object. The only other difference is that *TShape*'s three mouse procedures — *MouseUp*, *MouseDown*, and *MouseMove* — have been overridden.

Reviewing the implementation of these three event handlers, you can see that *MouseUp* sets *Grabbed* to *False*. Logically, if the user has released the mouse button, he or she is no longer grabbing the object, right? The *MouseDown* method first calls *BringToFront*. This method (which comes with *TShape*) tells the object to bring itself to the top of its parent's *Controls* list. Because the parent of the shape will be the form, *BringToFront* causes the shape to stack itself above all other shapes on the form.

Next, we assign a reference of this shape to a global variable called *SelectedShape*. This will allow methods of the form to access the selected shape without having to hunt for it. Then we save the *X* and *Y* values of the mouse pointer when the mouse was clicked inside the shape (relative to the top and left of the shape) and set its *Grabbed* value to *True*.

The *MouseMove* method verifies that the shape is selected, and moves it to the current mouse pointer location. To calculate the new location, we take the distance the mouse pointer moved, and add the shape's current *Left* and *Top* values.

The ObjDraw Program

Now let's take a look at the main form. This is where we will manage copies of the *TDrawShape* objects. Listing Five (beginning on page 25) shows the code for this project, so you can follow along as I describe its various capabilities.

As you can see from the finished product in Figure 2, there is a toolbar at the top of the form that accesses the program's primary functions. The first toolbar button, New Document, deletes any shapes on the form. The next two buttons, Open File and Save File, will be described in greater detail later.

Next come buttons controlling the six basic shapes we can create. Only one of these buttons can be down at a time (controlled by setting their *GroupIndex* values to 1). These buttons are linked to *ShapeSelectClick*'s event handler. This method looks at the button's *Tag* property from the generated event, casts it as a *TShapeType*, then saves that value in the *ObjType* variable. *ObjType* will be used to determine what kind of object will be created when the user draws on the form.

The *TShapeType* mentioned is defined in Delphi with *TShape*, and is an enumerated set of *TShape*'s possible styles. When Delphi builds an enumerated type, each element has an ordinal value assigned to it, based on where it was in the type declaration. For example, the definition of *TShapeType* resembles Delphi's Extctrls.pas unit:

```
TShapeType =
   (stRectangle, stSquare, stRoundRect,
    stRoundSquare, stEllipse, stCircle);
```

In this case, *stRectangle* is represented internally as a `0`, *stSquare* as a `1`, *stRoundRect* as a `2`, and so on. Therefore, to have all the shape buttons use the same event handler, I assign this ordinal value to the appropriate button's *Tag* property. To determine which shape is being requested, I typecast that *Tag* value to a *TShapeType* and save the result. (This handy trick is used in a few places in this project.)

Listening to Messages

So the user has clicked one of the shape-selection buttons and a *TShapeType* value has been stored in the variable *ObjType*. If we want the user to be able to click and drag a new copy of this shape onto the form, we must listen in on the form's mouse messages.

First, we take care of the form's *MouseDown* event. Here, we verify that it's the left mouse button that generated the event, save the mouse pointer's current *x* and *y* coordinates, then set *Creating* to *True*. This will tell the other mouse messages that we're creating a new shape.

Next comes the *MouseMove* event. We must verify the left mouse button is down, and the *Creating* flag is *True*. If these two conditions are met, make sure the *x* and *y* coordinates haven't moved left or above (respectively) of the original location of the mouse pointer. This stops the user from creating an object flipped backwards or upside down. After limiting the *X* and *Y* values, draw a "rubber-banding line" with a local procedure I call *DrawGhost*. This is a technique you see in many applications — including Windows itself. When you're on the Windows desktop and you want to select a group of objects, you can click and drag a selection rectangle around those objects. This is really nothing more than one of these rubber-banding lines. In our case, we want to draw the selected shape, but with a pen style of *psDot* and a pen mode of *pmXOr*. These values are properties of the form's canvas and are initialized to these values in the *FormCreate*.

The *psDot* pen style causes the rubber-banding line to draw dotted, and the *pmXOr* pen mode causes it to be drawn in

XOR mode. *XOR* mode is a neat trick you can do with Windows to draw a line and remove it from the form. In a nutshell, if you draw a line in *XOR* mode, you will see the line on the form. If you draw it again in the same location, it will restore the form to its original appearance. Additionally, an *XOR*ed line can be drawn on any color form and it will always appear.

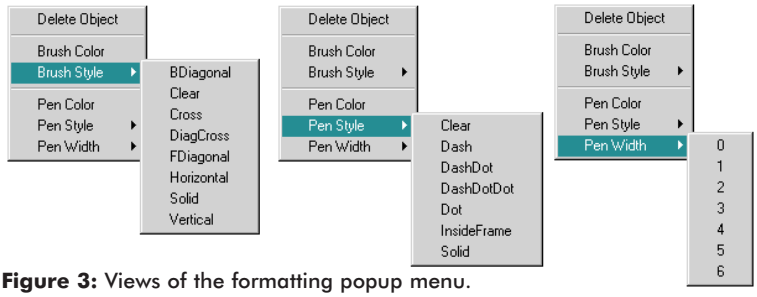


Figure 3: Views of the formatting popup menu.

Draw the object *XOR*ed once (by means of the *DrawGhost* procedure) to remove the object from its old location, then update the mouse coordinates and draw it again at its new location. As the mouse continues to move, we continue this "Undraw/Update/Redraw" sequence, until the user releases the mouse button.

This brings us to the form's *MouseUp* event. Verify the left mouse button was released and the *Creating* flag was set to *True*. Next, verify the mouse moved at least one pixel in both the *x* and *y* axis between *MouseDown* and *MouseUp*. This ensures that we don't create any shapes without width or height. Next, undraw the final piece of the rubber-banding image using the *DrawGhost* procedure.

Creating the Shape

Now it's time to create the instance of *TDrawShape*. To do this, we instantiate a copy of *TDrawShape* and assign it to the variable *NewObj*. Then we set various properties of *NewObj*:

- the *Parent* property is set to the main form;
- the *Shape* property is set to the value we saved when the user clicked the shape selection buttons;
- the *Cursor* property is set to *crHandPoint;*
- the *PopupMenu* property is set to *FormatMenu*.

The *TPopupMenu* has been placed on the form to allow various states of each shape to be changed. Because we don't want one popup menu for each shape, connect each shape created to the single popup menu on the main form. This ensures the cursor will automatically change to a hand pointer when a user passes over any shape on the form (because of the assignment to *NewObj*'s *Cursor* property). It also allows users to access a right-click formatting menu. (More on the menu a little later.)

The last few steps in *MouseUp* are to set the shape's bounds (keeping the width and height equal in the case of squares, rounded squares, and circles), and call a function to count the number of objects currently on the form (to update the "Objects" label at the top of the application). This function loops through the form's *Controls* array and counts the instances of *TDrawShape*.

A Few Extras

With the steps covered so far, we've built a functional shape-drawing program. The user can choose a shape from one of the buttons on the toolbar, and can click and drag a copy of that shape on the form. This portion is handled through the form's mouse methods that we supplanted. Once the shape has been created, the user can grab the shape and move it around the form. This portion is handled through the mouse methods we altered in *TDrawShape*.

We can do even better with just a little extra effort. As I mentioned earlier, there is a formatting menu; its contents are shown in Figure 3. This menu is linked to every shape placed on the form. When the user right-clicks on the shape, this menu is displayed. Note that when the user right-clicks on a shape, it moves to the top of the form and becomes the selected shape. A reference to this shape is kept in the *SelectedShape* variable.

The first item on the menu is **Delete Object**, which calls the *Free* method of the *SelectedShape* object. Because a shape has been deleted, it then needs to call *CountObjects* to update the caption on the toolbar. The **Brush Color** and **Pen Color** menu items call one of Delphi's ColorDialog components. The color of the brush or pen is passed into the dialog box (i.e. it's shown), and the color chosen by the user is then fed back into the shape's pen or brush *Color* property. This is handled through the *BrushColorClick* and *PenColorClick* methods, respectively.

The brush and pen styles are handled in a similar fashion to the way we selected the shape from the buttons on the toolbar. Each menu item under these sub-menus has a *Tag* property that corresponds to the ordinal value of that style. Then it's a simple matter of typecasting the chosen tag as a *TBrushStyle* or a *TPenStyle*, and assigning that value to the appropriate property in the currently selected shape.

The **Pen Width** menu items have the width itself in the menu item's *Tag* property, and are likewise copied into the currently selected shape's *Pen.Width* property. Note that the menu items in each sub-menu point to the same event handler. For example, all the **Pen Style** items point to the *PenStyleClick* event handler. This is a good example of how to take advantage of the power and flexibility of the *Sender* parameter in event handlers.

Saving and Retrieving the Shapes

A drawing program wouldn't be of much use if you couldn't save and load drawings. As it turns out, this too is simple to accomplish. Create an instance of a *TFileStream* object (specifying the name of the file) and read or write the appropriate objects to or from that stream. In the case of the reading from the stream, we present one of Delphi's OpenDialog components, and allow the user to pick a previously saved file. The dialog has a filter pre-defined for "DrawObj Files" using the

wildcard *.obd. After the user picks a file, all existing objects on the form are deleted and the *TFileStream* object is created.

Next, go into a loop that continues as long as the position of the *TFileStream* pointer is before the end of the file (meaning there is data left to read):

```
while FS.Position < FS.Size do begin
  NewObj := TDrawShape.Create(Self);
  NewObj.Parent := Self;
  NewObj.PopupMenu := FormatMenu;
  FS.ReadComponent(NewObj);
end;
```

If there is something to read, we create a new instance of a *TDrawShape* object, set the parent as the main form, and call Delphi's *ReadComponent* method to read its component data from the stream. When we reach the end of the file, the *TFileStream* object is destroyed.

To save the objects, apply the same logic in reverse. Present a SaveDialog component, obtain a file name, then create the *TFileStream* object. Loop through the form's *Controls* array, and for each *TDrawShape* object found, use *WriteComponent* to tell the component to write itself out to the stream:

```
for a := 0 to ControlCount-1 do
  if Controls[a] is TDrawShape then
    FS.WriteComponent(Controls[a] as TDrawShape);
```

Conclusion

As you can see, it doesn't take many brain cells to put together this simple drawing program. This is because we're taking advantage of Delphi's capabilities. Delphi includes the *TShape* component, which we use as the basis of the objects we create. It also provides simple streaming capabilities to allow us to save and retrieve groups of objects.

Could we go further with this project? Absolutely! There are a number of things I can come up with off the top of my head. The first would be to allow resizing of objects once they have been placed. Ideally, this should be integrated at the *TDrawShape* level so the remaining programs' behavior wouldn't require changes. We may want to add other features, such as plain lines or floating text objects. The topics covered in this article will give you a good start toward adding some of these features yourself.

This just goes to show that with Delphi's RAD capabilities, anyone can be quick on the draw. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\DEC\DI9712RV.

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

Begin Listing Five — The ObjDraw project

```
unit Unit1;

interface

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

type
  TDrawShape = class(TShape)
  private
    Grabbed: Boolean;
    XX,YY  : Integer;
    procedure MouseUp(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
    procedure MouseDown(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
    procedure MouseMove(Shift: TShiftState;
                        X, Y: Integer); override;
  end;

  TForm1 = class(TForm)
    FormatMenu: TPopupMenu;
    mnuDelete: TMenuItem;          // Main popup menu items.
    mnuSep1: TMenuItem;
    mnuBrushColor: TMenuItem;
    mnuBrushStyle: TMenuItem;
    mnuSep2: TMenuItem;
    mnuPenColor: TMenuItem;
    mnuPenStyle: TMenuItem;
    mnuPenWidth: TMenuItem;
    mnuBrushBDiag: TMenuItem;     // Brush Style menu items.
    mnuBrushClear: TMenuItem;
    mnuBrushCross: TMenuItem;
    mnuBrushDiagCross: TMenuItem;
    mnuBrushFDiag: TMenuItem;
    mnuBrushHorizontal: TMenuItem;
    mnuBrushSolid: TMenuItem;
    mnuBrushVertical: TMenuItem;
    mnuPenClear: TMenuItem;         // Pen Style menu items.
    mnuPenDash: TMenuItem;
    mnuPenDashDot: TMenuItem;
    mnuPenDashDotDot: TMenuItem;
    mnuPenDot: TMenuItem;
    mnuPenInsideFrame: TMenuItem;
    mnuPenSolid: TMenuItem;
    Width0: TMenuItem;               // Pen Width menu items.
    Width1: TMenuItem;
    Width2: TMenuItem;
    Width3: TMenuItem;
    Width4: TMenuItem;
    Width5: TMenuItem;
    Width6: TMenuItem;
    pnlToolbar: TPanel;
    spdNew: TSpeedButton;           // Toolbar buttons.
    spdOpenFile: TSpeedButton;
    spdSaveFile: TSpeedButton;
    spdSquare: TSpeedButton;
    spdSquareR: TSpeedButton;
    spdRect: TSpeedButton;
    spdRectR: TSpeedButton;
    spdCircle: TSpeedButton;
    spdEllipse: TSpeedButton;
    lblCount: TLabel;
    dlgColor: TColorDialog;
    dlgSave: TSaveDialog;
    dlgOpen: TOpenDialog;
    procedure FormCreate(Sender: TObject);
    procedure FormMouseUp(Sender: TObject;
      Button: TMouseButton; Shift: TShiftState;
      X, Y: Integer);
    procedure FormMouseDown(Sender: TObject;
      Button: TMouseButton; Shift: TShiftState;
      X, Y: Integer);
    procedure FormMouseMove(Sender: TObject;
      Shift: TShiftState; X, Y: Integer);
    procedure DrawGhost;
    procedure ShapeSelectClick(Sender: TObject);
    procedure DeleteClick(Sender: TObject);
    procedure BrushColorClick(Sender: TObject);
```

```
    procedure BrushStyleClick(Sender: TObject);
    procedure PenColorClick(Sender: TObject);
    procedure PenStyleClick(Sender: TObject);
    procedure PenWidthClick(Sender: TObject);
    procedure CountObjects;
    procedure DeleteObjects;
    procedure spdNewClick(Sender: TObject);
    procedure spdOpenFileClick(Sender: TObject);
    procedure spdSaveFileClick(Sender: TObject);
  private
    ObjType   : TShapeType;
    NewObj    : TDrawShape;
    X1,Y1     : Integer;
    X2,Y2     : Integer;
    Creating  : Boolean;
    FS        : TFileStream;
  end;

var
  Form1: TForm1;
  SelectedShape : TDrawShape;

implementation

{$R *.DFM}

function Min(A, B: Integer): Integer;
begin
  if A < B then
    Result := A
  else
    Result := B;
end;

procedure TDrawShape.MouseUp(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Grabbed := False;
end;

procedure TDrawShape.MouseDown(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  BringToFront;
  SelectedShape := Self;
  XX := X; YY := Y;
  Grabbed := True;
end;

procedure TDrawShape.MouseMove(Shift: TShiftState;
                               X, Y: Integer);
begin
  if Grabbed then
    SetBounds(Left+X-XX, Top+Y-YY, Width, Height);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Canvas.Pen.Mode := pmXOr;
  Canvas.Pen.Style := psDot;
  ObjType := stCircle;
  CountObjects;
  Creating := False;
  ShapeSelectClick(spdSquare);
end;

procedure TForm1.FormMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  L,T,W,H : Integer;
begin
  if not Creating then
    Exit;
  if Button <> mbLeft then
    Exit;

  DrawGhost;

  if (X<=X1) or (Y<=Y1) then
    Exit;

  NewObj := TDrawShape.Create(Self);
  with NewObj do begin
    Parent := Self;
    Shape := ObjType;
    Cursor := crHandPoint;
    PopupMenu := FormatMenu;
    L := X1; W := X-X1;
    T := Y1; H := Y-Y1;
    if ObjType in [stCircle,stSquare,stRoundSquare] then
      if W < H then
        H := W
      else
        W := H;
    SetBounds(L,T,W,H);
  end;

  CountObjects;
  Creating := False;
end;

procedure TForm1.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button <> mbLeft then
    Exit;
  X1 := X; Y1 := Y; X2 := X; Y2 := Y;
  Creating := True;
end;
procedure TForm1.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  if Creating and (ssLeft in Shift) then
    begin
      if X <= X1 then
        X := X1+1;
      if Y <= Y1 then
        Y := Y1+1;
      DrawGhost;  // Undraw the last image.
      X2 := X; Y2 := Y;
      DrawGhost;  // Draw the new image.
    end;
end;

procedure TForm1.DrawGhost;
var
  S : Integer;
begin
  S := Min(X2-X1,Y2-Y1);
  with Canvas do
    case ObjType of
      stCircle :
        Arc(X1,Y1,X1+S,Y1+S,X1,Y1,X1,Y1);
      stEllipse :
        Arc(X1,Y1,X2,Y2,X1,Y1,X1,Y1);
      stSquare,
      stRoundSquare :
        begin
          PolyLine([Point(X1,Y1),Point(X1+S,Y1),
                    Point(X1+S,Y1+S)]);
          PolyLine([Point(X1,Y1),Point(X1,Y1+S),
                    Point(X1+S,Y1+S)]);
        end;
      stRectangle,
      stRoundRect :
        begin
          PolyLine([Point(X1,Y1),Point(X2,Y1),
                    Point(X2,Y2)]);
          PolyLine([Point(X1,Y1),Point(X1,Y2),
                    Point(X2,Y2)]);
        end;
    end;
end;

procedure TForm1.ShapeSelectClick(Sender: TObject);
begin
  ObjType := TShapeType((Sender as TSpeedButton).Tag);
end;

procedure TForm1.DeleteClick(Sender: TObject);
begin
  SelectedShape.Free;
  CountObjects;
end;
```

```
procedure TForm1.PenColorClick(Sender: TObject);
begin
  dlgColor.Color := SelectedShape.Pen.Color;
  if dlgColor.Execute then
    SelectedShape.Pen.Color := dlgColor.Color;
end;

procedure TForm1.PenStyleClick(Sender: TObject);
begin
  SelectedShape.Pen.Style :=
    TPenStyle((Sender as TMenuItem).Tag);
end;

procedure TForm1.PenWidthClick(Sender: TObject);
begin
  SelectedShape.Pen.Width := (Sender as TMenuItem).Tag;
end;

procedure TForm1.BrushColorClick(Sender: TObject);
begin
  dlgColor.Color := SelectedShape.Brush.Color;
  if dlgColor.Execute then
     SelectedShape.Brush.Color := dlgColor.Color;
end;

procedure TForm1.BrushStyleClick(Sender: TObject);
begin
  SelectedShape.Brush.Style :=
    TBrushStyle((Sender as TMenuItem).Tag);
end;

procedure TForm1.CountObjects;
var
  a, b : Integer;
begin
  b := 0;
  for a := 0 to ControlCount-1 do
    if Controls[a] is TDrawShape then
       Inc(b);
  lblCount.Caption := 'Objects: ' + IntToStr(b);
end;

procedure TForm1.DeleteObjects;
var
  a : Integer;
  OneDeleted : Boolean;
begin
  repeat
    OneDeleted := False;
    for a := 0 to ControlCount-1 do
      if Controls[a] is TDrawShape then
        begin
          (Controls[a] as TDrawShape).Free;
          OneDeleted := True;
          Break;
        end;
  until not OneDeleted;

  CountObjects;
end;

procedure TForm1.spdNewClick(Sender: TObject);
begin
  DeleteObjects;
end;

procedure TForm1.spdOpenFileClick(Sender: TObject);
begin
  dlgOpen.FileName := '';

  if dlgOpen.Execute then
    begin
      DeleteObjects;
      FS := TFileStream.Create(dlgOpen.FileName,
              fmOpenRead or fmShareDenyWrite);
      try
        while FS.Position < FS.Size do begin
          NewObj := TDrawShape.Create(Self);
          NewObj.Parent := Self;
          NewObj.PopupMenu := FormatMenu;
          FS.ReadComponent(NewObj);
        end;
      finally FS.Free;
  end;
    end;
  CountObjects;
end;

procedure TForm1.spdSaveFileClick(Sender: TObject);
var
  a : Integer;
begin
  if dlgSave.Execute then
    begin
      FS := TFileStream.Create(dlgSave.FileName,
              fmCreate or fmShareDenyWrite);
      try
        for a := 0 to ControlCount-1 do
          if Controls[a] is TDrawShape then
             FS.WriteComponent(Controls[a] as TDrawShape);
      finally FS.Free;
      end;
    end;
end;

end.
```

End Listing Five

BDE API / Delphi 1, 2, 3

By Bill Todd

Using the BDE API

Part II: Troubleshooting and Problem Solving

Although much of the Borland Database Engine's API is surfaced in Delphi, some useful tools are only available by calling the BDE API directly. Last month's installment of this two-part series covered routine tasks such as table packing and copying, along with group record deletion and creating permanent aliases. This month, we'll explore the use of Paradox tables with read-only media, how to use persistent locks as semaphores, sorting local tables, and more.

Using Paradox Tables on Read-Only Media

The problem with trying to put Paradox tables on read-only media (such as a CD-ROM) is that the BDE must create and write to a lock-control file named Pdoxusrs.lck, which must be created in the directory that contains the tables. Because it can't be created at run time when the tables reside on a read-only device, you must create a special version of this file to include with the tables. This special version of Pdoxusrs.lck is called a *directory-lock* file. When the BDE sees this file, it assumes all tables in the directory are read-only, and doesn't try to write to the lock file. While the 32-bit BDE should detect read-only media without this lock file, the 16-bit BDE will not. In any event, creating the file doesn't cause any problems with the 32-bit BDE.

To create the directory-lock file, create a new project. Add a Database component and a Button component to the form. Modify the Database component's *DatabaseName* property to `C:\Lockdir` where C:\Lockdir is the directory in which you want the lock file created. Then set its *Connected* property to *True*. Now add the following code to the Button component's *OnClick* event handler:

```
Check(DbiAcqPersistTableLock(
        Database1.Handle,
          'PARADOX.DRO', 'PARADOX'));
```

where *Database1* is the name of the Database component. (Note: As discussed last month, you must include the BDE source files: In Delphi 1, the DbiProcs, DbiTypes, and DbiErrs units must be included in the **uses** clause; in Delphi 2 and 3, the BDE unit must be included.)

When you run the program and press the button, two files — Pdoxusrs.lck and Paradox.lck — will be written to the specified directory. Delete Paradox.lck, then copy Pdoxusrs.lck to the directory that contains the tables to be placed on the CD-ROM, so it will be included with the tables.

Using Persistent Locks as Semaphores

One difficult problem to solve in a database application is how to communicate between multiple users on different workstations. For example, suppose you have a system that requires some initialization code to be run by the first workstation that starts the system each day, but you don't want this code to be run more than once. One solution is to have the first workstation update a record in a table, and have the others read the table and check the date in the record.

However, there's a faster way that requires less network traffic and disk I/O. You can place a persistent, exclusive lock on a table that doesn't exist, then simply have the program, upon start-up, try to lock the non-existent table. If

```
procedure dgGetNetFileUsers(UserList: TStringList);
var
  UserCur: hDbiCur;
  pUserDes: pUserDesc;
begin
  Check(DbiOpenUserList(UserCur));
  GetMem(pUserDes, SizeOf(UserDesc));
  try
    FillChar(pUserDes^, SizeOf(UserDesc), #0);
    while (DbiGetNextRecord(UserCur, DbiNoLock,
                            pUserDes, nil) = DBIERR_NONE) do
      UserList.Add(StrPas(pUserDes^.szUserName));
  finally
    FreeMem(pUserDes, SizeOf(UserDesc));
    DbiCloseCursor(UserCur);
  end;
end;
```

Figure 1: Getting a list of users.

the lock succeeds, the program runs the initialization code. If the lock fails, another workstation is running the program, and the initialization code can be skipped.

To place a persistent, exclusive lock, call:

```
DbiAcqPersistTableLock(Db.Handle, 'foo.db', '');
```

where the first parameter is the *Handle* property of a *TDatabase* object pointing to the directory that contains the non-existent table (the directory where the lock file will be created). The second parameter is the name of the table. The third parameter is the driver type, and is optional if you include a file extension on the table name. To release the lock, call:

```
DbiRelPersistTableLock(Db.Handle, 'foo.db', '');
```

using the same parameters as in the call to **DbiAcqPersistTableLock**.

Getting a List of Network Users

If you have an application that uses Paradox tables on a network, you can get a list of all the users of a network control file by calling the function shown in Figure 1. The **var** block defines a cursor variable and a pointer to a user-descriptor structure. The call to **DbiOpenUserList** opens the cursor to the list of users in the network control file, the call to *GetMem* allocates memory to hold the user descriptor structure, and the call to *FillChar* initializes it to binary zero. **DbiGetNextRecord** reads the next record from the user list. The statement:

```
UserList.Add(StrPas(pUserDes^.szUserName));
```

extracts the *UserName* value from the user descriptor, converts it to a Pascal string, and adds it to the *UserList* string list passed to the function as a parameter. The **while** loop continues until **DbiGetNextRecord** returns an error code, indicating the end of the list. Finally, the memory for the descriptor is released and the cursor is closed.

Sorting Local Tables

Although Delphi lets you, by using different indices, view a table in any order you wish, it does not surface the BDE's ability to physically sort a table. If you sort an unkeyed table,

```
procedure dgSortTable(
  { Database that contains table to sort. }
  Db: TDatabase;
  { TTable to sort. }
  SortTbl: TTable;
  { Numbers of fields to sort on. }
  const SortFields: array of Word;
  { True for case-insensitive sort. }
  const SortCase: array of Bool;
  { sortASCEND or sortDESCEND. }
  const SortOrd: array of SORTOrder;
  { True to delete duplicate records. }
  RemoveDups: Boolean);
{ Sorts an unkeyed table into itself. The TDatabase passed
  as the first parameter must be connected. The TTable
  passed as the second parameter must be closed. If an
  exclusive lock cannot be obtained on the table, an
  EDBEngineError exception will be raised. }
var
  { Handle of database holding table. }
  hDb: hDBIdb;
  { Table name. }
  pTblName: array [0..DBIMAXPATHLEN] of Char;
  { Number of records to sort. }
  NumRecs: LongInt;
  { Numbers of fields to sort on. }
  SortFldCount: Word;
begin
  hDb := Db.Handle;
  StrPCopy(pTblName, SortTbl.TableName);
  SortFldCount := High(SortFields) + 1;
  Check(DbiSortTable(hDB, pTblName, szPARADOX, nil, nil,
                     nil, nil, SortFldCount, @SortFields,
                     @SortCase, @SortOrd, nil, RemoveDups,
                     nil, NumRecs));
end;
```

Figure 2: Sorting a table.

you can sort it *in place*; that is, the sorted result can be placed in the same table. Because keyed tables must be maintained in order by their primary keys when you sort a keyed table, the sorted records must be placed in a new table. The *dgSortTable* function shown in Figure 2 lets you sort an unkeyed table.

This function takes the following six parameters:

- *Db*: A *TDatabase* component connected to the database that contains the table you want to sort. The *Connected* property of the *TDatabase* must be set to *True.*
- *SortTbl*: A *TTable* component that contains the name of the table you want to sort. The *TTable* must be closed, because the **DbiSortTable** function that performs the sort must be able to open the table for exclusive use.
- *SortFields*: An array of type Word that contains one element for each field to be sorted. The elements contain the field numbers in the order they are to be used for the sort. For example, if the first field you want to sort on is the third field in the table's structure, then the value of the first element in the array would be 3. Note that the number of the first field in the table's structure is one, not zero.
- *SortCase*: An array of type Bool with one element for each field to be sorted. If the value of an element is *True*, then the sort on the corresponding field in *SortFields* is case-insensitive. If the value is *False*, the sort on that field is case-sensitive. The case-sensitivity setting affects only string fields.
- *SortOrd*: An array of type *SORTOrder* that specifies whether the fields in the *SortFields* array should be sorted in ascending or descending order. To sort a field

in ascending order, assign the constant *sortASCEND* to that field's element of the array. To sort in descending order, use *sortDESCEND*.
- *RemoveDups*: If *True*, then duplicate records will be removed. If *False*, duplicate records will be retained. Records are compared on the fields specified in *SortFields* to determine if they're duplicates.

The function begins by assigning the Database component's BDE handle to the *hDb* variable, and converting the table's name to a null-terminated string, then storing it in *pTblName*. The next line determines the number of fields used for the sort, by using the *High* function to get the highest value of the index of the *SortFields* array. Because the array is zero-based, the number of elements in the array is the highest index value, plus one. This works correctly because no matter how the array is declared inside the procedure, the open array parameter behaves as a zero-based array.

DbiSortTable is called using the *Check* procedure to raise an exception if an error occurs. The **nil** parameters are:
- *hSrcCur*: A cursor handle to the table to be sorted. You can use this to specify the table instead of using the table-name and driver-type parameters.
- *pszSortedName*: The name of the destination table if the sorted output is being placed in a new table.
- *phSortedHandle*: A cursor handle to the table that received the sorted output is returned in this parameter.
- *hDestCursor*: If you are sorting into a destination table other than the source table, you can pass a cursor handle to the destination table instead of identifying it by name and driver type. This parameter would be the *Handle* property of a *TTable* component.
- *ppfsortFn*: An array of pointers to functions that perform the comparison between fields to determine which is greater.
- *hDuplicatesCur*: If *RemoveDups* is *True*, you can provide a cursor to a table where you want any moved duplicate records to be placed.

Editing the BDE Configuration File

One problem with distributing Delphi programs that employ the BDE is that you may need to ensure that certain settings have been made in the BDE configuration program. While you can give the end user instructions for using the BDE configuration program, it's much easier — and more reliable for the user — if you can change any necessary settings under program control. This is particularly useful if you use an installation program that allows you to call custom DLLs as part of the installation process.

The BDE API function that provides access to the BDE configuration file is **DbiOpenCfgInfoList**; it takes the following parameters:
- *HCfg*: The configuration-file handle. This parameter must be **nil**.
- *eOpenMode*: Can be either *DbiReadWrite* or *DbiReadOnly*.
- *eConfigMode*: Must be *cfgPersistent*; it's the only valid value.
- *pszCfgPath*: The path to the node in the configuration file to work with.
- *phCur*: A cursor handle initialized by the call; this gives you a handle to the specified node in the configuration file.

Working with the BDE configuration file is different from working with tables, because the configuration file has a hierarchical structure much like the structure of the Windows 95/NT registry. When you open the configuration file, you must specify a path to a node in the hierarchy. You can then treat the nodes that are immediately below the node you specified as though they're records in a table, and read and write them.

However, there are restrictions on updating the nodes. First, you can update only bottom-level (leaf) nodes; that is, nodes that don't have any nodes below them in the hierarchy. Second, the only thing you can change about a leaf node is its value.

Suppose you want to set the value of the Local Share option on the System page of the BDE configuration program to *True*. The path to the node that contains the Local Share parameter is \SYSTEM\INIT. The root node of the configuration file is identified with a single backslash, and the nodes below it by their names. The *dgUpdateBDEConfig* procedure shown in Listing Six (on page 31) lets you update an entry in the BDE configuration file.

The procedure begins by declaring three variables. The first, *hCur*, is of type *hDbiCur*, and provides the cursor to the configuration file. The second variable, *pDesc*, is of type *CfgDesc*. This structure contains variables that, in turn, contain the information about each entry in the configuration file. (For a detailed description of the descriptor, see "CfgDesc" in the BDE online Help.) The call to *FillChar* fills the descriptor structure with nulls. The third variable, *pPath*, holds the path to the desired file node as a null-terminated string.

The call to **DbiInit** initializes the BDE session. Next, the path passed as a parameter is converted to a null-terminated string by calling *StrPCopy*. The statement that follows is the call to **DbiOpenCfgInfoList**. After this call, *hCur* is a cursor to the records (nodes) below the node specified in the *Path* parameter.

Next comes the **while** loop that reads each node by calling **DbiGetNextRecord**. The first parameter to **DbiGetNextRecord** is the open cursor to the configuration file. The second parameter specifies the type of record lock required: *DbiNoLock*, *DbiReadLock* or *DbiWriteLock*. In this case, a write lock is requested so the record can be modified. The next statement converts the *szNodeName* field of the descriptor structure to a Pascal string, and compares it to the *CfgNode* parameter passed to this procedure.

Make sure the parameter you pass matches the node name you're searching for exactly. If this is the desired node, the call to *StrPCopy* converts the new value passed to the procedure to a null-terminated string, and places it in the *szValue* field of the descriptor.

\SYSTEM\INIT\	\DRIVERS\PARADOX\INIT\	\DRIVERS\PARADOX\TABLE CREATE\
LOCAL SHARE	NET DIR	LEVEL
MINBUFSIZE	LANGDRIVER	BLOCK SIZE
MAXBUFSIZE		STRICTINTEGRITY
MAXFILEHANDLES		
LANGDRIVER		
AUTO ODBC		
DEFAULT DRIVER		

Figure 3: Parameter paths and names.

The call to **DbiModifyRecord** updates the record. The first two parameters to **DbiModifyRecord** are the cursor and descriptor structures, respectively. The third is a Boolean value that specifies whether the lock is to be released after the record is updated. Setting this parameter to *True* releases the lock. The code in the **finally** block closes the cursor and ends the BDE session.

By now you must be wondering how to find the paths and names to use for the various parameters you may want to change; the table in Figure 3 contains the ones you're most likely to find useful.

Further Investigation

If you need a BDE-programming technique not described in this series, you'll have to do a little detective work; the paths and parameter names aren't documented.

However, if you call **DbiOpenCfgInfoList** with a path of "\", you can then call **DbiGetNextRecord** in a loop to retrieve the name of all the first-level nodes. You can do the same for each of the first-level nodes, to display the names of the second-level nodes, and so on until you've displayed the entire tree. The sample program also includes a function that returns the value of a parameter in the BDE configuration file. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\DEC\DI9712BT.

Bill Todd is President of The Database Group, Inc., a Phoenix-area consulting and development company. He is a Contributing Editor of *Delphi Informant*; 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.

Begin Listing Six — Modify the BDE configuration file

```
procedure dgUpdateBDEConfig(CfgPath, CfgNode,
                            CfgValue: string);
{ Updates an entry in the BDE configuration file.
  Parameters:
   CfgPath:  Path to the node you want to change.
   CfgNode:  Name of the node you want to change.
   CfgValue: New value for the node.

 The following lists the path to some useful nodes.
 \SYSTEM\INIT
                  LOCAL SHARE
                  MINBUFSIZE
                  MAXBUFSIZE
                  MAXFILEHANDLES
                  LANGDRIVER
                  AUTO ODBC
                  DEFAULT DRIVER
 \DRIVERS\PARADOX\INIT
                  NET DIR
                  LANGDRIVER
 \DRIVERS\PARADOX\TABLE CREATE
                  LEVEL
                  BLOCK SIZE
                  STRICTINTEGRITY
}
var
  hCur:  hDbiCur;  { Cursor handle. }
  pDesc: CfgDesc;  { Configuration descriptor. }
  pPath: array [0..DBIMAXPATHLEN] of Char;
begin
  FillChar(pDesc, SizeOf(CfgDesc), #0);
  { Initialize the BDE session. }
  Check(DbiInit(nil));
  { Convert path to null terminated string. }
  StrPCopy(pPath, CfgPath);
  { Open the configuration file. }
  Check(DbiOpenCfgInfoList(nil, DbiReadWrite,
                           cfgPersistent, pPath, hCur));
  try
    { Read until you find the right node. }
    while DbiGetNextRecord(hCur, DbiWriteLock, @pDesc,
                           nil) = DBIERR_NONE do begin
      if UpperCase(StrPas(pDesc.szNodeName)) =
                                  UpperCase(CfgNode) then
        begin
          { Put the new value in the descriptor. }
          StrPCopy(pDesc.szValue, CfgValue);
          { Update the record. }
          Check(DbiModifyRecord(hCur, @pDesc, True));
          Break;
        end; { if }
    end; { while }
  finally
    { Close the configuration file's cursor. }
    Check(DbiCloseCursor(hCur));
    { End the BDE session. }
    DbiExit;
  end;  { try }
end;
```

End Listing Six

By Cary Jensen, Ph.D.

Combine and Conquer

An Introduction to Component Templates

Delphi 3 allows you to group one or more components, along with any event handlers you've assigned to them, and add them to your Component palette. Such collections of objects are called *component templates*, and they provide yet another tool for quickly creating sophisticated forms and dialog boxes.

While component templates can be useful in a number of situations, they're not a replacement for traditional component development. This month's "DBNavigator" discusses how to create component templates, how to use them, and when you should forgo them for true objects.

You create a component template by selecting one or more components you've placed onto a form, then selecting **Component | Create Component Template** to display the Component Template Information dialog box (see Figure 1). Enter the name of the template, choose the Component-palette page on which it will appear, and optionally define a custom bitmap that will represent the component on that page.

After you've created a component template, you can quickly duplicate the components that make up the template by selecting the icon representing the component template from the Component palette, and dropping it onto a form or data module. The components placed on the form duplicate the relative positions of the objects originally saved, as well as any design-time properties and event handlers assigned to them.

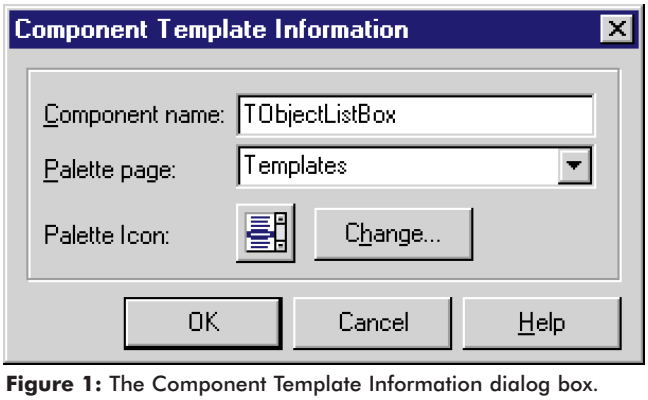


Figure 1: The Component Template Information dialog box.

Component templates are similar to the templates in the Object Repository. Those in the repository, however, must be based on a container, such as a form, data module, or QuickRep component. Because of this, the Object Repository is not well suited for small groups of related components that will normally appear in a container with other components.

The following example demonstrates how to create a component template that has the basic elements for displaying data in a DBGrid:

1) Create a new form; it can be associated with an existing project, or you can create it as a form outside of a project (it will be discarded at the end of the example anyway).
2) Add a Panel component and set its *BevelOuter* property to *bvNone*, and its *Align* property to *alClient*.
3) Add a DBNavigator and DBGrid component, making sure they are inside the panel. Set the DBNavigator's *Align* property to *alTop*, and the DBGrid's *Align* property to *alClient*.
4) Add a DataSource and Table component. Set the DataSource's *DataSet* property to *Table1*, and the DBNavigator's and the DBGrid's *DataSource* property to *DataSource1*. Your form should look like Figure 2.

You are now ready to save these components and their properties as a template. Select **Edit | Select All** to select all the components on the form. Next, select **Component | Create**

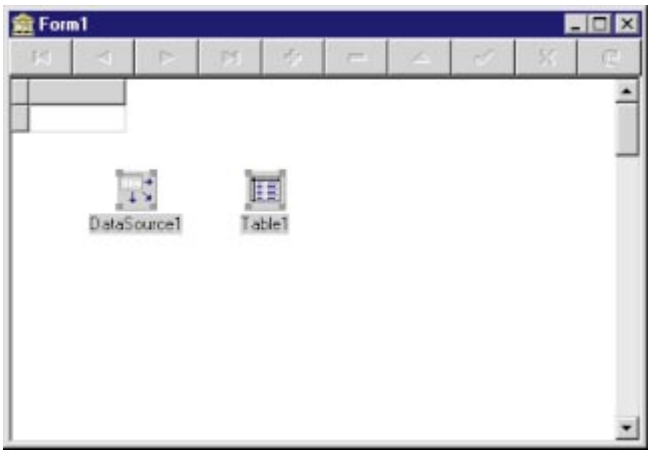


Figure 2: Creating a component template.

Figure 3: The controls on this form were placed from the component template.

Component Template to display the Component Template Information dialog box. In Component name, enter `PanelTemplate`, and select OK to save the template.

To demonstrate the use of the new template, close the form you've been working with (it's not necessary to save it). Next, select File | New Application to create a new application. Go to the Templates page of the Component palette, select the PanelTemplate component, and drop it onto the new project's form. All components that were part of the template are placed on your form. Furthermore, the properties you set at run time are already defined for these objects. To turn this form into a usable table view, set the *DatabaseName* property of *Table1* to `DBDEMOS`, the *TableName* property to `CUSTOMER.DB`, and the *Active* property to *True*. Now run the form. This form, similar to the one shown in Figure 3, displays the data from the Customer table, and permits you to edit it.

Creating Templates with Event Handlers

While the preceding example demonstrated the ease with which you can create customized sets of components, it didn't demonstrate all the power of this technique. Specifically, this example didn't include event handlers associated with the components. The following example demonstrates how to save a group of components that rely on an event handler.

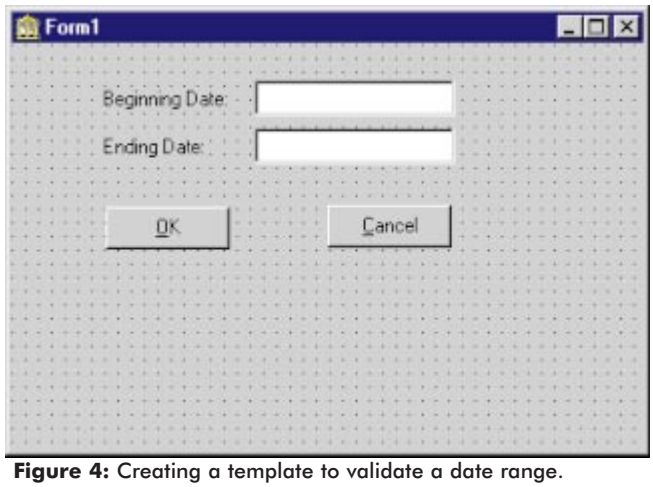


Figure 4: Creating a template to validate a date range.

In this example, we'll create a date-range component — one that enables a user to enter a beginning and an ending range of dates. What's more, this component will include the code necessary to validate the dates entered. It's designed to be placed on a modal dialog box.

Begin by creating a new form. Like the preceding example, this form can be associated with an existing project, or it can be a stand-alone form. Next, place onto this form the following components: two Labels, two Edits, and two Buttons. Set the *Caption* property of *Label1* to `Beginning Date`, and the *Caption* property of *Label2* to `Ending Date`. Next, set the *Text* property of both Edit components to a single blank space. (Don't set the *Text* property to an empty string; instead, press Spacebar at least once.)

The Buttons are next. Set the *Caption* property of *Button2* to `&Cancel`, and the *ModalResult* property to *mrCancel*. Next, set the *Caption* property of *Button1* to `&OK`. The form should look like Figure 4. Don't set the *ModalResult* property of *Button1*; the behavior of this button must be defined using an event handler. To do this, double-click *Button1* to create an *OnClick* event handler for it. Modify as follows:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  bd, ed: TDateTime;
begin
  bd := StrToDate(Edit1.Text);
  ed := StrToDate(Edit2.Text);

  if (bd > ed) then
    raise Exception.Create(
      'Ending date cannot precede beginning date');

  Self.ModalResult := mrOK;
end;
```

Save the new component template; select Edit | Select All, then select Component | Create Component Template. In the Component Template Information dialog box, set the Component name to `DateRange`, then select OK.

To test the new template, create a new project. Add a second form to the project by selecting File | New Form. On the newly created *Form2*, add the DateRange template. Return

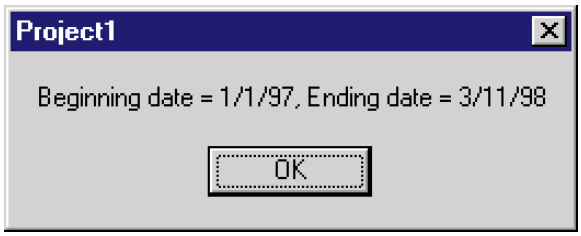


Figure 5: The ShowMessage dialog box displays the validated date range.

Figure 6: Entering an invalid date prompts this message.

now to *Form1*, add a Button, then add the following event handler to the button's *OnClick* event handler:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Form2.ShowModal = mrOK then
    ShowMessage('Beginning date = ' + Form2.Edit1.Text +
                ', Ending date = '  + Form2.Edit2.Text)
  else
    ShowMessage('Form2 was cancelled');
end;
```

Finally, add Unit2 to Unit1's **uses** clause by selecting **File | Use Unit**, then double-clicking Unit2 on the Use Unit dialog box. Now run the form. Click the button to display *Form2* with the DateRange template on it. Enter a valid date range, then click **OK**. *Form1* displays the entered date range, as shown in Figure 5. Now click the button again, and enter an invalid value in one of the date fields. Selecting **OK** displays the exception shown in Figure 6.

The code added to the component template was stored with the template. This code, however, uses explicit references to objects that are part of the template. Specifically, in the DateRange template, the saved code refers to *Edit1* and *Edit2*. What happens when the form on which you place the template already has an *Edit1* or *Edit2* component on it? The answer is that the components of the template are renamed, and the code in the template is updated.

Deleting Component Templates

Component templates, which are stored in the file DELPHI32.DCT in Delphi's \Bin directory, can easily be deleted. To delete a component template, right-click the Component palette and select **Properties**. Alternatively, select **Tools | Environment** to display the Environment Options dialog box, and select the Palette page. From the displayed dialog box (see Figure 7), select **Templates** in the **Pages** listbox. Then select the component template you want to delete in the **Components** listbox, and click **Delete**.

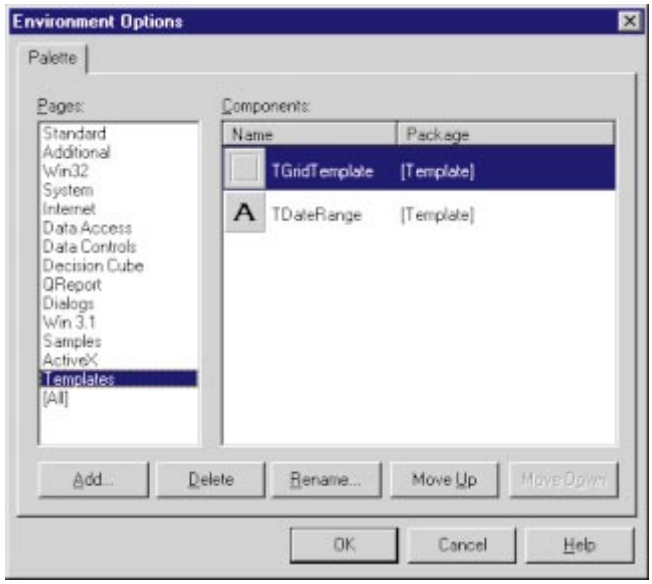


Figure 7: Use the Palette page of the Environment Options dialog box to delete component templates.

Component-Template Guidelines

While component templates are an important tool for rapid application development in Delphi, they can easily be overused. Avoid dependencies on components that aren't part of the template. For example, you might want to create a template for a RichEdit toolbar, without the RichEdit control being part of the template. This toolbar template, however, requires that a RichEdit component be present, to compile properly. If you must create dependent templates, be sure to add comments to the code references in the template's event handlers, indicating which other objects are required by the template, and what these components must be named.

If you use a particular component template a lot, consider turning it into a custom component. Templates do not support inheritance, whereas true components do. Only published properties can be saved as part of a component template. If your component must set properties that are not published, you'll need to create a true component. Also, note that not all published properties of a component template will be preserved. For example, a component template based on a panel whose *Caption* property has been set to an empty string will still be assigned a caption when you place the template. This is because a panel's caption is controlled by the presence of the *csSetCaption* flag in the *ControlStyle* property. (However, setting a Panel's *Caption* to a space will create a caption-less panel template.) Δ

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

table of CONTENTS

Mastering Delphi 3

Marco Cantù's *Mastering Delphi* [SYBEX, 1995] was one of the first Delphi books I bought. Despite its wide focus and encyclopedic nature, I found myself coming back to it fairly often as a reference. Like its predecessor, *Mastering Delphi 3* covers the Delphi landscape in surprising detail. It's this that sets it apart from so many other general introductory texts. For the same reason, I expect to continue to spend many additional hours with this book as I explore new areas of Delphi.

The book is organized into four large sections. The first chapter begins with building forms. The more advanced Delphi developers reading this are probably thinking, "I could skip all of that." However, there are useful tips throughout the entire volume, and you would be missing some if you did.

Chapter two provides an overview of the IDE, including a comprehensive list of file types that are part of Delphi, from .BMP, .OCX, and .~DF to .DPK with useful information on each. Chapter three introduces Delphi's Object Repository and built-in Experts (Wizards). The remaining chapters in the introductory section deal with Object Pascal and the VCL. The coverage is quite detailed with some unexpected topics. For example, how many introductory references bother to discuss procedural types and method pointers? This one does! On pages 180-182 there is a discussion on another rather advanced topic: Windows callback functions. Also within this section is an excellent discussion of the various types of strings available in Delphi.

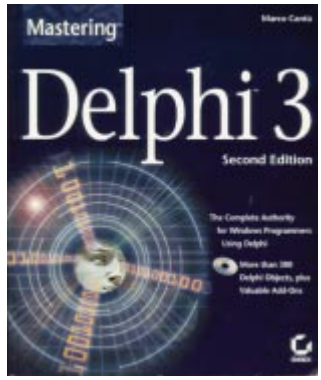
The second part is devoted to using components, starting with the simplest and ending with a couple of chapters on database components. Again, Cantù will delight you with his depth of coverage. How much can one write about using a simple button — a page or two? Cantù takes about five pages to demonstrate not only how to place and align a button (expected), but how to disable/enable, enlarge, shrink, hide, show, and change the font of a button.

Data entry forms are extremely common in Windows programming. Cantù devotes a lot of time to discussing the components used in data entry forms, including at least one that's new in Delphi 3: the DateTimePicker. Of course there are in-depth discussions on menus and toolbars, and yes, even the famous (or is it infamous?) CoolBar. In chapter ten, which focuses on forms and windows, Cantù explains how to set a form's style, use the *BorderStyle* property, avoid screen flicker, and enable direct keyboard input to a form.

Marco Cantù understands and appreciates that programming can be fun. (His popular presentation at this year's Borland Developers Conference in Nashville was devoted to the fun side of Delphi.) In this section, he shows how to use a graphical grid to write a simple Windows game with Delphi; toward the end of the book, he presents a chapter on Multimedia Fun.

Having devoted about a third of the book to an exposition of Delphi components, Cantù turns his attention to more advanced topics in part three, "Components and Libraries." The first two chapters provide an introduction to writing components, component and property editors, and experts. These are followed by chapters on DLLs, OLE, COM, ActiveX controls, and Internet programming.

The final section of the book is entitled "Advanced Delphi Programming." I'm not convinced the topics presented here are more advanced than some in the previous section, but every one of them is important. Among the clearly advanced topics, the one on threads and multitasking is probably the best introduction I've seen — and I've seen some good ones. While I wouldn't consider the chapter on printing in Delphi to be particularly advanced, I did find it useful and comprehensive. Likewise, the chapter on working with files provides an excellent introduction to working with Delphi streams. However, you'll probably want to read what Ray Lischner has to say about this topic in



Secrets of Delphi 2 [Waite Group Press, 1996] before trying anything too fancy!

Though it may seem I have written a lot of detailed information about Cantù's excellent book, I've only scratched the surface. From beginning to end, it's packed with useful tips, information about new features, and warnings about traps in the different versions of Delphi. For once, I agree with the publisher that this book is truly appropriate for all levels of Delphi programmers. I would particularly recommend it to someone just getting started with Delphi, or who is moving to Delphi 3 from an earlier version. Of all the general Delphi references, I think Marco Cantù's *Mastering Delphi 3* is the best written and the most comprehensive.

— Alan C. Moore, Ph.D.

Mastering Delphi 3 by Marco Cantù, SYBEX, 1151 Marina Village Parkway, Alameda, CA 94501, (510) 523-8233.
ISBN: 0-7821-2052-0
Price: US$49.99
(1,476 pages, CD-ROM)

table of CONTENTS

Searching for Our Industry's Soul

I recently came across an online Salon 21st article by Jennifer New (http://www.salonmagazine.com/-sept97/21st/gates970925.html) about her unhappy experiences working as a contractor for Microsoft. Jennifer tells tale after tale of the arrogance, cold-heartedness, and, frankly, amoral attitudes of many Microsoft employees. One anecdote captures the heart of her story when she describes greeting an individual in the hallway and having him respond: "Do I need to know you?" After reflecting on this, I asked myself whether this situation is unique to Microsoft, or does it reflect our software industry as a whole? While perhaps this attitude peaks in Redmond, it seems that when you look for the "soul" in our high-tech culture, it's hard to find.

To begin, we in the software industry are often arrogant. Part of this is due to the composition of the software developer community, made up largely of young, highly educated, well paid, and self-motivated individuals. While none of these qualities are bad, if left unchecked — and mixed with a large ego — they can lead to a feeling of invincibility and superiority. The extreme success the software industry has had over the past 10-15 years has surely helped facilitate a feeling of haughtiness. What other industry has produced so many millionaires under 30 years of age? Part of this arrogance is due to the "heroes" of our trade; names such as Bill Gates, Larry Ellison, Steve Jobs, and Phillipe Kahn don't ring of humility. Sadly, once considered a vice, "pride" is now considered a virtue. Somehow we have substituted "ego" for "self-confidence," the result of which inevitably leads to this arrogance.

Let's face it: The software industry is amoral. While you could say this about society as a whole, I believe this attitude reaches a zenith within our cyberculture. Ruthlessness and Machiavellian principles are usually considered prerequisites to achieving success in high tech. Decency tends to be equated with weakness, or even prudishness, and morality is left to personal interpretation.

A prime example of this "cybermorality" is Web pornography. Not only is its use widespread, it's sometimes even glorified by our digital culture in the name of free speech. In fact, I have yet to see anyone in our industry dare speak against the shortcomings of the Blue Ribbon campaign or address the impact pornography has on individuals. Somehow in our zeal for free speech, we've forgotten that responsibility must accompany that freedom. Dr Alan Keyes expresses this belief in *Our Character, Our Future* [Zondervan, 1996]: "Freedom requires that at the end of the day, we accept the constraint that is required." Our cybermorality rejects such constraints and dismisses any notion of "absolutes" apart from an individual.

You may be asking "so what?" What does this have to do with us? I think it points to the heart of the issue. Our cyberculture, which is largely influenced by the high-tech industry and digital media, has created a vacuum within which anything can propagate without anyone ever asking if it's right, decent, or good for the community at large.

Are all of us in the high-tech industry arrogant, cold-hearted, and amoral? Of course not. I know many software engineers and media professionals who have integrity and who are decent, humble, and respectful of their co-workers. I hope many of you would classify yourselves in that vein. But if we all think of ourselves that way, why is our industry known for producing clones of Bill Gates instead of Mother Teresa? Why are qualities like decency and humility often ridiculed? Why are greed and ruthlessness admired? Until such attitudes change, our industry will remain without a soul.

Does the software industry have a "soul?" Let me know what you think by writing me at rwagner@acadians.com. Δ

— Richard Wagner

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

table of CONTENTS